**Dependency Injection (DI):**

Any application is composed of many objects that collaborate with each other to perform some useful stuff. Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.

For example, consider a Car object.

A Car depends on wheels, engine, fuel, battery, etc. to run. Traditionally we define the brand of such dependent objects along with the definition of the Car object.

**Without Dependency Injection (DI):**

class Car{

private Wheel wh= new NepaliRubberWheel();

private Battery bt= new ExcideBattery();

//The rest

}

Here, the Car object *is responsible for creating the dependent objects.*

What if we want to change the type of its dependent object - say Wheel - after the initial NepaliRubberWheel() punctures? We need to recreate the Car object with its new dependency say ChineseRubberWheel(), but only the Car manufacturer can do that.

***Then what does the Dependency Injection do us for...?***

When using dependency injection, objects are given their dependencies *at run time rather than compile time (car manufacturing time)*. So that we can now change the Wheel whenever we want. Here, the dependency (wheel) can be injected into Car at run time.

**After using dependency injection:**

class Car{

private Wheel wh= [Inject an Instance of Wheel at runtime]

private Battery bt= [Inject an Instance of Battery at runtime]

Car(Wheel wh,Battery bt) {

this.wh = wh;

this.bt = bt;

}

//Or we can have setters

void setWheel(Wheel wh) {

this.wh = wh;

}

}

What is IOC or Dependency Injection?

You just need to describe the dependency, the Spring container is then responsible for hooking it all up. The basic concept of IOC (Dependency of Injection) is that you do not create your objects but describe how they should be created.

You don’t directly connect your component and services together in code but describe which services are needed by which component in configuration file.

**Inversion of Control** (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).

**Dependency Injection** (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.

## How you will decide when to use prototype scope and when singleton scope bean?

You should use the prototype scope for all beans that are stateful and the singleton scope should be used for stateless beans.

From spring perspective

* stateless beans: beans that are singleton and are initialized only once. The only state they have is a shared state. These beans are created while the ApplicationContext is being initialized. The SAME bean instance will be returned/injected during the lifetime of this ApplocationContext.
* stateful beans: beans that can carry state (instance variables). These are created EVERY time an object is required (like using the "new" operator in java).

## Differentiate between BeanFactory and ApplicationContext in spring.

- With ApplicationContext more than one config files are possible while only one config file or .xml file is possible with BeanFactory.

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-Common.xml",

"Spring-Connection.xml","Spring-ModuleA.xml"});

- ApplicationContext publishes events to beans that are registered as listeners while BeanFactory doesn't support this  
- ApplicationContext support internationalization messages, application life-cycle events, validation and many enterprise services like JNDI access, EJB integration, remoting etc. while BeanFactory doesn't support any of these.

## What are the different types of IoC (dependency injection) ?

a.) Constructor Injection : Here dependencies are provided as constructor parameters.   
b.) Setter Injection : Dependencies are assigned through JavaBeans properties.   
c.) Interface Injection : Injection is performed through an interface.

**What are Advice, Aspect, Join-point and point cut in spring?**

Advice: An advice is an action taken by the aspect at particular join-point is called Advice.

@Before("execution(public String getName())")

public void LoggingAdvice()

{

System.out.println("Aspect run.Get Method called");

}

Aspect: An aspect is a subprogram which is associated with specific property of a program (Example separating logging code from the main program).

An aspect is functionality or feature that cross cuts over object. AOP increase modularity of a program.

Join-Point: A join point is a point used in spring AOP framework to represent a method execution. It always point during execution of program, method or exception. A join point is basically an opportunity within the code to apply aspect.

@Before("allCircleMethods()")

public void LoggingAdvice(JoinPoint joinPoint)

{

}

Point Cut: In AOP a point cut is a set of many join points where an advice can execute. A chunk of code (known as Advice) associated with join point get executed.

@Pointcut("execution(\* get\*())")

public void allGetters()

{

}

## What is Bean Wiring ?

Bean wiring means creating associations between application components i.e. beans within the spring container.

## What are the different types of Advice?

There are different types of Advice.

Before Advice: The advice which executed before a join point called before advice. The before advice does not have the ability to prevent the execution flow proceeding at the join point (unless it throws an exception).

After Return Advice: The advice which executed after a join point completed normally without any exception.

Around Advice: It is responsible for choosing whether to proceeds to the join point or shortcut the advised method execution by returning its own return value or throwing an exception. This is most powerful kind of advice. With Around advice you can perform custom behavior before and after method execution.

