**SECTION2 :**

**package** com.in28minutes.learn\_spring\_framework;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

**import** com.in28minutes.learn\_spring\_framework.game.GameRunner;

**import** com.in28minutes.learn\_spring\_framework.game.GamingConsole;

**import** com.in28minutes.learn\_spring\_framework.game.MarioGame;

**import** com.in28minutes.learn\_spring\_framework.game.PacmanGame;

**import** com.in28minutes.learn\_spring\_framework.game.SuperContraGame;

@Configuration

**public** **class** App03GamingSpringBeans {

@Bean

**public** GamingConsole game()

{

**var** game = **new** PacmanGame();

**return** game;

}

@Bean

**public** GameRunner gameRunner()

{

**var** gameRunner = **new** GameRunner(game());

**return** gameRunner;

}

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(App03GamingSpringBeans.**class**);

context.getBean(GamingConsole.**class**).up();

context.getBean(GameRunner.**class**).run();

}

}

In Spring Boot, @Component and @ComponentScan are essential annotations used for component scanning and managing the application's beans. Here's a detailed explanation:

**1. @Component**

* **Definition**: The @Component annotation is a generic stereotype for any Spring-managed component or bean. It is a marker annotation that indicates that a class is a Spring component.
* **Purpose**: When a class is annotated with @Component, Spring's component scanning mechanism automatically detects it and registers it as a bean in the application context.
* **Usage**: Typically, you use @Component to annotate classes that are neither a @Service, @Repository, nor @Controller. It is a generic annotation, while the other three are more specific stereotypes.

**2. @ComponentScan**

* **Definition**: The @ComponentScan annotation is used with configuration classes to specify the packages that Spring should scan for annotated components (like @Component, @Service, @Repository, @Controller, etc.).
* **Purpose**: It instructs Spring to scan the specified packages to find and register beans. This is necessary when your components are not in the default package (the same package as the Spring Boot application class).
* **Usage**: @ComponentScan can be used on configuration classes (classes annotated with @Configuration), or directly on the main application class in a Spring Boot application.

**3. Working Procedure in Spring Boot**

1. **Startup**:
   * When a Spring Boot application starts, it scans the classpath for components to manage as beans. This scanning is usually configured in the main application class (annotated with @SpringBootApplication), which implicitly includes @ComponentScan.
2. **Component Scanning**:
   * Spring Boot will automatically scan the package where the main application class is located and its sub-packages for components, unless specified otherwise using @ComponentScan.
3. **Bean Registration**:
   * As the components are found, Spring registers them as beans in the application context. Each bean is managed by Spring's Inversion of Control (IoC) container, which handles their lifecycle, dependencies, and injection into other components.
4. **Dependency Injection**:
   * If a component needs to use another bean, it can do so via dependency injection. Spring Boot will automatically inject the required beans based on the constructor, field, or method annotations (@Autowired, for example).
5. **Application Context**:
   * Once all components are scanned and registered, Spring Boot's application context is fully initialized, and the application is ready to process incoming requests or perform tasks.

This mechanism allows for modular and scalable application development in Spring Boot, where each component is independently managed and easily reusable.

**EXAMPLE:**

**package** com.in28minutes.learn\_spring\_framework;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

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

**import** com.in28minutes.learn\_spring\_framework.game.GameRunner;

**import** com.in28minutes.learn\_spring\_framework.game.GamingConsole;

**import** com.in28minutes.learn\_spring\_framework.game.MarioGame;

**import** com.in28minutes.learn\_spring\_framework.game.PacmanGame;

**import** com.in28minutes.learn\_spring\_framework.game.SuperContraGame;

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.game")

**public** **class** App03GamingSpringBeans {

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(App03GamingSpringBeans.**class**);

context.getBean(GamingConsole.**class**).up();

context.getBean(GameRunner.**class**).run();

}

}

--

**package** com.in28minutes.learn\_spring\_framework.game;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** PacmanGame **implements** GamingConsole {

**public** **void** up()

{

System.***out***.println("PacmanGame JUMP");

}

**public** **void** down()

{

System.***out***.println("PacmanGame Go into hole");

}

**public** **void** left()

{

System.***out***.println(" PacmanGame Go back");

}

**public** **void** right()

{

System.***out***.println("PacmanGame Accerlate");

}

}

--

**package** com.in28minutes.learn\_spring\_framework.game;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** GameRunner {

// MarioGame game ;

GamingConsole game;

**public** GameRunner(GamingConsole game) {

**this**.game = game ;

}

**public** **void** run()

{

System.***out***.println("RUNNING GAME IS " +game);

game.down();

game.up();

game.left();

game.right();

}

}

In Spring, @Primary and @Qualifier are annotations used to resolve bean conflicts when multiple beans of the same type are available in the application context. Here's a detailed explanation:

**1. @Primary**

* **Definition**: The @Primary annotation is used to indicate that a specific bean should be given preference when multiple beans of the same type are available.
* **Purpose**: If there are multiple beans of the same type and Spring needs to autowire one of them without further specification, the one marked with @Primary will be chosen. This avoids the ambiguity of which bean to inject.
* **Usage**: You annotate one of the beans with @Primary to designate it as the default bean to be injected when no other qualifier is specified.

