Spring Framework

IoC : Inversion of Control

https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes

IoC : Inversion of Control

- The basic concept of the **dependency injection** (also known as **Inversion of Control pattern**) is that you do not create your objects but describe how they should be created.
  - Dependent components are never instantiated using a new operator within component classes. They are injected into the component by the container instance at run time.
- You don’t directly connect your components and services together in code but describe which services are needed by which components in a configuration file.
- A container (in the case of the Spring framework, **the IOC container**) is then responsible for hooking it all up.

Tasks/Types of IoC Container

The main tasks performed by IoC container are:

- Instantiate the application class
- Configure the object
- Assemble the dependencies between the objects

Types of IoC containers.

- BeanFactory Interface
- **ApplicationContext Interface**
  - It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context for web application.
  - It is better to use ApplicationContext than BeanFactory.

Spring Dependency Injection

Where to inject

- Constructor Injection
- Setter Injection
- Field Injection (annotation only)

Spring assigns the dependencies directly to the fields by using Java Reflections.

Difference between constructor and setter injection

- Partial dependency: it is possible by setter method only.
- Overriding: Setter injection overrides the constructor injection.
  - If we use both constructor and setter injection, IoC container will use the setter injection.
- Changes: We can easily change the value by setter injection.
  - It doesn't create a new bean instance always like constructor.
  - Setter injection is flexible than constructor injection.

Spring Dependency Injection

What to inject ?

- Literal Value Injection (primitive and String-based values)
- Object Injection
- Collection values etc.
  - List
  - Set
  - Map

```
public class Question {
  private int id;
  private String name;
  private List<String> answers;
  public Question() {
  }
  public Question(int id, String name) {
    super();
    this.id = id;
    this.name = name;
  }

  public Question(int id, String name, List<String> answers) {
    super();
    this.id = id;
    this.name = name;
    this.answers = new ArrayList<>();
  }
```

Collection values injection example

```
<bean id="q2" class="sit.int204.example.model.Question">
   <constructor-arg value="001"></constructor-arg>
   <constructor-arg value="What is java?"></constructor-arg>
   <property name="answers">
      <list>
         <value>Java is a programming language</value>
         <value>Java is a Platform</value>
         <value>Java is an Island of Indonesia</value>
         <value>Java is a language that make me sad</value>
      </list>
   </property>
</bean>
```

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

- no
  - It is the default autowiring mode. It means no autowiring bydefault.
- 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.
- byType
  - The byType mode injects the object dependency according to type.
  - Property name and bean name can be different.
  - It internally calls setter method.
  - There must be only one bean of a type.
- constructor
  - The constructor mode injects the dependency by calling the constructor of the class.
  - It calls the constructor having large number of parameters.
- autodetect
  - It is deprecated since Spring 3.

Autowiring example (1)

byName

```
 <bean id="engine" class="sit.int204.lab01.beans.DieselEngine">
   <property name="capacity" value="4990"/>
 </bean>
 <bean id="car" class="sit.int204.lab01.beans.Car" autowire="byName">
   <property name="chasisNumber" value="ZE3197-9485M"/>
   <property name="brand" value="Benz E520D"/>
</bean>
```

byType

```
<bean id=“oneKd" class="sit.int204.lab01.beans.DieselEngine">
  <property name="capacity" value="2982"/>
</bean>
<bean id="carX" class="sit.int204.lab01.beans.Car" autowire="byType">
  <property name="chasisNumber" value="ZE3197-9485M"/>
  <property name="brand" value="Benz E520D"/>
</bean>
```

Autowiring example (2)