After throwing advice: The advice executed when a method throws an exception.

After (finally) advice: The advice is executed when program exits the join points either normally or by throwing an exception.

## Q: 19 What is Weaving in Spring?

Weaving is the process of linking aspect with other application types or object to create an advised object. This can be performed at compile time, runtime and load time. In spring framework weaving is performed at runtime.

## Q:20 What is AOP Proxy?

AOP proxy is an object to implement the aspect contracts (advice method executions and so on). The AOP proxy is object is created by the AOP framework. In spring framework AOP proxy is JDK dynamic proxy or CGLIB proxy.

## Difference between FileSystemResource and ClassPathResource?

In FileSystemResource you need to give the configuration file (i.e. spring-config.xml) relative to your project or the absolute location of the file.

In ClassPathResource spring looks for the file in the ClassPath so configuration (i.e. spring-config.xml) file should be included in the classpath. If spring-config.xml is in classpath, you can simply give the name of the file.

For Example: If your configuration file is at src/main/java/com/test/loadresource then your FileSystemResource would be:

FileSystemResource resource = new FileSystemResource("src/main/java/com/test/loadresource/spring-config.xml");

And ClassPathResource would be:

ClassPathResource resource = new ClassPathResource("com/test/loadresource /spring-config.xml");

## 23 What is inner Bean Definition?

A bean definition added inside the property or constructor-arg elements are called inner bean.

Example:

<bean id="outerbean" class="...">

<!-- instead of using a reference to a target bean, simply define the target bean inline -->

<property name="targetbean">

<bean class="com.example.Person"> <!-- this is the inner bean -->

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

<property name="age" value="35"/>

</bean>

</property>

</bean>

**BeanFactory vs ApplicationContext**

Before seeing difference between ApplicationContext and BeanFactory, let see some similarity between both of them. Spring provides two kinds of IOC container, one is BeanFactory and other is ApplicationContext. Syntactically BeanFactory and ApplicationContext both are [Java interfaces](http://javarevisited.blogspot.in/2012/04/10-points-on-interface-in-java-with.html) and ApplicationContext extends BeanFactory. Both of them are configuration using [XML configuration file](http://javarevisited.blogspot.in/2012/03/how-to-read-properties-file-in-java-xml.html). In short BeanFactoryprovides basic IOC and DI features while ApplicationContext provides advanced features. Apart from these, Here are few more difference between BeanFactory and ApplicationContext which is mostly based upon features supported by them.

1) BeanFactory doesn't provide support for internationalization i.e. i18n but ApplicationContext provides support for it.

2) Another difference between BeanFactory vs ApplicationContext is ability to publish event to beans that are registered as listener.

3) One of the popular implementation of BeanFactory interface is XMLBeanFactory while one of the popular implementation of ApplicationContext interface is ClassPathXmlApplicationContext. On [Java web application](http://javarevisited.blogspot.sg/2012/08/what-is-jsessionid-in-j2ee-web.html) we use WebApplicationContext  which extends ApplicationContext interface and adds getServletContext method.

4) If you are using auto wiring and using BeanFactory than you need to register AutoWiredBeanPostProcessor using API which you can configure in XML if you are using  ApplicationContext. In summary BeanFactory is OK for testing and non [production](http://javarevisited.blogspot.in/2011/09/how-to-write-production-quality-code.html) use but ApplicationContext is more feature rich container implementation and should be favored over BeanFactory

These were some worth noting difference between BeanFactory and ApplicationContext in Spring framework. In most practical cases you will be using ApplicationContext but knowing about BeanFactory is important to understand fundamental concept of spring framework. I mostly use XML configuration file and ClassPathXmlApplicationContext to quickly run any Spring based Java program from [Eclipse](http://javarevisited.blogspot.sg/2012/10/eclipse-shortcut-to-remove-all-unused-imports-java.html) by using following snippet of code :

**public** **static** **void** main(**String** args[]){  
    ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");  
    Hello hello = (Hello) ctx.getBean("hello");  
    hello.sayHello("John");  
}

here beans.xml is your spring configuration file and “hello” is a bean defined in that spring configuration file. Here we have used ClassPathXmlApplicationContext  which is an implementation of ApplicationContext interface in Spring.

