**Spring IoC (Inversion of Control)** Container is the core of [Spring Framework](https://www.geeksforgeeks.org/introduction-to-spring-framework/). It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.  
**There are 2 types of IoC containers:**

[BeanFactory](https://www.geeksforgeeks.org/spring-beanfactory/)

[ApplicationContext](https://www.geeksforgeeks.org/spring-applicationcontext/)

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The followings are some of the main features of Spring IoC,

1. Creating Object for us
2. Managing our objects,
3. Helping our application to be configurable,
4. Managing dependencies

Implementation: So now let’s understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().

Java

|  |
| --- |
| ***// Java Program to Illustrate Sim Interface***  ***public interface Sim***  ***{***  ***void calling();***  ***void data();***  ***}*** |

Now we have created another two classes Airtel and Jio which implement the Sim interface and override the interface methods.

Java

|  |
| --- |
| ***// Java Program to Illustrate Airtel Class***    ***// Class***  ***// Implementing Sim interface***  ***public class Airtel implements Sim {***    ***@Override public void calling()***  ***{***  ***System.out.println("Airtel Calling");***  ***}***    ***@Override public void data()***  ***{***  ***System.out.println("Airtel Data");***  ***}***  ***}*** |

Java

|  |
| --- |
| ***// Java Program to Illustrate Jio Class***    ***// Class***  ***// Implementing Sim interface***  ***public class Jio implements Sim{***  ***@Override***  ***public void calling() {***  ***System.out.println("Jio Calling");***  ***}***    ***@Override***  ***public void data() {***  ***System.out.println("Jio Data");***  ***}***  ***}*** |

***+***  
So let’s now call these methods inside the main method. So by implementing the [Run time polymorphism](https://www.geeksforgeeks.org/difference-between-compile-time-and-run-time-polymorphism-in-java/) concept we can do something like this

Java

|  |
| --- |
| ***// Java Program to Illustrate Mobile Class***    ***// Class***  ***public class Mobile {***    ***// Main driver method***  ***public static void main(String[] args)***  ***{***    ***// Creating instance of Sim interface***  ***// inside main() method***  ***// with reference to Jio class constructor***  ***// invocation***  ***Sim sim = new Jio();***    ***// Sim sim = new Airtel();***    ***sim.calling();***  ***sim.data();***  ***}***  ***}*** |

But what happens if in the future another new Sim Vodafone came and we need to change again to the child class name in the code, like this

Sim sim = new Vodafone();

So we have to do our configuration in the source code. So how to make it configurable? We don’t want to touch the source code of this. The source code should be constant. And how can we make it? Here Spring IoC comes into the picture. So in this example, we are going to use ApplicationContext to implement an IoC container. First, we have to create an XML file and name the file as “beans.xml“.

Example: beans.xml File

XML

|  |
| --- |
| ***<?xml version="1.0" encoding="UTF-8"?>***  ***<beans xmlns="<http://www.springframework.org/schema/beans>"***  ***xmlns:xsi="<http://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="sim" class="Jio"></bean>***    ***</beans>*** |

Output Explanation: In the beans.xml file, we have created beans. So inside the id, we have to pass the unique id and inside the class, we have to pass the Class name for which you want to create the bean. Later on, inside the main method, we can tweek it out that will be described in the upcoming pr

ogram.

Bean Definition: In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Java

|  |
| --- |
| ***import org.springframework.context.ApplicationContext;***  ***import org.springframework.context.support.ClassPathXmlApplicationContext;***    ***public class Mobile {***  ***public static void main(String[] args) {***  ***// Using ApplicationContext tom implement Spring IoC***  ***ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");***    ***// Get the bean***  ***Sim sim = applicationContext.getBean("sim", Sim.class);***    ***// Calling the methods***  ***sim.calling();***  ***sim.data();***  ***}***  ***}*** |

Output:

Jio Calling

Jio Data

And now if you want to use the Airtel sim so you have to change only inside the beans.xml file. The main method is going to be the same.

