Spring Start Here

Chapter-3:
The Spring context: Wiring Beans

Upcode Software
Engineer Team

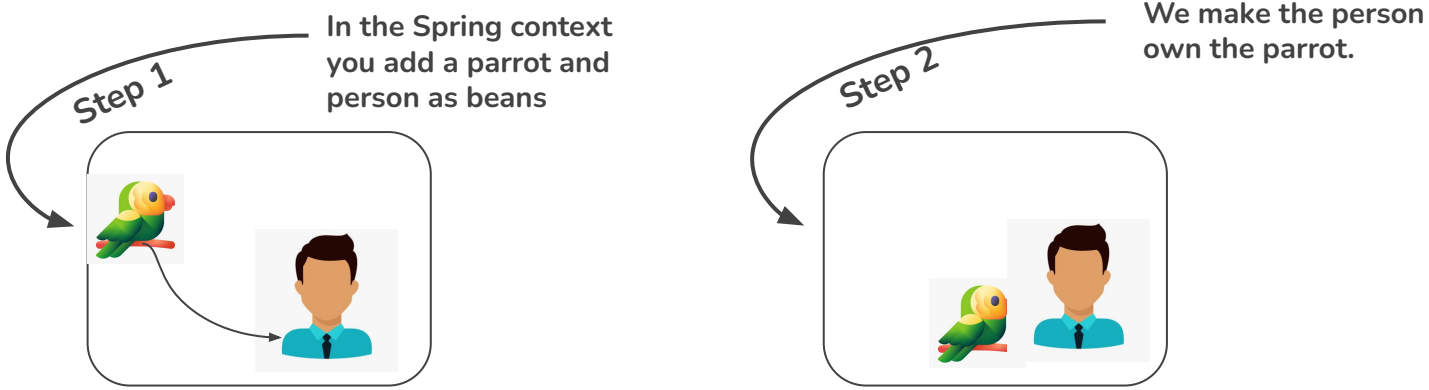
-2024-

CONTENT

Establishing Relationships among beans(1/3)

1.1 In this section, we will learn to implement the relationship between two beans defined in the configuration class annotating methods with the @Bean annotation. **Example:**

So, for each of the two approaches (wiring and autowiring), we have two steps



Establishing Relationships among beans(2/3)

```java
public class Parrot {
    private String name;
    // Omitted getters and setters
    @Override
    public String toString() {
        return "Parrot : " + name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

n.java

```java
public class Person {
    private String name;
    private Parrot parrot;

    public void setName(String name) {
        this.name = name;
    }
    public void setParrot(Parrot parrot) {
        this.parrot = parrot;
    }
```

```java
package com.company;

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

@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

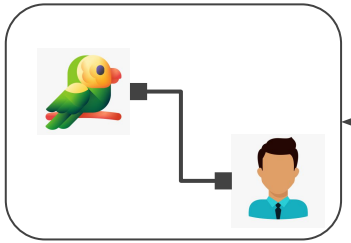
Establishing Relationships among beans(3/3)

