WHAT IS: JAVA
By Matthew Tyson, Java Developer, JavaWorld
OCT 10, 2019 10:42 AM PDT

About
Everything you need to know about Java programming tools and APIs, with code and examples.

THE SPRING SERIES

What is Spring? Component-based development for Java

Tutorial introduction to inversion of control and dependency injection, with Spring Web examples

Spring is perhaps the best of the component-based frameworks that emerged at the turn of the 21st century. It vastly improves the way that developers write and deliver infrastructure code in Java-based applications. Since its inception, Spring has been recognized as a leading framework for enterprise Java development. As an end-to-end application framework, Spring mirrors some of the Java EE capabilities , but it offers a combination of features and programming conventions you won't find elsewhere.

This article introduces Spring and its core programming philosophy and methodology: Inversion of control and dependency injection. You'll also get started with Spring annotations and a couple of hands-on coding examples.

Dependency injection and inversion of control

Spring's core idea is that instead of managing object relationships yourself, you offload them to the framework. Inversion of control (IOC) is the methodology used to manage object relationships. Dependency injection is the mechanism for implementing IOC. Since these two concepts are related but different, let's consider them more closely:

- **Inversion of control** (IOC) does just what its name says: it inverts the traditional hierarchy of control for fulfilling object relationships. Instead of relying on application code to

define how objects relate to each other, relationships are defined by the framework. As a methodology, IOC introduces consistency and predictability to object relations, but it does require you, as the developer, to give up some fine-grained control.

- **Dependency injection** (DI) is a mechanism where the framework "injects" dependencies into your app. It's the practical implementation of IOC. Dependency injection hinges on polymorphism, in the sense that it allows the fulfillment of a reference type to change based on configurations in the framework. The framework injects variable references rather than having them manually fulfilled in application code.

JSR-330

Like much in the Java world, what began as an in-the-wild innovation, Spring, has been in part absorbed by standard specification. In this case, JSR-330 is the Java standard. The nice thing about the JSR-330 spec is you can use it elsewhere, and will see it in use elsewhere, beyond Spring. You can use it without using Spring. However, Spring brings a whole lot more to the table.

Example #1: Spring dependency injection

Inversion of control and dependency injection are best understood by using them, so we'll start with a quick programming example.

Say you're modelling a car. If you're modeling in plain old Java, you might have an interface member on the `Car` class to reference an `Engine` interface, as shown in Listing 1.

Listing 1. Object relations in plain old Java

```
public Interface Engine() { ... }

public class Car {
  private Engine engine;
  public Engine getEngine() { ... }
  public void setEngine(Engine engine) { ... }
}
```

Listing 1 contains an interface for an `Engine` type, and a class for the concrete `Car` type, which references the `Engine`. (Note that in a real programming scenario these would be in separate files.) Now, when you're creating a `Car` instance, you'd set the association as shown

in Listing 2.

Listing 2. Creating a Car with the Engine interface

```
// ...
Car newCar = new Car();
Engine sixCylEngine = new InlineSixCylinderEngine();
newCar.setEngine(sixCylEngine );
// Do stuff with the car
```

Note that you create the `Car` object first. You then create a new object that fulfills the `Engine` interface, and assign it manually to the `Car` object. That is how object associations work in plain old Java.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

Modeling classes and objects in Spring

Now let's look at the same example in Spring. Here, you could do something like what's shown in Listing 3. You start with the `Car` class, but in this case you add an annotation to it: `@Inject`.

Listing 3. Example of using the @Inject annotation in Spring

```
public class Car {
  @Inject
  private Engine engine;
  // ...
}
```

Using the `@Inject` annotation (or `@Autowired`, if you prefer) tells Spring to search the context and automatically inject an object into the reference, based on a set of rules.

Next, consider the `@Component` annotation, shown in Listing 4.

Listing 4. @Component annotation

```
@Component
public class InlineSixCylinderEngine implements Engine{
  //...
}
```

Annotating a class with `@Component` tells Spring that it is available for fulfilling injections. In this case, the `InlineSixCylEngine` would be injected because it is available and satisfies the interface requirement of the association. In Spring, this is called an "autowired" injection. (See below for more about Spring's `@Autowired` annotation.)

Decoupling as a design principle

Inversion of control with dependency injection removes a source of concrete dependency from your code. Nowhere in the program is there a hard-coded reference to the `Engine` implementation. This is an example of *decoupling* as a software design principle. Decoupling application code from implementation makes your code easier to manage and maintain. The application knows less about how its parts fit together, but it's much easier to make changes at any point in the application lifecycle.

@Autowired vs @Inject

`@Autowired` and `@Inject` do the same thing. However, `@Inject` is the Java standard annotation, whereas `@Autowired` is specific to Spring. They both serve the same purpose of telling the DI engine to inject the field or method with a matching object. You can use either one in Spring.

Overview of the Spring framework

Now that you've seen some Spring code, let's take an overview of the framework and its components. As you can see, the framework consists of four main modules, which are broken into packages. Spring gives you a fair amount of flexibility with the modules you'll use.