<bean id="sim" class="Airtel"></bean>

Java

|  |
| --- |
| ***import org.springframework.context.ApplicationContext;***  ***import org.springframework.context.support.ClassPathXmlApplicationContext;***    ***public class Mobile {***  ***public static void main(String[] args) {***    ***// Using ApplicationContext tom implement Spring IoC***  ***ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");***    ***// Get the bean***  ***Sim sim = applicationContext.getBean("sim", Sim.class);***    ***// Calling the methods***  ***sim.calling();***  ***sim.data();***  ***}***  ***}*** |

Output:

Airtel Calling

Airtel Data

# **Spring – BeanFactory**

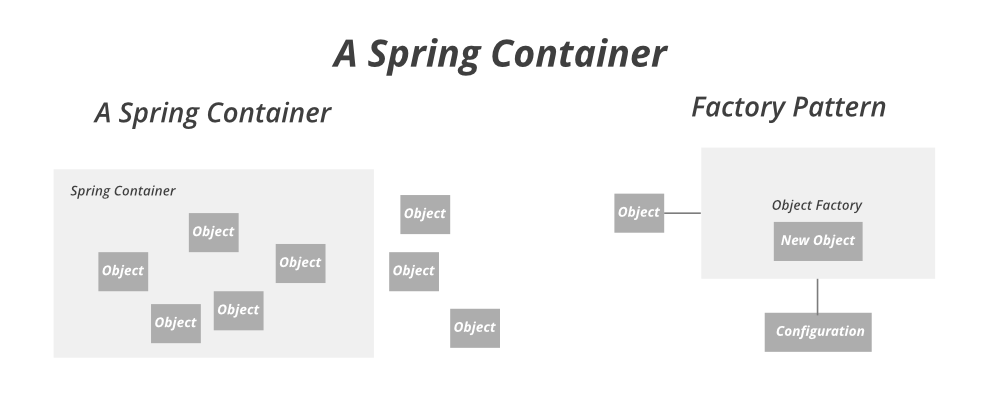
**The first foremost thing when we talk about spring is dependency injection which is possible because spring is actually a container and behaves as a factory of Beans.**

**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 file, etc.**

**BeanFactory does not support Annotation-based configuration whereas ApplicationContext does.**

****

**Let us do first go through some of the methods of Bean factory before landing up on implementation which are shown below in tabular format below as follows:**

| **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 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.** |

**Creating a Spring project using [start.spring.io](https://start.spring.io/" \t "https://www.geeksforgeeks.org/spring-beanfactory/_blank).**

**Creating a POJO class.**

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

**Writing it to application class.**

**Implementation:**

**Step 1: Bean Definition: Create a Student POJO class.**

**// 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: XML Bean Configuration: Configure the Student bean in the bean-factory-demo.xml file.**

**<?xml version = "1.0" encoding="UTF-8"?>**

**<beans xmlns = "http://www.springframework.org/schema/beans"**

**xmlns:xsi="http://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: Main Class**

**// Application class**

**@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'}**

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

# **Spring – ApplicationContext**

Spring IoC container is responsible for instantiating, wiring, configuring, and managing the entire life cycle of objects. BeanFactory and ApplicationContext represent the Spring IoC Containers. ApplicationContext is the sub-interface of BeanFactory. BeanFactory provides basic functionalities and is recommended to use for lightweight applications like mobile and applets. ApplicationContext provides basic features in addition to enterprise-specific functionalities which are as follows:

* Publishing events to registered listeners by resolving property files.
* Methods for accessing application components.
* Supports Internationalization.
* Loading File resources in a generic fashion.

***Note:****It is because of these additional features, developers prefer to use ApplicationContext over BeanFactory.*

**ApplicationContext Implementation Classes**

There are different types of Application containers provided by Spring for different requirements as listed below which later onwards are described alongside with declaration, at lastly providing an example to get through the implementation part with the pictorial aids. Containers are as follows:

1. AnnotationConfigApplicationContext container
2. AnnotationConfigWebApplicationContext
3. XmlWebApplicationContext

**Container 1:**AnnotationConfigApplicationContext

AnnotationConfigApplicationContext class was introduced in Spring 3.0. It accepts classes annotated with @Configuration, @Component**,** and JSR-330 compliant classes. The constructor of AnnotationConfigApplicationContext accepts one or more classes. For example, in the below declaration, two Configuration classes **Appconfig and AppConfig1**are passed as arguments to the constructor. The beans defined in later classes will override the same type and name beans in earlier classes when passed as arguments. For example, AppConfig and AppConfig1 have the same bean declaration. The bean defined in AppConfig1 overrides the bean in AppConfig.

