Dependency Injection using Annotations

Annotations in Java provide a way to add metadata to classes, methods, or fields. By using annotations for dependency injection, developers can clearly and concisely specify dependencies directly in the code.

@Component

used to indicate that a class is a candidate for auto-detection and instantiation as a **bean** within the Spring container.

When you annotate a class with `@Component`, Spring scans the **classpath** during the application startup and automatically detects and registers the annotated class as a Spring bean. Spring then manages the lifecycle and dependencies of these beans.
The `@Component` annotation is a generic stereotype annotation, which means it can be used as a base annotation for more specific stereotypes like `@Service`, `@Repository`, and `@Controller`. These specific annotations provide additional semantics and help in distinguishing different types of components in the application.

When Spring finds a class annotated with **`@Component`**, it automatically creates an instance of that class and registers it as a bean in the application context. These beans can then be injected into other beans or retrieved from the context as needed.

By default, beans annotated with **`@Component`** are singleton-scoped, meaning that Spring creates only one instance of the bean and shares it across the application context.

Example:

Student.java

```
package com.itsc.ioc.springioc;

import org.springframework.stereotype.Component;

@Component
public class Student {
     public void Details() {
```

```
            System.out.println("Student Details");
      }
}
```

Create a class called AppConfig class in the Java configuration approach as an alternative to the XML configuration file.

AppConfig.java

```
package com.itsc.ioc.springioc;

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

@Configuration
@ComponentScan("com.itsc.ioc.springioc")
public class AppConfig {
      //config code
}
```

The **AppConfig** class is a Java configuration class in the Spring framework. It is used to define the configuration for your Spring application and is an alternative to the traditional XML-based configuration.

The **AppConfig** class is typically annotated with @**Configuration**, indicating that it contains Spring bean definitions and other configuration details. Inside this class, you can define beans, specify component scanning, configure aspects, and set up other application-related configurations.

In the above example, the **AppConfig** class is annotated with @**Configuration**, indicating that it is a configuration class.

The @**ComponentScan** annotation specifies the base package (com.itsc.ioc.springioc) to scan for **components**, such as classes annotated with @**Component**, @**Service**, @**Repository**, or other stereotype annotations.

Component scanning enables automatic detection and registration of these components as Spring beans.

In `AppConfig`, we annotate the class with `@Configuration` to indicate that this class contains bean configurations. Additionally, we use `@ComponentScan` to specify the base package(s) where Spring should scan for components (like the `Student` class annotated with `@Component`).

The **`AppConfig`** class is typically loaded by the Spring application context to initialize and configure the beans and other application components. You can create an instance of the application context using code like this:

```
package com.itsc.ioc.springioc;

public class Demo {

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

		Student s = context.getBean(Student.class);
	//	Student s = context.getBean("student", Student.class);

		s.Details();
	}
}
```

If you want to name the bean (by default the name of the class camel case is the name of the bean "student"), you can do so as:

```
@Component("fresh")
public class Student {
	public void Details() {
.........
```

Then you can use the name in the getBean() method as:

```
Student s = context.getBean("fresh", Student.class);
```

Adding properties to the Student class

```
package com.itsc.ioc.springioc;

@Component
public class Student {
	private String firstName, lastName;

	// setters here

	public void Details() {
		Sys.out("Full Name: " + firstName + " " + lastName);
	}
}
```

Put AppConfig class intact and modify your launcher class like:

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

	Student s = context.getBean(Student.class);

	s.setFirstName("John");
	s.setLastName("Doe");
	s.Details();
}
```

Using Setter Injection

Let's modify the **`AppConfig`** class to define a **`Student`** bean and inject its dependencies using setter injection.

