

Gen-C Spring 1

Table of Contents

1. Code	1
2. Spring Framework Core Concepts	1
2.1. Terminologies	1
2.2. Inversion of Control (IoC)?	2
2.2.1. Demo	2
2.2.2. Autowiring	2
2.2.3. AutowiringXML	2
2.2.4. Loose Coupling: Sorting	2
2.2.5. Scenario: Chef	4
2.3. Dependency Injection	5
2.4. Bean Scopes	7
2.5. Bean Lifecycle Methods	8
2.6. Java Config Bean Overview	8
2.7. What is AOP?	9

1. Code

¥ GenC-SpringFramework-1

! IoC

2. Spring Framework Core Concepts

¥ Introduction to Spring Framework

¥ Objective: Understand key concepts of Spring Framework, including IoC, Dependency Injection, and Bean Management.

2.1. Terminologies

- " Beans
- " Autowiring
- " Dependency Injection
- " Inversion of Control
- " IOC Container
- " Application Context

2.2. Inversion of Control (IoC)?

¥ Definition:

¥ IoC is a design principle where the control of object creation and dependency management is handed over to a container (Spring Framework).

¥ Helps in reducing tight coupling and improving testability.

2.2.1. Demo

¥ example/demo

```
@Configuration
public class AppConfig {
    @Bean
    public MessageService messageService() {
        return new MessageServiceImpl();
    }
}
```

2.2.2. Autowiring

¥ autowiring

```
@Configuration
@ComponentScan(basePackages = "com.autowiring")
public class AppConfig {
}
```

2.2.3. AutowiringXML

2.2.4. Loose Coupling: Sorting

¥ RECAP: GenC-SpringCore1/sorting

Loose coupling means that classes or modules are independent and can be modified or replaced without affecting other parts of the system.

Scenario: Sorting Strategies with Loose Coupling

We create an interface that defines a sorting behavior. Different sorting algorithms will implement this interface.

Step 1: Define the Interface

```
public interface SortAlgorithm {  
    void sort(int[] numbers);  
}
```

This interface defines a method `sort()`, which different sorting algorithms will implement.

Step 2: Implement Different Sorting Algorithms

Bubble Sort Implementation

```
@Component  
public class BubbleSort implements SortAlgorithm {  
    @Override  
    public void sort(int[] numbers) {  
    }  
}
```

Quick Sort Implementation

```
@Component  
public class QuickSort implements SortAlgorithm {  
    @Override  
    public void sort(int[] numbers) {  
  
    }  
}
```

Step 3: Create a Service That Uses the Sorting Algorithm

```
@Component  
public class SortService {  
    private final SortAlgorithm sortAlgorithm;  
  
    @Autowired  
    public SortService(SortAlgorithm sortAlgorithm) {  
        this.sortAlgorithm = sortAlgorithm;  
    }  
}
```

```

    }

    public void performSorting(int[] numbers) {
        sortAlgorithm.sort(numbers);
    }
}

```

¥ The `SortService` depends on `SortAlgorithm` but does not know which sorting algorithm it is using.

¥ Spring injects the required sorting algorithm at runtime, ensuring loose coupling.

Step 4: Configure Sorting Algorithm in Spring

```

@Configuration
public class AppConfig {
    @Bean
    public SortAlgorithm sortAlgorithm() {
        return new BubbleSort(); // Can be changed to QuickSort easily
    }
}

```

¥ If we change `BubbleSort` to `QuickSort`, the system will work without modifying `SortService`.

Key Benefits of Loose Coupling

¥ Easier to extend: We can add new sorting algorithms without modifying `SortService`.

¥ Improved flexibility: We can switch sorting strategies at runtime.

¥ Better maintainability: The classes are independent and reusable.

This is how loose coupling makes systems more scalable and maintainable! #

2.2.5. Scenario: Chef

Imagine you love eating delicious home-cooked food but don't have time to cook. You have two choices:

Without IoC (Traditional Approach)

¥ You go grocery shopping.

¥ You buy ingredients.

¥ You cook the meal yourself.

¥ You serve it and clean up after.

Problem: You control every step, making it time-consuming and tightly coupled to your effort.