#### Explain Bean lifecycle in Spring framework.

1. The spring container finds the bean’s definition from the XML file and instantiates the bean.  
2. Using the dependency injection, spring populates all of the properties as specified in the bean definition.  
3. If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.  
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.  
5. If there are any BeanPostProcessors associated with the bean, their post- ProcessBeforeInitialization() methods will be called.  
6. If an init-method is specified for the bean, it will be called.  
7. Finally, if there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

#### What is bean wiring?

Combining together beans within the Spring container is known as bean wiring or wiring. When wiring beans, you should tell the container what beans are needed and how the container should use dependency injection to tie them together.

**Question 3: what is Bean Factory, have you used XMLBeanFactory?**

Ans: BeanFactory is factory Pattern which is based on IOC [design principles](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html).it is used to make a clear separation between application configuration and dependency from actual code.

XmlBeanFactory is one of the implementation of bean Factory which we have used in our project.

# [What are the disadvantages of Aspect-Oriented Programming (AOP)?](http://stackoverflow.com/questions/875512/what-are-the-disadvantages-of-aspect-oriented-programming-aop)

Maintenance and Debugging. With aop, you suddenly have code that is being run at a given point ( method entry, exit, whatever ) but in just looking at the code, you have no clue that it's even getting called, especially if the aop configuration is in another file, like xml config. If the advice causes some changes, then while debugging an application, things may look strange with no explanation. This doesn't affect only newbies.

### Which are the important beans lifecycle methods? Can you override them?

There are two important bean lifecycle methods. The first one is setup which is called when the bean is loaded in to the container. The second method is the teardown method which is called when the bean is unloaded from the container.  
The bean tag has two important attributes (init-method and destroy-method) with which you can define your own custom initialization and destroy methods. There are also the correspondive annotations(@PostConstruct and @PreDestroy).

### How can you inject a Java Collection in Spring?

Spring offers the following types of [collection configuration elements](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-collections-list-set-map-and-properties-example/):

* The <list> type is used for injecting a list of values, in the case that duplicates are allowed.
* The <set> type is used for wiring a set of values but without any duplicates.
* The <map> type is used to inject a collection of name-value pairs where name and value can be of any type.
* The <props> type can be used to inject a collection of name-value pairs where the name and value are both Strings.

### Are there limitations with autowiring?

Limitations of autowiring are:

* **Overriding:**You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
* **Primitive data types:** You cannot autowire simple properties such as primitives, Strings, and Classes.
* **Confusing nature:** Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

### 35. Can you inject null and empty string values in Spring?

Yes, you can.

### How do you turn on annotation wiring?

Annotation wiring is not turned on in the Spring container by default. In order to use annotation based wiring we must enable it in our Spring configuration file by configuring <context:annotation-config/> element.

### @Required annotation

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws BeanInitializationException if the affected bean property has not been populated.

### @Qualifier annotation

When there are more than one beans of the same type and only one is needed to be wired with a property, the@Qualifier annotation is used along with @Autowired annotation to remove the confusion by specifying which exact bean will be wired.

### What is a Proxy?

A proxy is an object that is created after applying advice to a target object. When you think of client objects the target object and the proxy object are the same.

**ContextLoaderListener**is the listener to start up and shut down Spring’s root **WebApplicationContext**. It’s important functions are to tie up the lifecycle of **ApplicationContext** to the lifecycle of the **ServletContext** and to automate the creation of **ApplicationContext**. We can use it to define shared beans that can be used across different spring contexts.

**ViewResolver** implementations are used to resolve the view pages by name. Usually we configure it in the spring bean configuration file. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->  <beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">      <beans:property name="prefix" value="/WEB-INF/views/" />      <beans:property name="suffix" value=".jsp" />  </beans:bean> |

**InternalResourceViewResolver** is one of the implementation of ViewResolver interface and we are providing the view pages directory and suffix location through the bean properties. So if a controller handler method returns “home”, view resolver will use view page located at /WEB-INF/views/home.jsp.

