Spring 3.0 Interview Questions-Answers

---

1. 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.

2. 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

3. 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.

4. 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.

5. 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.

6. 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

7. How many modules are there in Spring? What are they? (**Roll over to view the Image**)

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.

8. 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.

9. 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.

10. 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.

11. 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

12. 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");
- **WebXmlApplicationContext :** It loads context definition from an XML file contained within a web application.

13. How is a typical spring implementation look like ?

For a typical Spring Application we need the following files:

- An interface that defines the functions.
- An Implementation that contains properties, its setter and getter methods, functions etc.,
- Spring AOP (Aspect Oriented Programming)
- A XML file called Spring configuration file.
- Client program that uses the function.

14. What is the typical Bean life cycle in Spring Bean Factory Container ?

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

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

15. 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.

16. 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*.

- byName
- byType
- constructor
- autodetect (Available in Spring 2.5 not in Spring 3.0)

17. What is DelegatingVariableResolver?

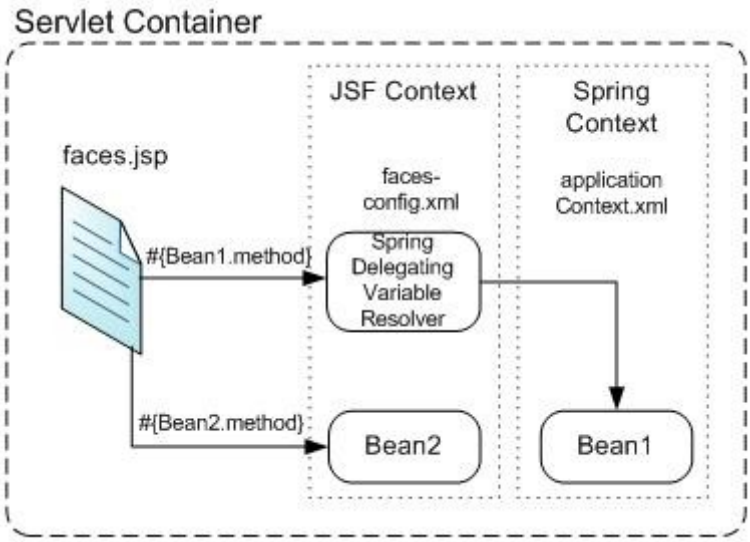
Spring provides a custom JavaServer Faces VariableResolver implementation that extends the standard Java Server Faces managed beans mechanism which lets you use JSF and Spring together. This variable resolver is called as *DelegatingVariableResolver*

18. 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());
```

19. 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.

20. 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.

21. 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.

22. What are ORM's Spring supports ?

Spring supports the following ORM's :

- Hibernate
- iBatis
- JPA (Java Persistence API)
- TopLink
- JDO (Java Data Objects)
- OJB

23. 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

24. 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

25. 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.
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 SpringApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware SpringApplicationContext.
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.

26. 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.

27. 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.

28. 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).

29. 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.

30. 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.

31. What are the types of Advice?

Types of advice:

- *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).
- *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
- *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
- *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- *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

32. What are the types of the transaction management Spring supports ?

Spring Framework supports:

- Programmatic transaction management.
- Declarative transaction management.

33. 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:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.

34. 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.

35. 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:

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

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

37. 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.

38. 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.

39. What are the exceptions thrown by the Spring DAO classes ?

Spring DAO classes throw exceptions which are subclasses ofDataAccessException(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.

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-agnosticorg.springframework.dao.DataAccessException.

41. 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);
```

42. 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.

43. What is SQLProvider ?

SQLProvider:

- Has one method – getSql()
- Typically implemented byPreparedStatementCreator implementers.
- Useful for debugging.

44. 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.

45. 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 itsPlatformTransactionManager 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.

1. **Q. Explain DI or IOC pattern.**
   A: Dependency injection (DI) is a programming design pattern and architectural model, sometimes also referred to as inversion of control or IOC, although technically speaking, dependency injection specifically refers to an implementation of a particular form of IOC. Dependancy Injection describes the situation where one object uses a second object to provide a particular capacity. For example, being passed a database connection as an argument to the constructor instead of creating one internally. The term "Dependency injection" is a misnomer, since it is not a dependency that is injected, rather it is a provider of some capability or resource that is injected. There are three common forms of dependency injection: setter-, constructor- and interface-based injection. Dependency injection is a way to achieve loose coupling. Inversion of control (IOC) relates to the way in which an object obtains references to its dependencies. This is often done by a lookup method. The advantage of inversion of control is that it decouples objects from specific lookup mechanisms and implementations of the objects it depends on. As a result, more flexibility is obtained for production applications as well as for testing.

Q. What are the different IOC containers available?
A. Spring is an IOC container. Other IOC containers are HiveMind, Avalon, PicoContainer.