**2. @Qualifier**

* **Definition**: The @Qualifier annotation is used to specify which bean should be injected when multiple beans of the same type are available in the application context.
* **Purpose**: @Qualifier provides more granular control over dependency injection by allowing you to explicitly name the bean that should be injected, even when @Primary is used on another bean.
* **Usage**: You annotate the injection point with @Qualifier followed by the name of the bean you want to inject.

--

**package** com.in28minutes.learn\_spring\_framework.game;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** GameRunner {

// MarioGame game ;

GamingConsole game;

**public** GameRunner(@Qualifier("SuperContraGameQualifier")GamingConsole game) {

**this**.game = game ;

}

**public** **void** run()

{

System.***out***.println("RUNNING GAME IS " +game);

game.down();

game.up();

game.left();

game.right();

}

}

--

**package** com.in28minutes.learn\_spring\_framework.game;

**import** org.springframework.context.annotation.Primary;

**import** org.springframework.stereotype.Component;

@Component

@Primary

**public** **class** MarioGame **implements** GamingConsole{

**public** **void** up()

{

System.***out***.println("JUMP");

}

**public** **void** down()

{

System.***out***.println("Go into hole");

}

**public** **void** left()

{

System.***out***.println("Go back");

}

**public** **void** right()

{

System.***out***.println("Accerlate");

}

}

--

**package** com.in28minutes.learn\_spring\_framework.game;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.stereotype.Component;

@Component

@Qualifier("SuperContraGameQualifier")

**public** **class** SuperContraGame **implements** GamingConsole {

**public** **void** up()

{

System.***out***.println(" SUPER JUMP");

}

**public** **void** down()

{

System.***out***.println(" SUPER Go into hole");

}

**public** **void** left()

{

System.***out***.println(" SUPER Go back");

}

**public** **void** right()

{

System.***out***.println(" SUPER Accerlate");

}

}

**- \*\*@Primary\*\*:** Use `@Primary` to make one bean the default when there are multiple beans of the same type.

**- \*\*@Qualifier\*\*:** Use `@Qualifier` to choose a specific bean when there are multiple beans of the same type.

**Dependency Injection in Spring Boot**

**Dependency Injection (DI)** is a design pattern used in Spring Boot to achieve Inversion of Control (IoC) between classes and their dependencies. Instead of a class controlling its dependencies (by creating or fetching them), the control is inverted, and the framework (Spring) injects the required dependencies into the class. This promotes loose coupling, making the system easier to manage, test, and extend.

**Types of Dependency Injection in Spring Boot**

1. **Constructor Injection**:
   * **Description**: Dependencies are provided through a class constructor.
   * **Usage**: When a class requires mandatory dependencies to be provided at the time of object creation. This is the recommended way of injection in Spring because it ensures that all dependencies are injected at object creation, making the class immutable.
   * **Example**:

java

Copy code

@Service

public class MyService {

private final MyRepository repository;

@Autowired

public MyService(MyRepository repository) {

this.repository = repository;

}

}

1. **Setter Injection**:
   * **Description**: Dependencies are provided through setter methods after the object is created.
   * **Usage**: Useful when you want to inject optional dependencies or when a class has multiple dependencies that can be set independently.
   * **Example**:

java

Copy code

@Service

public class MyService {

private MyRepository repository;

@Autowired

public void setRepository(MyRepository repository) {

this.repository = repository;

}

}

1. **Field Injection**:
   * **Description**: Dependencies are injected directly into the fields of a class.
   * **Usage**: This is a simpler and less verbose method but not recommended due to difficulties in unit testing and violation of the principle of dependency injection. It is more difficult to create immutable objects with field injection.
   * **Example**:

java

Copy code

@Service

public class MyService {

@Autowired

private MyRepository repository;

}

1. **Interface Injection (Less Common)**:
   * **Description**: The dependency provides an injector method that the client class must implement. Spring does not support interface injection out-of-the-box, but you can achieve it with custom implementations.
   * **Usage**: Rarely used in practice, as constructor injection and setter injection are more commonly employed.

**Uses of Dependency Injection in Spring Boot**

1. **Loose Coupling**:
   * DI decouples the classes from their dependencies, making the codebase more modular and easier to maintain. If a dependency needs to be changed, it can be done with minimal changes to the code.
2. **Ease of Testing**:
   * By injecting dependencies, especially through constructors, it's easier to create mock objects for unit testing. This leads to better test coverage and more reliable code.
3. **Improved Code Readability and Maintenance**:
   * DI helps in organizing code more logically, making it easier to understand and maintain. The dependencies are clearly outlined and injected, which improves the structure of the application.
4. **Flexibility**:
   * DI allows the configuration of dependencies to be changed or extended without modifying the classes that use them. This is particularly useful in large applications where different implementations of a dependency might be needed for different environments (e.g., dev, test, prod).
5. **Scalability**:
   * With DI, it's easier to scale an application because the components are loosely coupled and can be extended or replaced without affecting other parts of the application.