```
package com.itsc.ioc.springioc;


@Configuration
@ComponentScan("com.itsc.ioc.springioc")
public class AppConfig {

	@Bean
	public Student student() {
		Student student = new Student();
```

```
            student.setFirstName("John");
            student.setLastName("Doe");
            return student;
        }
}
```

In this modified **`AppConfig`** class:

- We've annotated the class with **`@Configuration`** to indicate that this class contains bean configurations.
- We've defined a **`student()`** method annotated with **`@Bean`**. This method creates an instance of the **`Student`** class, sets its properties using setter injection, and returns it as a bean.
- Setter injection is performed within the **`student()`** method by invoking the **`setFirstName()`** & **`setLastName()`** methods to set the name of the **`Student`** bean.

Now, let's update the **`Main`** class to retrieve the **`Student`** bean from the application context created using the **`AppConfig`** class.

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

      Student s = context.getBean(Student.class);

      s.Details();
}
```

@Autowired annotation

Used to inject the bean automatically.

Used in constructor injection, setter injection and field injection.

Setter injection

Suppose we have the following classes.

```
package com.itsc.ioc;

public class School {
     private String schoolName;

     //setter and getter for the field here

}
```

```
package com.itsc.ioc;

public class Student {
     private String name;
     private School school;

     …setters and getters for the fields here

}
```

Next, let's implement setter injection using Spring annotations:

```
package com.itsc.ioc;

@Component
public class School {
     private String schoolName;

     //setter and getter for the field here

}
```

```
package com.itsc.ioc;

@Component
public class Student {
	private String name;
	private School school;

	…setters and getters for the fields here

	@Autowired
	public void setSchool(School school) {
		this.school = school;
	}

	public void getDetails() {
		Sout(getName() + " is learning at : " +
					getSchool().getSchoolName());
	}


}
```

Here, the **`setSchool()`** method is annotated with **`@Autowired`**. When Spring creates an instance of the **`Student`** class, it looks for a bean of type **`School`** and automatically injects it into the **`school`** property using the setter method. Spring will automatically invoke this method and provide the School dependency.

In the above example, we have a **`Student`** class annotated with **@Component** to indicate that it is a Spring bean. It has a name field of type String and a school field of type **School**.

Now, let's create a launcher class to demonstrate the usage:

```
package com.itsc.ioc;


public class SpringIocApplication {

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

		Student student = context.getBean(Student.class);
		student.setName("John");
```

```
            School school = context.getBean(School.class);

            school.setSchoolName("AAiT");

            student.getDetails();

      }

}
```

Constructor Injection

```
package com.itsc.ioc;


@Component
public class Student {
      private String name;
      private School school;

      @Autowired
      public Student(School school) {
            this.school = school;
      }

      public School getSchool() {
            return school;
      }
      public String getName() {
            return name;
      }
      public void setName(String name) {
            this.name = name;
      }

      public void getDetails() {
            System.out.println(getName() + " is learning at : " +
            getSchool().getSchoolName());
      }
}
```

In this version of the `Student` class, we've removed the setter method for `School` injection and instead created a constructor that takes a `School` parameter annotated with `@Autowired`. This indicates that Spring should inject the `School` bean into this constructor.

The rest remains the same.

In this setup, Spring will automatically inject the `School` bean into the `Student` bean using constructor injection. When you run the `Main` class, you should get the same output as before, with the `School` dependency being injected into the `Student` bean.

Field injection (discouraged)

Field injection involves directly injecting dependencies into fields using annotations.

```
package com.itsc.ioc;


@Component
public class Student {
	private String name;

	@Autowired
	private School school;


	public School getSchool() {
		return school;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public void getDetails() {
		System.out.println(getName() + " is learning at : " +
getSchool().getSchoolName());
	}
}
```

In this version of the `Student` class, we've removed the setter and constructor for `School` injection and instead directly annotated the `school` field with `@Autowired`. This tells Spring to inject the `School` bean directly into this field.

Now, let's keep the `Main` class the same as before, and check your result.

