Spring Framework

Spring is a **lightweight framework**. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises **several modules** such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. Let's understand this with the following code:

```
class Employee{
Address address;
Employee(){
address=new Address();
}
}
```

In such case, there is dependency between the Employee and Address (**tight coupling**). In the Inversion of Control scenario, we do this something like this:

```
class Employee{
Address address;
Employee(Address address){
this.address=address;
}
}
```

Thus, IOC makes the code **loosely coupled**. In such case, there is **no need to modify** the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to **inject the dependency**. We provide metadata to the IOC container either by **XML file or annotation**.

Advantage of Dependency Injection

- makes the code loosely coupled so easy to maintain
- makes the code easy to test

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So, there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JDBC Template, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it saves a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Dependency Injection in Spring: -

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

Problems of Dependency Lookup: -

There are mainly two problems of dependency lookup.

- **Tight Coupling: -** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
- **Not easy for testing: -** This approach creates a lot of problems while testing the application especially in black box testing.

Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

```
class Employee{
Address address;

Employee(Address address){
this.address=address;
}
public void setAddress(Address address){
this.address=address;
}

}
```

In such case, instance of Address class is provided by external source such as XML file either by constructor or setter method.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

- **By Constructor**
- **By Setter method**

Design patterns are an essential part of software development. These solutions not only solve recurring problems but also help developers understand the design of a framework by recognizing common patterns.

The most common **design patterns** used in the Spring Framework:

- Singleton pattern
- Factory Method pattern
- Proxy pattern
- Template pattern

1. Singleton Pattern

The singleton pattern is a mechanism that ensures only one instance of an object exists per application. This pattern can be useful when managing shared resources or providing cross-cutting services, such as logging.

Spring's approach differs from the strict definition of a singleton since an application can have more than one Spring container. Therefore, multiple objects of the same class can exist in a single application if we have multiple containers.

For example, we can create two controllers within a single application context and inject a bean of the same type into each.

First, we create a BookRepository that manages our Book domain objects.

Next, we create LibraryController, which uses the BookRepository to return the number of books in the library:

```
@RestController
public class LibraryController {
    @Autowired
    private BookRepository repository;

    @GetMapping("/count")
    public Long findCount() {
        System.out.println(repository);
        return repository.count();
    }
}
@RestController
public class BookController {
     @Autowired
    private BookRepository repository;
 @GetMapping("/book/{id}")
    public Book findById(@PathVariable long id) {
        System.out.println(repository);
        return repository.findById(id).get();
    }
}
```

2. Factory Method Pattern

The factory method pattern entails a factory class with an abstract method for creating the desired object.

```
public interface BeanFactory {
    getBean(Class<T> requiredType);
    getBean(Class<T> requiredType, Object... args);
```

getBean(String name);

// ...

]

3. Proxy Pattern

Proxies are a handy tool in our digital world, and we use them very often outside of software (such as network proxies). In code, the proxy pattern is a technique that allows one object — the proxy — to control access to another object — the subject or service.

4. Template Method Pattern

In many frameworks, a significant portion of the code is boilerplate code.

For example, when executing a query on a database, the same series of steps must be completed:

i. Establish a connection
ii. Execute query
iii. Perform cleanup
iv. Close the connection

Spring - Bean: -

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

S.No.	Properties & Description
1	**Class** This attribute is mandatory and specifies the bean class to be used to create the bean.
2	**Name** This attribute specifies the bean identifier uniquely. In XML based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
3	**Scope** This attribute specifies the scope of the objects created from a particular bean definition.

4	**Constructor-arg** This is used to inject the dependencies.
5	**Properties** This is used to inject the dependencies.
6	**Autowiring mode** This is used to inject the dependencies.
7	**Lazy-initialization mode** A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.
8	**Initialization method** A callback to be called just after all necessary properties on the bean have been set by the container.
9	**Destruction method** A callback to be used when the container containing the bean is destroyed.