- Placement of new objects in the context

```java
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

We define the relationship between the person bean and the parrot bean by directly calling the method that returns the bean we want to set.

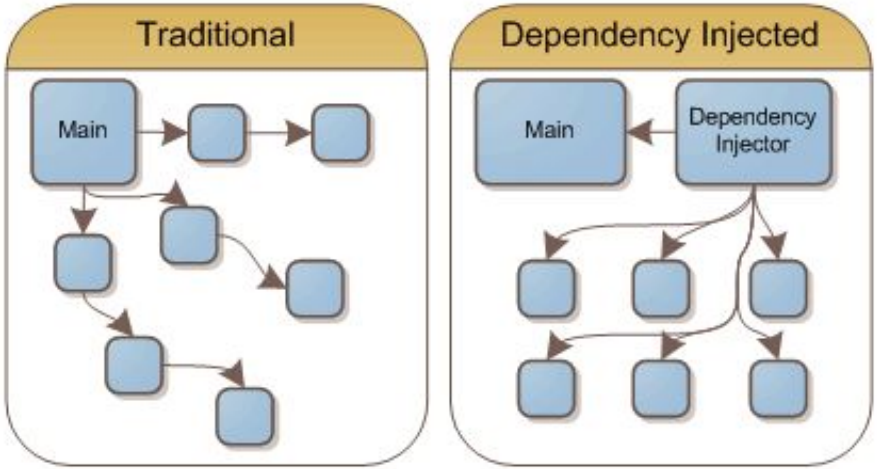
Context



The result is the has-A relationship between the two beans. The person has-A (owns) the parrot

Dependency Injection (DI)

- Dependency Injection (DI) is a design pattern and technique that allows objects to receive their dependencies from external sources, rather than creating or managing them internally.



Dependency Inversion (DI)

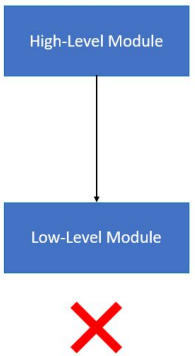
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

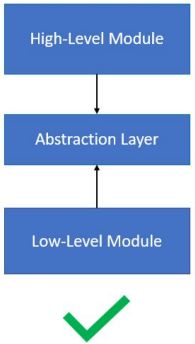
OOP

Abstraction

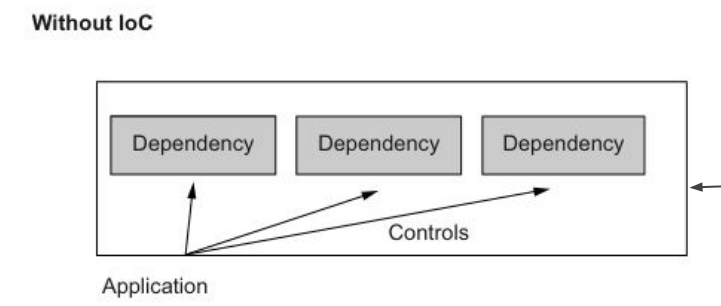
SOLID (D)

Dependency inversion Principle

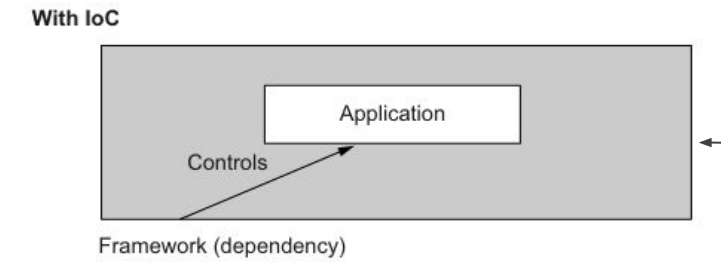




Inversion of control (IoC)



The application executes and controls (uses) the dependencies it needs.



The application executes being controlled by the framework (dependency).

Using Dependency injection(1/3)

```java
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person(Parrot parrot) {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot);
        return p;
    }
}
```

Spring injects the parrot bean into this parameter.

Using Dependency injection(2/3)

Configuration annotation in Spring

- `@Configuration` is a class-level annotation indicating that an object is a source of bean definitions.
- `@Configuration` classes declare beans through `@Bean`-annotated methods.
- Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies

```
@Configuration
class VehicleFactoryConfig{

    @Bean
    Engine engine(){
        return new Engine();
    }
}
```

Using Dependency injection(3/3)

1. Spring calls the parrot() method to create the parrot bean and add it to the context

2. Spring calls the person() method to create the person bean and add it to the context.

```
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

3. Does this mean a second parrot instance is created here when Spring creates the person bean?

The definition of the Main class

```
public class Main {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext
                (ProjectConfig.class);
        Person person =
                context.getBean(Person.class);
        Parrot parrot =
                context.getBean(Parrot.class);
        System.out.println(
                "Person's name: " + person.getName());
        System.out.println(
                "Parrot's name: " + parrot.getName());
        System.out.println(
                "Person's parrot: " + person.getParrot());
    }
}
```

Creates an instance of the Spring context based on the configuration class

Gets a reference to the Person bean from the Spring context

Gets a reference to the Parrot bean from the Spring context

Prints the person , parrot and person's parrot instances

This is Result in console

The result is the has-A relationship between the two beans. The person has-A (owns) the parrot.