Q. What are the different types of dependency injection. Explain with examples.
A: There are two types of dependency injection: setter injection and constructor injection.
Setter Injection: Normally in all the java beans, we will use setter and getter method to set and get the value of property as follows:

```
public class namebean {
 String      name;
 public void setName(String a) {
    name = a; }
 public String getName() {
    return name; }
}
```

We will create an instance of the bean 'namebean' (say bean1) and set property as bean1.setName("tom"); Here in setter injection, we will set the property 'name' in spring configuration file as showm below:

```
<bean id="bean1"   class="namebean">
  <property   name="name" >
     <value>tom</value>
  </property>
</bean>
```

The subelement <value> sets the 'name' property by calling the set method as setName("tom"); This process is called setter injection.
To set properties that reference other beans <ref>, subelement of <property> is used as shown below,

```
 <bean id="bean1"   class="bean1impl">
  <property name="game">
     <ref bean="bean2"/>
  </property>
</bean>
<bean id="bean2"   class="bean2impl" />
```

Constructor injection: For constructor injection, we use constructor with parameters as shown below,

```
public class namebean {
    String name;
    public namebean(String a) {
      name = a;
    }
}
```

We will set the property 'name' while creating an instance of the bean 'namebean' as namebean bean1 = new namebean("tom");

Here we use the <constructor-arg> element to set the the property by constructor injection as

```
 <bean id="bean1"  class="namebean">
   <constructor-arg>
     <value>My Bean Value</value>
  </constructor-arg>
</bean>
```

Q. What is spring? What are the various parts of spring framework? What are the different persistence frameworks which could be used with spring?
A. 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. The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed, as shown in the following figure. Each of the modules (or components) that comprise the Spring framework can stand on its own or be

implemented jointly with one or more of the others. The functionality of each component is as follows:

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.

Q. What is AOP? How does it relate with IOC? What are different tools to utilize AOP?
A: 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. AOP and IOC are complementary technologies in that both apply a modular approach to complex problems in enterprise application development. In a typical object-oriented development approach you might implement logging functionality by putting logger statements in all your methods and Java classes. In an AOP approach you would instead modularize the logging services and apply them declaratively to the components that required logging. The advantage, of course, is that the Java class doesn't need to know about the existence of the logging service or concern itself with any related code. As a result, application code written using Spring AOP is loosely coupled. The best tool to utilize AOP to its capability is AspectJ. However AspectJ works at he byte code level and you need to use AspectJ compiler to get the aop features built into your compiled code. Nevertheless AOP functionality is fully integrated into the Spring context for transaction management, logging, and various other features. In general any AOP framework control aspects in three possible ways:

Joinpoints: Points in a program's execution. For example, joinpoints could define calls to specific methods in a class
Pointcuts: Program constructs to designate joinpoints and collect specific context at those points
Advices: Code that runs upon meeting certain conditions. For example, an advice could log a message before executing a joinpoint

Q. Can you name a tool which could provide the initial ant files and directory structure for a new spring project.

A: Appfuse or equinox.

Q. Explain BeanFactory in spring.
A: Bean factory is an implementation of the factory design pattern and its function is to create and dispense beans. As the bean factory knows about many objects within an application, it is able to create association between collaborating objects as they are instantiated. This removes the burden of configuration from the bean and the client. There are several implementation of BeanFactory. The most useful one is "org.springframework.beans.factory.xml.XmlBeanFactory" It loads its beans based on the definition contained in an XML file. To create an XmlBeanFactory, pass a InputStream to the constructor. The resource will provide the XML to the factory. BeanFactory factory = new XmlBeanFactory(new FileInputStream("myBean.xml"));
This line tells the bean factory to read the bean definition from the XML file. The bean definition includes the description of beans and their properties. But the bean factory doesn't instantiate the bean yet. To retrieve a bean from a 'BeanFactory', the getBean() method is called. When getBean() method is called, factory will instadntiate the bean and begin setting the bean's properties using dependency injection. myBean bean1 = (myBean)factory.getBean("myBean");

Q. Explain the role of ApplicationContext in spring.
A. While Bean Factory is used for simple applications, the Application Context is spring's more advanced container. Like 'BeanFactory' it can be used to load bean definitions, wire beans together and dispense beans upon request. It also provide
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.
Because of additional functionality, 'Application Context' is preferred over a BeanFactory. Only when the resource is scarce like mobile devices, 'BeanFactory' is used. The three commonly used implementation of 'Application Context' 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.

