1) What is Spring ?

Spring is an open source framework created to address the complexity of enterprise application development. One of the chief advantages of the Spring framework is its layered architecture, which allows you to be selective about which of its components you use while also providing a cohesive framework for J2EE application development.

2) What are the advantages of Spring framework?

The advantages of Spring are as follows:

Spring has layered architecture. Use what you need and leave you don't need now.

Spring Enables POJO Programming. There is no behind the scene magic here. POJO programming enables continuous integration and testability.

Dependency Injection and Inversion of Control Simplifies JDBC

Open source and no vendor lock-in.

3) What are features of Spring ?

Lightweight: spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.

Inversion of control (IOC): Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.

Aspect oriented (AOP): Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.

Container: Spring contains and manages the life cycle and configuration of application objects.

MVC Framework: Spring comes with MVC web application framework, built on core Spring functionality. This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.

Transaction Management: Spring framework provides a generic abstraction layer for transaction management. This allowing the developer to add the pluggable transaction managers, and making it easy to demarcate transactions without dealing with low-level issues. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.

JDBC Exception Handling: The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy. Integration with Hibernate, JDO, and iBATIS: Spring provides best Integration services with Hibernate, JDO and iBATIS

4) What is IOC (or Dependency Injection)?

The basic concept of the Inversion of Control pattern (also known as dependency injection) is that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (in the case of the Spring framework, the IOC container) is then

responsible for hooking it all up.

i.e., Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

5) What are the different types of IOC (dependency injection) ?
There are three types of dependency injection:

Constructor Injection (e.g. Pico container, Spring etc): Dependencies are provided as constructor parameters.

Setter Injection (e.g. Spring): Dependencies are assigned through JavaBeans properties (ex: setter methods).

Interface Injection (e.g. Avalon): Injection is done through an interface.

Note: Spring supports only Constructor and Setter Injection

6) What are the benefits of IOC (Dependency Injection)?
Benefits of IOC (Dependency Injection) are as follows:

Minimizes the amount of code in your application. With IOC containers you do not care about how services are created and how you get references to the ones you need. You can also easily add additional services by adding a new constructor or a setter method with little or no extra configuration.

Make your application more testable by not requiring any singletons or JNDI lookup mechanisms in your unit test cases. IOC containers make unit testing and switching implementations very easy by manually allowing you to inject your own objects into the object under test.

Loose coupling is promoted with minimal effort and least intrusive mechanism. The factory design pattern is more intrusive because components or services need to be requested explicitly whereas in IOC the dependency is injected into requesting piece of code. Also some containers promote the design to interfaces not to implementations design concept by encouraging managed objects to implement a well-defined service interface of your own.

IOC containers support eager instantiation and lazy loading of services. Containers also provide support for instantiation of managed objects, cyclical dependencies, life cycles management, and dependency resolution between managed objects etc.

7) How many modules are there in Spring? What are they?
Spring comprises of seven modules. They are..
The core container:
The core container provides the essential functionality of the Spring framework. A primary component of the core container is the BeanFactory, an implementation of the Factory pattern. The BeanFactory applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code.
Spring context:

The Spring context is a configuration file that provides context information to the Spring framework. The Spring context includes enterprise services such as JNDI, EJB, e-mail, internalization, validation, and scheduling functionality.

Spring AOP:

The Spring AOP module integrates aspect-oriented programming functionality directly into the Spring framework, through its configuration management feature. As a result you can easily AOP-enable any object managed by the Spring framework. The Spring AOP module provides transaction management services for objects in any Spring-based application. With Spring AOP you can incorporate declarative transaction management into your applications without relying on EJB components.

Spring DAO:

The Spring JDBC DAO abstraction layer offers a meaningful exception hierarchy for managing the exception handling and error messages thrown by different database vendors. The exception hierarchy simplifies error handling and greatly reduces the amount of exception code you need to write, such as opening and closing connections. Spring DAO's JDBC-oriented exceptions comply to its generic DAO exception hierarchy.

Spring ORM:

The Spring framework plugs into several ORM frameworks to provide its Object Relational tool, including JDO, Hibernate, and iBatis SQL Maps. All of these comply to Spring's generic transaction and DAO exception hierarchies.

Spring Web module:

The Web context module builds on top of the application context module, providing contexts for Web-based applications. As a result, the Spring framework supports integration with Jakarta Struts. The Web module also eases the tasks of handling multi-part requests and binding request parameters to domain objects.

