Spring

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.

Beans

Beans are the java objects which form the backbone of a Spring application and are managed by Spring IoC container. Other than being managed by the container, there is nothing special about a bean (in all other respects it's one of many objects in the application).

The Spring container is responsible for instantiating, configuring, and assembling the beans. The container gets its information on what objects to instantiate, configure, and manage by reading configuration metadata we define for the application.

Spring Container

Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name **Inversion Of Control**.

Bean factory

Beans are java objects that are configured at run-time by Spring IoC Container. BeanFactory represents a basic **IoC container** which is a parent interface of **ApplicationContext. BeanFactory** uses Beans and their dependencies metadata to create and configure them at run-time. BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file(XML) or the beans can be directly returned when required using Java Configuration. There are other types of configuration files like LDAP, RDMS, properties file, etc. BeanFactory **does not support Annotation-based configuration** whereas ApplicationContext does.

ApplicationContext

Spring IoC container is responsible for instantiating, wiring, configuring, and managing the entire life cycle of objects. BeanFactory and ApplicationContext represent the Spring IoC Containers. ApplicationContext is the **sub-interface of BeanFactory**. BeanFactory provides basic functionalities and is recommended to use for lightweight applications like mobile and applets. ApplicationContext provides basic features in addition to enterprise-specific functionalities which are as follows:

- Publishing events to registered listeners by resolving property files.
- Methods for accessing application components.
- Supports Internationalization.
- Loading File resources in a generic fashion.

It is because of these additional features, developers prefer to use ApplicationContext over BeanFactory.

Inversion of Control (IoC)

Singleton class

```
public final class Singleton {

   private static Singleton singletonObj;
   private String info = "Initial info class";

   private Singleton() {
   }

   public static Singleton getInstance() {
      if(singletonObj == null) {
         singletonObj= new Singleton();
      }

      return singletonObj;
   }

   // getters and setters
}
```

Bean Scopes in Spring

The spring framework provides five scopes for a bean. We can use three of them only in the context of web-aware **Spring ApplicationContext** and the rest of the two is available for both **IoC container and Spring-MVC container**. The following are the different scopes provided for a bean:

1. **Singleton:** Only one instance will be created for a single bean definition per Spring IoC container and the same object will be shared for each request made for that bean.
2. **Prototype:** A new instance will be created for a single bean definition every time a request is made for that bean.

Spring Dependency Injection

There are two types of Spring Dependency Injection. They are:

- **Setter Dependency Injection (SDI):** This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file For this, the property to be set with the SDI is declared under the <property> tag in the bean-config file.
- **Constructor Dependency Injection (CDI):** In this, the DI will be injected with the help of constructors. Now to set the DI as CDI in bean, it is done through the bean-configuration file For this, the property to be set with the CDI is declared under the <constructor-arg> tag in the bean-config file.

Setter Injection

Employee.java

```
public class Employee {

	private int empId;
	private String name;

	public int getEmpId() {
		return empId;
	}

	public void setEmpId(int empId) {
		this.empId = empId;
	}

	public String getName() {
		return name;
	}

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

	@Override
	public String toString() {
		return "Employee [empId=" + empId + ", name=" + name + "]";
	}

}
```

App.java

```
public class App
{
    public static void main( String[] args )
    {
       BeanFactory factory = new ClassPathXmlApplicationContext("config.xml");

			Employee obj1 = (Employee) factory.getBean("emp");
			System.out.println(obj1.toString());


    }
}
```

config.xml

```
<?xml version = "1.0" encoding="UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation = "http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">


        <bean id="emp" class="com.pmm.SpringDemo1.Employee">
                <property name="empId" value="1001"></property>
                <property name="name" value="Arya"></property>

        </bean>


</beans>
```

Constructor Injection

Employee.java

```
public class Employee {

        private int empId;
        private String name;


        public Employee(int empId, String name) {

                this.empId = empId;
                this.name = name;
        }


        @Override
        public String toString() {
                return "Employee [empId=" + empId + ", name=" + name + "]";
        }

}
```

App.java

```
public class App
{
   public static void main( String[] args )
   {
      BeanFactory factory = new ClassPathXmlApplicationContext("config.xml");

                Employee obj1 = (Employee) factory.getBean("emp");
                System.out.println(obj1.toString());


   }
}
```

config.xml

```
<?xml version = "1.0" encoding="UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation = "http://www.springframework.org/schema/beans
          https://www.springframework.org/schema/beans/spring-beans.xsd">


         <bean id="emp" class="com.pmm.SpringDemo1.Employee">
                <constructor-arg name="empId" value="1001"></constructor-arg>
                <constructor-arg name="name" value="Bala"></constructor-arg>

         </bean>


</beans>
```

Autowiring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

There are many autowiring modes:

No.	Mode	Description
1)	no	It is the default autowiring mode. It means no autowiring bydefault.
2)	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3)	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4)	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
5)	autodetect	It is deprecated since Spring 3.

Employee.java

```
public class Employee {

	private int empId;
	private String name;
	private Computer comp;

	public Computer getComp() {
		return comp;
	}


	public void setComp(Computer comp) {
		this.comp = comp;
	}
```

```
	public Employee(int empId, String name) {

		this.empId = empId;
		this.name = name;
	}

	void empStatus()
	{
		comp.working();
	}
	@Override
	public String toString() {
		return "Employee [empId=" + empId + ", name=" + name + "]";
	}

}
```

App.java

```
public class App
{
   public static void main( String[] args )
   {
      BeanFactory factory = new ClassPathXmlApplicationContext("config.xml");

		Employee obj1 = (Employee) factory.getBean("emp");
		//System.out.println(obj1.toString());
     obj1.empStatus();

   }
}
```

config.xml

```
<?xml version = "1.0" encoding="UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation = "http://www.springframework.org/schema/beans
	https://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
	<bean id="emp" class="com.pmm.SpringDemo1.Employee" autowire="byName">
		<constructor-arg name="empId" value="1001"></constructor-arg>
		<constructor-arg name="name" value="Bala"></constructor-arg>
		<property name="comp" ref="lap"></property>
	</bean>

	<bean id="lap" class="com.pmm.SpringDemo1.Laptop"></bean>

	<bean id="desk" class="com.pmm.SpringDemo1.Desktop"></bean>


</beans>
```