Q. How does Spring supports DAO in hibernate?
A. Spring's HibernateDaoSupport class is a convenient super class for Hibernate DAOs. It has handy methods you can call to get a Hibernate Session, or a SessionFactory. The most convenient method is getHibernateTemplate(), which returns a HibernateTemplate. This template wraps Hibernate checked exceptions with runtime exceptions, allowing your DAO interfaces to be Hibernate exception-free.
Example:

```
public class UserDAOHibernate extends HibernateDaoSupport {
public User getUser(Long id) {
return (User) getHibernateTemplate().get(User.class, id);
}
public void saveUser(User user) {
getHibernateTemplate().saveOrUpdate(user);
if (log.isDebugEnabled()) {
log.debug("userId set to: " + user.getID());
}
}
public void removeUser(Long id) {
Object user = getHibernateTemplate().load(User.class, id);
getHibernateTemplate().delete(user);
}
}
```

Q. What are the id generator classes in hibernate?
A: increment: It generates identifiers of type long, short or int that are unique only when no other

process is inserting data into the same table. It should not the used in the clustered environment.
identity: It supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.
sequence: The sequence generator uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int
hilo: The hilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. Do not use this generator with connections enlisted with JTA or with a user-supplied connection.
seqhilo: The seqhilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid: The uuid generator uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.
guid: It uses a database-generated GUID string on MS SQL Server and MySQL.
native: It picks identity, sequence or hilo depending upon the capabilities of the underlying database.
assigned: lets the application to assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.
select: retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.
foreign: uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

Q. How is a typical spring implementation look like?
A. For a typical Spring Application we need the following files
1. An interface that defines the functions.
2. An Implementation that contains properties, its setter and getter methods, functions etc.,
3. A XML file called Spring configuration file.
4. Client program that uses the function.

Q. How do you define hibernate mapping file in spring?
A. Add the hibernate mapping file entry in mapping resource inside Spring's applicationContext.xml file in the web/WEB-INF directory.

```
 <property name="mappingResources">
    <list>
       <value>org/appfuse/model/User.hbm.xml</value>
    </list>
</property>
```

Q. How do you configure spring in a web application?
A. It is very easy to configure any J2EE-based web application to use Spring. At the very least, you can simply add Spring's ContextLoaderListener to your web.xml file:

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Q. Can you have xyz.xml file instead of applicationcontext.xml?
A. ContextLoaderListener is a ServletContextListener that initializes when your webapp starts up. By default, it looks for Spring's configuration file at WEB-INF/applicationContext.xml. You can change this default value by specifying a <context-param> element named "contextConfigLocation." Example:

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    <context-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/xyz.xml</param-value>
    </context-param>
    </listener-class>
```

```
</listener>
```

Q. How can you configure JNDI instead of datasource in spring applicationcontext.xml?
A. Using "org.springframework.jndi.JndiObjectFactoryBean". Example:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiName">
      <value>java:comp/env/jdbc/appfuse</value>
   </property>
</bean>
```

Ques: 1 What is a BeanFactory?
Ans: The BeanFactory is provide an advanced configuration mechanism capable of managing beans of any kind of storage facility. The ApplicationContext builds on top of the BeanFactory and adds other functionality like integration with Springs AOP features, message resource handling for use in internationalization, event propagation, declarative mechanisms to create the ApplicationContext and optional parent contexts, and application-layer specific contexts such as the WebApplicationContext, among other enhancements.

Spring Framework Transaction Management

Transaction management is critical in any form of applications that will interact with the database. The application has to ensure that the data is consistent and the integrity of the data is maintained. There are many popular data frameworks like **JDBC**, **JPA**, **Hibernate** etc.. and **Spring Framework** provides a seamless way of integrating with these frameworks.

Spring has its own Transaction Model which is common for all the persistence implementations without Spring, we need to use persistence provider specific API to manage the Txs. but with Spring, we can use uniform API to manage the Txs.
Various Transaction managers are provided for various persistence providers.
*** PlatformTransactionManager is the root for all the Transaction Manager in Spring.**
Following are various Transaction managers provided which are sub clasess of PlatformTransactionManager :

1. DataSourceTransactionManager (JDBC)
2. HibernateTransactionManager (Hibernate)
3. JpaTransactionManager (JPA)

Spring supports the following ways to manager the Tx. :
*** Programmatic Transaction :** This is used when developer would like to write the transaction related code himself
*** Declarative transaction :** This is enabled by Spring AOP and as the transactional aspects code comes with Spring and may be used in a boilerplate fashion

a) using TransactionProxyFactoryBean
b) using Schema support
c) using Annotation support

Note : Most developers prefer **declarative transaction** as it has least impact on application code.
Spring provides a unique transaction management abstraction, which enables a consistent programming model over a variety of underlying transaction technologies, such as JTA or JDBC. Supports declarative transaction management. Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA. Integrates very well with Spring's various data access abstractions.

Chapter 9. Transaction management

9.1. Introduction

One of the most compelling reasons to use the Spring Framework is the comprehensive transaction support. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

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

- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes up with some discussion of best practices surrounding transaction management (for example, choosing between declarative and programmatic transaction management).

- The first section, entitled Motivations, describes *why* one would want to use the Spring Framework's transaction abstraction as opposed to EJB CMT or driving transactions via a proprietary API such as Hibernate.
- The second section, entitled Key abstractions outlines the core classes in the Spring Framework's transaction support, as well as how to configure and obtain `DataSource` instances from a variety of sources.
- The third section, entitled Declarative transaction management, covers the Spring Framework's support for declarative transaction management.
- The fourth section, entitled Programmatic transaction management, covers the Spring Framework's support for programmatic (that is, explicitly coded) transaction management.