- Core container
    - Core

    - Bean
    - Context
    - Expression Language
- Aspect-oriented programming (AOP)
    - AOP
    - Aspects
    - Instrumentation
- Data access and integration
    - JDBC
    - JPA/ORM
    - JMS
    - Transactions
- Web
    - Web/REST
    - Servlet
    - Struts

Rather than cover everything here, let's get started with two of the more commonly used Spring features.

Starting up a new project: Spring Boot

We'll use Spring Boot to create an example project, which we'll use to demo Spring features. Spring Boot makes starting new projects much easier, as you'll see for yourself. To begin, take a look at the main class shown below. In Spring Boot, we can take a main class with a `main()` method, and then choose to run it standalone, or package for deployment in a container like Tomcat.

Listing 5 has the outlines of our main class, which will live at the standard `src/main/java/hello` location.

Listing 5. Main class with Spring Boot

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

Note two things about the above code: First, all of the work is abstracted into the framework. The main class boots up the app, but it doesn't know anything about how the app works or delivers its functionality. Second, the `SpringApplication.run()` does the actual job of booting the app and passing in the `Application` class itself. Again, the work the app does is not apparent here.

The `@SpringBootApplication` annotation wraps up a few standard annotations and tells Spring to look at the package where the main class exists for components. In our previous example, with the car and engine, this would allow Spring to find all classes annotated with `@Component` and `@Inject`. The process itself, called *component scanning*, is highly customizable.

You can build the app with the standard `mvn clean install`, and you can run it with the Spring Boot goal (`mvn spring-boot:run`). Before doing that, let's look at this application's `pom.xml` file.

Listing 6. Starter pom.xml

```
<groupId>com.javaworld</groupId>
    <artifactId>what-is-spring</artifactId>
    <version>1.0.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
    </parent>

    <dependencies>
    </dependencies>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

Note two important features in the above code:

1. The `parent` element relies on the `spring-boot-starter-parent` project. This parent project defines a number of useful defaults, such as the default compiler level of JDK 1.8. For the most part, you can just trust that it knows what it's doing. As an example, you can omit the version number for many common dependencies, and `SpringBootParent` will set the versions to be compatible. When you bump up the parent's version number, the dependency versions and defaults will also change.
2. The `spring-boot-maven-plugin` allows for the executable JAR/WAR packaging and in-place `run` (via the `mvn spring-boot:run` command).

Adding Spring Web as a dependency

So far, we've been able to use `spring-boot` to limit how much work we put in to get an app up and running. Now let's add a dependency and see how quickly we can get something in a browser.

Listing 7. Adding Spring Web to a project

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Note
Spring will automatically detect what files have changed and compile accordingly. You can just execute `mvn spring-boot:run` to pickup changes.

Now that we've got a basic project setup, we're ready for our two examples.

Example #2: Building RESTful endpoints with Spring Web

We've used `spring-boot-starter-web` to bring in several dependencies that are useful for building web applications. Next we'll create a route handler for a URL path. Spring's web support is part of the Spring MVC (Model-View-Controller) module, but don't let that worry you: Spring Web has full and effective support for building RESTful endpoints, as well.

The class whose job it is to field URL requests is known as a *controller*, as shown in Listing 8.

Listing 8. Spring MVC REST controller

```
package hello;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @RequestMapping(value = "/hi", method = RequestMethod.GET)
    public String hi(@RequestParam(name="name", required=false, defaultValue="JavaWorld") Str
        return "Hello " + name;
    }

}
```

The @Controller annotation

The `@Controller` annotation identifies a class as a controller. A class marked as a controller is also automatically identified as a component class, which makes it a candidate for auto-wiring. Wherever this controller is needed, it will be plugged into the framework. In this case, we'll plug it into the MVC system to handle requests.

The controller is a specialized kind of component. It supports the `@RequestMapping` and `@ResponseBody` annotations that you see on the `hi()` method. These annotations tell the framework how to map URL requests to the app.

At this point, you can run the app with `mvn spring-boot:run`. When you hit the `/hi` URL, you'll get a response like "Hello, JavaWorld."

Notice how Spring has taken the basics of autowiring components, and delivered a whole web framework. With Spring, you don't have to explicitly connect anything together!

The @Request annotations

The `@RequestMapping` allows you to define a handler for a URL path. Options include defining the HTTP method you want, which is what we've done in this case. Leaving `RequestMethod` off would instruct the program to handle all HTTP method types.

The `@RequestParam` argument annotation allows us to map the request parameters directly into the method signature, including requiring certain params and defining default values as we've done here. We can even map a request body to a class with the `@RequestBody` argument annotation.

REST and JSON response

If you are creating a REST endpoint and you want to return JSON from the method, you can annotate the method with `@ResponseBody`. The response will then be automatically packaged as JSON. In this case you'll return an object from the method.

Using MVC with Spring Web

Similar to Struts, the Spring Web module can easily be used for a true model-view-controller setup. In that case, you would return a mapping in the given templating language (like Thymeleaf), and Spring would resolve the mapping, provide the model you pass to it, and render the response.