Parameterised Beans

Introduction:

In Spring Framework, dependency injection is a core concept; constructor injection is one of the popular ways to inject dependency. If we want to inject a bean that takes specific parameters, we can do so in Spring Framework using the parameterised beans concept.

Both techniques have their benefits and drawbacks, and it is essential to understand the differences between them to choose the one that fits the specific requirements of an application. We will explore parameterised beans and constructor injection in detail and provide examples to illustrate their differences. We will also use a real-life example of an employee with two dependencies - salaries and a department - to demonstrate the practical application of these concepts. We will be using XML configuration to define beans and inject dependencies.

Constructor Injection:

Constructor injection is a popular technique used in Spring to inject dependencies into a class via its constructor. It provides a way to create and initialise objects with all the dependencies required to function correctly. Let us consider an example of an Employee class with two dependencies - Salary and Department - which are injected using constructor injection.

```
public class Employee {

    private final Salary salary;
    private final Department department;

    public Employee(Salary salary, Department department) {
        this.salary = salary;
        this.department = department;
    }

    public void displayDetails() {
        System.out.println("Salary: " + salary.getAmount());
        System.out.println("Department: " + department.getName());
    }
}
```

In this example, the Employee class has two dependencies - Salary and Department - that are injected using constructor injection. The constructor takes two arguments, the Salary and Department objects, and assigns them to the respective instance variables. The displayDetails() method can then access these instance variables to display the employee's details.

Parameterised Beans:

Parameterised beans are a specific implementation of constructor injection that involves creating beans with a constructor that takes one or more parameters. These parameters are typically used to customise the bean's behaviour in some way, such as providing configuration information or injecting dependencies that are known in runtime. In the case of the Employee example, we can create parameterised beans for the Salary and Department classes to provide additional information to the Employee class.

Employee Example:

```
```
public class Salary {

    private final double amount;

    public Salary(double amount) {
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```
```

```
```
public class Department {

    private final String name;

    public Department(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```
```

```
```

In these examples, we have created parameterised beans for the Salary and Department classes. Each bean has a constructor that takes a single parameter - salary amount and department name. The value of these parameters can be set at runtime to customise the behavior of the beans. For example, we can create a Salary bean with a value of 5000:

```
<bean id="salary" class="com.example.Salary">
    <constructor-arg value="5000.0"/>
</bean>
```

We can then inject this Salary bean into the Employee class using constructor injection, as follows:

```
<bean id="employee" class="com.example.Employee">
    <constructor-arg ref="salary"/>
    <constructor-arg ref="department"/>
</bean>
```

In this example, the Employee bean is created with the Salary bean injected via constructor injection. The value of the Salary bean is set to 5000.0, and the Department bean is injected at runtime.

Let us take o more examples to understand the concept. Consider a blog application as an example. A typical blog application might have a class called BlogPost that represents a blog post. The BlogPost class might have two dependencies - a User object representing the author of the blog post and a Category object representing the blog post's category.

To inject these dependencies using constructor injection, we could define a constructor in the BlogPost class that takes two arguments - a User object and a Category object - like this:

Blog Example

```java
// User Class
public class User {
    private String username;
    private String email;

    public User(String username, String email) {
        this.username = username;
        this.email = email;
    }
```

```
    // getters and setters
}

// Category Class
public class Category {
    private String name;

    public Category(String name) {
        this.name = name;
    }

    // getters and setters
}


// Blog Post Class
public class BlogPost {
    private User author;
    private Category category;

    public BlogPost(User author, Category category) {
        this.author = author;
        this.category = category;
    }

    // getters and setters
}
```

This example defines the User and Category classes with constructors that take the necessary arguments. We've also updated the BlogPost class to use the User and Category objects passed in via the constructor.

To use this constructor, we would define the User and Category beans first and then define the BlogPost bean, passing in the User and Category beans as constructor arguments like this:

```
<bean id="author" class="com.example.User">
    <constructor-arg value="johndoe"/>
    <constructor-arg value="johndoe@example.com"/>
</bean>

<bean id="category" class="com.example.Category">
    <constructor-arg value="Technology"/>
</bean>

<bean id="blogPost" class="com.example.BlogPost">
    <constructor-arg ref="author"/>
    <constructor-arg ref="category"/>
```

```
</bean>
```

In this example, we've defined a BlogPost bean with two constructor arguments - a User object and a Category object - and we've defined the values of these arguments directly in the XML configuration file. By using constructor injection or parameterised beans, we can easily inject dependencies into our classes and configure our application with ease.

Comparison:

Now that we have seen parameterised beans and constructor injection examples let us compare the two techniques based on their benefits and drawbacks.

1. **Flexibility:** Constructor injection is more flexible than parameterised beans because it can inject any type of dependency. On the other hand, Parameterised beans can only inject dependencies that can be instantiated with a constructor that takes parameters. However, parameterised beans offer more flexibility than regular beans as you can customise their behavior at runtime by passing different values for constructor parameters.

2. **Ease of use:** Constructor injection is generally easier than parameterised beans. You don't need to define beans with constructor arguments explicitly. You can specify the beans in the XML configuration file and let Spring handle the injection automatically. Parameterised beans require more work as you must define the beans explicitly and provide constructor arguments while creating the beans.

3. **Testability:** Constructor injection is more testable than parameterised beans because creating mock objects for dependencies is easier. Parameterised beans require creating a new bean for each test case, which can be cumbersome and time-consuming.

4. **Configuration management:** Parameterized beans provide better configuration management because you can specify the values of constructor parameters in the XML configuration file. This makes it easier to manage configuration changes and ensures that the same configuration is used across different bean instances.

Conclusion:

In conclusion, parameterised beans and constructor injection are valuable techniques for injecting dependencies in Spring Framework. Constructor injection is more flexible and easier to use, whereas parameterised beans provide more customisation and better configuration management.

When choosing between the two techniques, it is essential to consider the application's specific requirements and choose the best method. Using the real-life example of an *Employee class with dependencies on Salary and Department classes*, we have illustrated the practical application of these concepts and demonstrated how they could be implemented using XML configuration in Spring.