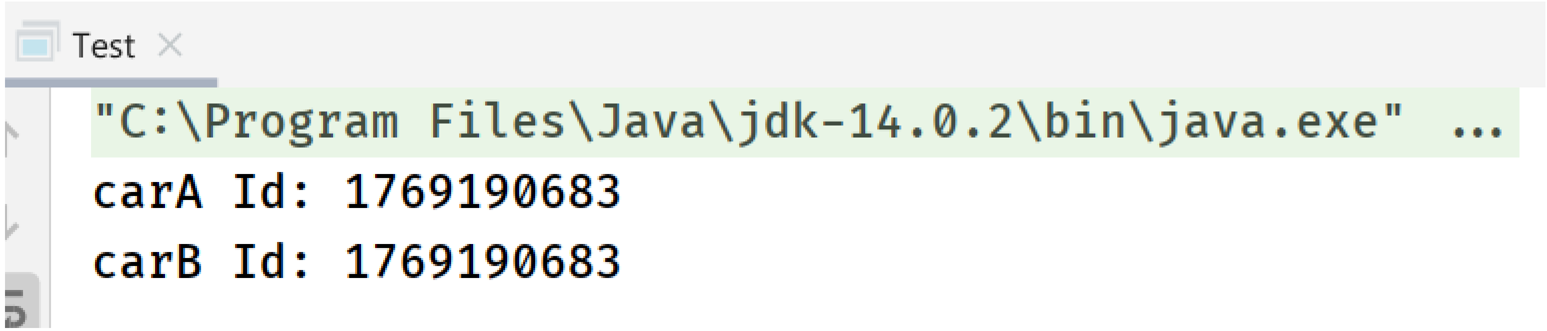
```
 <bean id="1KD-FTV" class="sit.int204.lab01.beans.DieselEngine">
   <property name="capacity" value="2982"/>
</bean>
<bean id="brand" class="java.lang.String">
   <constructor-arg value="Toyota"/>
</bean>
<bean id="chasisNumber" class="java.lang.String">
   <constructor-arg value="ZQ12345MZ"/>
</bean>

<bean id="carX" class="sit.int204.lab01.beans.Car" autowire="constructor“/>
```

Spring Bean Scopes

- **Singleton**
  - (Default) Scopes a single bean definition to **a single object instance for each Spring IoC container.**
- **prototype**
  - Scopes a single bean definition to **any number of object instances.**
- request
  - Scopes a single bean definition to the lifecycle of a single HTTP request.
- session
  - Scopes a single bean definition to the lifecycle of an HTTP Session.
- application
  - Scopes a single bean definition to the lifecycle of a ServletContext.
- websocket
  - Scopes a single bean definition to the lifecycle of a WebSocket.

Singleton scope (Default)

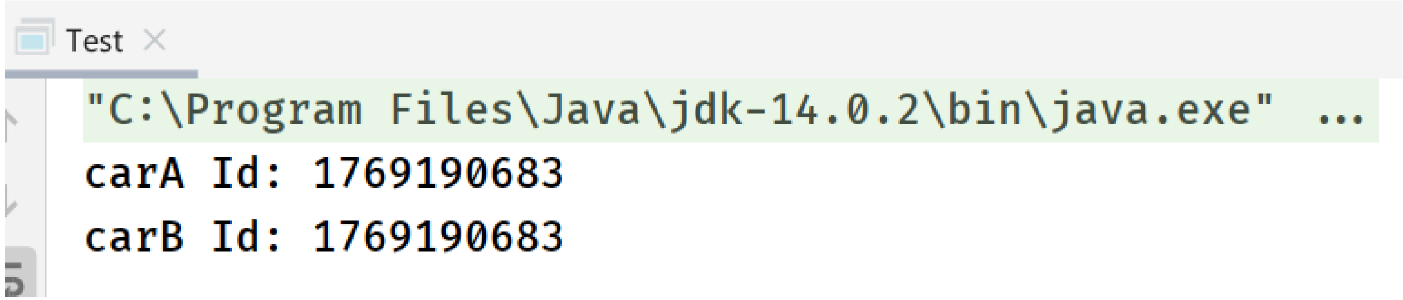
```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        Car carA = (Car) context.getBean("carX");
        System.out.println("carA Id: "+ System.identityHashCode(carA));
        Car carB = (Car) context.getBean("carX");
        System.out.println("carB Id: "+ System.identityHashCode(carB));
```



```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
carA Id: 1769190683
carB Id: 1769190683
```

Singleton scope (Default)

```java
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        Car carA = (Car) context.getBean("carX");
        System.out.println("carA Id: "+ System.identityHashCode(carA));
        Car carB = (Car) context.getBean("carX");
        System.out.println("carB Id: "+ System.identityHashCode(carB));
```

