Java SE & Spring

Module 2: Spring

02.Inversion of Control(IOC) and Dependency Injection(DI)

What is Inversion of Control (IoC) ?

- In software engineering, inversion of control (IoC) is a programming technique in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis.
- In software engineering, inversion of control (IoC) is a programming principle. IoC inverts the flow of control as compared to traditional control flow. In IoC, custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.

What is Inversion of Control (IoC) ?

Actually, the definitions of the IoC in previous slide is book definition and hard to understand. But the definition below is more compact and easy to understand. 😉

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.

Further reading: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

What is Inversion of Control (IoC) ?

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

---

Further reading: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Inversion of Control (IoC)

Inversion of control can be implemented by using these mechanisms:

- Strategy Pattern
- Service Locator Pattern
- Factory Pattern
- **Dependency Injection**

We will discuss Dependency Injection(DI) in this course as an implementation of IoC.

Further reading: https://www.martinfowler.com/articles/injection.html

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:

```
1  public class Store {
2      private Item item;
3
4      public Store() {
5          item = new ItemImpl1();
6      }
7  }
```

Dependency Injection

In the example in previous slide, 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:

```
1  public class Store {
2      private Item item;
3      public Store(Item item) {
4          this.item = item;
5      }
6  }
```

Pros of Dependency Injection

Dependency Injection provides loosely coupled structure, which means an object is not strongly depend on another object. To implement this, we can use interfaces rather than specific classes. by this way, we can use different implementations of an interface. We can realise this case more clearly in examples.

```
1  public class Car {
2      DieselEngine engine;
3      public void drive() {
4          String engineStart = engine.start();
5      }
6  }
```

```
1  public class DieselEngine {
2      public String start() {
3          return "DieselEngine started ";
4      }
5  }
```

Pros of Dependency Injection

Problems on this example:

- **The code is not flexible**: The class Car is strongly depend on DieselEngine class. Any modification or change in DieselEngine will probably cause changing Car class.
- **Lack of reusability**: If we want to use DieselEngine's start method in another class, probably it will not fully suitable for the another class.

Pros of Dependency Injection

Solution is making both Car and Engine interface, thus we can implement these interfaces and in a possible different use cases, we can create and use another implementations. By this way, a car object not depend on DieselEngine object.

Pros of Dependency Injection

In the example in previous slide, 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:

```
1  public interface Car {
2      void drive();
3  }
4
5  public interface Engine {
6      String start();
7  }
```

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.

Dependency Injection Types

Dependency Injection in Spring can be done through

- constructors
- setters
- fields

Constructor Injection

If we inject the dependency through constructor then this is called constructor injection. For example, in our SuperCar example, the dependency Engine is injected through constructor:

```
1  public class SuperCar implements Car{
2
3      Engine engine;
4      //this is known as constructor injection;
5      public SuperCar (Engine engine){
6          this.engine = engine;
7      }
8  }
```

Setter Injection

If we inject the dependency through a setter method, then this is called setter injection. For example, in our SuperCar example, the dependency Engine is injected through constructor:

```
1  public class SuperCar implements Car{
2
3      //this is called setter injection
4      public void setEngine(Engine engine){
5          this.engine = engine;
6      }
7
8  }
```

Field Injection

If we inject the dependency through the annotation **@Autowired** just above the field, then this is called field injection. For example, in our SuperCar example, the dependency Engine is injected through constructor:

```
1  public class SuperCar implements Car{
2
3      //this is field injection
4      @Autowired
5      Engine engine;
6  }
```

It is not recommended to use field injection rather than setter or constructor injection. Spring documentation recommends setter injection rather than other types of injection.

Questions ?

Next:Introduction to Spring Boot