9.2. Motivations

Is an application server needed for transaction management?

The Spring Framework's transaction management support significantly changes traditional thinking as to when a J2EE application requires an application server.

In particular, you don't need an application server just to have declarative transactions via EJB. In fact, even if you have an application server with powerful JTA capabilities, you may well decide that the Spring Framework's declarative transactions offer more power and a much more productive programming model than EJB CMT.

Typically you need an application server's JTA capability only if you need to enlist multiple transactional resources, and for many applications being able to handle transactions across multiple resources isn't a requirement. For example, many high-end applications use a single, highly scalable database (such as Oracle 9i RAC). Standalone transaction managers such as Atomikos Transactions and JOTM are other options. (Of course you may need other application server capabilities such as JMS and JCA.)

The most important point is that with the Spring Framework *you can choose when to scale your application up to a full-blown application server*. Gone are the days when the only alternative to using EJB CMT or JTA was to write code using local transactions such as those on JDBC connections, and face a hefty rework if you ever needed that code to run within global, container-managed transactions. With the Spring Framework, only configuration needs to change so that your code doesn't have to.

Traditionally, J2EE developers have had two choices for transaction management: *global* or *local* transactions. Global transactions are managed by the application server, using the Java Transaction API (JTA). Local transactions are resource-specific: the most common example would be a transaction associated with a JDBC connection. This choice has profound implications. For instance, global transactions provide the ability to work with multiple transactional resources (typically relational databases and message queues). With local transactions, the application server is not involved in transaction management and cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.)

Global Transactions. Global transactions have a significant downside, in that code needs to use JTA, and JTA is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be sourced from JNDI: meaning that we need to use *both* JNDI *and* JTA to use JTA. Obviously all use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment. Previously, the preferred way to use global transactions was via EJB *CMT* (*Container Managed Transaction*): CMT is a form of **declarative transaction management** (as distinguished from **programmatic transaction management**). EJB CMT removes the need for transaction-related JNDI lookups - although of course the use of EJB itself necessitates the use of JNDI. It removes most of the need (although not entirely) to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in

EJBs, or at least behind a transactional EJB facade. The negatives around EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

Local Transactions. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Another downside is that local transactions tend to be invasive to the programming model.

Spring resolves these problems. It enables application developers to use a *consistent* programming model *in any environment*. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence don't depend on the Spring Framework's transaction API (or indeed on any other transaction API).

9.3. Key abstractions

The key to the Spring transaction abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by the`org.springframework.transaction.PlatformTransactionManager` interface, shown below:

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with the Spring Framework's philosophy,`PlatformTransactionManager` is an *interface*, and can thus be easily mocked or stubbed as necessary. Nor is it tied to a lookup strategy such as JNDI:`PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework's IoC container. This benefit alone makes it a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring's philosophy, the `TransactionException` that can be thrown by any of the `PlatformTransactionManager` interface's methods is*unchecked* (i.e. it extends the `java.lang.RuntimeException` class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle `TransactionException`. The salient point is that developers are not *forced* to do so.

The `getTransaction(..)` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned`TransactionStatus` might represent a new or existing transaction (if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a `TransactionStatus` is associated with a **thread** of execution).

The `TransactionDefinition` interface specifies:

- **Isolation**: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Propagation**: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. *Spring offers all of the transaction propagation options familiar from EJB CMT*.
- **Timeout**: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).

- **Read-only status**: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

These settings reflect standard transactional concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed any other transaction management solution.

The `TransactionStatus` interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `PlatformTransactionManager`implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

`PlatformTransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, etc The following examples from the `dataAccessContext-local.xml` file from Spring's **jPetStore** sample application show how a local `PlatformTransactionManager`implementation can be defined. (This will work with plain JDBC.)

We must define a JDBC `DataSource`, and then use the Spring `DataSourceTransactionManager`, giving it a reference to the `DataSource`.

```
<bean    id="dataSource"    class="org.apache.commons.dbcp.BasicDataSource"    destroy-
method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

The related `PlatformTransactionManager` bean definition will look like this:

```
<bean                                                                    id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

If we use JTA in a J2EE container, as in the `'dataAccessContext-jta.xml'` file from the same sample application, we use a container `DataSource`, obtained via JNDI, in conjunction with Spring's `JtaTransactionManager`. The `JtaTransactionManager` doesn't need to know about the `DataSource`, or any other specific resources, as it will use the container's global transaction management infrastructure.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
                                          http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                            http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

                                           <bean                        id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />
```

