Coding Cheat Sheet: Introduction to Spring Framework

This reading provides a reference list of code that you'll encounter as you learn and use the Spring framework in Java. Understanding these concepts will help you write and debug Java programs that utilize Spring framework. Let's explore the following Java coding concepts:

- Learning Spring annotations
- Using Maven with Spring
- Defining Spring projects

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin coding with Java after this course!

Learning Spring annotations

Spring annotations are special metadata in the Spring framework that help configure applications by reducing the need for XML-based configuration. They simplify dependency injection, bean management, transaction handling, and AOP (Aspect-Oriented Programming). Common annotations such as @Component, @Autowired, and @Transactional enable efficient and modular development. By using annotations, developers can write cleaner, more maintainable, and easily testable code.

Description	**Example**

`@Component` marks a class as a Spring-managed component for auto-detection and registration in the application context.	`import org.springframework.stereotype.Component;`
	
	`@Component`
	`public class BookService {`
	`    public void listBooks() {`
	`        System.out.println("Listing all books");`
	`    }`
	`}`

`@Controller` is a specialized `@Component` for Spring MVC controllers that handle web requests.	`import org.springframework.stereotype.Controller;`
	`import org.springframework.web.bind.annotation.GetMapping;`
	
	`@Controller`
	`public class BookController {`
	`    @GetMapping("/books")`
	`    public String showBooks() {`
	`        return "books"; // Returns view name "books"`
	`    }`
	`}`

`@Autowired` enables automatic dependency injection in Spring-managed beans.	`import org.springframework.beans.factory.annotation.Autowired;`
	`import org.springframework.stereotype.Controller;`
	
	`@Controller`
	`public class BookController {`
	`    @Autowired`
	`    private BookService bookService;`
	
	`    public void displayBooks() {`
	`        bookService.listBooks();`
	`    }`
	`}`

Description | **Example**

`@Configuration` defines a configuration class that declares beans and configurations for the Spring container.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public BookService bookService() {
        return new BookService();
    }
}
```

`@RequestMapping` maps web requests to handler methods in Spring MVC applications.

```java
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class BookController {
    @RequestMapping("/books")
    public String getBooks() {
        return "books";
    }
}
```

`@PathVariable` extracts values from the URL and binds them to method parameters.

```java
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class BookController {
    @GetMapping("/books/{id}")
    public String getBookById(@PathVariable("id") String bookId) {
        System.out.println("Book ID: " + bookId);
        return "bookDetails";
    }
}
```

`@RestController` is a combination of `@Controller` and `@ResponseBody`, used for building RESTful web services.

```java
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
import java.util.Arrays;
import java.util.List;

@RestController
public class BookRestController {
    @GetMapping("/api/books")
    public List<String> getAllBooks() {
        return Arrays.asList("Spring Boot", "Spring Cloud");
    }
}
```

`@RequestParam` extracts query parameters from the URL and binds them to method parameters.

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
```

Description	Example

	`@RestController`
	`public class BookRestController {`
	`    @GetMapping("/api/book")`
	`    public String getBookByTitle(@RequestParam("title") String title) {`
	`        return "Book title: " + title;`
	`    }`
	`}`

`@ResponseBody` indicates that a method's return value should be written directly to the HTTP response body.	`import org.springframework.web.bind.annotation.GetMapping;`
	`import org.springframework.web.bind.annotation.ResponseBody;`
	`import org.springframework.web.bind.annotation.RestController;`
	
	`@RestController`
	`public class BookRestController {`
	`    @GetMapping("/api/message")`
	`    @ResponseBody`
	`    public String getMessage() {`
	`        return "Hello, Spring!";`
	`    }`
	`}`

`@Value` injects values from properties files or environment variables into Spring beans.	`import org.springframework.beans.factory.annotation.Value;`
	`import org.springframework.stereotype.Component;`
	
	`@Component`
	`public class Library {`
	`    @Value("${library.name}")`
	`    private String libraryName;`
	
	`    public void printLibraryName() {`
	`        System.out.println("Library Name: " + libraryName);`
	`    }`
	`}`

`@Scope` defines the scope of a bean, such as singleton or prototype.	`import org.springframework.context.annotation.Scope;`
	`import org.springframework.stereotype.Component;`
	
	`@Component`
	`@Scope("prototype")`
	`public class Book {`
	`    // Prototype-scoped bean`
	`}`

Using Maven with Spring

Maven is used in Spring to manage dependencies, build projects, and automate tasks such as compiling, packaging, and deploying applications. It simplifies project configuration with a standardized pom.xml file, ensuring consistent builds and easy integration of Spring dependencies.

