

## Autowire

**Autowire** allows Spring to automatically resolve dependencies by type, injecting the appropriate beans into fields, constructors, or methods.

### Steps:

1. Create a @Bean method in the configuration class.
2. Use @Autowired to automatically inject the dependency (in the constructor, field, or method).
3. In newer versions of Spring, the @Autowired annotation can be skipped as Spring performs automatic autowiring by type.

There are several ways to perform autowiring in Spring: **field-level**, **setter-based**, and **constructor-based** injection. Each method has its use case and benefits. Here's a detailed explanation of each, along with the code examples.

### 1. Field-Level Autowiring

**Field-level autowiring** is the simplest and most commonly used form of dependency injection. The dependency is injected directly into the field of the class using the @Autowired annotation.

#### How it works:

- The @Autowired annotation is placed directly on the field.
- Spring will automatically inject the appropriate bean by matching the type of the field with the bean.

#### Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
```

```

@Configuration
public class AppConfig {

    @Bean
    public Alien alien() {
        return new Alien();
    }

    @Bean
    public Desktop desktop() {
        return new Desktop();
    }
}

@Component
class Alien {
    @Autowired // Field-level Autowiring
    private Computer computer; // Injected by Spring

    public void code() {
        System.out.println("Coding...");
        computer.compile(); // Calls the compile method of injected Computer
    }
}

interface Computer {
    void compile();
}

@Component
class Desktop implements Computer {
    @Override
    public void compile() {

```

```

        System.out.println("Compiling using Desktop");
    }
}

// Main class
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
        AnnotationConfigApplicationContext(AppConfig.class);

        Alien alien = context.getBean(Alien.class);
        alien.code(); // Output: "Coding..." followed by "Compiling using Desktop"
    }
}

```

## Key Points:

- **Pros:** Easy to use and clean code.
- **Cons:** Less testable since the field is private, and you cannot pass dependencies through the constructor easily for testing.
- **Use case:** Quick injection where testing flexibility is not a concern.

## 2. Setter-Based Autowiring

**Setter-based autowiring** uses setter methods to inject dependencies into a class. It is a good option when you want to have some control over when the dependency is injected.

### How it works:

- The @Autowired annotation is placed on the setter method.
- Spring calls the setter method to inject the dependency.

### Example:

```
@Component
```

```
class Alien {
    private Computer computer;

    public void code() {
        System.out.println("Coding...");
        computer.compile(); // Calls the compile method of injected Computer
    }

    @Autowired // Setter-based Autowiring
    public void setComputer(Computer computer) {
        this.computer = computer;
    }
}

// Rest of the code remains the same as above
```

## Key Points:

- **Pros:** More flexibility and better for testing (you can inject mocks/stubs later).
- **Cons:** Slightly more verbose than field injection.
- **Use case:** When you need to perform additional logic before setting the dependency or if you want to make the dependency optional.

## 3. Constructor-Based Autowiring

**Constructor-based autowiring** is considered the best practice by many developers. It ensures that the required dependencies are injected when the object is created, making the object immutable after construction.

### How it works:

- The @Autowired annotation is placed on the constructor. In Spring 4.3+, if a class has only one constructor, you can omit the @Autowired annotation.

- Spring automatically injects the necessary dependencies into the constructor.

### Example:

```
@Component
class Alien {
    private final Computer computer;

    // Constructor-based Autowiring
    @Autowired
    public Alien(Computer computer) {
        this.computer = computer; // Dependency injected through constructor
    }

    public void code() {
        System.out.println("Coding...");
        computer.compile(); // Calls the compile method of injected Computer
    }
}

// Rest of the code remains the same as above
```

### Key Points:

- **Pros:**
  - Enforces dependency injection at the time of object creation, making the object immutable.
  - Makes the class easier to test by passing mock objects in the constructor.
  - In newer versions of Spring, @Autowired can be omitted if the class has a single constructor.
- **Cons:** Slightly more code to write, but it is the preferred method in many cases.
- **Use case:** When you want to ensure immutability and the dependency is mandatory.