### Can we have multiple Spring configuration files?

For Spring MVC applications, we can define multiple spring context configuration files through **contextConfigLocation**. This location string can consist of multiple locations separated by any number of commas and spaces. For example;

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <servlet>      <servlet-name>appServlet</servlet-name>      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>      <init-param>          <param-name>contextConfigLocation</param-name>          <param-value>/WEB-INF/spring/appServlet/servlet-context.xml,/WEB-INF/spring/appServlet/servlet-jdbc.xml</param-value>      </init-param>      <load-on-startup>1</load-on-startup>  </servlet> |

We can also define multiple root level spring configurations and load it through context-param. For example;

|  |  |
| --- | --- |
| 1  2  3  4 | <context-param>      <param-name>contextConfigLocation</param-name>      <param-value>/WEB-INF/spring/root-context.xml /WEB-INF/spring/root-security.xml</param-value>  </context-param> |

Another option is to use import element in the context configuration file to import other configurations, for example:

|  |  |
| --- | --- |
| 1 | <beans:import resource="spring-jdbc.xml"/> |

### Can we send an Object as the response of Controller handler method?

Yes we can, using **@ResponseBody** annotation. This is how we send JSON or XML based response in restful web services.

### What is Spring MVC Interceptor and how to use it?

Sometimes we want to intercept the HTTP Request and do some processing before handing it over to the controller handler methods. That’s where Spring MVC Interceptors come handy.

HandlerInterceptor declares three methods based on where we want to intercept the HTTP request.

1. **boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**: This method is used to intercept the request before it’s handed over to the handler method. This method should return ‘true’ to let Spring know to process the request through another interceptor or to send it to handler method if there are no further interceptors.

If this method returns ‘false’ Spring framework assumes that request has been handled by the interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

Object *handler* is the chosen handler object to handle the request. This method can throw Exception also, in that case [Spring MVC Exception Handling](http://www.journaldev.com/2651/spring-mvc-exception-handling-exceptionhandler-controlleradvice-handlerexceptionresolver-json-response-example) should be useful to send error page as response.

1. **void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**: This interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this interceptor to determine the time taken by handler method to process the client request.
2. **void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**: This is a callback method that is called once the handler is executed and view is rendered.

**Bean definition inheritance**

<bean id="parenttriangle" class="spring.practice.Triangle" >

<property name="pointA" ref="pointA" />

</bean>

<bean id="triangle1" class="spring.practice.Triangle" parent="parenttriangle">

<property name="pointB" ref="pointB" />

<property name="pointC" ref="pointC" />

</bean>

JSON is JavaScript Object Notation. It is a much-more compact way of transmitting sets of data across network connections as compared to XML. I suggest JSON be used in any AJAX-like applications where XML would otherwise be the "recommended" option. The verbosity of XML will add to download time and increased bandwidth consumption ($$$). You can accomplish the same effect with JSON and its mark-up is almost exclusively dedicated to the data itself and not the underlying structure.

{

"firstName": "John",

"lastName": "Smith",

"address": {

"streetAddress": "21 2nd Street",

"city": "New York",

"state": "NY",

"postalCode": 10021

},

"phoneNumbers": [

"212 555-1234",

"646 555-4567"

]

}

A servlet is simply a class which responds to a particular type of network request - most commonly an HTTP request. Basically servlets are usually used to implement web applications - but there are also various frameworks which operate on top of servlets (e.g. Struts) to give a higher-level abstraction than the "here's an HTTP request, write to this HTTP response" level which servlets provide.

Servlets run in a servlet container which handles the networking side (e.g. parsing an HTTP request, connection handling etc). One of the best-known open source servlet containers is [Tomcat](http://tomcat.apache.org/).

What is Hashing

[**Hashing**](http://en.wikipedia.org/wiki/Hash_function) in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:

**Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.**

All objects in java inherit a default implementation of hashCode() function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

When your class implements [Comparable](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html), the compareTo method of the class is defining the "natural" ordering of that object. That method is contractually obligated (though not demanded) to be in line with other methods on that object, such as a 0 should always be returned for objects when the .equals() comparisons return true.

A [Comparator](http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html) is its own definition of how to compare two objects, and can be used to compare objects in a way that might not align with the natural ordering.

For example, Strings are generally compared alphabetically. Thus the "a".compareTo("b") would use alphabetical comparisons. If you wanted to compare Strings on length, you would need to write a custom comparator.

In short, there isn't much difference. They are both ends to similar means. In general implement comparable for natural order, (natural order definition is obviously open to interpretation), and write a comparator for other sorting or comparison needs.

**Comparable**

A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.

**Comparator**

A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

Implementing Comparable means "I can compare myself with another object." This is typically useful when there's a single natural default comparison.

Implementing Comparator means "I can compare two other objects." This is typically useful when there are multiple ways of comparing two instances of a type - e.g. you could compare people by age, name etc.

Local variables are stored on the stack. Instance and static variables are stored on the heap.

Binary Search trees by definition cannot contain duplicates