In the context of Dependency Injection (DI), a \*\*dependency\*\* refers to an object or service that a class requires to perform its function. These dependencies are the building blocks that a class relies on to execute its operations.

### Example Breakdown:

Consider a class `MyService` that needs to interact with a database. To do this, it requires an instance of `MyRepository` (which might represent a database access layer). In this scenario, `MyRepository` is a \*\*dependency\*\* of `MyService`.

Without Dependency Injection:

- The `MyService` class might directly create an instance of `MyRepository` inside itself:

```java

public class MyService {

private MyRepository repository = new MyRepository();

}

```

Here, `MyService` is tightly coupled with `MyRepository`, meaning any change in how `MyRepository` is created or behaves could require changes in `MyService`.

With Dependency Injection:

- Instead of creating its own dependencies, `MyService` would receive them from an external source, typically through a constructor, setter method, or directly into a field (as discussed earlier).

```java

@Service

public class MyService {

private final MyRepository repository;

@Autowired

public MyService(MyRepository repository) {

this.repository = repository;

}

}

```

Here, `MyRepository` is injected into `MyService`, making `MyService` dependent on `MyRepository`, but without the responsibility of managing or creating it. This makes `MyService` more flexible and easier to test and maintain.

### Key Points about Dependencies:

- \*\*Dependencies are the required objects/services a class needs to function.\*\*

- \*\*Dependencies can be other classes, interfaces, or services.\*\*

- \*\*In DI, these dependencies are provided (injected) by a framework like Spring rather than being created directly by the class.\*\*

- \*\*This approach leads to more modular, testable, and maintainable code.\*\*

**`@Autowired`** in Spring Boot is an annotation used for automatic dependency injection, where Spring automatically resolves and injects the necessary beans (objects) into a class's fields, constructors, or setter methods. This allows for loose coupling and easier management of dependencies within the application.

**package** com.in28minutes.learn\_spring\_framework.example.a1;

**import** java.util.Arrays;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

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

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

**import** org.springframework.stereotype.Component;

//YourBusinessclass

//dependency1

//dependency2

@Component

**class** YourBusinessClass

{

// @Autowired

Dependency1 dependency1;

// @Autowired

Dependency2 dependency2;

@Autowired

**public** YourBusinessClass(Dependency1 dependency1, Dependency2 dependency2) {

**super**();

System.***out***.println("no problem if we dont write autoired for constrcutor and Constructor injection done sucessfully");

**this**.dependency1 = dependency1;

**this**.dependency2 = dependency2;

}

// @Autowired

// public void setDependency1(Dependency1 dependency1) {

// System.out.println("Setter Injection done sucessfully");

// this.dependency1 = dependency1;

// }

// @Autowired

// public void setDependency2(Dependency2 dependency2) {

// System.out.println("Setter Injection done sucessfully");

// this.dependency2 = dependency2;

// }

**public** String toString()

{

**return** "using" + dependency1 + "and" +dependency2;

}

}

@Component

**class** Dependency1

{

}

@Component

**class** Dependency2

{

}

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.a1")

**public** **class** DepInjectionLauncherApplication {

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(DepInjectionLauncherApplication.**class**);

Arrays.*stream*(context.getBeanDefinitionNames()).forEach(System.***out***::println);

System.***out***.println(context.getBean(YourBusinessClass.**class**));

}

}

package com.in28minutes.learn\_spring\_framework.example.c1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.c1")

public class RealWorldSpringontextLauncherApplication {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext(RealWorldSpringontextLauncherApplication.class);

Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

System.out.println(context.getBean(BusinessCalculationService.class).findMax());

}

}

---

package com.in28minutes.learn\_spring\_framework.example.c1;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.c1")

public class RealWorldSpringontextLauncherApplication {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext(RealWorldSpringontextLauncherApplication.class);

Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

System.out.println(context.getBean(BusinessCalculationService.class).findMax());

}

}

--

**package** com.in28minutes.learn\_spring\_framework.example.c1;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MysqlDataService **implements** DataService{

@Override

**public** **int**[] retrieveData() {

// **TODO** Auto-generated method stub

**return** **new** **int**[] {1,2,3,4,5};

}

}

--

**package** com.in28minutes.learn\_spring\_framework.example.c1;

**import** java.util.Arrays;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** BusinessCalculationService {

**private** DataService dataService;

**public** BusinessCalculationService(DataService dataService) {

**super**();

**this**.dataService = dataService;

}

**public** **int** findMax()

{

**return** Arrays.*stream*(dataService.retrieveData()).max().orElse(0);

}

}

--

**package** com.in28minutes.learn\_spring\_framework.example.c1;

**public** **interface** DataService {

**int**[] retrieveData();

}