With IoC (Using a Chef - Inversion of Control)

¥ You hire a personal chef and tell them what kind of food you want.

¥ The chef takes care of buying ingredients, cooking, and serving.

¥ You simply enjoy the meal.

IoC Concept: Instead of you controlling the process, the chef takes control of cooking.

How This Relates to Spring Framework?

¥ In a traditional Java application, we create and manage objects ourselves (like cooking on our own).

¥ With IoC, Spring takes over object creation and management, just like a chef handling the cooking.

¥ We just request what we need (like ordering a dish), and Spring provides the required object (like a cooked meal).

2.3. Dependency Injection

¥ Dependency Injection (DI) is a technique where one object supplies dependencies of another object.

¥ Types of DI:

! Constructor Injection

! Setter Injection

! Field Injection

```
//Constructor Injection
@Component
public class Car {
    private Engine engine;

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

```
//Setter Injection
@Component
public class Car {
    private Engine engine;
    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

```
//Field Injection
@Component
public class Car {
    @Autowired
    private Engine engine;
}
```

Component Scanning

¥ Definition:

¥ Spring automatically detects and registers beans using `@ComponentScan`.

¥ Requires annotating classes with `@Component`, `@Service`, or `@Repository`.

¥ Example:

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

Qualifiers Overview

¥ Definition:

¥ Used to specify which bean to inject when multiple beans of the same type exist.

¥ Example:

```

@Component("dieselEngine")
public class DieselEngine implements Engine {}

@Component("petrolEngine")
public class PetrolEngine implements Engine {}

@Component
public class Car {
    private Engine engine;

    @Autowired
    public Car(@Qualifier("dieselEngine") Engine engine) {
        this.engine = engine;
    }
}

```

Lazy Initialization Overview

¥ Definition:

¥ By default, Spring initializes beans eagerly.

¥ **@Lazy** delays bean creation until the first request.

¥ Example:

```

@Component
@Lazy
public class HeavyComponent {
    public HeavyComponent() {
        System.out.println("HeavyComponent initialized");
    }
}

```

2.4. Bean Scopes

¥ Definition:

¥ Spring provides different bean scopes:

¥ **singleton** (default)

¥ **prototype**

¥ **request, session, application** (Web only)

¥ Example:

```
@Component
@Scope("prototype")
public class PrototypeBean {
}
```

2.5. Bean Lifecycle Methods

¥ Definition:

¥ Spring provides lifecycle callbacks using `@PostConstruct` and `@PreDestroy`.

¥ Example:

```
@Component
public class LifecycleBean {
    @PostConstruct
    public void init() {
        System.out.println("Bean initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean destroyed");
    }
}
```

2.6. Java Config Bean Overview

¥ Definition:

¥ Instead of XML, Java-based configuration defines beans using `@Configuration` and `@Bean`.

¥ Example:

```
@Configuration
public class AppConfig {
    @Bean
    public Engine engine() {
        return new Engine();
    }
}
```


2.7. What is AOP?

Aspect-Oriented Programming (AOP) helps you separate cross-cutting concerns (like logging, security, transactions) from your main business logic.

For example, instead of writing logging code in every method, you write it once in an "Aspect" and apply it where needed.

Key AOP Concepts

Concept	Description
Aspect	A class that contains cross-cutting logic (e.g., logging).
Advice	The code to be executed at a join point (e.g., before a method runs).
Join Point	A point in the execution of your program (like a method call).
Pointcut	An expression that matches join points (e.g., all methods in a package).

Maven Dependency

To use AOP in Spring, add this to your `pom.xml`:

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

Example Code

1. Business Logic `MyService.java`

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {
    public void doWork() {
        System.out.println("Doing actual work...");
    }
}
```

2. Logging Aspect `LoggingAspect.java`

```

package com.example.demo.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.demo.service.*.*(..))")
    public void logBeforeMethod() {
        System.out.println("Logging before method execution...");
    }
}

```

3. Spring Boot Application DemoApplication.java

```

package com.example.demo;

import com.example.demo.service.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    @Autowired
    private MyService myService;

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

    @Override
    public void run(String... args) {
        myService.doWork();
    }
}

```

Output

```

Logging before method execution...

```

Doing actual work...