**Syntax:**Declaration

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class, AppConfig1.class);

***Note:****Add the following to the properties file in the IDE to allow the spring to override beans.*

spring.main.allow-bean-definition-overriding=true

**Container 2:**AnnotationConfigWebApplicationContext

AnnotationConfigWebApplicationContext class was introduced in Spring 3.0. It is similar to AnnotationConfigApplicationContext for a web environment. It accepts classes annotated with @Configuration, @Component, and JSR-330 compliant classes. These classes can be registered via ***register() method***or passing base packages to ***scan() method***. This class may be used when we configure ContextLoaderListener servlet listener or a [DispatcherServlet](https://www.geeksforgeeks.org/spring-mvc-with-jsp-view/) in a web.xml. From Spring 3.1, this class can be instantiated and injected to DispatcherServlet using java code by implementing WebApplicationInitializer, an alternative to web.xml.

**Example**

// Class

// Implementing WebApplicationInitializer

public class MyWebApplicationInitializer implements WebApplicationInitializer {

// Servlet container

public void onStartup(ServletContext container) throws ServletException {

AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();

context.register(AppConfig.class);

context.setServletContext(container);

// Servlet configuration

}

}

**Container 3:**XmlWebApplicationContext

Spring MVC Web-based application can be configured completely using XML or Java code. Configuring this container is similar to the AnnotationConfigWebApplicationContext container, which implies we can configure it in web.xml or using java code.

// Class

// Implementing WebApplicationInitializer

public class MyXmlWebApplicationInitializer implements WebApplicationInitializer {

// Servlet container

public void onStartup(ServletContext container) throws ServletException {

XmlWebApplicationContext context = new XmlWebApplicationContext();

context.setConfigLocation("/WEB-INF/spring/applicationContext.xml");

context.setServletContext(container);

// Servlet configuration

}

}

**Container 4:**FileSystemXmlApplicationContext

FileSystemXmlApplicationContext is used to load XML-based Spring Configuration files from the file system or from URL. We can get the application context using Java code. It is useful for standalone environments and test harnesses. The following code shows how to create a container and use the XML as metadata information to load the beans.

**Illustration:**

String path = "Documents/demoProject/src/main/resources/applicationcontext/student-bean-config.xml";

ApplicationContext context = new FileSystemXmlApplicationContext(path);

AccountService accountService = context.getBean("studentService", StudentService.class);

**Container 5:**ClassPathXmlApplicationContext

FileSystemXmlApplicationContext is used to load XML-based Spring Configuration files from the classpath. We can get the application context using Java code. It is useful for standalone environments and test harnesses. The following code shows how to create a container and use the XML as metadata information to load the beans.

**Illustration:**

ApplicationContext context = new ClassPathXmlApplicationContext("applicationcontext/student-bean-config.xml");

StudentService studentService = context.getBean("studentService", StudentService.class);

