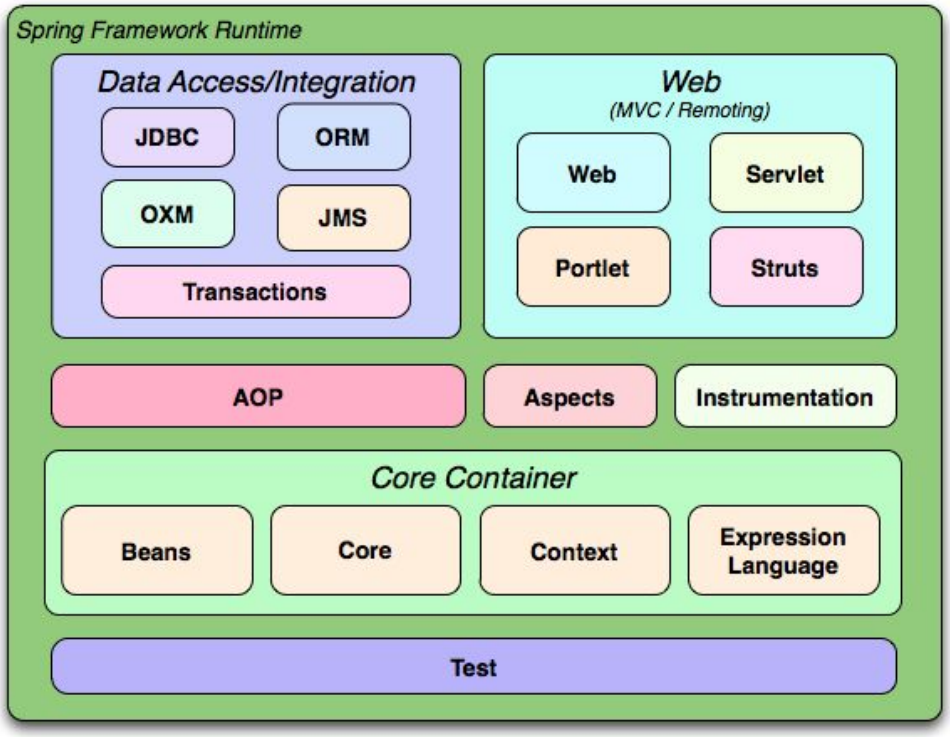
SPRING FRAMEWORK

Why do we need Spring Framework?

- helps us to "wire" different components together, swap them with alternatives or combine them in order.
- dependency injection - an outside force to supply them with what they need
- good way to structure your application into layers



Framework of Frameworks

Spring Core :

- Core part of Spring and consists
- Core, Beans, Context and Expression Language.

Core - fundamental module of the framework
- container for IOC (Inversion of control) and Dependency Injection

Beans - Type of Objects

Context - How to access the objects
- it is a medium to access any objects defined and configured

Expression Language - GUI based
- querying and manipulating an object

Data Access :

- DB Access
- JDBC, ORM, OXM (object into Xml & vice versa)
- JMS – Java messaging service – features for consuming & producing messaging
- Transaction – managing special interfaces & POJOs.

Web :

- Web - web-oriented application context
- Web Servlet, Web Struts, Web Portlets

Others :

- There are few other important modules in Spring, which plays vital role in the framework to use all the features in various scenario. The modules are AOP, Aspect, Instrumentation, and Test.

AOP : It contains API for AOP aspect-oriented programming implementations on various layers. You can introduce new functionalities into existing code without modifying it.

Aspectj : The separate Aspects module provides integration with AspectJ.

Instrumentation : The Instrumentation module provides class instrumentation support and classloader implementations to be used in certain application servers.

Test : Testing related APIs

Types of Spring Bean Scopes

In the spring bean configurations, bean attribute called 'scope' defines what kind of object has to created and returned. There are 5 types of bean scopes available, they are:

1) singleton: Returns a single bean instance per Spring IoC container. (default)

2) prototype: Returns a new bean instance each time when requested.

3) request: Returns a single instance for every HTTP request call.

4) session: Returns a single instance for every HTTP session.

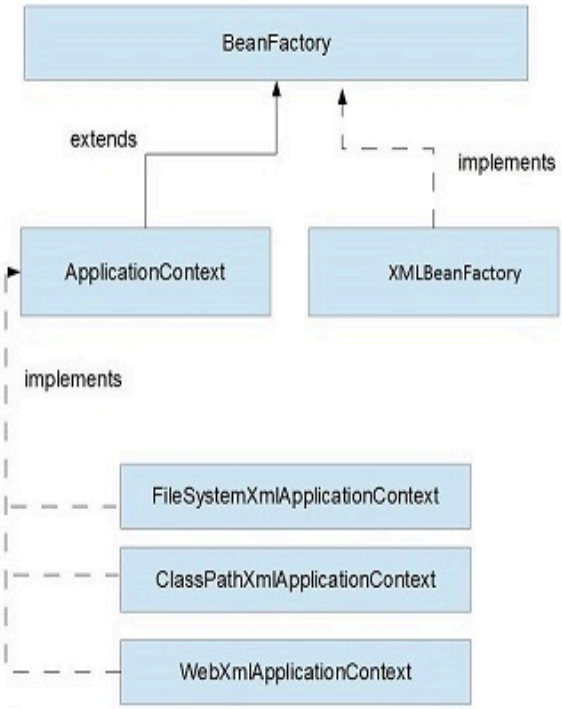
5) global session: global session scope is equal as session scope on portlet-based web applications.

```
<bean id="clrBean" class="com.java2novice.bean.ColorBean"
scope="prototype"/>
```

Desktop apps - 1, 2
Web apps - 1, 2, 3 ,4
Portlet apps - 1,2,3,4,5

BeanFactory vs ApplicationContext

	XMLBeanFactory	ApplicationContext
Annotation support	No	Yes
BeanPostProcessor Registration	Manual	Automatic
implimentation	XMLBeanFactory	ClassPath/FileSystem/WebX
internationalization	No	Yes
Enterprise services	No	Yes
ApplicationEvent publication	No	Yes



1. FileSystemXmlApplicationContext: Beans loaded through the full path.
2. ClassPathXmlApplicationContext: Beans loaded through the CLASSPATH
3. WebXmlApplicationContext: Beans loaded through the web application context.

Spring.xml

```
<bean id="employee" class="com.java.Spring.SpringDemo.Employee">
        <property name="empId" value="1" />
        <property name="name" value="Abc" />
        <property name="sal" value="10000" />
        <property name="ad" ref="address" />
</bean>
```

```
<bean id="address" class="com.java.Spring.SpringDemo.Address">
            <property name="addID" value="1" />
            <property name="pincode" value="412207" />
            <property name="city" value="Pune" />
</bean>
```

BeanFactory example [Lazy Initialisation]

```
Resource resource = new FileSystemResource(
                        new File("//Users//adnaan//eclipse-workspace-
JAVA//SpringDemo//src//main//java//spring.xml"));
            BeanFactory beanFactory = new XmlBeanFactory(resource);
            Employee e = (Employee) beanFactory.getBean("employee");
            System.out.println(e);
```

ApplicationContext example [Eager Initialisation]

```
ApplicationContext context = new FileSystemXmlApplicationContext(
                        "//Users//adnaan//eclipse-workspace-
JAVA//SpringDemo//src//main//java//spring.xml");
            Employee e = (Employee) context.getBean("employee");
            System.out.println(e);
```

Bean Factory:

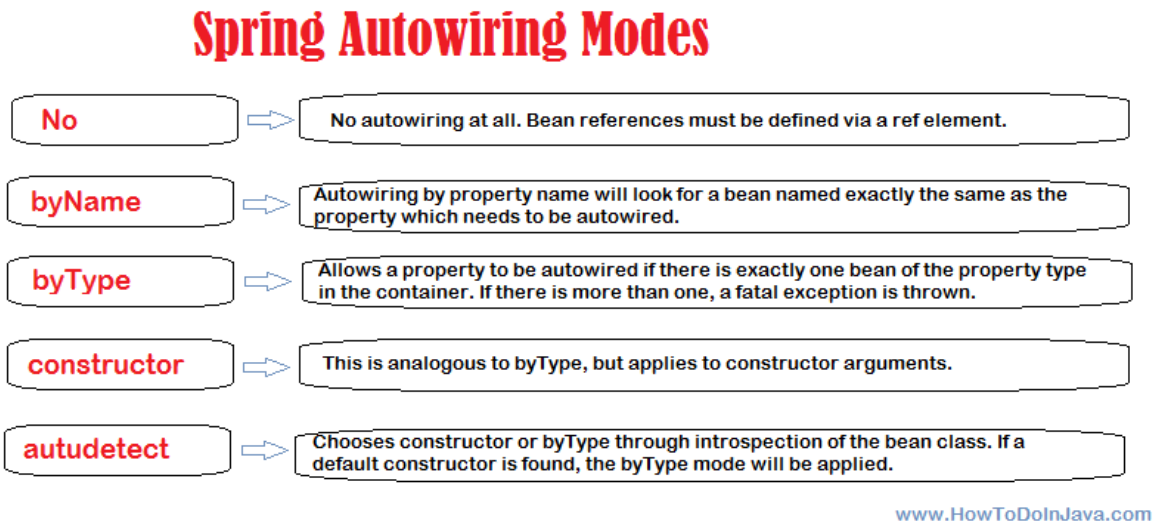
-- Lazy loading
-- Lightweight container (better for mobile application)
-- Doesn’t support AOP features.
-- Not supported international language.
-- Doesn’t support any text msg.

What are the types of Dependency Injection? Differentiate.

Setter Injection
Constructor Injection (for mandatory injection)

Setter Injection	Constructor Injection
1. In Setter Injection, partial injection of dependencies can be possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives	**1.** In constructor injection, partial injection of dependencies cannot possible, because for calling constructor we must pass all the arguments right, if not so we may get error
2. Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property [i already told regarding this, hope you remember]	**2.** But, constructor injection cannot overrides the setter injected values
3. If we have more dependencies for example 15 to 20 are there in our bean class then, in this case setter injection is not recommended as we need to write almost 20 setters right, bean length will increase.	**3.** In this case, Constructor injection is highly recommended, as we can inject all the dependencies with in 3 to 4 lines [i mean, by calling one constructor]
4. Setter injection makes bean class object as mutable [We can change]	**4.** Constructor injection makes bean class object as immutable [We cannot change]

Autowiring



Autowiring Example

```
<!-- Using Setter Getters -->
        <!-- <bean id="empObj" class="com.scp.spring.SpringDemo.Employee" scope =
"prototype">
                <property name="employeeId" value="1" />
                <property name="emplyeeName" value="Adnaan" />
                <property name="address" ref="addressObj" />
        </bean>

        <bean id="addressObj" class="com.scp.spring.SpringDemo.Address">
                <property name="city" value="Pune" />
                <property name="pincode" value="414005" />
        </bean> -->

<!-- Using Default Construtor -->
        <!-- <bean id="empObj" class="com.scp.spring.SpringDemo.Employee" >
        <constructor-arg ref="addressObj"> </constructor-arg>
        </bean>

        <bean id="addressObj" class="com.scp.spring.SpringDemo.Address" >
        </bean>
 -->
 <!-- Using Parameterized Construtor -->

<!--    <bean id="empObj" class="com.scp.spring.SpringDemo.Employee" >
        <constructor-arg name="employeeId" value="1"> </constructor-arg>
        <constructor-arg name="emplyeeName" value="Adnaan">  </constructor-arg>
```

```
        <constructor-arg name="address" ref="addressObj"> </constructor-arg>
        </bean>

        <bean id="addressObj" class="com.scp.spring.SpringDemo.Address" >
        <constructor-arg name="city" value="Pune"> </constructor-arg>
        <constructor-arg name="pincode" value="414005">  </constructor-arg>
        </bean>
 -->
        <!-- Using Autowiring -->
        <bean id="empObj" class="com.scp.spring.SpringDemo.Employee" >
                <property name="employeeId" value="1" />
                <property name="emplyeeName" value="Adnaan" />
                <property name="address" ref="addressObj" />
        </bean>

        <bean id="empObj1" class="com.scp.spring.SpringDemo.Employee" >
                <property name="employeeId" value="1" />
                <property name="emplyeeName" value="Adnaan" />
                <property name="address" ref="addressObj1" />
        </bean>

        <bean id="addressObj" class="com.scp.spring.SpringDemo.Address">
                <property name="city" value="Pune" />
                <property name="pincode" value="414005" />
        </bean>

        <bean id="addressObj1" class="com.scp.spring.SpringDemo.Address">
                <property name="city" value="Ahmednagar" />
                <property name="pincode" value="414010" />
        </bean>
</beans>
```

Circular dependency & how to resolve it

--It happens when a bean A depends on another bean B, and the bean B depends on the bean A as well
Bean A → Bean B → Bean A

By Using Constructor:

--Spring cannot decide which of the beans should be created first, since they depend on one another

Exception raised: *BeanCurrentlyInCreationException* while loading context.

By using setter Base:

-- if you change the ways your beans are wired to use setter injection (or field injection) instead of constructor injection – that does address the problem. This way Spring creates the beans, but the dependencies are not injected until they are needed.

By using @Lazy

```
public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired

    public CircularDependencyA(@Lazy CircularDependencyB circB) {
        this.circB = circB;
```

By making Bean to implement ApplicationContextAware interface

If one of the beans implements ApplicationContextAware, the bean has access to Spring context and can extract the other bean from there.

How to resolve constructor ambiguities in spring?

```
public Employee(String name, String designation) {
    this.name = name;
    this.designation = designation;
  }

  public Employee(float salary, String name) {
    this.salary = salary;
    this.name = name;
```

```
<bean id="employee" class="com.jwt.spring.Employee">
    <constructor-arg value="10000" />                //by default String – put in name
    <constructor-arg value="Mukesh" />
  </bean>
```

Name: 10000
Designation: Mukesh
Salary: 0.0

--To avoid such ambiguities, it is always better to make the constructor-arg injection as clear as possible. The use of type is adviced.

--Usage of 'type' attribute along with 'constructor-arg' tag avoids ambiguity in the constructor being called

```
bean id="employee" class="com.jwt.spring.Employee">
    <constructor-arg type="float" value="10000" />
    <constructor-arg type="java.lang.String" value="Mukesh" />
  </bean>
```

Name: Mukesh
Designation: null
Salary: 10000.0

```
public Employee(String name, String designation) {
      this.name = name;
      this.designation = designation;
      System.out.println("1st constructor called");
   }

   public Employee(String designation, float salary) {
      this.designation = designation;
      this.salary = salary;
      System.out.println("2nd constructor called");
   }

   public Employee(float salary, String name) {
      this.salary = salary;
      this.name = name;
      System.out.println("3rd constructor called");
   }
```

Name: null
Designation: Mukesh
Salary: 10000.0

--'index' attribute resolves the ambiguity in this case

```
 <bean id="employee" class="com.jwt.spring.Employee">
     <constructor-arg type="float" index="0" value="10000" />
     <constructor-arg type="java.lang.String" index="1"
        value="Mukesh" />
  </bean>
```

3rd constructor called
Name: Mukesh
Designation: null
Salary: 10000.0