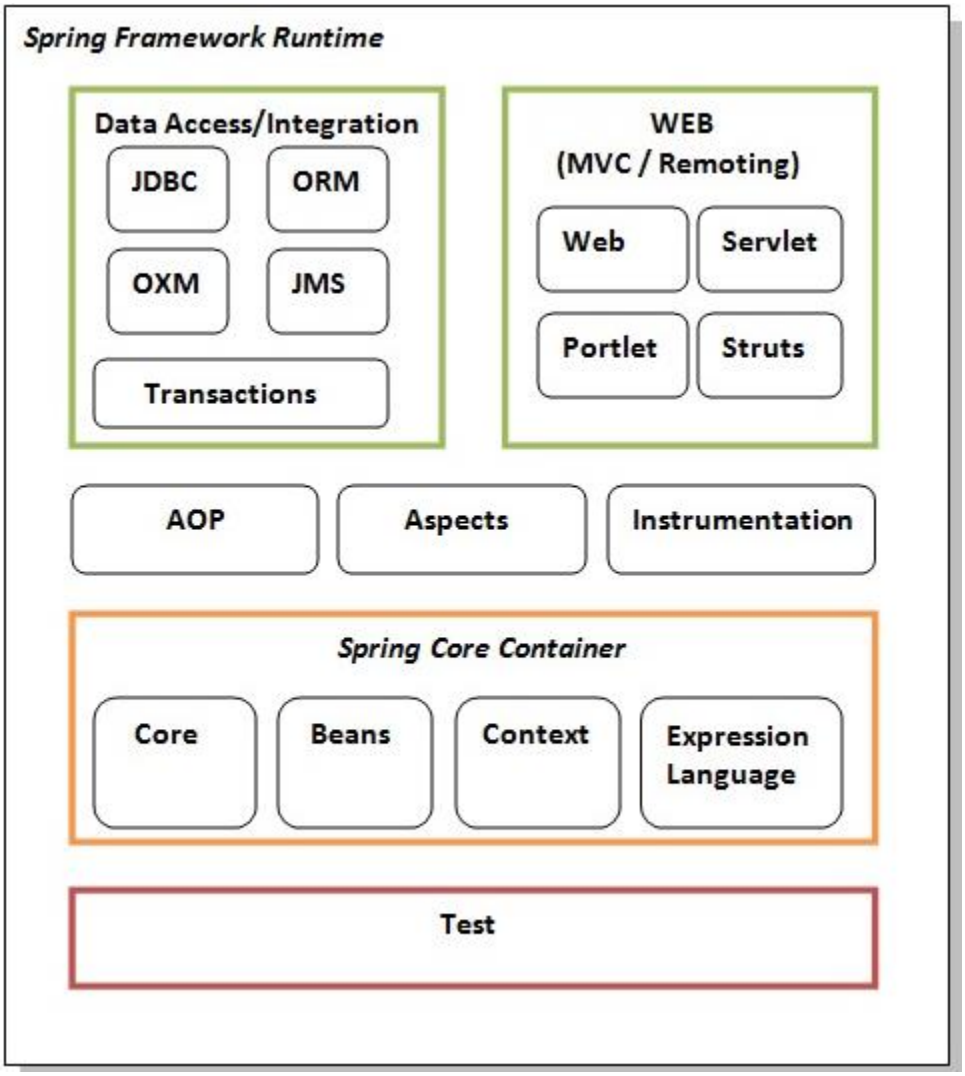
Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

**Spring Framework Runtime**




# Inversion Of Control (IOC) and Dependency Injection

These are the design patterns that are used to remove dependency from the programming code.

```
1. class Employee{
2. Address address;
```

```
3. Employee(){
4. address=new Address();
5. }
6. }
```

In the Inversion of Control scenario, we do this something like this:

```
1. class Employee{
2. Address address;
3. Employee(Address address){
4. this.address=address;
5. }
6. }
```

## Steps to create spring application

1. Create POJO (Plain old java object)

```
public class Student {
private String name;

public String getName() {
   return name;
}

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

public void displayInfo(){
   System.out.println("Hello: "+name);
}
}
```

2. Create XML mapping file (applicationContext.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"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
                http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="studentbean" class="com.javatpoint.Student">
    <property name="name" value="Vimal Jaiswal"></property>
    </bean>

    </beans>
```

3. Create test class

```
    public class Test {
    public static void main(String[] args) {
      Resource resource=new ClassPathResource("applicationContext.xml");
       BeanFactory factory=new XmlBeanFactory(resource);

       Student student=(Student)factory.getBean("studentbean");
       student.displayInfo();
    }
    }
```

The **Resource** object represents the information of applicationContext.xml file. The Resource is the interface and the **ClassPathResource** is the implementation class of the Reource interface. The **BeanFactory** is responsible to return the bean.
The **XmlBeanFactory** is the implementation class of the BeanFactory. There are many methods in the BeanFactory interface. One method is **getBean()**, which returns the object of the associated class.

# IoC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

- to instantiate the application class
- to configure the object
- to assemble the dependencies between the objects

## 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.

# Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

**By Consructor**

```
<bean id="e" class="com.javatpoint.Employee">
        <constructor-arg value="10" type="int" ></constructor-arg>
        <constructor-arg value="Sonoo"></constructor-arg>
</bean>
```

**By Setter**

```
<bean id="obj" class="com.javatpoint.Employee">
<property name="id">
<value>20</value>
</property>
<property name="name">
<value>Arun</value>
</property>
<property name="city">
<value>ghaziabad</value>
</property>

</bean>
```

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.

# Autowiring in Spring

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.

| | | |
|---|---|---|
| 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. |

**B.java**

This class contains a constructor and method only.

```
1. package org.sssit;
2. public class B {
3. B(){System.out.println("b is created");}
4. void print(){System.out.println("hello b");}
5. }
```

**A.java**

This class contains reference of B class and constructor and method.

```
1. package org.sssit;
2. public class A {
3. B b;
4. A(){System.out.println("a is created");}
5. public B getB() {
6.     return b;
7. }
8. public void setB(B b) {
9.     this.b = b;
10. }
11. void print(){System.out.println("hello a");}
12. void display(){
13.     print();
```

```
14.     b.print();
15. }
16. }
```

**applicationContext.xml**

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans
3.      xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xmlns:p="http://www.springframework.org/schema/p"
6.      xsi:schemaLocation="http://www.springframework.org/schema/beans
7.  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
8.
9.  <bean id="b" class="org.sssit.B"></bean>
10. <bean id="a" class="org.sssit.A" autowire="byName"></bean>
11.
12. </beans>
```

**By Type**

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byType"></bean>
```

# Spring AOP Tutorial

**Aspect Oriented Programming** (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class

**Understanding Scenario** I have to maintain log and send notification after calling methods that starts from m.

**Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

**Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

## *Join point*

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

---

## *Advice*

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

- **Before Advice**: it executes before a join point.
- **After Returning Advice**: it executes after a joint point completes normally.
- **After Throwing Advice**: it executes if method exits by throwing an exception.
- **After (finally) Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return.
- **Around Advice**: It executes before and after a join point.

---

## *Pointcut*

It is an expression language of AOP that matches join points.