```
  <!-- other <bean/> definitions here -->

</beans>
```

Note : The above definition of the 'dataSource' bean uses the <jndi-lookup/> tag from the 'jee' namespace. For more information on schema-based configuration, see Appendix A, *XML Schema-based configuration*, and for more information on the <jee/> tags see the section entitledSection A.2.3, "The jee schema".

We can also use Hibernate local transactions easily, as shown in the following examples from the Spring Framework's **PetClinic** sample application. In this case, we need to define a Hibernate LocalSessionFactoryBean, which application code will use to obtain Hibernate Session instances.

The DataSource bean definition will be similar to the one shown previously (and thus is not shown). If the DataSource is managed by the JEE container it should be non-transactional as the Spring Framework, rather than the JEE container, will manage transactions.

The 'txManager' bean in this case is of the HibernateTransactionManager type. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the SessionFactory.

```
<bean                                                          id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
          hibernate.dialect=${hibernate.dialect}
        </value>
  </property>
</bean>

<bean                                                          id="txManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

With Hibernate and JTA transactions, we can simply use the JtaTransactionManager as with JDBC or any other resource strategy.

```
<bean                                                          id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Note that this is identical to JTA configuration for any resource, as these are global transactions, which can enlist any transactional resource.

In all these cases, application code will not need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

9.4. Resource synchronization with transactions

It should now be clear how different transaction managers are created, and how they are linked to related resources which need to be synchronized to transactions (i.e.DataSourceTransactionManager to a JDBC DataSource, HibernateTransactionManager to a Hibernate SessionFactory, etc). There remains the question however of how the application code, directly or indirectly using a persistence API (JDBC, Hibernate, JDO, etc), ensures that these resources are obtained and handled properly in terms of proper creation/reuse/cleanup and trigger (optionally) transaction synchronization via the relevant PlatformTransactionManager.

9.4.1. High-level approach

The preferred approach is to use Spring's highest level persistence integration APIs. These do not replace the native APIs, but internally handle resource creation/reuse, cleanup, optional transaction synchronization of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same *template* approach is used for all persistence APIs, with examples including the`JdbcTemplate`, `HibernateTemplate`, and `JdoTemplate` classes (detailed in subsequent chapters of this reference documentation.

9.4.2. Low-level approach

At a lower level exist classes such as `DataSourceUtils` (for JDBC), `SessionFactoryUtils` (for Hibernate), `PersistenceManagerFactoryUtils` (for JDO), and so on. When it is preferable for application code to deal directly with the resource types of the native persistence APIs, these classes ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions which happen in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you would instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

If an existing transaction exists, and already has a connection synchronized (linked) to it, that instance will be returned. Otherwise, the method call will trigger the creation of a new connection, which will be (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, this has the added advantage that any `SQLException` will be wrapped in a Spring Framework `CannotGetJdbcConnectionException` - one of the Spring Framework's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the `SQLException`, and ensures portability across databases: even across different persistence technologies.

It should be noted that this will also work fine without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you are using Spring for transaction management.

Of course, once you've used Spring's JDBC support or Hibernate support, you will generally prefer not to use `DataSourceUtils` or the other helper classes, because you'll be much happier working via the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring `JdbcTemplate` or `jdbc.object`package to simplify your use of JDBC, correct connection retrieval happens behind the scenes and you won't need to write any special code.

9.4.3. `TransactionAwareDataSourceProxy`

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a J2EE server.

It should almost never be necessary or desirable to use this class, except when existing code exists which must be called and passed a standard JDBC `DataSource`interface implementation. In that case, it's possible to still have this code be usable, but participating in Spring managed transactions. It is preferable to write your new code using the higher level abstractions mentioned above.

9.5. Declarative transaction management

Most users of the Spring Framework choose declarative transaction management. 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.

The Spring Framework's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with 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:

- 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.
- The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*: a feature with no EJB equivalent, which we'll discuss below. Rollback can be controlled declaratively, not merely programmatically.
- The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than `setRollbackOnly()`.
- The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature. Normally, we do not want transactions to span remote calls.

Where is `TransactionProxyFactoryBean`?

Declarative transaction configuration in versions of Spring 2.0 and above differs considerably from previous versions of Spring. The main difference is that there is no longer any need to configure `TransactionProxyFactoryBean`beans.

The old, pre-Spring 2.0 configuration style is still 100% valid configuration; under the covers think of the new`<tx:tags/>` as simply defining`TransactionProxyFactoryBean` beans on your behalf.

The concept of rollback rules is important: they enable us to specify which exceptions (and throwables) should cause automatic roll back. We specify this declaratively, in configuration, not in Java code. So, while we can still call `setRollbackOnly()` on the `TransactionStatus` object to roll the current transaction back programmatically, most often we can specify a rule that`MyApplicationException` must always result in rollback. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other.

While the EJB default behavior is for the EJB container to automatically roll back the transaction on a*system exception* (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an *application exception* (i.e. a checked exception other than`java.rmi.RemoteException`). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this.

9.5.1. Understanding the Spring Framework's declarative transaction implementation

The aim of this section is to dispel the mystique that is sometimes associated with the use of declarative transactions. It is all very well for this reference documentation simply to tell you to annotate your classes with the `@Transactional` annotation, add the line (`'<tx:annotation-driven/>'`) to your configuration, and then expect you to understand how it all works. This section will explain the inner workings of the Spring Framework's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.

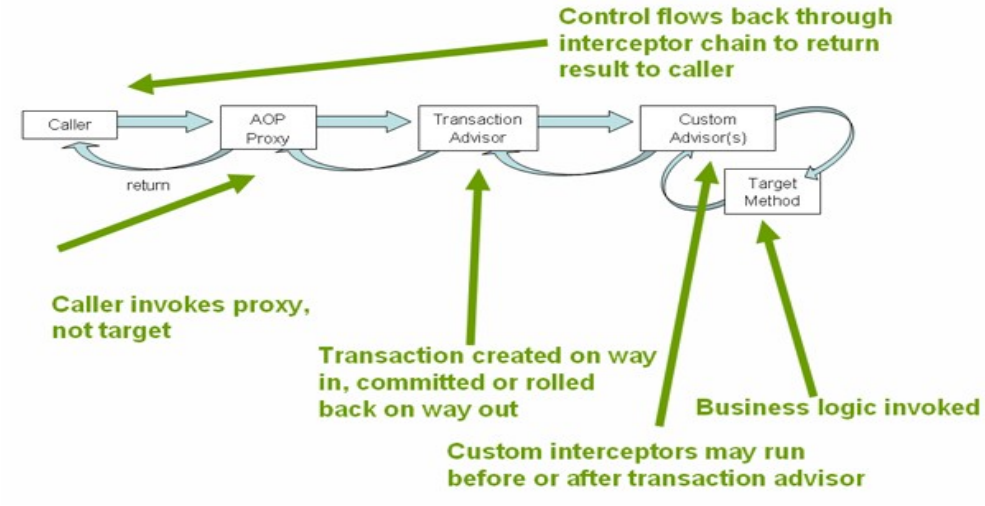
Note : Looking at the Spring Framework source code is a good way to get a real understanding of the transaction support. We also suggest turning the logging level to `'DEBUG'` during development to better see what goes on under the hood.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled *via AOP proxies*, and that the transactional advice is driven by *metadata* (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a`TransactionInterceptor` in conjunction with an appropriate `PlatformTransactionManager` implementation to drive transactions *around method invocations*.

Note

Although knowledge of Spring AOP is not required to use Spring's declarative transaction support, it can h
AOP is thoroughly covered in the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring*.

Conceptually, calling a method on a transactional proxy looks like this...



9.5.2. A first example

Consider the following interface, and its attendant implementation. (The intent is to convey the concepts, and using the rote `Foo` and `Bar` tropes means that you can concentrate on the transaction usage and not have to worry about the domain model.)

