Java Spring - Using @Scope Annotation to Set a POJO's Scope

Last Updated : 18 Mar, 2025

In the Spring Framework, when we declare a POJO (Plain Old Java Object) instance, what we are essentially creating is a template for a bean definition. This means that, just like a class, we can have multiple object instances created from a single template. These beans are managed by the Spring IoC (Inversion of Control) container. When a bean definition is created, Spring allows us to control not only the dependencies and configuration values injected into the object but also the scope of the objects created from the bean definition.

Bean scope refers to the lifecycle of an object and dictates how many instances of a bean will be created and how long they will exist. Typically, when a bean is requested using the getBean() method, the Spring Framework decides which bean instance should be returned based on the scope. However, Spring provides flexibility in configuring the scope, so we can control the number of instances and their lifecycle

Bean Scopes Supported by Spring

The Spring Framework supports the following five scopes

Scopes	Definition
singleton	This creates a single bean instance per Spring IoC container

Scopes	Definition
prototype	This creates a new bean instance each time it is requested
request	This creates a single bean instance per HTTP request. It is valid only in the context of a web application
session	This creates a single bean instance per HTTP session. It is valid only in the context of a web application
globalSession	This creates a single bean instance per global HTTP session. It is valid only in the context of a portal application

@Scope Annotation

The @Scope annotation is used to define the scope of a bean. By default, Spring creates a single instance of each bean declared in the IoC container. This default scope is singleton, meaning that the same bean instance is shared throughout the container and returned for all subsequent getBean() calls.

The @Scope annotation can be applied to a class or a factory method, depending on the scenario.

Syntax of @Scope Annotation

```
@Target(value={TYPE, METHOD})
@Retention(value=RUNTIME)
@Documented
public @interface Scope{
    String value() default "singleton";
```

```
}
```

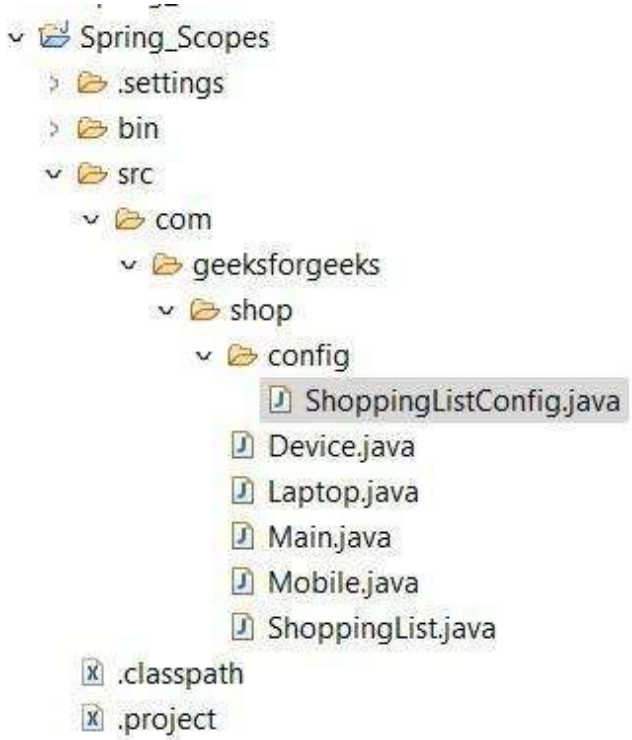
@Scope annotations can only be used on the concrete bean class (for annotated components) or the factory method (for @Bean methods). When used on the concrete bean class as a type-level annotation together with @Component, *@Scope* indicates the name of a scope to use for instances of the annotated type. When used on the factory method as a method-level annotation together with @Bean, *@Scope* indicates the name of a scope to use for the instance returned from the method.

How @Scope Works

Let's create a simple shopping application to demonstrate how bean scopes work. The application will allow customers to create separate wish lists for different products.

Project Structure:

The final project structure will be as follows,



Step 1: Creating the ShoppingList.java Class

We will first create a ShoppingList.java class, where we define the shopping list bean without explicitly specifying the scope. By default it uses the singleton scope.

```
package com.geeksforgeeks.shop;

import java.util.ArrayList;
import java.util.List;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
public class ShoppingList {

    private List<Device> items = new ArrayList<>();

    public void addItem(Device item) {
        items.add(item);
    }

    public List<Device> getItems() {
        return items;
    }

}
```

Here, we created the bean with @Component and we are not providing any scope explicitly. So, the default scope - Singleton will be applied.

Step 2: Creating the Device.java Class

Now, we will create products/devices in the shopping application. Create a Device.java bean class and provide variables, getter/setter methods as follows:

```
package com.geeksforgeeks.shop;

public class Device {

    private String name;
    private double price;

    public Device() {
```

```
    }

    public Device(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String toString() {
        return name + " " + price;
    }

}
```

Step 3: Creating Product Classes (Laptop.java and Mobile.java)

Now we are creating 2 product classes, **Mobile.java** and **Laptop.java** as follows:

Laptop.java:

```
package com.geeksforgeeks.shop;

public class Laptop extends Device{

    private boolean touchScreen;

    public Laptop() {
        super();
    }

    public Laptop(String name, double price) {
        super(name, price);
    }

    public boolean isTouchScreen() {
```

```
        return touchScreen;
    }

    public void setTouchScreen(boolean touchScreen) {
        this.touchScreen = touchScreen;
    }

}
```

Mobile.java:

```
package com.geeksforgeeks.shop;

public class Mobile extends Device{

    private int batteryCapacity;

    public Mobile() {
        super();
    }

    public Mobile(String name, double price) {
        super(name, price);
    }

    public int getBatteryCapacity() {
        return batteryCapacity;
    }

    public void setBatteryCapacity(int capacity) {
        this.batteryCapacity = capacity;
    }

}
```