Spring MVC framework:

The Model-View-Controller (MVC) framework is a full-featured MVC implementation for building Web applications. The MVC framework is highly configurable via strategy interfaces and accommodates numerous view technologies including JSP, Velocity, Tiles, iText, and POI

7) What are the types of Dependency Injection Spring supports?>

Setter Injection: Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Constructor Injection: Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator.

8) What is Bean Factory ?

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.

9) What is Application Context?

A bean factory is fine to simple applications, but to take advantage of the full power of the Spring framework, you may want to move up to Springs more advanced container, the application context. 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:

A means for resolving text messages, including support for internationalization.

A generic way to load file resources.

Events to beans that are registered as listeners.

10) What is the difference between Bean Factory and Application Context ?

On the surface, an application context is same as a bean factory. But application context offers much more..

Application contexts provide a means for resolving text messages, including support for i18n of those messages.

Application contexts provide a generic way to load file resources, such as images.

Application contexts can publish events to beans that are registered as listeners.

Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.

ResourceLoader support: Spring's Resource interface us a flexible generic abstraction for handling low-level resources. An application context itself is a ResourceLoader, Hence provides an application with access to deployment-specific Resource instances.

MessageSource support: The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable

11) What are the common implementations of the Application Context ?

The three commonly used implementation of 'Application Context' are

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");

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");
XmlWebApplicationContext : It loads context definition from an XML file contained within a web application.

12) How is a typical spring implementation look like ?

For a typical Spring Application we need the following files:

a) An interface that defines the functions.

b) An Implementation that contains properties, its setter and getter methods, functions etc.,

c) Spring AOP (Aspect Oriented Programming)

d) A XML file called Spring configuration file.

e) Client program that uses the function.

13) What is the typical Bean life cycle in Spring Bean Factory Container ?

Bean life cycle in Spring Bean Factory Container is as follows:

f) The spring container finds the bean's definition from the XML file and instantiates the bean.

g) Using the dependency injection, spring populates all of the properties as specified in the bean definition

h) If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean's ID.

i) If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.

j) If there are any BeanPostProcessors associated with the bean, their post-ProcessBeforeInitialization() methods will be called.

k) If an init-method is specified for the bean, it will be called.

l) Finally, if there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

14) What do you mean by Bean wiring ?

The act of creating associations between application components (beans) within the Spring container is reffered to as Bean wiring.

15) What do you mean by Auto Wiring?

The Spring container is 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. The autowiring functionality has five modes.

no

byName

byType

constructor

autodirect

16) What is DelegatingVariableResolver?

Spring provides a custom JavaServer Faces VariableResolver implementation that extends the standard Java Server Faces managed beans mechanism which lets you use JSF and Spring together. This variable resolver is called as DelegatingVariableResolver

17) How to integrate Java Server Faces (JSF) with Spring?

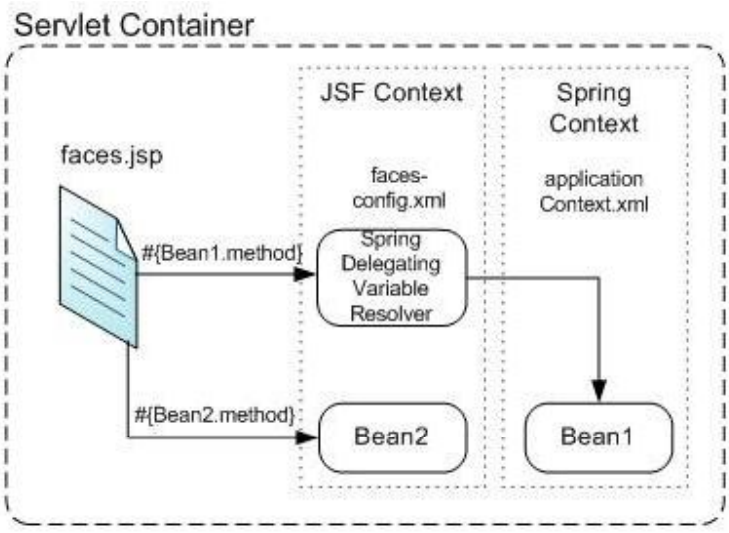
JSF and Spring do share some of the same features, most noticeably in the area of IOC services. By declaring JSF managed-beans in the faces-config.xml configuration file, you allow the FacesServlet to instantiate that bean at startup. Your JSF pages have access to these beans and all of their properties.We can integrate JSF and Spring in two ways:

DelegatingVariableResolver: Spring comes with a JSF variable resolver that lets you use JSF and Spring together.

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

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">

<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
</faces-config>
```



The DelegatingVariableResolver will first delegate value lookups to the default resolver of the underlying JSF implementation, and then to Spring's 'business context' WebApplicationContext. This allows one to easily inject dependencies into one's JSF-managed beans.

FacesContextUtils:custom VariableResolver works well when mapping one's properties to beans in faces-config.xml, but at times one may need to grab a bean explicitly. The FacesContextUtils class makes this easy. It is similar to WebApplicationContextUtils, except that it takes a FacesContext parameter rather than a ServletContext parameter.

ApplicationContext ctx =
FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());

18) What is Java Server Faces (JSF) - Spring integration mechanism?

Spring provides a custom JavaServer Faces VariableResolver implementation that extends the standard JavaServer Faces managed beans mechanism. When asked to resolve a variable name, the following algorithm is performed:

Does a bean with the specified name already exist in some scope (request, session, application)? If so, return it

Is there a standard JavaServer Faces managed bean definition for this variable name? If so, invoke it in the usual way, and return the bean that was created.

Is there configuration information for this variable name in the Spring WebApplicationContext for this application? If so, use it to create and configure an instance, and return that instance to the caller.

If there is no managed bean or Spring definition for this variable name, return null instead.

BeanFactory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.

As a result of this algorithm, you can transparently use either JavaServer Faces or Spring facilities to create beans on demand.

19) What is Significance of JSF- Spring integration ?

Spring - JSF integration is useful when an event handler wishes to explicitly invoke the bean factory to create beans on demand, such as a bean that encapsulates the business logic to be performed when a submit button is pressed.

20) How to integrate your Struts application with Spring?

To integrate your Struts application with Spring, we have two options:

Configure Spring to manage your Actions as beans, using the ContextLoaderPlugin, and set their dependencies in a Spring context file.

Subclass Spring's ActionSupport classes and grab your Spring-managed beans explicitly using a getWebApplicationContext() method.

21) What are ORM's Spring supports ?

Spring supports the following ORM's :

Hibernate

iBatis

JPA (Java Persistence API)

TopLink

JDO (Java Data Objects)

OJB

22) What are the ways to access Hibernate using Spring ?

There are two approaches to Spring's Hibernate integration:

Inversion of Control with a HibernateTemplate and Callback

Extending HibernateDaoSupport and Applying an AOP Interceptor

23) How to integrate Spring and Hibernate using HibernateDaoSupport?

Spring and Hibernate can integrate using Spring's SessionFactory called LocalSessionFactory. The integration process is of 3 steps.

Configure the Hibernate SessionFactory

Extend your DAO Implementation from HibernateDaoSupport

Wire in Transaction Support with AOP

24) What are Bean scopes in Spring Framework ?

The Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware ApplicationContext). The scopes supported are listed below**:**

Scope	Description
Singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
Prototype	Scopes a single bean definition to any number of object instances.

Scope	Description
Request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
Session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

25) What is AOP?

Aspect-oriented programming, or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management. The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules.

26) **How the AOP used in Spring?**

AOP is used in the Spring Framework: To provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management, which builds on the Spring Framework's transaction abstraction.To allow users to implement custom aspects, complementing their use of OOP with AOP.

27) What do you mean by Aspect ?

A modularization of a concern that cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (@AspectJ style).

28) What do you mean by JointPoint?

A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

29) What do you mean by Advice?

Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors "around" the join point.

30) What are the types of Advice?

Types of advice:

a) Before advice: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

b) After returning advice: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

c) After throwing advice: Advice to be executed if a method exits by throwing an exception.

d) After (finally) advice: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

e) Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception

31) What are the types of the transaction management Spring supports ?

Spring Framework supports:

Programmatic transaction management.

Declarative transaction management.

32) What are the benefits of the Spring Framework transaction management ?

The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

a) Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.

b) Supports declarative transaction management.

c) Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.

d) Integrates very well with Spring's various data access abstractions.

33) Why most users of the Spring Framework choose declarative transaction management ?

Most users of the Spring Framework choose declarative transaction management because it is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

34) Explain the similarities and differences between EJB CMT and the Spring Framework's declarative transaction management ?

The basic approach is similar: it is possible to specify transaction behavior (or lack of it) down to individual method level. It is possible to make a setRollbackOnly() call within a transaction context if necessary. The differences are:

a) Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.

b) The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.

c) The Spring Framework offers declarative rollback rules: this is a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.

d) The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. With EJB CMT, you have no way to influence the container's transaction management other than setRollbackOnly().

e) The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers.

35) When to use programmatic and declarative transaction management ?

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. On the other hand, if your application has numerous transactional operations,

declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure.

36) Explain about the Spring DAO support ?

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows one to switch between the persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

37) What are the exceptions thrown by the Spring DAO classes ?

Spring DAO classes throw exceptions which are subclasses of DataAccessException(org.springframework.dao.DataAccessException).Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception. These exceptions wrap the original exception.

38) 40. What is SQLExceptionTranslator ?

SQLExceptionTranslator, is an interface to be implemented by classes that can translate between SQLExceptions and Spring's own data-access-strategy-agnostic org.springframework.dao.DataAccessException.

39) What is Spring's JdbcTemplate ?

Spring's *JdbcTemplate* is central class to interact with a database through JDBC. JdbcTemplate provides many convenience methods for doing things such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.

JdbcTemplate template = new JdbcTemplate(myDataSource);

40) What is PreparedStatementCreator ?

PreparedStatementCreator:

Is one of the most common used interfaces for writing data to database.

Has one method – createPreparedStatement(Connection)

Responsible for creating a PreparedStatement.

Does not need to handle SQLExceptions.

41) What is SQLProvider ?

SQLProvider:

Has one method – getSql()

Typically implemented by PreparedStatementCreator implementers

Useful for debugging.

42) What is RowCallbackHandler ?

The RowCallbackHandler interface extracts values from each row of a ResultSet.

Has one method – processRow(ResultSet)

Called for each row in ResultSet.

Typically stateful.

43) What are the differences between EJB and Spring ?

Spring and EJB feature comparison.

Feature	EJB	Spring
Transaction management	Must use a JTA transaction manager. Supports transactions that span remote method calls.	Supports multiple transaction environments through its PlatformTransactionManager interface, including JTA, Hibernate, JDO, and JDBC. Does not natively support distributed transactions—it must be used with a JTA transaction manager.
Declarative transaction support	Can define transactions declaratively through the deployment descriptor. Can define transaction behavior per method or per class by using the wildcard character *. Cannot declaratively define rollback behavior—this must be done programmatically.	Can define transactions declaratively through the Spring configuration file or through class metadata. Can define which methods to apply transaction behavior explicitly or by using regular expressions. Can declaratively define rollback behavior per method and per exception type.
Persistence	Supports programmatic bean-managed persistence and declarative container managed persistence.	Provides a framework for integrating with several persistence technologies, including JDBC, Hibernate, JDO, and iBATIS.
Declarative security	Supports declarative security through users and roles. The management and implementation of users and roles is container specific. Declarative security is configured in the deployment descriptor.	No security implementation out-of-the box. Acegi, an open source security framework built on top of Spring, provides declarative security through the Spring configuration file or class metadata.
Distributed computing	Provides container-managed remote method calls.	Provides proxying for remote calls via RMI, JAX-RPC, and web services.

44) What is aspect oriented programming (AOP)? Do you have any experience with AOP?

Aspect-Oriented Programming (AOP) complements OOP (Object Oriented Programming) by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements.

AOP allows you to dynamically modify your static model consisting mainly of business logic to include the code required to fulfill the secondary requirements or in AOP terminology called cross-cutting concerns (i.e. secondary requirements) like auditing, logging, security, exception handling etc without having to modify the original static model (in fact, we don't even need to have the original code). Better still, we can often keep this additional code in a single location rather than having to scatter it across the existing model, as we would have to if we were using OOP on its own.

For example; A typical Web application will require a servlet to bind the HTTP request to an object and then pass it to the business handler object to be processed and finally return the response back to the

user. So only a minimum amount of code is initially required. But once you start adding all the other additional secondary requirements or cross-cutting concerns like logging, auditing, security, exception-handling, transaction demarcation, etc the code will inflate to 2-4 times its original size. This is where AOP can assist by separately modularizing these cross-cutting concerns and integrating theses concerns at runtime or compile time through. aspect weaving. AOP allows rapid development of an evolutionary prototype using OOP by focusing only on the business logic by omitting concerns such as security, auditing, logging etc. Once the prototype is accepted, additional concerns like security, logging, auditing etc can be weaved into the prototype code to transfer it into a production standard application.
AOP nomenclature is different from OOP and can be described as shown below:

Join points: represents the point at which a cross-cutting concern like logging, auditing etc intersects with a main concern like the core business logic. Join points are locations in programs' execution path like method & constructor call, method & constructor execution, field access, class & object initialization, exception handling execution etc.
pointcut: is a language construct that identifies specific join points within the program. A pointcut defines a collection of join points and also provides a context for the join point.
Advice: is an implementation of a cross-cutting concern which is a piece of code that is executed upon reaching a pointcut within a program.
Aspect: encapsulates join points, pointcuts and advice into a reusable module for the cross-cutting concerns which is equivalent to Java classes for the core concerns in OOP. Classes and aspects are independent of one another. **Classes are unaware of the presence of aspects, which is an important AOP concept**. Only pointcut declaration binds classes and aspects.
Weaving is the process for interleaving separate cross-cutting concerns such as logging into core concerns such as business logic code to complete the system. AOP weaving composes different implementations of aspects into a cohesive system based on weaving rules. The weaving process (aka injection of aspects into Java classes) can happen at:

Compile-time: Weaving occurs during compilation process.
Load-time: Weaving occurs at the byte-code level at class loading time.
Runtime: Similar to load-time where weaving occurs at byte-code level during runtime as join points are reached in the executing application.

So which approach to use? Load-time and runtime weaving have the advantages of being highly dynamic and
enabling changes on the fly without having to rebuild and redeploy. But Load-time and runtime weaving adversely affect system performance. Compile time weaving offers better performance but requires rebuilding and redeployment to effect changes.

Two of the most interesting modules of the Spring framework are **AOP** (Aspect Oriented Programming) and Inversion Of Control (**IoC**) container (aka Dependency Injection). Let us look at a simple AOP example.

STEP 1: Define the interface and the implementation classes. Spring promotes the code to interface design concept.

```
public interface Hello {
public void hello();
}
public class HelloImpl implements Hello{
public void hello() {
System.out.println("Printing hello. ");
}
}
```

STEP 2: Configure the Spring runtime via the **SpringConfig.xml** file. Beans can be configured and subsequently

injected into the calling ***Test*** class.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/springbeans.
dtd">
<beans>
<!-- bean configuration which enables dependency injection -->
<bean id="helloBean" class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="target">
<bean class="HelloImpl" singleton="false" />
</property>
</bean>
</beans>
```

STEP 3: Write your *Test* class. The "**SpringConfig.xml**" configuration file should be in the classpath.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Test {
public static void main(String[] args) {
ApplicationContext ctx = new FileSystemXmlApplicationContext("SpringConfig.xml");
Hello h = (Hello)ctx.getBean("helloBean");
h.hello();
}
}
```

If you run the ***Test*** class, you should get an output of :

Printing hello.

Now, if you want to trace your methods like **hello()** *before* and *after* in your ***Hello*** class, then you can make use of the Spring AOP.

STEP 4: Firstly you need to define the classes for the *before* and *after* advice for the method tracing as follows:

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class TracingBeforeAdvice implements MethodBeforeAdvice {
public void before(Method arg0, Object[] arg1, Object arg2) throws Throwable {
System.out.println("Just before method call...");
}

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
public class TracingAfterAdvice implements AfterReturningAdvice {
public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3)
throws Throwable {
System.out.println("Just after returning from the method call...");
}
}
```

STEP 5: In order to attach the advice to the appropriate joint points, you must make a few amendments to the **SpringConfig.xml** file as shown below in bold:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/springbeans.
dtd">
<beans>
<!-- bean configuration which enables dependency injection -->
<bean id="helloBean"
class="org.springframework.aop.framework.ProxyFactoryBean">
```

```
<property name="target">
<bean class="HelloImpl" singleton="false" />
</property>
<property name="interceptorNames">
<list>
<value>traceBeforeAdvisor</value>
<value>traceAfterAdvisor</value>
</list>
</property>
</bean>
<!-- Advice classes -->
<bean id="tracingBeforeAdvice" class="TracingBeforeAdvice" />
<bean id="tracingAfterAdvice" class="TracingAfterAdvice" />
<!-- Advisor: way to associate advice beans with pointcuts -->
<!-- pointcut definition for before method call advice -->
<bean id="traceBeforeAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<property name="advice">
<ref local="tracingBeforeAdvice" />
</property>
<property name="pattern">
<!-- apply the advice to Hello class methods -->
<value>Hello.*</value>
</property>
</bean>
<!-- Advisor: way to associate advice beans with pointcuts -->
<!-- pointcut definition for after returning from the method advice -->
<bean id="traceAfterAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<property name="advice">
<ref local="tracingAfterAdvice" />
</property>
<!-- apply the advice to Hello class methods -->
<property name="pattern">
<value>Hello.*</value>
</property>
</bean>
</beans>
```

If you run the ***Test*** class again, you should get an output with AOP in action:

```
Just before method call...
Printing hello.
Just after returning from the method call...
```

45) What are the differences between OOP and AOP?

OOP:

OOP looks at an application as a set of collaborating objects. OOP code scatters system level code like logging, security etc with the business logic code.
OOP nomenclature has classes, objects, interfaces etc.

Provides benefits such as code reuse, flexibility, improved maintainability, modular architecture, reduced development time etc with the help of polymorphism, inheritance and encapsulation.

AOP:

AOP looks at the complex software system as combined implementation of multiple concerns like business logic, data persistence, logging, security, multithread safety, error handling, and so on. Separates business logic code from the system level code. In fact one concern remains unaware of other concerns.

AOP nomenclature has join points, point cuts, advice, and aspects.
AOP implementation coexists with the OOP by choosing OOP as the base language. **For example:** AspectJ uses Java as the base language.
AOP provides benefits provided by OOP plus some additional benefits which are discussed in the next question.

46) What are the benefits of AOP?

OOP can cause the system level code like logging, transaction management, security etc to scatter throughout the business logic. AOP helps overcome this problem by centralizing these cross-cutting concerns.

AOP addresses each aspect separately in a modular fashion with minimal coupling and duplication of code. This modular approach also promotes code reuse by using a business logic concern with a separate logger aspect.

It is also easier to add newer functionalities by adding new aspects and weaving rules and subsequently regenerating the final code. This ability to add newer functionality as separate aspects enable application designers to delay or defer some design decisions without the dilemma of over designing the application.

Promotes rapid development of evolutionary prototypes using OOP by focusing only on the business logic by omitting cross-cutting concerns such as security, auditing, logging etc. Once the prototype is accepted, additional concerns like security, logging, auditing etc can be weaved into the prototype code to transfer it into a production standard application.

Developers can concentrate on one aspect at a time rather than having to think simultaneously about business logic, security, logging, performance, multithread safety etc. Different aspects can be developed by different developers based on their key strengths. **For example:** A security aspect can be developed by a security expert or a senior developer who understands security.

47) What is inversion of control (IoC) (also known more specifically as dependency injection)?

Inversion of control or dependency injection (which is a specific type of IoC) is a term used to resolve object dependencies by injecting an instantiated object to satisfy dependency as opposed to explicitly requesting an object. So objects will not be explicitly requested but objects are provided as needed with the help of an Inversion Of Controller container (e.g. Spring, Hivemind etc). This is analogous to the Hollywood principal where the servicing objects say to the requesting client code (i.e. the caller) “don’t call us, we’ll call you”. Hence it is called inversion of control.
Most of you all are familiar with the software development context where client code (requesting code) collaborates with other dependent objects (or servicing objects) by knowing which objects to talk to, where to locate them and how to talk with them. This is achieved by embedding the code required for locating and instantiating the requested components within the client code. The above approach will tightly couple the dependent components with the client code.

Caller code:

```
class CarBO {
public void getCars(String color) {
//if you need to use a different implementation class say FastCarDAOImpl then need to
//make a change to the caller here (i.e. CarDAO dao = new FastCarDAOImpl()). so the
//caller is tightly coupled. If this line is called by 10 different callers then you
//need to make changes in 10 places.
CarDAO dao = new CarDAOImpl();
List listCars = dao.findCarsByColor(color);
}
}
```

Being called code:

