01 - Spring Framework

Table of Contents

Agenda

- What is Spring Framework?
- Why Spring?
- Spring Goals
- Spring Architecture
- IOC (Inversion of Control)
- IOC Container
- Hands-On
- File Structure

What is Spring Framework

The Spring Framework is a powerful, feature-rich framework for building Java-based enterprise applications. It provides comprehensive infrastructure support and aims to make Java programming easier by simplifying development processes.

- Open-Source Framework
- Loose Coupling
- The lightweight alternative of JAVA EE

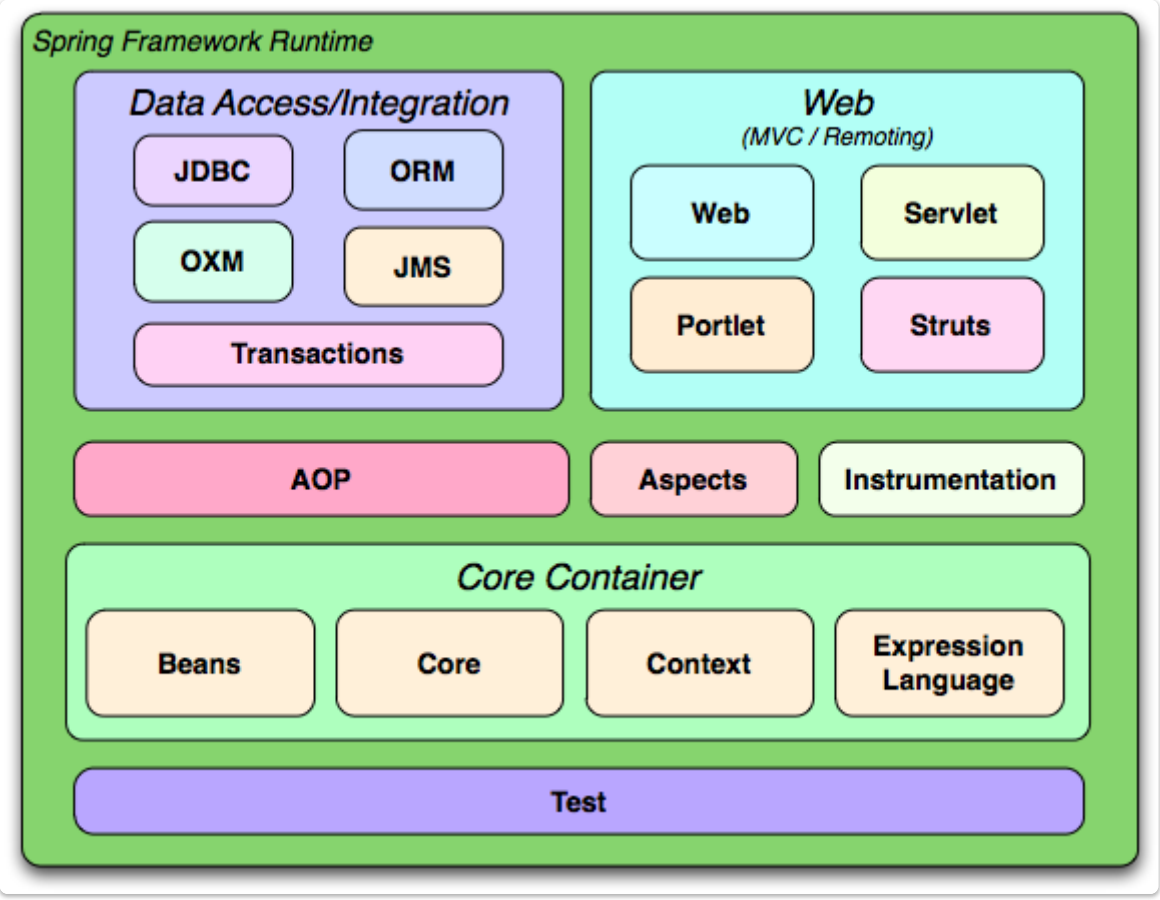
Goals of spring

- Simplicity: Reduce complexity in enterprise Java development by providing a simple, consistent, and comprehensive programming and configuration model.

- **Loose Coupling:** Use dependency injection to achieve loose coupling between components, making applications easier to manage and extend.
- **Modularity:** Allow developers to use only the parts of the framework they need, promoting a modular and lightweight approach to application development.
- **Integration:** Provide seamless integration with various other technologies and frameworks, enabling developers to build robust, scalable applications using their preferred tools and libraries.

  Use Cases:
  - Spring Framework is widely used in enterprise Java applications, including web applications, microservices, RESTful APIs, batch processing, and integration with external systems.
  - It is trendy in the finance, healthcare, e-commerce, and telecommunications industries, where robustness, scalability, and maintainability are critical.

Basic Spring Architecture



Inversion of Control (IoC)

- It is a design principle followed by Spring
- The control of object creation and dependency management is transferred from the application code to a container or framework.
- To Achieve IOC we use DI (or Dependency Injection)

- Promotes loose coupling, enhances testability, and makes code more modular and easier to maintain.

IOC Container

1. **Definition:** The IoC container is a core component of the Spring Framework responsible for managing the lifecycle and configuration of application objects (beans).
2. **Types:**
   - **BeanFactory:** The simplest container, providing basic DI capabilities.
   - **ApplicationContext:** A more advanced container that builds on BeanFactory with additional features such as event propagation, declarative mechanisms to create a bean, and various ways to look up beans.
3. **Bean Management:** The IoC container creates, configures, and manages beans based on the configuration metadata (e.g., XML configuration, annotations).
4. **Dependency Injection:** The container automatically injects the required dependencies into beans, either through constructors, setters, or fields.
5. **Lifecycle Management:** Manages the complete lifecycle of beans, including instantiation, dependency injection, initialization, and destruction, allowing for custom initialization and destruction methods.

IOC Concepts

1. **Dependency Injection:** The most common form of IoC, where dependencies are injected into objects rather than being created internally. This can be done through constructor injection, setter injection, or field injection.
2. **Flexibility:** Allows developers to easily switch implementations or configurations without changing the code, as dependencies are managed externally.
3. **Example:** Instead of an object creating its dependencies, the IoC container injects the required dependencies at runtime.

Hands-On

1. Create 2 classes - TennisCoach and CricketCoach
2. Create a GetWorkout() Method in both the classes and create objects in Main to call the method.

```
public class CricketCoach {
    public String getWorkout(){
        return "Practice 10 coverdrives";
```

```
    }
}
```

```
public class TennisCoach {
    public  String getWorkout(){
        return "Practice 50 back hands";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
                TennisCoach tennisCoach = new TennisCoach();
                System.out.println(tennisCoach.getWrokout());
    }
}
```

Now there can be 100-1000 coaches in an academy.
We will create an `interface` to manage this.

3. Create interface - Coach

```
public interface Coach {
    String getWorkout();
}
```

Now every Coach class can implement the Coach interface

```
public class CricketCoach implements Coach {
    public String getWorkout(){
        return "Practice 10 coverdrives";
    }
}
```

```
public class TennisCoach implements Coach {
    public  String getWorkout(){
        return "Practice 50 back hands";
    }
}
```

So now in `Main`

```
public class Main {
    public static void main(String[] args) {
                Coach coach1 = new TennisCoach();
```

```
                System.out.println(coach1.getWrokout());
                Coach coach2 = new TennisCoach();
                System.out.println(coach2.getWrokout());

    }
}
```

Still, we have to create objects like coach1, and coach2 and change the source code if any changes are required
To solve this we will add in resources -> `applicationContext.xml`
Also add `spring-framework` dependency in `pom.xml`

4. Add `applicationContext.xml` and `pom.xml`

`pom.xml`

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.1.10</version>
    </dependency>
</dependencies>
```

`applicationContext.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="" class="">
    </bean>
    <!-- more bean definitions go here -->

</beans>
```

5. Add bean definitions in applicationContext file and define context in Main

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <bean id="coach" class="org.example.CricketCoach">
    </bean>

    <bean id="coach1" class="org.example.TennisCoach ">
    </bean>
    <!-- more bean definitions go here -->

</beans>
```

`Main.java`

```
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
Coach coach = context.getBean("coach",Coach.class);
System.out.println(coach.getWorkout());

Coach coach1 = context.getBean("coach1",Coach.class);
System.out.println(coach1.getWorkout());
```

Types of Dependency Injection

- Constructor Based
- Setter Based

To implement dependency Injections

1. Create a new class Greetings

```
public class Greetings {
    public String sayHello(){
        return "Hello! Welcome to Spring Framework Training";
    }
}
```

2. In applicationContext add the reference objects and create a bean for Greetings.java

```
<bean id="coach" class="org.example.CricketCoach">
    <constructor-arg ref="obj"/>
</bean>

<bean id="coach1" class="org.example.TennisCoach ">
    <property name="ObjectClass" ref="obj"/>
</bean>
<bean id="obj" class="org.example.ObjectClass">
```

```
</bean>
```

3. Perform Constructor based injection in CricketCoach

```
public class CricketCoach  implements Coach{

    Greetings greetings;

    public CricketCoach(Greetings greetings) {
        this.objectClass = objectClass;
    }

        @Override
    public String sayHelloUsingGreetings(){
        return greetings.sayHello();
    }

    public String getWorkout(){
        return "Practice 10 coverdrives";
    }
}
```

4. Perform Setter Based Injection in TennisCoach

```
public class TennisCoach implements Coach {
    ObjectClass objectClass;

    public void setGreetings(Greetings greetings) {
        this.Greetings = greetings;
    }

    public  String getWorkout(){
        return "practice 50 back hands";
    }

    @Override
    public String sayHelloUsingGreetings() {
        return greetings.sayHello();
    }
}
```

5. Add sayHelloUsingGreetings() method into the Coach interface.

```
public interface Coach {
    String getWorkout();
```

```
    public String sayHelloUsingObjectClass();
}
```

6. Now you can call it in the Main.class and the sayHello() method will be called from the reference of coach classes.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Worldz!");

        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Coach coach  = context.getBean("coach1",Coach.class);
        System.out.println(coach.getWorkout());
        System.out.println(coach.sayHelloUsingObjectClass());
    }
}
```