Spring - Bean Life Cycle

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Initialization callbacks: -

```
public class ExampleBean implements InitializingBean {
   public void afterPropertiesSet() {
      // do some initialization work
   }
```

```
}
```

Destruction callbacks: -

```
public class ExampleBean implements DisposableBean {
   public void destroy() {
      // do some destruction work
   }
}
```

Steps	Description
1	Create a project with a name *SpringExample* and create a packge *com.XYZ* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Create Java classes *HelloWorld* and *MainApp* under the *com.XYZ* package.
4	Create Beans configuration file *Beans.xml* under the **src** folder.
5	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

```
package com.XYZ;
public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
```

```
  public void init(){
    System.out.println("Bean is going through init.");
  }
  public void destroy() {
    System.out.println("Bean will destroy now.");
  }
}
```

Default initialization and destroy methods

If you have too many beans having initialization and/or destroy methods with the same name, you don't need to declare init-method and destroy-method on each individual bean. Instead, the framework provides the flexibility to configure such situation using default-init-method and default-destroy-method attributes on the <beans> element.

Bean Scopes

When you create a bean definition what you are actually creating is a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware ApplicationContext).

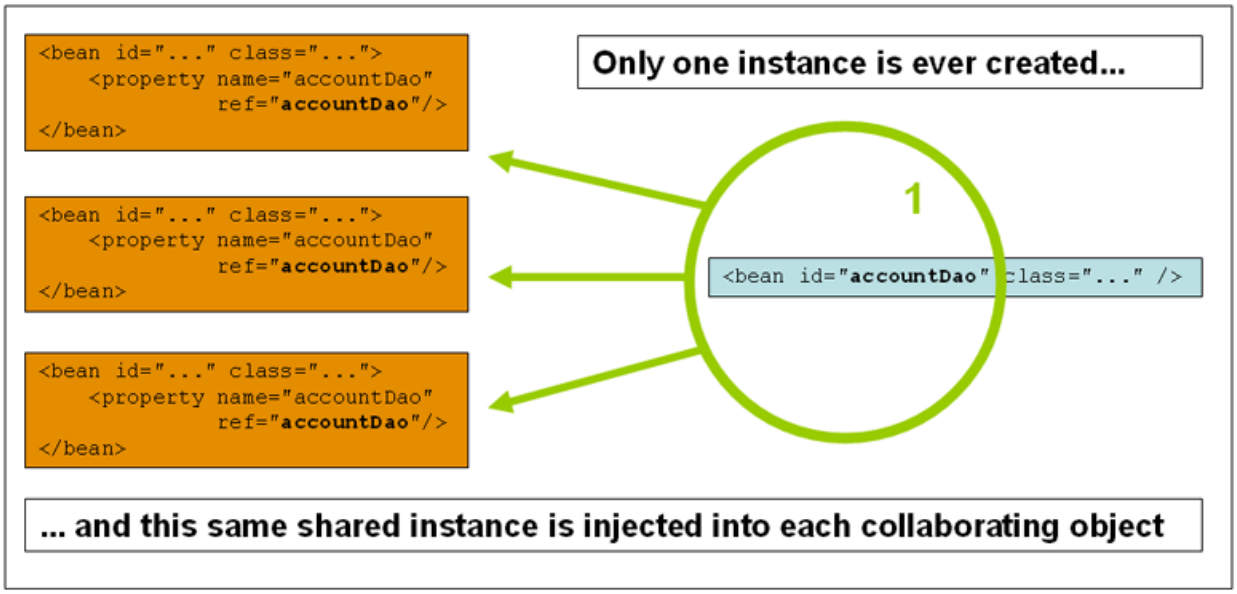
Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of a HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.

Scope	Description
global session	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically, only valid when used in a portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.