```
// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);

}
// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }
    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }
}
```

(For the purposes of this example, the fact that the `DefaultFooService` class throws `UnsupportedOperationException` instances in the body of each implemented method is good;

it will allow us to see transactions being created and then rolled back in response to the `UnsupportedOperationException` *instance being thrown.)*

Let's assume that the first two methods of the `FooService` interface (`getFoo(String)` and `getFoo(String, String)`) have to execute in the context of a transaction with read-only semantics, and that the other methods (`insertFoo(Foo)` and `updateFoo(Foo)`) have to execute in the context of a transaction with read-write semantics. Don't worry about taking the following configuration in all at once; everything will be explained in detail in the next few paragraphs.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
                                           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                             http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
                                            http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

   <!-- the transactional advice (i.e. what 'happens'; see the <aop:advisor/> bean
below) -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- the transactional semantics... -->
    <tx:attributes>
      <!-- all methods starting with 'get' are read-only -->
      <tx:method name="get*" read-only="true"/>
      <!-- other methods use the default transaction settings (see below) -->
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
      of an operation defined by the FooService interface -->
  <aop:config>
                  <aop:pointcut    id="fooServiceOperation"    expression="execution(*
x.y.service.FooService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>

  <!-- don't forget the DataSource -->
     <bean  id="dataSource"  class="org.apache.commons.dbcp.BasicDataSource"  destroy-
method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
                                          <bean                        id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Let's pick apart the above configuration. We have a service object (the `'fooService'` bean) that we want to make transactional. The transaction semantics that we want to apply are encapsulated in

the `<tx:advice/>` definition. The `<tx:advice/>` definition reads as "*... all methods on starting with `'get'` are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics*". The `'transaction-manager'` attribute of the`<tx:advice/>` tag is set to the name of the `PlatformTransactionManager` bean that is going to actually *drive* the transactions (in this case the `'txManager'`bean).

Tip

You can actually omit the `'transaction-manager'` attribute in the transactional advice (`<tx:advice/>`) name of the`PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the`PlatformTransactionManager` bean that you want to wire in has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

The `<aop:config/>` definition ensures that the transactional advice defined by the `'txAdvice'` bean actually executes at the appropriate points in the program. First we define a pointcut that matches the execution of any operation defined in the `FooService` interface (`'fooServiceOperation'`). Then we associate the pointcut with the `'txAdvice'` using an advisor. The result indicates that at the execution of a `'fooServiceOperation'`, the advice defined by `'txAdvice'` will be run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression; see the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring* for more details on pointcut expressions in Spring 2.0.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