Step 4: Declaring Device Beans in ShoppingListConfig.java

Then, we declare some device beans in a Java configuration file so that they can be added to the shopping list later. Create the **ShoppingListConfig.java** class as follows:

```
package com.geeksforgeeks.shop.config;

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

```
import com.geeksforgeeks.shop.Device;
import com.geeksforgeeks.shop.Laptop;
import com.geeksforgeeks.shop.Mobile;

@Configuration
@ComponentScan("com.geeksforgeeks.shop")
public class ShoppingListConfig {

    @Bean
    public Device lenovo() {
        Laptop d1 = new Laptop();
        d1.setName("LENOVO");
        d1.setPrice(65000);
        d1.setTouchScreen(true);
        return d1;
    }

    @Bean
    public Device dell() {
        Laptop d1 = new Laptop();
        d1.setName("DELL");
        d1.setPrice(57000);
        d1.setTouchScreen(false);
        return d1;
    }

    @Bean
    public Device moto() {
        Mobile d2 = new Mobile();
        d2.setName("MOTOROLA");
        d2.setPrice(40000);
        d2.setBatteryCapacity(4000);
        return d2;
    }

    @Bean
    public Device iQ() {
        Mobile d3 = new Mobile();
        d3.setName("iQOO");
        d3.setPrice(55000);
        d3.setBatteryCapacity(4700);
        return d3;
    }
}
```

Step 5: Testing the Shopping Application with Main.java

Main.java:

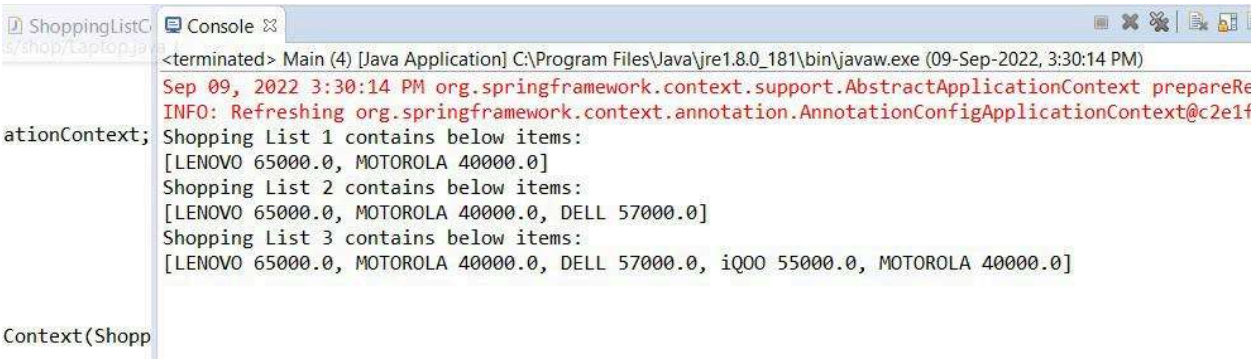
Now, we need to define a **Main.java** class to test the shopping cart by adding some products to it.

```
package com.geeksforgeeks.shop;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.geeksforgeeks.shop.config.ShoppingListConfig;

public class Main {

    public static void main(String[] args) throws Exception{

        ApplicationContext context = new
AnnotationConfigApplicationContext(ShoppingListConfig.class);

        Device lenovo = context.getBean("lenovo", Device.class);
        Device dell = context.getBean("dell", Device.class);
        Device moto = context.getBean("moto", Device.class);
        Device iQ = context.getBean("iQ", Device.class);


        ShoppingList list1 = context.getBean("shoppingList",
ShoppingList.class);
        list1.addItem(lenovo);
        list1.addItem(moto);
        System.out.println("Shopping List 1 contains below items:");
        System.out.println(list1.getItems());

        ShoppingList list2 = context.getBean("shoppingList",
ShoppingList.class);
        list2.addItem(dell);
        System.out.println("Shopping List 2 contains below items:");
        System.out.println(list2.getItems());

        ShoppingList list3 = context.getBean("shoppingList",
ShoppingList.class);
        list3.addItem(iQ);
        list3.addItem(moto);
        System.out.println("Shopping List 3 contains below items:");
        System.out.println(list3.getItems());

    }
}
```

Suppose a customer is trying to create separate wish lists for different products. Then, the first one gets a shopping list 1 by the getBean() method and adds two products. The second wish list gets a shopping list 2 by the getBean() method and adds another product. The third wish list gets a shopping list 3 by the getBean() method and adds two products. As a result of the preceding bean declaration, all three shopping lists get the

same bean instance. This is because, the default scope is Singleton, and every time you call the getBean() method, spring provides the same instance. Hence the output will be as follows.

Output with Singleton Scope:



Changing the Scope to prototype

In shopping applications, customers expect to get different lists to be created. To achieve this behavior, the scope of the **shoppingList.java** bean needs to be set to **prototype**. Then Spring framework creates a new bean instance for each getBean() method call. So, we need to add the '**@Scope("prototype")**' on the bean class as below.

Updated ShoppingList.java:

```
package com.geeksforgeeks.shop;

import java.util.ArrayList;
import java.util.List;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class ShoppingList {

    private List<Device> items = new ArrayList<>();

    public void addItem(Device item) {
        items.add(item);
    }

    public List<Device> getItems() {
        return items;
    }
```

```
}
```

Output with Prototype Scope:

Now if we rerun the Main.java class, we can see the three different shopping lists. The output will be as follows.

This way we can use the @Scope annotation to define the bean scope at the class level or at the method level based on our requirement.