1. The Singleton Scope: -

When a bean is a singleton, only one shared instance of the bean will be managed, and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

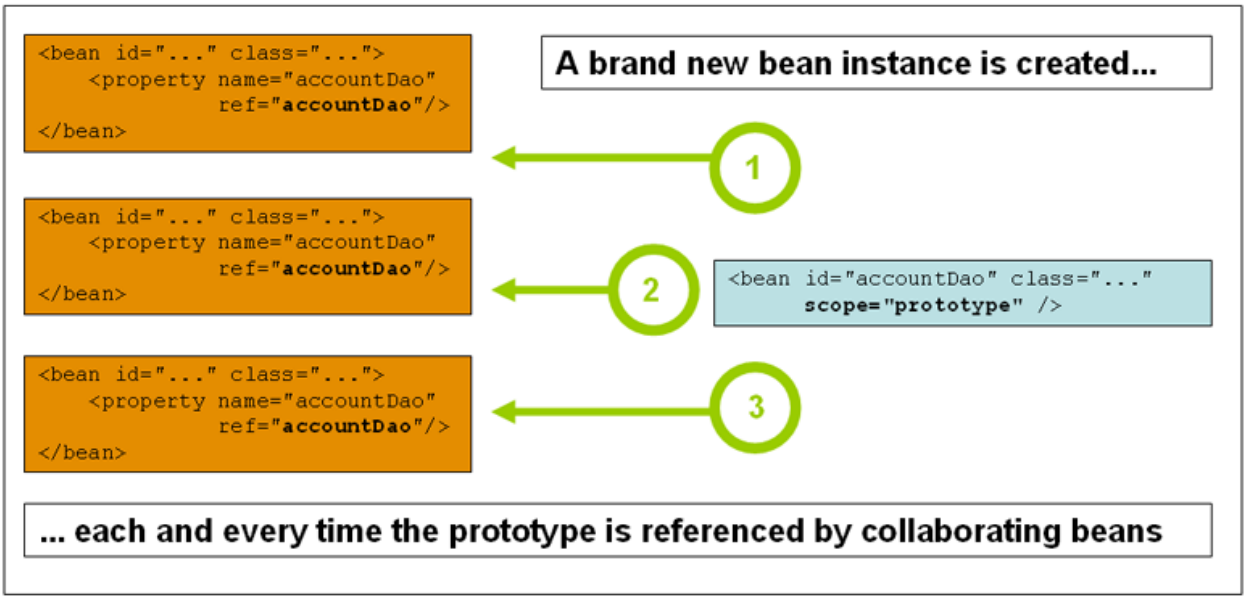
To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and all subsequent requests and references for that named bean will result in the cached object being returned.



2. The prototype scope: -

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic getBean() method call on the container). As a rule of thumb, you should use the prototype scope for all beans that are **stateful beans**, while the singleton scope should be used for **stateless beans**.

The following diagram illustrates the Spring prototype scope. Please note that a DAO would not typically be configured as a prototype, since a typical DAO would not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.



3. The request scope: -

Consider the following bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

With the above bean definition in place, the Spring container will create a brand-new instance of the LoginAction bean using the 'loginAction' bean definition for each and every HTTP request. That is, the 'loginAction' bean will be effectively scoped at the HTTP request level. You can change or dirty the internal state of the instance that is created as much as you want, safe in the knowledge that other requests that are also using instances created off the back of the same 'loginAction' bean definition will not be seeing these changes in state since they are particular to an individual request. When the request is finished processing, the bean that is scoped to the request will be discarded.

4. The session scope: -

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

With the above bean definition in place, the Spring container will create a brand-new instance of the UserPreferences bean using the 'userPreferences' bean definition for the lifetime of a single HTTP Session. In other words, the 'userPreferences' bean will be effectively scoped at the HTTP Session level. Just like request-scoped beans, you can change the internal state of the in-

stance that is created as much as you want, safe in the knowledge that other HTTP Session instances that are also using instances created off the back of the same 'userPreferences' bean definition will not be seeing these changes in state since they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session will also be discarded.