Intro to Inversion of Control and Dependency Injection with Spring

By Loredana Crusoveanu December 28, 2016

1. Overview

In this tutorial, we'll introduce the concepts of IoC (Inversion of Control) and DI (Dependency Injection), as well as take a look at how these are implemented in the Spring framework.

2. What Is Inversion of Control?

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

- decoupling the execution of a task from its implementation
- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

We're going to look at DI next.

3. What Is Dependency Injection?

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.

Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

Here's how we would create an object dependency in traditional programming:

```
public class Store {
     private Item item;
     public Store() {
          item = new ItemImpl1();
     }
}
```

In the example above, we need to instantiate an implementation of the *Item* interface within the *Store* class itself.

By using DI, we can rewrite the example without specifying the implementation of the *Item* that we want:

```
public class Store {
     private Item item;
     public Store(Item item) {
          this.item = item;
     }
}
```

In the next sections, we'll look at how we can provide the implementation of Item through metadata.

Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

4. The Spring IoC Container

An IoC container is a common characteristic of frameworks that implement IoC.

In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles.

The Spring framework provides several implementations of the *ApplicationContext* interface: *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* for standalone applications, and *WebApplicationContext* for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Here's one way to manually instantiate a container:

```
ApplicationContext context
     = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

To set the *item* attribute in the example above, we can use metadata. Then the container will read this metadata and use it to assemble beans at runtime.

Dependency Injection in Spring can be done through constructors, setters or fields.

5. Constructor-Based Dependency Injection

In the case of constructor-based dependency injection, the container will invoke a constructor with arguments each representing a dependency we want to set.

Spring resolves each argument primarily by type, followed by name of the attribute, and index for disambiguation. Let's see the configuration of a bean and its dependencies using annotations:

```
@Configuration
public class AppConfig {
```

```
    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

The *@Configuration* annotation indicates that the class is a source of bean definitions. We can also add it to multiple configuration classes.

We use the *@Bean* annotation on a method to define a bean. If we don't specify a custom name, then the bean name will default to the method name.

For a bean with the default *singleton* scope, Spring first checks if a cached instance of the bean already exists, and only creates a new one if it doesn't. If we're using the *prototype* scope, the container returns a new bean instance for each method call.

Another way to create the configuration of the beans is through XML configuration:

```
<bean id="item1" class="org.baeldung.store.ItemImpl1" />
<bean id="store" class="org.baeldung.store.Store">
<constructor-arg type="ItemImpl1" index="0" name="item"
ref="item1" />
</bean>
```

6. Setter-Based Dependency Injection

For setter-based DI, the container will call setter methods of our class after invoking a noargument constructor or no-argument static factory method to instantiate the bean. Let's create this configuration using annotations:

```
@Bean
```

```
public Store store() {
     Store store = new Store();
     store.setItem(item1());
     return store;
}
```

We can also use XML for the same configuration of beans:

```
<bean id="store" class="org.baeldung.store.Store">
     <property name="item" ref="item1" />
</bean>
```

We can combine constructor-based and setter-based types of injection for the same bean. The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

7. Field-Based Dependency Injection

In case of Field-Based DI, we can inject the dependencies by marking them with an *@Autowired* annotation:

```
public class Store {
     @Autowired
     private Item item;
}
```

While constructing the *Store* object, if there's no constructor or setter method to inject the *Item* bean, the container will use reflection to inject *Item* into *Store*.

We can also achieve this using XML configuration.

This approach might look simpler and cleaner, but we don't recommend using it because it has a few drawbacks such as:

- This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection.
- It's really easy to keep adding multiple dependencies using this approach. If we were using constructor injection, having multiple arguments would make us think

that the class does more than one thing, which can violate the Single Responsibility Principle.

More information on the *@Autowired* annotation can be found in the Wiring In Spring article.

8. Autowiring Dependencies

Wiring allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

There are four modes of autowiring a bean using an XML configuration:

- ***no:*** the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies.
- ***byName:*** autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set.
- ***byType:*** similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
- ***constructor:*** autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments.

For example, let's create the *store* bean with its dependency, the *item*, injected into it by type.

```
public class AppConfig {
     @Bean
     public Item item() {
          return new ItemImpl1();
     }

     @Bean(autowire = Autowire.BY_TYPE)
     public Store store() {
          return new Store();
     }
}
```

Note that the *autowire* property is deprecated as of Spring 5.1.

We can also inject beans using the *@Autowired* annotation for autowiring by type:

```
public class Store {

     @Autowired
     private Item item;
}
```

If there's more than one bean of the same type, we can use the *@Qualifier* annotation to reference a bean by name:

```
public class Store {

     @Autowired
     @Qualifier("item1")
     private Item item;
}
```

Now let's autowire beans by type through XML configuration:

```
<bean id="store" class="org.baeldung.store.Store"
autowire="byType"> </bean>
```

Next, let's inject a bean named *item* into the *item* property of *store* bean by name through XML:

```
<bean id="item" class="org.baeldung.store.ItemImpl1" />

<bean id="store" class="org.baeldung.store.Store"
autowire="byName">
</bean>
```

We can also override the autowiring by defining dependencies explicitly through constructor
arguments or setters.

9. Lazy Initialized Beans

By default, the container creates and configures all singleton beans during initialization. To avoid this, we can use the *lazy-init* attribute with value true on the bean configuration:

```
<bean id="item1" class="org.baeldung.store.ItemImpl1"
lazy-init="true" />
```

Consequently, the *item1* bean will only be initialized when it's first requested, and not at startup. The advantage of this is faster initialization time, but the trade-off is that we won't discover any configuration errors until after the bean is requested, which could be several hours or even days after the application has already been running.

10. Conclusion

In this article, we presented the concepts of Inversion of Control and Dependency Injection, and exemplified them in the Spring framework.

We can read more about these concepts in Martin Fowler's articles:

- Inversion of Control Containers and the Dependency Injection pattern.
- Inversion of Control

Furthermore, we can learn about the Spring implementations of IoC and DI in the Spring Framework Reference Documentation.

This text is presented here in case of changed or broken links, so that the learner may still have access to the information.
https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring