# Introduction to Spring Framework

The **Spring Framework** is a comprehensive framework for building Java-based enterprise applications. It simplifies the development of large-scale applications and promotes best practices in software design, such as loose coupling and separation of concerns. Spring provides a wide range of features, from dependency injection and aspect-oriented programming to robust data access and transaction management.

### Key Features of the Spring Framework:

1. **Inversion of Control (IoC)**: Spring promotes loose coupling through **dependency injection**, which simplifies object creation and dependency management.
2. **Aspect-Oriented Programming (AOP)**: Supports modularization of cross-cutting concerns such as logging, transaction management, and security.
3. **Spring MVC**: A popular Model-View-Controller web framework for building web applications.
4. **Transaction Management**: Provides a consistent abstraction for transaction management, allowing you to work with different transaction models.
5. **Data Access**: Simplifies JDBC and integrates with ORM frameworks like Hibernate and JPA.
6. **Security**: Includes features for authentication, authorization, and access control.
7. **Testing**: Provides integration with testing frameworks like JUnit and TestNG, and it includes built-in support for mock objects.

### Spring Core and IoC (Inversion of Control)

**Spring Core** refers to the foundational modules of the Spring Framework that provide essential functionality for building Spring-based applications. The most significant aspect of the Spring Core is **Inversion of Control (IoC)**, which is implemented through **Dependency Injection (DI)**.

#### What is IoC (Inversion of Control)?

Inversion of Control is a design principle in software engineering in which the control of object creation and dependency management is transferred from the application code to a container. The container (often referred to as the **Spring container**) is responsible for creating and managing the lifecycle of objects and resolving their dependencies.

In traditional programming, an object is responsible for creating and managing its own dependencies. With IoC, the responsibility for creating objects is given to the framework (e.g., Spring), which "injects" the required dependencies at runtime.

### Benefits of IoC:

* **Loose Coupling**: Classes are not tightly coupled to their dependencies, making it easier to change and maintain code.
* **Better Testability**: With IoC, it becomes easier to test individual components, as dependencies can be mocked or substituted during unit tests.
* **Improved Maintainability**: By centralizing the management of dependencies, code is easier to maintain and modify over time.
* **Flexibility**: The configuration of beans and their dependencies can be externalized into XML files or annotations, allowing for easy modification without changing the source code.

### Dependency Injection (DI)

**Dependency Injection** is a core aspect of IoC and refers to the technique where an object’s dependencies are provided to it by an external source, rather than the object creating the dependencies itself.

In other words, DI is the mechanism through which Spring injects the required dependencies (like other objects or services) into a class when it is instantiated. This enables the class to focus solely on its own business logic, without worrying about how to create or manage its dependencies.

#### Types of Dependency Injection

There are three primary ways to inject dependencies in Spring:

1. **Constructor Injection**: Dependencies are provided through the constructor of a class. This method is preferred when the dependency is mandatory for the class to function correctly.

Example:

public class Car {

private Engine engine;

// Constructor Injection

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

Spring would create a Car instance and inject an Engine dependency by invoking the constructor.

1. **Setter Injection**: Dependencies are provided via setter methods. This method is typically used when the dependency is optional, or if it can be set after the object is constructed.

Example:

public class Car {

private Engine engine;

// Setter Injection

public void setEngine(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

In this case, Spring will invoke the setter method to inject the Engine dependency after creating the Car object.

1. **Field Injection**: Dependencies are injected directly into fields via reflection. While convenient, it is generally less preferred because it can make the code less explicit and harder to test.

Example:

public class Car {

@Autowired // Spring injects the Engine automatically

private Engine engine;

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

The @Autowired annotation tells Spring to inject the Engine dependency directly into the field.

### How Spring Handles IoC and DI

In Spring, the **ApplicationContext** (the Spring IoC container) manages the lifecycle of beans (objects) and handles the injection of dependencies. The container is responsible for creating beans, wiring them together, and managing their lifecycle.

#### Defining Beans in Spring

In Spring, **beans** are the objects managed by the Spring IoC container. Beans can be defined in several ways, but typically they are defined in either an XML configuration file or using annotations.

1. **XML-based Configuration** (traditional approach): In this approach, beans are declared in an XML file (applicationContext.xml or beans.xml), where you can specify properties and dependencies.

Example of XML-based configuration:

<bean id="engine" class="com.example.Engine" />

<bean id="car" class="com.example.Car">

<constructor-arg ref="engine" />

</bean>

In this example, the Car bean depends on the Engine bean. Spring will inject the Engine bean into the Car bean’s constructor.

1. **Annotation-based Configuration** (modern approach): Spring 2.5 introduced annotations to reduce the need for XML configuration. Common annotations include:
   * @Component: Marks a class as a Spring bean.
   * @Autowired: Injects dependencies automatically.
   * @Configuration and @Bean: Marks a configuration class and defines beans using methods.

Example of annotation-based configuration:

@Component

public class Engine {

public void start() {

System.out.println("Engine started!");

}

}

@Component

public class Car {

private final Engine engine;

@Autowired // Dependency injection via constructor

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

And in your main class:

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

// Spring will automatically scan the package for components

}

public class Main {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Car car = context.getBean(Car.class);

car.drive();

}

}

In this example, Spring automatically scans the com.example package for beans and injects the Engine dependency into the Car bean.

### Advantages of Dependency Injection in Spring

1. **Loose Coupling**: By removing the responsibility for object creation from the class itself, DI allows the class to be more flexible and reusable.
2. **Easier Testing**: DI makes unit testing easier by allowing you to inject mock dependencies or change dependencies without modifying the class.
3. **Cleaner Code**: The classes focus on their core functionality, and dependencies are managed externally by the container.
4. **Flexible Configuration**: Dependencies can be configured externally (via XML, annotations, or Java configuration), making it easy to swap implementations or adjust configuration settings.

### 3. ****Injecting Collections****

In Spring, you can inject **collections** (like List, Set, and Map) into beans using dependency injection. You can either define collections in the XML configuration or use annotations to inject them.

#### XML-Based Collection Injection

**applicationContext.xml**:

<bean id="car" class="com.example.Car">

<property name="engines">

<list>

<ref bean="engine1"/>

<ref bean="engine2"/>

</list>

</property>

</bean>

<bean id="engine1" class="com.example.Engine"/>

<bean id="engine2" class="com.example.Engine"/>

**Car.java**:

import java.util.List;

public class Car {

private List<Engine> engines;

public void setEngines(List<Engine> engines) {

this.engines = engines;

}

public void drive() {

for (Engine engine : engines) {

engine.start();

}

System.out.println("Car is driving...");

}

}

In this example:

* A List<Engine> is injected into the Car bean. The engine1 and engine2 beans are injected as elements of the list.

#### Annotation-Based Collection Injection

**Car.java**:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import java.util.List;

@Component

public class Car {

private List<Engine> engines;

@Autowired

public void setEngines(List<Engine> engines) {

this.engines = engines;

}

public void drive() {

for (Engine engine : engines) {

engine.start();

}

System.out.println("Car is driving...");

}

}

In this example:

* @Autowired is used on the setter method to inject a list of Engine objects.

Spring automatically wires all beans of type Engine into the engines list.

### 4. ****Bean Scopes****

Spring provides different **bean scopes** that define the lifecycle and visibility of a bean. The most common bean scopes are:

* **Singleton** (default): Only one instance of the bean is created, and the same instance is shared throughout the application context.
* **Prototype**: A new instance of the bean is created each time it is requested from the Spring context.
* **Request**: A new instance is created for each HTTP request (available in web applications).
* **Session**: A new instance is created for each HTTP session (available in web applications).
* **Global Session**: A new instance is created for each global HTTP session (less commonly used in Portlet applications).

#### Example: Bean Scopes in XML

<bean id="car" class="com.example.Car" scope="singleton"/> <!-- Singleton scope (default) -->

<bean id="engine" class="com.example.Engine" scope="prototype"/> <!-- Prototype scope -->

#### Example: Bean Scopes with Annotations

In annotation-based configuration, bean scopes can be set using the @Scope annotation:

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("prototype") // Prototype scope

public class Engine {

public void start() {

System.out.println("Engine started!");

}

}

In this example, the Engine bean is scoped as a prototype, meaning a new instance will be created every time it is requested.

### 5. ****Autowiring****

**Autowiring** is the process of automatically injecting dependencies into Spring beans. Spring provides several ways to autowire dependencies:

#### 1. ****Autowiring by Type (****@Autowired****)****

You can use the @Autowired annotation to inject dependencies by their type. Spring will automatically inject the bean that matches the type of the property or constructor parameter.

Example:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class Car {

private Engine engine;

@Autowired // Autowires Engine by type

public void setEngine(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

#### 2. ****Autowiring by Name****

Autowiring by name allows Spring to inject a bean based on the name of the property. This requires the bean name to match the name of the property in the target bean.

<bean id="car" class="com.example.Car" autowire="byName">

<property name="engine" ref="engine"/>

</bean>

#### 3. ****Autowiring by Constructor****

You can autowire dependencies via the constructor. If there is a single constructor in a class, Spring will automatically inject the required dependencies.

@Component

public class Car {

private Engine engine;

@Autowired // Autowires Engine by constructor

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}