```
interface CarDAO (){
public abstract List findCarsByColor(color);
}
interface CarDAOImpl extends CarDAO (){
public List findCarsByColor(color) {
//data access logic goes here
}
}
```

This tight coupling can be resolved by applying the **factory design pattern** and **program to interfaces not to implementations driven development**.

Simplified factory class implemented with a singleton design pattern:

```
class CarDAOFactory {
private static final CarDAOFactory onlyInstance = new CarDAOFactory();
private CarDAOFactory(){}//private so that cannot be instantiated from outside the class
public CarDAOFactory getInstance(){
return onlyInstance;
}
public CarDAO getDAO(){
//if the implementation changes to FastCarDAOImpl then change here only instead of 10
//different places.
return new CarDAOImpl();
}
}
```

Now the caller code should be changed to:

```
class CarBO {
public void getCars(String color) {
//if you need to use a different implementation class say FastCarDAOImpl then need to
//make one change only to the factory class CarDAOFactory to return a different
//implementation (i.e. FastCarDAOImpl) rather than having to change all the callers.
CarDAO dao = CarDAOFactory.getInstance().getDAO();
List listCars = dao.findCarsByColor(color);
}
}
```

But the factory design pattern is still an intrusive mechanism because servicing objects need to be requested explicitly. Also if you work with large software systems, as the system grows the number of factory classes can become quite large. All the factory classes are simple singleton classes that make use of static methods and field variables, and therefore cannot make use of inheritance. This results in same basic code structure repeated in all the factory classes.

Let us look at how dependency injection comes to our rescue. It takes the approach that clients declare their

dependency on servicing objects through a configuration file (like spring-config.xml) and some external piece of supplying the relevant references when needed to the client code whereby acting as the factory objects. This external piece of code is often referred to as IoC (specifically known as dependency injection) container or framework.

SpringConfig.xml

```
<beans>
<bean id="car" class="CarBO" singleton="false" >
<constructor-arg>
<ref bean="carDao" />
</constructor-arg>
</bean>
<bean id="carDao" class="CarDAOImpl" singleton="false" />
</beans>
```

Now your ***CarBO*** code changes to:

```
class CarBO {
private CarDAO dao = null;
public CarBO(CarDAO dao) {
this.dao = dao;
}
public void getCars(String color) {
//if you need to use a different implementation class say FastCarDAOImpl then need to
//make one change only to the SpringConfig.xml file to use a different implementation
//class(i.e. class="FastCarDAOImpl") rather than having to change all the callers.
List listCars = dao.findCarsByColor(color);
}
}
```

Your calling code would be (e.g. from a Web client or EJB client):

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("SpringConfig.xml");
//lookup "car" in your caller where "carDao" is dependency injected using the constructor.
CarBO bo = (CarBO)ctx.getBean("car"); //Spring creates an instance of the CarBO object with
//an instance of CarDAO object as the constructor arg.
String color = red;
bo.getCars(color)
```

You can use IoC containers like Spring framework to inject your business objects and DAOs into your calling classes. Dependencies can be wired by either using annotations or using XML as shown above. Tapestry 4.0 makes use of the Hivemind IoC container for injecting application state objects, pages etc.

IoC or dependency injection containers generally control creation of objects (by calling "new") and resolve dependencies between objects it manages. Spring framework, Pico containers, Hivemind etc are IoC containers to name a few. IoC containers support **eager instantiation**, which is quite useful if you want self-starting services that "come up" on their own when the server starts. They also support **lazy loading**, which is useful when you have many services which may only be sparsely used.

48) What are the benefits of IoC (aka Dependency Injection)?

Minimizes the amount of code in your application. With IoC containers you do not care about how services are created and how you get references to the ones you need. You can also easily add additional services by adding a new constructor or a setter method with little or no extra configuration.

Makes your application more testable by not requiring any singletons or JNDI lookup mechanisms in your unit test cases. IoC containers make unit testing and switching implementations very easy by manually allowing you to inject your own objects into the object under test.

Loose coupling is promoted with minimal effort and least intrusive mechanism. The factory design pattern is more intrusive because components or services need to be requested explicitly whereas in IoC the dependency is injected into the requesting code. Also some containers promote the design to interfaces not to implementations design concept by encouraging managed objects to implement a well-defined service interface of your own.

IoC containers support eager instantiation and lazy loading of services. Containers also provide support for instantiation of managed objects, cyclical dependencies, life cycle management, and dependency resolution between managed objects etc.