Java-Based Configuration

Java-based configuration in Spring allows us to configure beans using Java classes instead of traditional XML configuration.

Steps:

1. Create a class annotated with @Configuration. This class will contain methods that return instances of the beans you need.
2. Use AnnotationConfigApplicationContext to initialize the Spring container and provide it with the configuration class.
3. Retrieve beans from the container using context.getBean().

Configuration Class:

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// Configuration class
@Configuration
public class AppConfig {
        @Bean
        public Desktop desktop() {
        return new Desktop();
        }
}
```

Bean Class:

```
// Bean class
class Desktop {
        public Desktop() {
      System.out.println("Desktop Object Created");
        }

        public void compile() {
      System.out.println("Compiling using Desktop");
        }
}
```

Main Class:

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

        // Retrieve the Desktop bean from the context
        Desktop dt = context.getBean(Desktop.class);
      dt.compile();  // Output: "Compiling using Desktop"
        }
}
```

Key Points:

- **@Configuration**: Marks the class as a source of bean definitions.
- **@Bean**: Defines a bean, and the method name (desktop) becomes the bean name by default.
- **AnnotationConfigApplicationContext**: Initializes the Spring context with the provided configuration class.

Bean Name

When creating beans using @Bean, Spring assigns a default bean name as the method name. However, you can customize this name by providing a custom name using the name attribute in @Bean.

Steps:

1. The default bean name is the name of the method that defines the bean.
2. To change the default name, you can use the @Bean(name = "customName") annotation.

Configuration class:

```
@Configuration
public class AppConfig {

        // Default name of the bean is "desktop"
        @Bean
        public Desktop desktop() {
        return new Desktop();
        }

        // Bean with multiple names
        @Bean(name = {"com2", "desktop1", "beast"})
        public Desktop desktop2() {
        return new Desktop();
        }
}
```

Main Class:

```
 // Main class to retrieve the bean
public class App {
```

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

        // Retrieve bean using the default name
        Desktop dt1 = context.getBean("desktop", Desktop.class);
      dt1.compile(); // Output: "Compiling using Desktop"

        // Retrieve bean using custom names
        Desktop dt2 = context.getBean("desktop1", Desktop.class);
      dt2.compile();  // Output: "Compiling using Desktop"

        Desktop dt3 = context.getBean("beast", Desktop.class);
      dt3.compile();  // Output: "Compiling using Desktop"
        }
}
```

Key Points:

- **Default Bean Name**: The method name (desktop).
- **Custom Bean Name**: You can give a bean multiple names (com2, desktop1, beast), and it can be retrieved using any of those names.

Scope

Spring beans have different scopes that define their lifecycle. The two most commonly used scopes are:

- **Singleton**: (Default) Only one instance of the bean is created and shared across the application.
- **Prototype**: A new instance of the bean is created every time it is requested.

Steps:

1. By default, Spring beans are singleton-scoped.
2. To change the scope, use the @Scope annotation and specify the scope type (e.g., "prototype").

Configuration class:

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

@Configuration
public class AppConfig {

        // Singleton scope (default)
        @Bean
        public Desktop desktopSingleton() {
        return new Desktop();
        }

        // Prototype scope (new instance each time)
        @Bean
        @Scope("prototype")
        public Desktop desktopPrototype() {
```

```
        return new Desktop();
        }
}
```

Main Class:

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

        // Singleton scope
        Desktop dt1 = context.getBean("desktopSingleton", Desktop.class);
        Desktop dt2 = context.getBean("desktopSingleton", Desktop.class);
       System.out.println(dt1 == dt2);  // Output: true (Same instance)

        // Prototype scope
        Desktop dt3 = context.getBean("desktopPrototype", Desktop.class);
        Desktop dt4 = context.getBean("desktopPrototype", Desktop.class);
       System.out.println(dt3 == dt4);  // Output: false (Different instances)
        }
}
```

Key Points:

- **Singleton Scope**: One shared instance for the entire application.
- **Prototype Scope**: A new instance is created each time the bean is requested.

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;
```

```java
@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.

Primary and Qualifier

In some cases, there are multiple beans of the same type, and Spring does not know which bean to inject. This is where @Qualifier comes in. It allows you to specify which bean should be injected.

Example with @Qualifier:

```
@Configuration
public class AppConfig {

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

	@Bean
	public Laptop laptop() {
	return new Laptop();
	}
}

@Component
class Alien {
	private Computer computer;

	// Use @Qualifier to specify which bean to inject
	@Autowired
	@Qualifier("laptop")
	public Alien(Computer computer) {
	this.computer = computer;
	}

	public void code() {
```

```
        System.out.println("Coding...");
    computer.compile();  // Calls compile on the Laptop object
        }
}

@Component
class Laptop implements Computer {
        @Override
        public void compile() {
    System.out.println("Compiling using Laptop");
        }
}

@Component
class Desktop implements Computer {
        @Override
        public void compile() {
    System.out.println("Compiling using Desktop");
        }
}

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

        // Get the Alien bean
        Alien alien = context.getBean(Alien.class);
    alien.code();  // Output: "Coding..." followed by "Compiling using Laptop"
        }
}
```

Key Points:

- **@Qualifier**: Used to resolve ambiguity when multiple beans of the same type exist.
- **Example Output**: "Compiling using Laptop", as we specified to inject the Laptop bean using @Qualifier("laptop").

Using @Primary for Default Bean Injection

You can also use the @Primary annotation to specify the default bean that should be autowired when multiple candidates are present.

Example with @Primary:

```
@Configuration
public class AppConfig {

        @Bean
        @Primary  // This will be the default bean if no @Qualifier is specified
        public Laptop laptop() {
        return new Laptop();
        }

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

With this configuration, if no @Qualifier is used, Spring will inject the Laptop bean by default because of the @Primary annotation.

Conclusion:

1. **Field-Level Autowiring**: The simplest, but can be harder to test due to private fields.
2. **Setter-Based Autowiring**: More flexible and allows optional dependencies or pre-injection logic.
3. **Constructor-Based Autowiring**: The most robust and preferred method for mandatory dependencies, enforcing immutability.
4. **Autowiring with @Qualifier**: Used when multiple beans of the same type exist, to explicitly specify which one to inject.
5. **Autowiring with @Primary**: Used to mark one bean as the default if multiple beans of the same type are present.

Component Stereotype Annotations

1. What are Stereotype Annotations?

Stereotype annotations in Spring are specialized annotations that define and manage Spring Beans in the application context. These annotations assign specific roles to classes within the application, enabling Spring to detect and register them automatically as beans.

2. Types of Stereotype Annotations

1. **@Component**:
    - General-purpose annotation to mark a class as a Spring-managed component.
    - Automatically registers the class as a bean when component scanning is enabled.

Example:
@Component

```
public class Alien {
    public void code() {
        System.out.println("Alien is coding!");
    }
}
```

2. **@Service**:
    - Specialization of @Component for the service layer.
    - Indicates a class containing business logic.

Example:

```
@Service
public class AlienService {
    public String assist() {
        return "Assistance provided by AlienService.";
```

```
    }
}
```

3. **@Repository**:
    - Specialization of @Component for the data access layer.
    - Indicates a class interacting with the database.

Example:

```
@Repository
public class AlienRepository {
    public List<String> fetchAliens() {
        return List.of("Alien1", "Alien2");
    }
}
```

4. **@Controller**:
    - Specialization of @Component for the presentation layer.
    - Handles HTTP requests in a Spring MVC application.

Example:

```
@Controller
public class AlienController {
    @GetMapping("/alien")
    public String greet() {
        return "Greetings from Alien!";
    }
}
```

3. The @Component Annotation

- **Purpose**:
    - Marks a class as a Spring component.
    - Simplifies bean creation and management.

- **Key Features**:
    - Generic stereotype annotation for any Spring-managed component.
    - Automatically detected during component scanning.

Example:

```
@Component
public class Alien {
    public void code() {
        System.out.println("Alien is coding!");
    }
}
```

4. @ComponentScan with @Configuration

1. **What is @ComponentScan?**
    - Specifies which packages Spring should scan for components, services, repositories, and controllers.
    - Detects and registers classes annotated with @Component, @Service, @Repository, and @Controller.
2. **Use of @ComponentScan with @Configuration**:
    - Often used alongside @Configuration to specify the base packages for scanning.

Example:

```
@Configuration
@ComponentScan("com.telusko")
public class AppConfig {
}
```

3. **Explanation**:
    - **@Configuration**:
        - Indicates the class provides Spring configuration.
    - **@ComponentScan("com.telusko")**:

        - Scans the com.telusko package and sub-packages for stereotype annotations.
    - Ensures all annotated classes in the specified package are registered as beans.

5. Advantages of Stereotype Annotations

- **Simplifies Configuration**:
    - No need for explicit bean definitions in XML or Java-based configuration.
- **Improves Readability**:
    - Clarifies the role of a class in the application.
- **Enables Component Scanning**:
    - Automatically detects and registers beans, reducing boilerplate code.

6. Practical Implementation

1. **Define Components**:
    - Create a class annotated with @Component.

```
@Component
public class Alien {
   public void code() {
      System.out.println("Alien is coding!");
   }
}
```

2. **Set Up Configuration**:
    - Use @Configuration and @ComponentScan to enable component scanning.

```
@Configuration
@ComponentScan("com.telusko")
public class AppConfig {
```

```
}
```

3. **Access Beans**:
    - Retrieve the bean from the Spring ApplicationContext.

```
public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Alien alien = context.getBean(Alien.class);
        alien.code();
    }
}
```

Autowire field, constructor, Setter

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;
```

```java
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.

Primary Annotation

Using @Primary for Default Bean Injection

You can also use the @Primary annotation to specify the default bean that should be autowired when multiple candidates are present.

Example with @Primary:

```
@Configuration
public class AppConfig {

        @Bean
        @Primary  // This will be the default bean if no @Qualifier is specified
        public Laptop laptop() {
        return new Laptop();
        }

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

With this configuration, if no @Qualifier is used, Spring will inject the Laptop bean by default because of the @Primary annotation.

Note: What happen if we are using @Primary and @Qualifier
Qualifier has more precedence over primary

Scope and Value Annotations

1. Scope Annotation (@Scope)

- **Purpose**:
  Specifies the lifecycle and scope of a bean in a Spring application.
  The default scope in Spring is **singleton**, which means only one instance of the bean is created for the application context.
- **Usage**:
  - @Scope("prototype"): Creates a new instance of the bean every time it is requested.
  - Other scopes: singleton (default), request, session, application, etc.

Example:

```
@Component
@Primary
@Scope("prototype")
public class Desktop implements Computer {
    public Desktop() {
        System.out.println("Desktop Object Created..");
    }

    @Override
    public void compile() {
        System.out.println("Compiling using Desktop");
    }
}
```

  - **Explanation**:
    - The @Scope("prototype") ensures that a **new instance of Desktop** is created every time it is injected or retrieved from the Spring context.

        - The @Primary annotation marks this as the default bean for injection when there are multiple implementations of the Computer interface.

2. Value Annotation (@Value)

- **Purpose**:
  Used to inject a specific value into a field, method parameter, or constructor. It can be used to set default values, read values from property files, or inject constant values.
- **Usage**:
    - Can inject literals, expressions, or property values.
    - Format: @Value("value").

Example:

```
@Component
public class Alien {
    @Value("21")
    private int age;

    private Computer com;

    public Alien() {
        System.out.println("Alien Object Created");
    }
}
```

    - **Explanation**:
        - The @Value("21") injects the value 21 into the age field of the Alien class.
        - The Alien bean will always have age set to 21 unless overridden by external configuration.

Key Points to Remember

1. **@Scope("prototype")**:
    - Creates a new instance of the bean for every request.
    - Useful for beans with independent state for each use.
2. **@Value**:
    - Injects values directly into fields.
    - Can read values from property files or inject hardcoded constants.
3. **Practical Usage**:
    - Use @Scope("prototype") for beans where state isolation is required.
    - Use @Value for initializing fields with constant or configurable values.