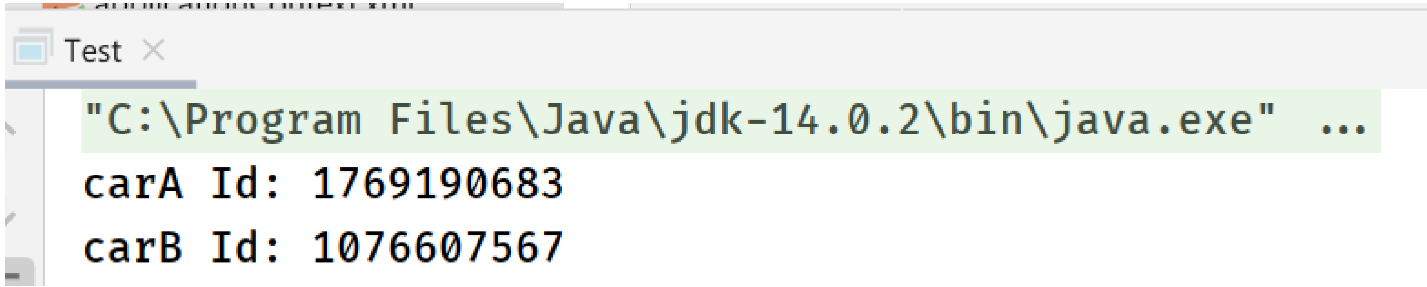


```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
carA Id: 1769190683
carB Id: 1769190683
```

Prototype Scope

```
<bean id="carX" class="sit.int204.lab01.beans.Car"
autowire="constructor" scope="prototype“/>
```

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "applicationContext.xml");
        Car carA = (Car) context.getBean("carX");
        System.out.println("carA Id: "+ System.identityHashCode(carA));
        Car carB = (Car) context.getBean("carX");
        System.out.println("carB Id: "+ System.identityHashCode(carB));
```



```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
carA Id: 1769190683
carB Id: 1076607567
```

Creational Design Patterns for creating an object

- **Singleton** – only one and shared object for a class
- **Prototype** – cloning an existing object
- **Factory Method** – creating without specifying an exact class
- Abstract Factory – creating objects with a theme

Factory Method

- Builder – creating a complex object possibly with many options

Factory Method Types

There can be three types of factory method:

1. A static factory method that returns instance of its own class. It is used in singleton design pattern.
2. A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.
3. A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

Type 1

```
<bean id=“x" class=“sit.int204.example.factory.X" factory-method="getX“/>
```

```
public class X {
    private static final X obj = new X();
    private X() {
        System.out.println("private constructor");
    }
    public static X getX() {
        System.out.println("factory method ");
        return obj;
    }
    public void msg() {
        System.out.println("hello user");
    }
}
```

Type 2

```
<bean id="p" class=“package.PrintableFactory" factory-method="getPrintable"/>
```

Printable.java

```
public interface Printable {
void print();
}
```

A.java

```
public class A implements Printable{
   @Override
   public void print() {
      System.out.println("hello a");
   }
}
```

B.java

```
public class B implements Printable{
   @Override
   public void print() {
      System.out.println("hello b");
   }
}
```

PrintableFactory.java

```
public class PrintableFactory {
  public static Printable getPrintable(){
      //return new B();
     return new A();//return any one instance, either A or B
  }
}
```

Type 3

- All files are same as previous, you need to change only 2 files: PrintableFactory and applicationContext.xml.

```
public class PrintableFactory {
    //non-static factory method
    public Printable getPrintable(){
        return new A();//return any one instance, either A or B
   }
}
```

```
<bean id="pfactory" class=“package.PrintableFactory"/>

<bean id="p" class=“package.PrintableFactory" factory-method="getPrintable"
factory-bean="pfactory"/>
```

Code Based Configuration

```
package sit.int204.lab01.config;
@Configuration
public class ApplicationConfig {
    @Bean(name = "car")
    public Car getCar() {
        return new Car("ZM4969JXX", "Toyota-Fortuner");
    }
}
```

```
public static void main(String[] args) {
   ApplicationContext context = new AnnotationConfigApplicationContext(ApplicationConfig.class);
   Car car = context.getBean("car", Car.class);
   car.start();
   System.out.println(car);
}
```