Description	Example

Managing external libraries with dependencies: This part of the `pom.xml` file ensures that required external libraries are included in the project. Each dependency specifies a `<groupId>` (organization or vendor), an `<artifactId>` (library name), and a `<version>` (specific release). Maven automatically downloads and manages these dependencies.	`<dependencies>`
	`    <dependency>`
	`        <groupId>org.springframework.boot</groupId>`
	`        <artifactId>spring-boot-starter-web</artifactId>`
	`        <version>2.7.0</version>`
	`    </dependency>`
	`</dependencies>`

Configuring the build process: This part defines how the project is compiled and packaged. It includes plugins such as the `maven-compiler-plugin`, which specifies the Java version for source code compatibility.	`<build>`
	`    <plugins>`
	`        <plugin>`
	`            <groupId>org.apache.maven.plugins</groupId>`
	`            <artifactId>maven-compiler-plugin</artifactId>`
	`            <version>3.8.1</version>`
	`            <configuration>`
	`                <source>1.8</source>`
	`                <target>1.8</target>`
	`            </configuration>`
	`        </plugin>`
	`    </plugins>`
	`</build>`

Adding custom repositories for dependencies: If required dependencies are not available in the default Maven Central repository, this part allows you to specify additional repositories where Maven can look for them.	`<repositories>`
	`    <repository>`
	`        <id>spring-releases</id>`
	`        <url>https://repo.spring.io/release</url>`
	`    </repository>`
	`</repositories>`

Defining project-wide properties: This feature allows setting reusable values such as the Java version,	`<properties>`
	`    <java.version>1.8</java.version>`
	`</properties>`

Description	Example
making configuration easier to maintain across the project.	

Managing different environments with profiles: Profiles help configure different settings for various environments (e.g., development, testing, production). They can be activated using command-line options.	`<profiles>`
	`    <profile>`
	`        <id>dev</id>`
	`        <properties>`
	`            <env>development</env>`
	`        </properties>`
	`    </profile>`
	`</profiles>`

Complete example of a Maven project (pom.xml): This is a full example of a `pom.xml` file that manages dependencies, build configurations, and plugins for a simple Spring Boot application.	`<project xmlns="<a href="http://maven.apache.org/POM/4.0.0"" target="_blank" rel="noopener noreferrer">http://maven.apach`
	`         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
	`         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">`
	`    <modelVersion>4.0.0</modelVersion>`
	
	`    <groupId>com.example</groupId>`
	`    <artifactId>spring-demo</artifactId>`
	`    <version>1.0-SNAPSHOT</version>`
	`    <packaging>jar</packaging>`
	
	`    <dependencies>`
	`        <dependency>`
	`            <groupId>org.springframework.boot</groupId>`
	`            <artifactId>spring-boot-starter-web</artifactId>`
	`            <version>2.7.0</version>`
	`        </dependency>`
	`    </dependencies>`
	
	`    <build>`
	`        <plugins>`
	`            <plugin>`
	`                <groupId>org.springframework.boot</groupId>`
	`                <artifactId>spring-boot-maven-plugin</artifactId>`
	`            </plugin>`
	`        </plugins>`
	`    </build>`
	`</project>`

Defining Spring projects

Defining Spring projects is important to establish a clear structure, manage dependencies efficiently, and ensure smooth integration with frameworks such as Spring Boot. A well-defined project simplifies development, testing, and deployment while maintaining scalability and maintainability.

Description	**Example**

Verify installation: Open your terminal or command prompt and execute the following commands to verify the installations.	`java -version`
	`mvn -version`
	
	Both commands should return version information if installed correctly.

Create a new Maven project using the command line: Open your terminal, navigate to your desired directory, and run the following command.	`mvn archetype:generate -DgroupId=com.example -DartifactId=spring-beginner-project -DarchetypeArtifactId=maven-archetype-q` `groupId`: A unique identifier for your project (e.g., com.example). artifactId: The name of your project (e.g., spring-beginner-project).

Understand the project structure: A standard Maven project layout looks like this.	`spring-beginner-project`
	`├── src`
	`│   ├── main`
	`│   │   ├── java`
	`│   │   └── resources`
	`│   └── test`
	`│       ├── java`
	`│       └── resources`
	`├── pom.xml`

Add Spring dependencies: Open `pom.xml` and add the necessary Spring dependencies.	`<dependencies>`
	`    <dependency>`
	`        <groupId>org.springframework</groupId>`
	`        <artifactId>spring-context</artifactId>`
	`        <version>5.3.28</version>`
	`    </dependency>`
	`    <dependency>`
	`        <groupId>org.springframework</groupId>`
	`        <artifactId>spring-webmvc</artifactId>`
	`        <version>5.3.28</version>`
	`    </dependency>`
	`</dependencies>`
	
	Run `mvn clean install` to download dependencies.

Create a configuration class: Defines beans and configurations for the application.	`package com.example;`
	`import org.springframework.context.annotation.Bean;`
	`import org.springframework.context.annotation.Configuration;`
	
	`@Configuration`
	`public class AppConfig {`
	`    @Bean`
	`    public HelloWorld helloWorld() {`
	`        return new HelloWorld();`
	`    }`
	`}`

Description	Example

Create a simple bean: A basic class to demonstrate a Spring-managed bean.	`package com.example;`
	
	`public class HelloWorld {`
	`    public void sayHello() {`
	`        System.out.println("Hello, World!");`
	`    }`
	`}`

Create a main application class: Loads the Spring application context and retrieves the bean.	`package com.example;`
	`import org.springframework.context.ApplicationContext;`
	`import org.springframework.context.annotation.AnnotationConfigApplicationContext;`
	
	`public class MainApp {`
	`    public static void main(String[] args) {`
	`        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);`
	`        HelloWorld helloWorld = context.getBean(HelloWorld.class);`
	`        helloWorld.sayHello();`
	`    }`
	`}`

Run your application: Compile and run the application using the following commands.	`mvn compile`
	`mvn exec:java -Dexec.mainClass="com.example.MainApp"`

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan