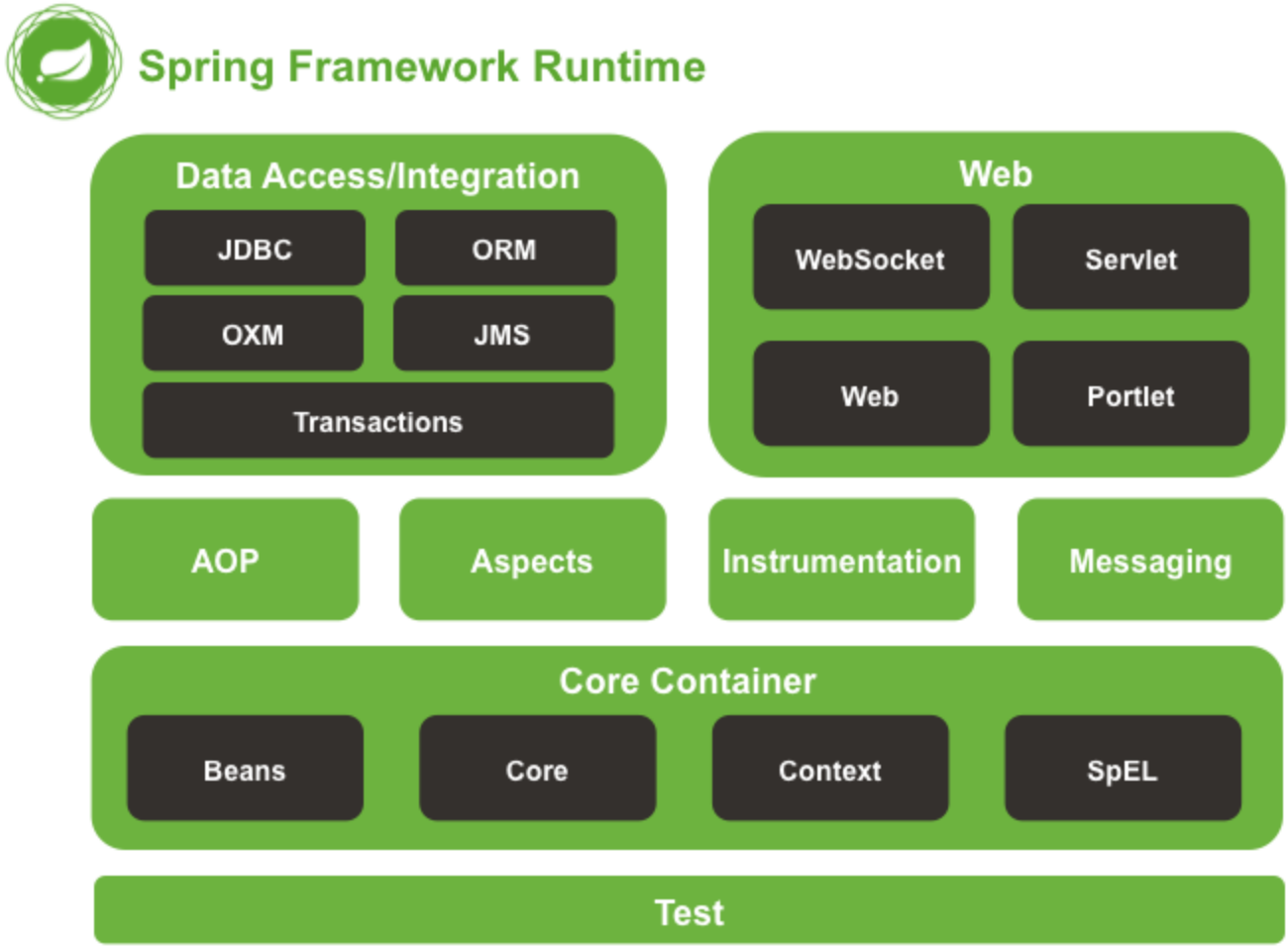
Spring Core Interview Questions

1) What is Spring Framework? What are it's main modules?

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure part so you can focus on your application part. Inside itself, Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s) without worrying too much about how they work in backend.

At present, Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test, as shown in the following diagram.



Read More : Spring Framework Tutorials

2) What are the benefits of using Spring Framework?

Following is the list of few of the great benefits of using Spring Framework --

- With the **Dependency Injection(DI)** approach, dependencies are explicit and evident in constructor or JavaBean properties.

- IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.
- **Testing an application** written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.
- Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

3) What is Inversion of Control (IoC) and Dependency Injection?

In software engineering, **inversion of control** (IoC) is a programming technique in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. In traditional programming, the flow of the business logic is determined by objects that are statically assigned to one another. With inversion of control, the flow depends on the object graph that is instantiated by the assembler and is made possible by object interactions being defined through abstractions. The binding process is achieved through "dependency injection". Inversion of control is a design paradigm with the goal of giving more control to the targeted components of your application, the ones that are actually doing the work.

Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism

is needed in order to activate the components providing the specific functionality. Otherwise how will the framework know which components to create if it is no longer in control?

In Java, dependency injection may happen through 3 ways:

1. A constructor injection
2. A setter injection
3. An interface injection

4) Explain IoC in Spring Framework?

The `org.springframework.beans` and `org.springframework.context` packages provide the basis for the Spring Framework's IoC container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing objects of any nature. The `ApplicationContext` interface builds on top of the `BeanFactory` (it is a sub-interface) and adds other functionality such as easier integration with **Spring's AOP features**, **message resource handling** (for use in internationalization), event propagation, and application-layer specific contexts such as the `WebApplicationContext`for use in web applications.
The `org.springframework.beans.factory.BeanFactory` is the actual representation of the Spring IoC container that is responsible for containing and otherwise managing the aforementioned beans. The BeanFactory interface is the central IoC container interface in Spring.

5) Difference between BeanFactory and ApplicationContext?

A `BeanFactory` is like a factory class that contains a collection of beans.
The `BeanFactory` holds Bean Definitions of multiple beans within itself and then instantiates the bean whenever asked for by clients. `BeanFactory` is able to create associations between collaborating objects as they are instantiated. This removes the

burden of configuration from bean itself and the beans client. `BeanFactory` also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.
On the surface, an application context is same as a bean factory.Both load bean definitions, wire beans together, and dispense beans upon request. But it also provides:

1. A means for resolving text messages, including support for internationalization.
2. A generic way to load file resources.
3. Events to beans that are registered as listeners.

The three commonly used implementations of `ApplicationContext` are:

1. **ClassPathXmlApplicationContext** : It Loads context definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code.
```
ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```
2. `FileSystemXmlApplicationContext` : It loads context definition from an XML file in the filesystem. The application context is loaded from the file system by using the code.
```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```
3. `XmlWebApplicationContext` : It loads context definition from an XML file contained within a web application.

6) In how many ways, you can configure Spring into our application?

You can configure spring into your application in 3 ways:

1. XML Based Configuration
2. Annotation-based configuration
3. Java-based configuration

7) What is Spring XML-Based Configuration?

In Spring framework, dependencies and the services needed by beans are specified in configuration files, which are typically in an XML format. These configuration files usually start with `<beans>` tag and contain a lot of bean definitions AND application specific configuration options.
The main goal of Spring XML Configuration is to have all the Spring components configured by using xml files.
This means that there will not be present any other type of Spring Configuration (like annotations or configuration via Java classes).

A Spring XML Configuration uses Spring namespaces to make available the sets of XML tags used in the configuration; the main Spring namespaces are: *context, beans, jdbc, tx, aop, mvc, aso*.

```
<beans>


    <!-- JSON Support -->

    <bean name="viewResolver"

class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

    <bean name="jsonTemplate"

class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>


    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate"/>


</beans>
```

And simplest `web.xml` file to make your application load configuration file and configure the runtime components for you is below where you configure only **DispatcherServlet**.

```
<web-app>

  <display-name>Archetype Created Web Application</display-name>
```

```
  <servlet>
        <servlet-name>spring</servlet-name>
            <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

8) What is Spring Java-Based Configuration?

The central artifacts in Spring's new Java-configuration support are `@Configuration` annotated classes and `@Bean`annotated methods.
The `@Bean` annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container. `@Bean` annotation plays the same role as the `<bean/>` element.
Annotating a class with `@Configuration` indicates that its primary purpose is as a source of bean definitions. Furthermore,`@Configuration` classes allow inter-bean dependencies to be defined by simply calling other `@Bean` methods in the same class. The simplest possible `@Configuration` class would read as follows:

```
@Configuration

public class AppConfig

{
```

```
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

The equivalent XML configuration for above java config would be:

```
<beans>
    <bean id="myService" class="com.howtodoinjava.services.MyServiceImpl"/>
</beans>
```

To instantiate such config, you will need the help of `AnnotationConfigApplicationContext` class.

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

To enable component scanning, just annotate your `@Configuration` class as follows:

```
@Configuration
@ComponentScan(basePackages = "com.howtodoinjava")
public class AppConfig  {
    ...
}
```

In the example above, the com.acme package will be scanned, looking for any `@Component` annotated classes, and those classes will be registered as Spring bean definitions within the container.

If you are using above configuration in a web application then you will be using `AnnotationConfigWebApplicationContext` class. This implementation may be used when configuring the Spring `ContextLoaderListener` servlet listener, Spring MVC `DispatcherServlet` etc.

```
<web-app>
    <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.support.
AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited
        fully-qualified @Configuration classes. Fully-qualified packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.howtodoinjava.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual
using ContextLoaderListener -->
    <listener>
        <listener-class>
org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
```

```
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Configure DispatcherServlet to use
AnnotationConfigWebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.web.context.support.
AnnotationConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or
more comma- or space-delimited
            and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.howtodoinjava.web.MvcConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /app/* to the dispatcher servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
```

```
</web-app>
```

9) What is Spring Annotation-based Configuration?

Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration. Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider to have following configuration file in case you want to use any annotation in your Spring application.

```
<beans>


   <context:annotation-config/>

   <!-- bean definitions go here -->


</beans>
```

Once `<context:annotation-config/>` is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.
Few important annotations which you will be using in this type of configuration are :

1. **@Required** : The @Required annotation applies to bean property setter methods.
2. **@Autowired** : The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.

3. **@Qualifier** : The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
4. **JSR-250 Annotations** : Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

10) Explain Spring Bean lifecycle?

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.

Spring bean factory is responsible for managing the life cycle of beans created through spring container. The life cycle of beans consist of **call back methods which can be categorized broadly in two groups**:

1. Post initialization call back methods
2. Pre destruction call back methods

Spring framework provides following **4 ways for controlling life cycle events** of bean:

- InitializingBean and DisposableBean callback interfaces
- Other Aware interfaces for specific behavior
- Custom init() and destroy() methods in bean configuration file
- @PostConstruct and @PreDestroy annotations

For example, `customInit()` and `customDestroy()` methods are example of life cycle method.

```
<beans>

    <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"

            init-method="customInit" destroy-method="customDestroy"></bean>

</beans>
```

Read More: Spring Bean Life Cycle

11) What are different Spring Bean Scopes?

The beans in spring container can be created in **five scopes**. All the scope names are self-explanatory but lets make them clear so that there will not be any doubt.

1. **singleton**: This bean scope is default and it enforces the container to have only one instance per spring container irrespective of how much time you request for its instance. This singleton behavior is maintained by bean factory itself.
2. **prototype**: This bean scope just reverses the behavior of singleton scope and produces a new instance each and every time a bean is requested.
3. **request**: With this bean scope, a new bean instance will be created for each web request made by client. As soon as request completes, bean will be out of scope and garbage collected.
4. **session**: Just like request scope, this ensures one instance of bean per user session. As soon as user ends its session, bean is out of scope.
5. **global-session**: global-session is something which is connected to Portlet applications. When your application works in Portlet container it is built of some amount of portlets. Each portlet has its own session, but if your want to store variables global for all portlets in your application than you should store them in global-session. This scope doesn't have any special effect different from session scope in Servlet based applications.

Read More : Spring Bean Scopes

12) What are inner beans in Spring?

In Spring framework, whenever a bean is used for only one particular property, it's advise to declare it as an inner bean. And the inner bean is supported both in setter injection '**property**' and constructor injection '**constructor-arg**'.
For example, let's say we one `Customer` class having reference of `Person` class. In our application, we will be creating only one instance of `Person` class, and use it inside `Customer`.

```
public class Customer

{

    private Person person;
```

```
    //Setters and Getters
}
public class Person
{
    private String name;
    private String address;
    private int age;

    //Setters and Getters
}
```

Now inner bean declaration will look like this:

```
<bean id="CustomerBean" class="com.howtodoinjava.common.Customer">
    <property name="person">
        <!-- This is inner bean -->
        <bean class="com.howtodoinjava.common.Person">
            <property name="name" value="lokesh" />
            <property name="address" value="India" />
            <property name="age" value="34" />
        </bean>
    </property>
</bean>
```

13) Are Singleton beans thread safe in Spring Framework?

Spring framework does not do anything under the hood concerning the multi-threaded behavior of a **singleton** bean. It is the developer's responsibility to deal with concurrency issue and **thread safety** of the singleton bean.
While practically, most spring beans have no mutable state (e.g. Service and DAO clases), and as such are trivially thread safe. But if your bean has mutable state (e.g. View Model Objects), so you need to ensure thread safety. The most easy and obvious solution for this problem is to change bean scope of mutable beans from "**singleton**" to "**prototype**".

14) How can you inject a Java Collection in Spring? Give example?

Spring offers four types of collection configuration elements which are as follows:

<**list**> : This helps in wiring ie injecting a list of values, allowing duplicates.
<**set**> : This helps in wiring a set of values but without any duplicates.
<**map**> : This can be used to inject a collection of name-value pairs where name and value can be of any type.
<**props**> : This can be used to inject a collection of name-value pairs where the name and value are both Strings.
Let's see example of each type.

```
<beans>


   <!-- Definition for javaCollection -->

   <bean id="javaCollection" class="com.howtodoinjava.JavaCollection">
```

```
 <!-- java.util.List -->
 <property name="customList">
   <list>
      <value>INDIA</value>
      <value>Pakistan</value>
      <value>USA</value>
      <value>UK</value>
   </list>
 </property>

<!-- java.util.Set -->
<property name="customSet">
   <set>
      <value>INDIA</value>
      <value>Pakistan</value>
      <value>USA</value>
      <value>UK</value>
   </set>
 </property>

<!-- java.util.Map -->
<property name="customMap">
   <map>
      <entry key="1" value="INDIA"/>
      <entry key="2" value="Pakistan"/>
      <entry key="3" value="USA"/>
      <entry key="4" value="UK"/>
```

```
        </map>
      </property>

      <!-- java.util.Properties -->
    <property name="customProperies">
        <props>
            <prop key="admin">admin@nospam.com</prop>
            <prop key="support">support@nospam.com</prop>
        </props>
    </property>

   </bean>

</beans>
```

15) How to inject a java.util.Properties into a Spring Bean?

First way is to use <**props**> tag as below.

```
<bean id="adminUser" class="com.howtodoinjava.common.Customer">

    <!-- java.util.Properties -->
    <property name="emails">
        <props>
            <prop key="admin">admin@nospam.com</prop>
            <prop key="support">support@nospam.com</prop>
        </props>
```

```
    </property>


</bean>
```

You can use “**util:**” namespace as well to create properties bean from properties file, and use bean reference for setter injection.

```
<util:properties id="emails" location="classpath:com/foo/emails.properties" />
```

16) Explain Spring Bean Autowiring?

In spring framework, setting bean dependencies in configuration files is a good practice to follow, but the spring container is also able to autowire relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory. **Autowiring** is specified per bean and can thus be enabled for some beans, while other beans will not be autowired.
The following excerpt from the XML configuration file shows a bean being autowired by name.

```
<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />
```

Apart from the autowiring modes provided in bean configuration file, autowiring can be specified in bean classes also using `@Autowired` annotation. To use `@Autowired` annotation in bean classes, you must first enable the annotation in spring application using below configuration.

```
<context:annotation-config />
```

Same can be achieved using `AutowiredAnnotationBeanPostProcessor` bean definition in configuration file.

```
<bean class ="org.springframework.beans.factory.

annotation.AutowiredAnnotationBeanPostProcessor"/>
```

Now, when annotation configuration has been enables, you are free to autowire bean dependencies using `@Autowired`, the way you like.

```
@Autowired
```

```
public EmployeeDAOImpl ( EmployeeManager manager ) {
    this.manager = manager;
}
```

17) Explain different modes of bean autowiring?

There are **five auto wiring modes** in spring framework. Lets discuss them one by one.

1. **no**: This option is default for spring framework and it means that autowiring is OFF. You have to explicitly set the dependencies using tags in bean definitions.
2. **byName**: This option enables the dependency injection based on bean names. When autowiring a property in bean, property name is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.
3. **byType**: This option enables the dependency injection based on bean types. When autowiring a property in bean, property's class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, a error is raised.
4. **constructor**: Autowiring by constructor is similar to byType, but applies to constructor arguments. In autowire enabled bean, it will look for class type of constructor arguments, and then do a autowire by type on all constructor arguments. Please note that if there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.
5. **autodetect**: Autowiring by autodetect uses either of two modes i.e. constructor or byType modes. First it will try to look for valid constructor with arguments, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the autowire byType mode is chosen.

18) How do you turn on annotation based autowiring?

To enable `@Autowired`, you have to register `AutowiredAnnotationBeanPostProcessor`, and you can do it in two ways.

1. Include `<context:annotation-config >` in bean configuration file.

```
<beans>

    <context:annotation-config />

</beans>
```

2. Include `AutowiredAnnotationBeanPostProcessor` directly in bean configuration file.

```
<beans>

    <bean class="org.springframework.beans.factory.annotation.

AutowiredAnnotationBeanPostProcessor"/>

</beans>
```

19) Explain @Required annotation with example?

In a production-scale application, there may be hundreds or thousands of beans declared in the IoC container, and the dependencies between them are often very complicated. One of the shortcomings of setter injection is that it's very hard for you to check if all required properties have been set or not. To overcome this problem, you can set "**dependency-check**" attribute of <bean> and set one of four attributes i.e. none, simple, objects or all (none is default option).
In real life application, you will not be interested in checking all the bean properties configured in your context files. Rather you would like to check if particular set of properties have been set or not in some specific beans only. Spring's dependency checking feature using "**dependency-check**" attribute, will not able to help you in this case. So solve this problem, you can use`@Required` annotation.
To Use the `@Required` annotation over setter method of bean property in class file as below:

```
public class EmployeeFactoryBean extends AbstractFactoryBean<Object>

{

    private String designation;


    public String getDesignation() {

        return designation;
```

```
    }

    @Required
    public void setDesignation(String designation) {
        this.designation = designation;
    }

    //more code here
}
```

`RequiredAnnotationBeanPostProcessor` is a spring bean post processor that checks if all the bean properties with the`@Required` annotation have been set. To enable this bean post processor for property checking, you must register it in the Spring IoC container.

```
<bean class="org.springframework.beans.factory.annotation.
RequiredAnnotationBeanPostProcessor" />
```

If any properties with `@Required` have not been set, a `BeanInitializationException` will be thrown by this bean post processor.

20) Explain @Autowired annotation with example?

The `@Autowired` annotation provides more fine-grained control over where and how autowiring should be accomplished. The `@Autowired` annotation can be used to autowire bean on the setter method just like `@Required` annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.
E.g. You can use `@Autowired` annotation on setter methods to get rid of the `<property>` element in XML configuration file. When Spring finds an `@Autowired` annotation used with setter methods, it tries to perform **byType** autowiring on the method.
You can apply `@Autowired` to constructors as well. A constructor `@Autowired` annotation indicates that the constructor should be autowired when creating the bean, even if no `<constructor-arg>` elements are used while configuring the bean in XML file.

```
public class TextEditor {
```

```
   private SpellChecker spellChecker;

   @Autowired
   public TextEditor(SpellChecker spellChecker){
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }

   public void spellCheck(){
      spellChecker.checkSpelling();
   }
}
```

And it’s configuration without constructor arguments.

```
<beans>

   <context:annotation-config/>

   <!-- Definition for textEditor bean without constructor-arg  -->
   <bean id="textEditor" class="com.howtodoinjava.TextEditor">
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.howtodoinjava.SpellChecker">
   </bean>

</beans>
```

21) Explain @Qualifier annotation with example?

`@Qualifier` means, which bean is qualify to autowired on a field. The qualifier annotation helps disambiguate bean references when Spring would otherwise not be able to do so.

See below example, it will autowired a "**person**" bean into customer's person property.

```
public class Customer

{

    @Autowired

    private Person person;

}
```

And we have two bean definitions for `Person` class.

```
<bean id="customer" class="com.howtodoinjava.common.Customer" />



<bean id="personA" class="com.howtodoinjava.common.Person" >

    <property name="name" value="lokesh" />

</bean>



<bean id="personB" class="com.howtodoinjava.common.Person" >

    <property name="name" value="alex" />

</bean>
```

Will Spring know which person bean should autowired? NO. When you run above example, it hits below exception :

```
Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

    No unique bean of type [com.howtodoinjava.common.Person] is defined:

        expected single matching bean but found 2: [personA, personB]
```

To fix above problem, you need `@Quanlifier` to tell Spring about which bean should autowired.

```
public class Customer
```

```
{
    @Autowired
    @Qualifier("personA")
    private Person person;
}
```

22) Difference between constructor injection and setter injection?

Please find below the noticeable differences:

1. In Setter Injection, partial injection of dependencies can 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. In constructor injection, partial injection of dependencies is not 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. But, constructor injection cannot overrides the setter injected values. It's obvious because constructors are called to first to create the instance.
3. Using setter injection you can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.
4. In setter injection, if Object A and B are dependent each other i.e A is depends on B and vice-versa, Spring throws`ObjectCurrentlyInCreationException` while creating objects of A and B because A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection because Objects are constructed before setter methods invoked.

23) What are the different types of events in spring framework?

Spring's `ApplicationContext` provides the functionality to support events and listeners in code. We can create beans that listen for events which are published through our `ApplicationContext`. Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and `ApplicationListener` interface. So if a bean implements the`ApplicationListener`, then every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean is notified.

```
public class AllApplicationEventListener

implements ApplicationListener < ApplicationEvent >

{

    @Override

    public void onApplicationEvent(ApplicationEvent applicationEvent)

    {

        //process event

    }

}
```

Spring provides the following **5 standard events**:

1. **ContextRefreshedEvent** : This event is published when the ApplicationContext is either initialized or refreshed. This can also be raised using the refresh() method on the `ConfigurableApplicationContext` interface.
2. **ContextStartedEvent** : This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can re/start any stopped application after receiving this event.
3. **ContextStoppedEvent** : This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do required house keeping work after receiving this event.
4. **ContextClosedEvent** : This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.

5. **RequestHandledEvent** : This is a web-specific event telling all beans that an HTTP request has been serviced.

Apart from above, you can create your own custom events by extending `ApplicationEvent` class. e.g.

```
public class CustomApplicationEvent extends ApplicationEvent

{

    public CustomApplicationEvent ( Object source, final String msg )

    {

        super(source);

        System.out.println("Created a Custom event");

    }

}
```

To listen this event, create a listener like this:

```
public class CustomEventListener implements

ApplicationListener < CustomApplicationEvent >

{

    @Override

    public void onApplicationEvent(CustomApplicationEvent applicationEvent) {

        //handle event

    }

}
```

And to publish this event, you will need the help of applicationContext instance.

```
CustomApplicationEvent customEvent =

new CustomApplicationEvent( applicationContext, "Test message" );

applicationContext.publishEvent ( customEvent );
```

24) Difference between FileSystemResource and ClassPathResource?

In `FileSystemResource` you need to give path of `spring-config.xml` (Spring Configuration) file relative to your project or the absolute location of the file.
In `ClassPathResource` spring looks for the file using `ClassPath` so `spring-config.xml` should be included in classpath. If `spring-config.xml` is in "**src**" so we can give just its name because **src** is in classpath path by default.
In one sentence, ClassPathResource looks in the class path and FileSystemResource looks in the file system.

25) Name some of the design patterns used in Spring Framework?

There are loads of different design patterns used, but there are a few obvious ones:

- **Proxy** -- used heavily in AOP, and remoting.
- **Singleton** -- beans defined in spring config files are singletons by default.
- **Template method** -- used extensively to deal with boilerplate repeated code e.g. **RestTemplate**, `JmsTemplate`,`JpaTemplate`.
- **Front Controller** -- Spring provides `DispatcherServlet` to ensure an incoming request gets dispatched to your controllers.
- **View Helper** -- Spring has a number of custom JSP tags, and velocity macros, to assist in separating code from presentation in views.
- **Dependency injection** -- Center to the whole `BeanFactory` / `ApplicationContext` concepts.
- **Factory pattern** -- BeanFactory for creating instance of an object.