Further reading:

Wiring in Spring: @Autowired, @Resource and @Inject (/spring-annotations-resource-inject-autowire)

This article will compare and contrast the use of annotations related to dependency injection, namely the @Resource, @Inject, and @Autowired annotations.

Read more (/spring-annotations-resource-inject-autowire) →

@Component vs @Repository and @Service in Spring (/spring-component-repository-service)

Learn about the differences between the @Component, @Repository and @Service annotations and when to use them.

Read more (/spring-component-repository-service) →

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.