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Person's name: Ella
Parrot's name: Koko
Person's parrot: Parrot : Koko

Process finished with exit code 0
```

Relationships in Java

- Has-A relationship essentially implies that an example of one class has a reference to an occasion of another class

Composition(Has-A) relationship(1/2)

- Composition is a concept where a class contains an object of another class, and it forms a “has-a” relationship.
- It is an alternative to inheritance and allows for building complex objects by combining simpler ones.
- It promotes flexibility, as the components can be changed or replaced independently.
- In Java, composition is typically achieved by creating an instance variable within a class.

Composition(Has-A) relationship(2/2)

```java
class Engine {
    public void start() {
        // Additional Engine logic
    }
}

public class Car {

  private Engine engine;

    public Car() {
        engine = new Engine();
    }

    public void start() {
        engine.start();
        // Additional car-specific logic
    }
}
```

Has-a

@Component annotation in Spring

Spring Container will automatically create and manage the spring bean for the above class because it is annotated with `@Component` annotation

Case -1

```java
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

Case -2

```java
@Component
class ComponentDemo{
    public String getValue() {
        return "Hello World";
    }
```

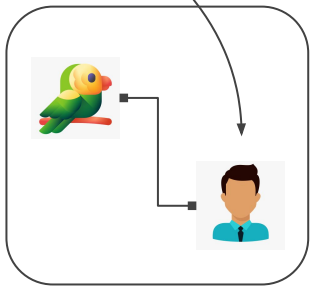
@Autowired annotation to inject beans(1/3)

```java
@Component
public class Person {
    private String name="John";
    @Autowired
    private Parrot parrot;

    public void setName(String name) {
        this.name = name;
    }
    public void setParrot(Parrot parrot) {
        this.parrot = parrot;
    }
    public String getName() {
        return name;
    }
    public Parrot getParrot() {
        return parrot;
    }
}
```

The stereotype annotation @Component instructs Spring to create and add a bean to the context of the type of this class: Person.

We instruct Spring to provide a bean from its context and set it directly as the value of the field, annotated with @Autowired. This way we establish a relationship between the two beans

Context

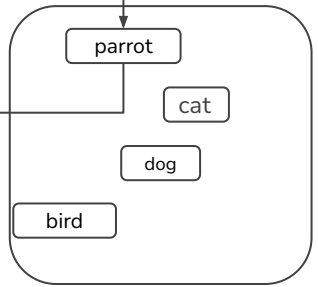


@Autowired annotation to inject beans(2/3)

```java
@Component
public class Person {

    private String name = "Ella";
    @Autowired
    private Parrot parrot;

    public void setParrot(Parrot parrot) { this.parrot = parrot; }

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }

    public Parrot getParrot() { return parrot; }
}
```

Step 1 . @Autowired says: I need to find the Parrot type bean from the context

Step 2 . @Autowired says: I found it and now inject it into this field



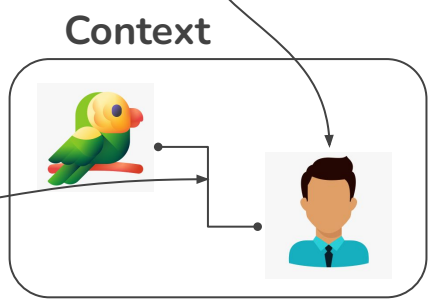
Context

@Autowired annotation to inject beans(3/3)

```java
@Component
public class Person {

    private String name = "Ella";

    private final Parrot parrot;

    @Autowired
    public Person(Parrot parrot) {
        this.parrot = parrot;
    }
```

The stereotype annotation @Component instructs Spring to create and add a bean to the context of the type of this class: Person.

When Spring creates the bean of type Person, it calls the constructor annotated with @Autowired. Spring provides a bean of type Parrot from its context as value of the parameter.