@Qualifier Annotation

used to disambiguate between multiple beans of the same type when performing autowiring.

let's suppose we have two different types of schools: a primary school and a high school. We'll define two separate classes for them:

```
package com.itsc.ioc;

public interface SchoolInterface {
	void setSchoolName(String schoolName);
	String getSchoolName();
}
```

This interface will be implemented by both the `PrimarySchool` and `HighSchool` classes:

```
@Component
public class PrimarySchool implements SchoolInterface{
	private String schoolName;

	public String getSchoolName() {
		return "Primary School: " + schoolName;
	}

	public void setSchoolName(String schoolName) {
		this.schoolName = schoolName;
	}
}
```

```
@Component
public class HighSchool implements SchoolInterface{
	private String schoolName;

	public String getSchoolName() {
		return "HighSchool : " + schoolName;
	}

	public void setSchoolName(String schoolName) {
		this.schoolName = schoolName;
	}
```

```
}
```

Both `PrimarySchool` and `HighSchool` classes implement the `School` interface and provide their own implementation for the `setSchoolName()` and `getSchoolName()` methods.

Now, when we annotate these classes with `@Component`, they will be automatically registered as beans in the Spring application context, allowing us to inject them into other components.

The `Student` class can then be modified to use `@Autowired` and `@Qualifier` annotations to specify which bean to inject:

```
package com.itsc.ioc;


@Component
public class Student {
	private String name;

	// use @Qualifier to specify which bean to inject

	@Autowired
	@Qualifier("primarySchool")
	private SchoolInterface school;


	public SchoolInterface getSchool() {
		return school;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public void getDetails() {
		System.out.println(getName() + " is learning at : " +
getSchool().getSchoolName());
	}
}
```

Here, we've annotated the `school` field with `@Autowired` and `@Qualifier("primarySchool")`. This tells Spring to inject the bean named "primarySchool" into this field.

AppConfig class

```
package com.itsc.ioc;


@Configuration
@ComponentScan("com.itsc.ioc")
public class AppConfig {
	@Bean
	public SchoolInterface primarySchool() {
		return new PrimarySchool();
	}

	@Bean
	public SchoolInterface highSchool() {
		return new HighSchool();
	}
//	@Bean
//	public Student student() { //optional
//		return new Student();
//	}
}
```

To change the name to something else (eg. primary), you have to use that name in the **AppConfig** class as well and use it while calling the bean in the main class using getBean() method.

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

		SchoolInterface school =
				context.getBean(PrimarySchool.class);

		Student s = context.getBean(Student.class);

		s.setName("John");

		school.setSchoolName("AAiT");

		s.getDetails();

	}
```

If you're relying on component scanning to detect your `@Component`-annotated classes, you can leave the `AppConfig` class empty.

```
package com.itsc.ioc;


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

With this setup, Spring will automatically scan the specified package (`"your.package.name"`) for classes annotated with `@Component`, `@Service`, `@Repository`, and `@Controller` and register them as beans in the application context.

Ensure to replace `"your.package.name"` with the actual package where your `Student`, `School`, `PrimarySchool`, and `HighSchool` classes are located.

The @Qualifier annotation resolves the problem of which School interface implementation should be injected.

The `@Qualifier` annotation provides a solution to resolve ambiguity when there are multiple beans of the same type. By specifying the bean name, you can indicate which specific bean should be injected. This is particularly useful when you have multiple implementations of an interface or multiple beans of the same type, and you need to specify which one should be injected into a particular dependency.

In our example, we had two implementations of the `School` interface: `PrimarySchool` and `HighSchool`. By using `@Qualifier("primarySchool")`, we instructed Spring to inject the bean named "primarySchool" into the `Student` class. This ensures that the correct implementation of the `School` interface is injected, resolving any ambiguity that might arise due to multiple bean definitions.

Using `@Qualifier` along with `@Autowired` provides fine-grained control over dependency injection and helps maintain clarity and consistency in your Spring application.

In Constructor-based injection

If you're using constructor injection and want to use `@Qualifier`, you can apply it to the constructor parameters. Let's modify the `Student` class to use constructor injection and demonstrate how to use `@Qualifier` in that scenario:

```
@Component
public class Student {
	private String name;
	private SchoolInterface school;

	@Autowired
	public Student(@Qualifier("primarySchool") SchoolInterface
school) {
		this.school = school;
	}

	public SchoolInterface getSchool() {
		return school;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public void getDetails() {
		System.out.println(getName() + " is learning at : " +
getSchool().getSchoolName());
	}
}
```

Here, we've removed `@Autowired` from the `school` field and applied it directly to the constructor. Additionally, we've added `@Qualifier("primarySchool")` to specify that the `PrimarySchool` bean should be injected into this constructor.

Now, when Spring creates an instance of the `Student` class, it will inject the `PrimarySchool` bean into the constructor parameter based on the `@Qualifier` annotation.

The `AppConfig` class remains unchanged, and the `Main` class can stay the same as well.

In Setter-based Injection

```
@Component
public class Student {
	private String name;
	private SchoolInterface school;


	@Autowired
	public void
		setSchool(@Qualifier("primarySchool")SchoolInterface
	school) {
		this.school = school;
	}
	public SchoolInterface getSchool() {
		return school;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public void getDetails() {
		System.out.println(getName() + " is learning at : " +
getSchool().getSchoolName());
	}
}
```


You can remove the AppConfig class altogether

you can remove the `AppConfig` class altogether if you're relying on component scanning to automatically detect and register your components.

In the absence of the `AppConfig` class, you'll need to ensure that your `@Component`-annotated classes (`Student`, `PrimarySchool`, `HighSchool`) are within the package(s) specified in the component scan. Then Spring will automatically detect these classes and register them as beans in the application context.

```
ApplicationContext context = new
                AnnotationConfigApplicationContext("com.itsc.ioc");
```

- Instead of passing a class to `AnnotationConfigApplicationContext`, we're passing the base package name `"your.package.name"` to specify where Spring should perform component scanning.
- Spring will scan this package and its sub-packages for classes annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, etc., and register them as beans in the application context.
- The rest of the `Main` class remains unchanged. We retrieve the `Student` bean from the context and interact with it as before.

More Examples

Setter Injection Example

To demonstrate the usage of setter injection, let's create a few interfaces and classes.

MessageService

```
public interface MessageService {
    void sendMessage(String message);
}
```

EmailService

```
@Component
public class EmailService implements MessageService{

    @Override
    public void sendMessage(String message){
        System.out.println(message);
    }
}
```

We have annotated *EmailService* class with **@Component annotation** so the Spring container automatically creates a Spring bean and manages its life cycle.

SMSService

```
@Component("smsService")
public class SMSService implements MessageService{

    @Override
    public void sendMessage(String message){
        System.out.println(message);
    }
}
```

We have annotated *SMSService* class with **@Component annotation** so the Spring container automatically creates a Spring bean and manages its life cycle.

MessageSender

In setter injection, Spring will find the **@Autowired annotation** and call the setter to inject the dependency.

```
@Component
public class MessageSender {

    private MessageService messageService;

    @Autowired
    public void setMessageService(@Qualifier("emailService") MessageService
messageService) {
        this.messageService = messageService;
        System.out.println("setter based dependency injection");
    }

    public void sendMessage(String message){
        this.messageService.sendMessage(message);
    }
}
```

@Qualifier annotation is used in conjunction with **Autowired** to avoid confusion when we have two or more beans configured for the same type.

Spring container uses the below setter method to inject dependency on any Spring-managed bean (MessageSender is a Spring bean):

```
    @Autowired
    public void setMessageService(@Qualifier("emailService") MessageService
messageService) {
        this.messageService = messageService;
        System.out.println("setter based dependency injection");
    }
```

AppConfig

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

@Configuration: Used to indicate that a configuration class declares one or more @Bean methods. These classes are processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

@ComponentScan: This annotation is used to specify the base packages to scan for spring beans/components.

Testing

Let's create ApplicationContext and test this example:

```
public class Client {

    public static void main(String[] args) {

        String message = "Hi, good morning have a nice day!.";
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AppConfig.class);

        MessageSender messageSender =
applicationContext.getBean(MessageSender.class);
        messageSender.sendMessage(message);
    }
}
```