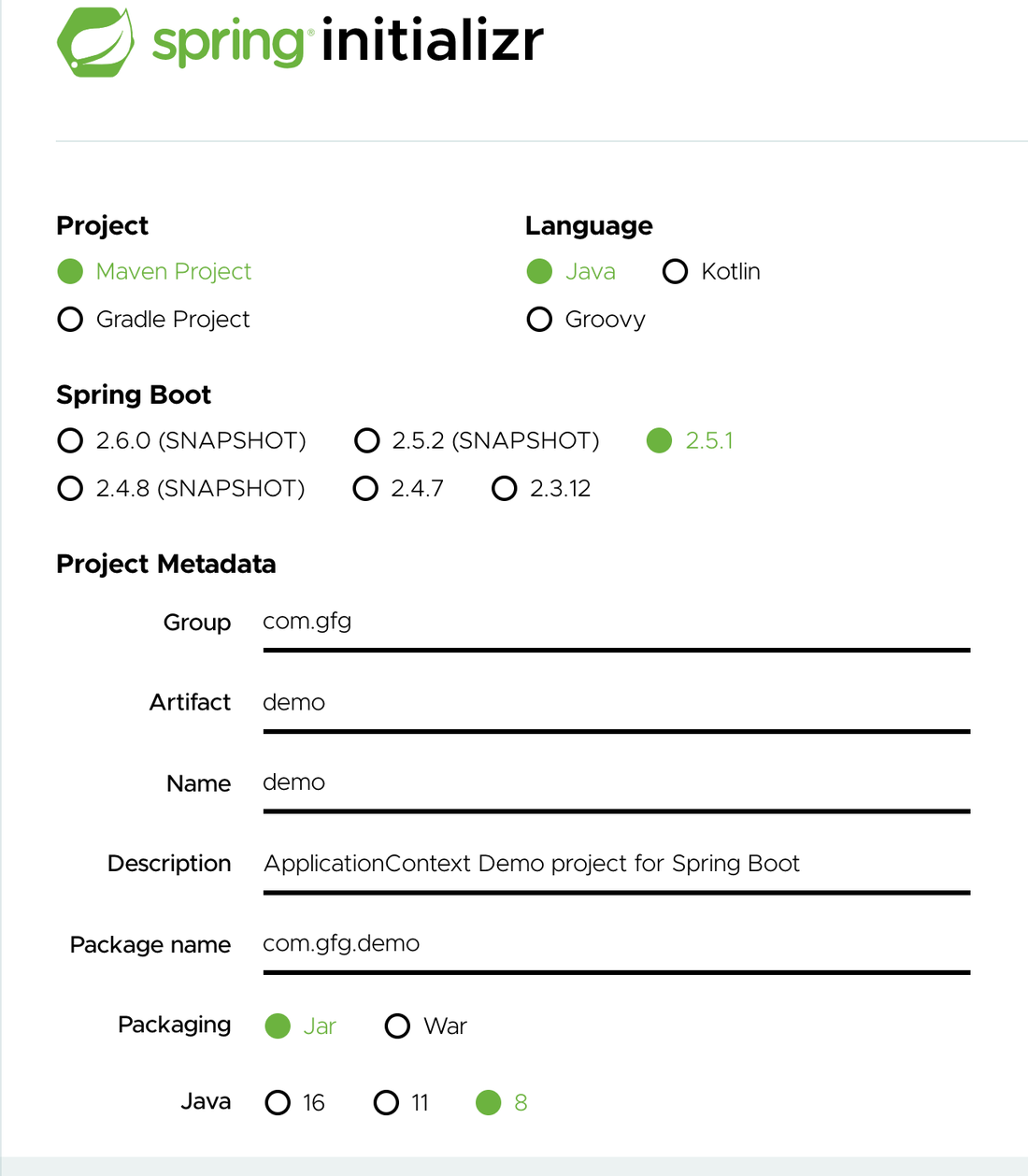
Now, let us implement the same showcasing an example which is as follows:

**Implementation:**

* Create a Spring Project using [Spring Initializer](https://start.spring.io/" \t "https://www.geeksforgeeks.org/spring-applicationcontext/_blank).
* Create Student class under **com.gfg.demo.domain**
* Similarly,AppConfig class under **com.gfg.demo.config**packages.
* The main application class at the root contains the creation of a container.
* Lastly, the **SpringApplication.run()**method is provided by default in the main class when the SpringBoot project is created.

**Example**

Step 1: Creating a Spring Project using [Spring Initializer](https://start.spring.io/" \t "https://www.geeksforgeeks.org/spring-applicationcontext/_blank) as pictorially depicted below.



**Step 2:** Create Student class under ***com.gfg.demo.domain***and**AppConfig class** under ***com.gfg.demo.config***packages. The AppConfig is the configuration class that contains all the Java beans configured using Java Based Configuration. The Student class is the [POJO class](https://www.geeksforgeeks.org/pojo-vs-java-beans/).

* **Class 1:** AppConfig class

@Configuration

// Class

public class AppConfig {

@Bean

// Method

public Student student() {

return new Student(1, "Geek");

}

}

* **Class 2:**Student class

// Class

public class Student {

// member variables

private int id;

private String name;

// Constructor 1

public Student() {}

// Constructor 2

public Student(int id, String name) {

this.id = id;

this.name = name;

}

// Method of this class

// @Override

public String toString() {

return "Student{" + "id=" + id + ", name='" + name + '\'' + '}';

}

}

**Step 3:** Now the Main Application class at the root contains the creation of a container.

// Class

// @SpringBootApplication

public class DemoApplication {

// Main driver method

public static void main(String[] args) {

// SpringApplication.run(DemoApplication.class, args);

// Creating its object

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Student student = context.getBean(Student.class);

// Print and display

System.out.println(student);

}

}

**Step 4:**The **SpringApplication.run()**method is provided by default in the main class when the SpringBoot project is created. It creates the container, creates beans, manages dependency injection and life cycle of those beans.This is done using [@SpringBootApplication annotation](https://www.geeksforgeeks.org/introduction-to-spring-boot/).

// Main driver method

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(DemoApplication.class, args);

Student student = context.getBean(Student.class);

// Print and display

System.out.println(student);

}