@Autowired annotation to inject beans(3/3)

this. keyword in java

```java
public class Main1 {
    int x;

    // Constructor with a parameter
    public Main1(int x) {
        this.x = x;
    }

    // Call the constructor
    public static void main(String[] args) {
        Main1 myObj = new Main1( x: 5);
        System.out.println("Value of x = " + myObj.x);
    }
}
```

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If we omit the keyword in the example above, the output would be "0" instead of "5".

Dealing with circular dependencies

1. Spring needs to create a bean of type Parrot

2. To call the Parrot class constructor, Spring needs a bean of type Person.

```java
@Component
public class Parrot {
    String name;
    private final Person person;

    @Autowired
    public Parrot(Person person) {
        this.person = person;
    }
```

3. Spring tries to create the bean of type Person.

4. To create a bean of type Person, Spring needs a bean of type Parrot. Spring is now in a deadlock!

```java
@Component
public class Person {

    private String name = "Ella";

    private final Parrot parrot;

    @Autowired
    public Person(Parrot parrot) {
        this.parrot = parrot;
    }
```

A circular dependency. Spring needs to create a bean of type Parrot. But because Parrot has as a dependency a Person, Spring needs first to create a Person. However, to create a Person, Spring already needs to have built a Parrot. Spring is now in a deadlock. It cannot create a Parrot because it needs a Person, and it cannot create a Person because it needs a Parrot.

While the project is running

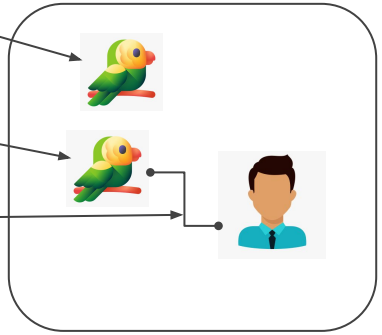
```
Caused by:
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error
creating bean with name 'parrot': Requested bean is currently in creation:
Is there an unresolvable circular reference?
    at
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.before
SingletonCreation(DefaultSingletonBeanRegistry.java:347)
```

Choosing from multiple beans in Spring context(1/3)

```java
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot1() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Parrot parrot2() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
    @Bean
    public Person person(Parrot parrot2) {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot2);
        return p;
    }
}
```

When the Spring context contains multiple beans of the same type, Spring will select the bean whose name matches the name of the parameter

The name of the parameter matches the name of the bean representing parrot Miki.



Choosing from multiple beans in Spring context(1/3)

While the project is running

```
Parrot created
Person's name: Ella
Person's parrot: Parrot : Miki
```

Choosing from multiple beans in Spring context(2/3)

- ## @Qualifier annotation in Java

  Using the @Qualifier annotation, we clearly mark our intention to inject a specific bean from the context. By using the @*Qualifier* annotation, we can eliminate the issue of which bean needs to be injected.

Choosing from multiple beans in Spring context(3/3)

```java
@Configuration
public class MyProjectConfig {
    @Bean
    public Parrot parrot1() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Parrot parrot2() {
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }
```

```java
@Bean
public Person person(
        @Qualifier("parrot2") Parrot parrot) {
    Person p = new Person();
    p.setName("Ella");
    p.setParrot(parrot);
    return p;
}
```

Conclusion

- When implementing an app, you need to refer from one object to another. This way, an object can delegate actions to other objects when executing their responsibilities. To implement this behavior, you need to establish relationships among the beans in the Spring context.
- Whenever you allow Spring to provide a value or reference through an attribute of the class or a method or constructor parameter, we say Spring uses DI, a technique supported by the IoC principle.
- The creation of two beans that depend on one another generates a circular dependency. Spring cannot create the beans with a circular dependency, and the execution fails with an exception. When configuring your beans, make sure you avoid circular dependencies.

Resources

MANNING

Reference

1. Spring Start Here
2. Baeldung.com
3. w3schools.com

Thank you!

Presented by

Qodirov Xudoyberdi

(qodirovhudoberdi4@gmail.com)