```
<aop:config>
                    <aop:pointcut     id="fooServiceMethods"     expression="execution(*
x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

(This example assumes that all your service interfaces are defined in the `'x.y.service'` package; see the chapter entitled Chapter 6, Aspect Oriented Programming with Spring for more details.)

Now that we've analyzed the configuration, you may be asking yourself, "*Okay... but what does all this configuration actually do?*".

The above configuration is going to effect the creation of a transactional proxy around the object that is created from the `'fooService'` bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked *on the proxy*, a transaction *may* be started, suspended, be marked as read-only, etc., depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration.

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
            ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml",
Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}
```

The output from running the above program will look something like this. *(Please note that the Log4J output and the stacktrace from the`UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated in the interest of clarity.)*

```
    <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
        for bean 'fooService' with 0 common interceptors and 1 specific interceptors
    <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]
```

```
    <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo
    <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name
[x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

    <!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction
should
        rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on
x.y.service.FooService.insertFoo
        due to throwable [java.lang.UnsupportedOperationException]

    <!-- and the transaction is rolled back (by default, RuntimeException instances
cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

9.5.3. Rolling back

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled `Exception` as it bubbles up the call stack, and will mark the transaction for rollback.

However, please note that the Spring Framework's transaction infrastructure code will, by default, *only* mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Errors` will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional method will *not* result in the transaction being rolled back.

Exactly which `Exception` types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific `Exception` type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
              <tx:method        name="get*"        read-only="false"        rollback-
for="NoProductInStockException"/>
        <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

The second way to indicate to the transaction infrastructure that a rollback is required is to do so *programmatically*. Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure. Find below a snippet of code that does programmatic rollback of a Spring Framework-managed transaction:

```
public void resolvePosition() {
    try {
        // some business logic...
```

```
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you need it, but its usage flies in the face of achieving a clean POJO-based application model.

9.5.4. Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply *totally different* transactional configuration to each of them. This is achieved by defining distinct `<aop:advisor/>` elements with differing `'pointcut'` and `'advice-ref'` attribute values.

Let's assume that all of your service layer classes are defined in a root `'x.y.service'` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `'Service'` have the default transactional configuration, you would write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
                                            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                              http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
                                             http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="serviceOperation"
                    expression="execution(* x.y.service..*Service.*(..))"/>

        <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

    </aop:config>

    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

    <!-- ... and these two beans won't -->
      <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right
package) -->
     <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in
'Service') -->

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

     <!-- other transaction infrastructure beans such as a PlatformTransactionManager
omitted... -->

</beans>
```

Find below an example of configuring two distinct beans with totally different transactional settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
                                            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                               http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
                                             http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="defaultServiceOperation"
                      expression="execution(* x.y.service.*Service.*(..))"/>

        <aop:pointcut id="noTxServiceOperation"
                      expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>

                          <aop:advisor   pointcut-ref="defaultServiceOperation"   advice-
ref="defaultTxAdvice"/>

        <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

    </aop:config>

     <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut)
-->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this bean will also be transactional, but with totally different transactional
settings -->
    <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <tx:advice id="noTxAdvice">
        <tx:attributes>
            <tx:method name="*" propagation="NEVER"/>
        </tx:attributes>
    </tx:advice>

     <!-- other transaction infrastructure beans such as a PlatformTransactionManager
omitted... -->

</beans>
```

9.5.5. `<tx:advice/>` settings

This section summarises the various transactional settings that can be specified using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are:

- The propagation setting is `REQUIRED`
- The isolation level is `DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can, of course, be changed; the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and`<tx:attributes/>` tags are summarized below:

Table 9.1. `<tx:method/>` settings

Attribute	Required?	Default	Description
`name`	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `'get*'`, `'handle*'`, `'on*Event'`, etc.
`propagation`	No	REQUIRED	The transaction propagation behavior
`isolation`	No	DEFAULT	The transaction isolation level
`timeout`	No	-1	The transaction timeout value (in seconds)
`read-only`	No	false	Is this transaction read-only?
`rollback-for`	No		The `Exception(s)` that will trigger rollback; comma-delimited. For example,`'com.foo.MyBusinessException,ServletException'`
`no-rollback-for`	No		The `Exception(s)` that will *not* trigger rollback; comma-delimited. For example,`'com.foo.MyBusinessException,ServletException'`

At the time of writing it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example `'com.foo.BusinessService.handlePayment'`.

9.5.6. Using `@Transactional`

Note

The functionality offered by the `@Transactional` annotation and the support classes is only available to you if you are using Java 5+.

In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the `@Transactional` annotation is best illustrated with an example, after which all of the details will be explained. Consider the following class definition:

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely *one* line of XML configuration, like so:

```
<!-- from the file 'context.xml' -->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- (this dependency is defined somewhere else) -->
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Note : You can actually omit the `'transaction-manager'` attribute in the `<tx:annotation-driven/>` tag if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the `PlatformTransactionManager` bean that you want to dependency inject has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

Method visibility and `@Transactional`

When using proxies, the `@Transactional` annotation should only be applied to methods with *public* visibility. If you do annotate protected, private or package-visible methods with the `@Transactional` annotation, no error will be raised, but the annotated method will not exhibit the configured transactional settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods.

The `@Transactional` annotation may be placed before an interface definition, a method on an interface, a class definition, or a *public* method on a class. However, please note that the mere presence of the `@Transactional` annotation is not enough to actually turn on the transactional behavior - the `@Transactional` annotation *is simply metadata* that can be consumed by something that is `@Transactional`-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the case of the above example, it is the presence of the `<tx:annotation-driven/>` element that *switches on* the transactional behavior.

The Spring team's recommendation is that you only annotate concrete classes with the `@Transactional` annotation, as opposed to annotating interfaces. You certainly can place the `@Transactional` annotation on an interface (or an interface method), but this will only work as you would expect it to if you are using interface-based proxies. The fact that annotations are *not inherited* means that if you are using class-based proxies then the transaction settings will not be recognised by the class-based proxying infrastructure and the object will not be wrapped in a transactional proxy (which would be decidedly *bad*). So please do take the Spring team's advice and only annotate concrete classes (and the methods of concrete classes) with the `@Transactional` annotation.

Note: Since this mechanism is based on proxies, only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object

calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with`@Transactional`!

Table 9.2. `<tx:annotation-driven/>` settings

Attribute	Required?	Default	Description
`transaction-manager`	No	transactionManager	The name of transaction manager to use. Only required if the name of the transaction manager is not`transactionManager`, as in the example above.
`proxy-target-class`	No		Controls what type of transactional proxies are created for classes annotated with the `@Transactional`annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled Section 6.6, "Proxying mechanisms" for a detailed examination of the different proxy types.)
`order`	No		Defines the order of the transaction advice that will be applied to beans annotated with<code>@Transactional</code>. More on the rules related to ordering of AOP advice can be found in the AOP chapter (see section Section 6.2.4.7, "Advice ordering"). Note that not specifying any ordering will leave the decision as to what order advice is run in to the AOP subsystem.

Note

The "`proxy-target-class`" attribute on the `<tx:annotation-driven/>` element controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled Section 6.6, "Proxying mechanisms" for a detailed examination of the different proxy types.)

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the `DefaultFooService`class is annotated at the class level with the settings for a read-only transaction, but the `@Transactional` annotation on the `updateFoo(Foo)` method in the same class takes precedence over the transactional settings defined at the class level.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
```

```
}
```

9.5.6.1. `@Transactional` settings

The `@Transactional` annotation is metadata that specifies that an interface, class, or method must have transactional semantics; for example, "*start a brand new read-only transaction when this method is invoked, suspending any existing transaction*". The default `@Transactional` settings are:

- The propagation setting is `PROPAGATION_REQUIRED`
- The isolation level is `ISOLATION_DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can, of course, be changed; the various properties of the `@Transactional` annotation are summarized in the following table:

Table 9.3. `@Transactional` properties

Property	Type	Description
`propagation`	enum: `Propagation`	optional propagation setting
`isolation`	enum: `Isolation`	optional isolation level
`readOnly`	boolean	read/write vs. read-only transaction
`timeout`	int (in seconds granularity)	the transaction timeout
`rollbackFor`	an array of `Class` objects, which must be derived from`Throwable`	an optional array of exception classes which **must** cause rollback
`rollbackForClassname`	an array of class names. Classes must be derived from`Throwable`	an optional array of names of exception classes that **must** cause rollback
`noRollbackFor`	an array of `Class` objects, which must be derived from`Throwable`	an optional array of exception classes that **must not** cause rollback.
`noRollbackForClassname`	an array of `String` class names, which must be derived from`Throwable`	an optional array of names of exception classes that **must not**cause rollback

At the time of writing it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example `'com.foo.BusinessService.handlePayment'`.

9.5.7. Advising transactional operations

Consider the situation where you would like to execute *both* transactional *and* (to keep things simple) some basic profiling advice. Hhow do you effect this in the context of using `<tx:annotation-driven/>`?

What we want to see when we invoke the `updateFoo(Foo)` method is:

- the configured profiling aspect starting up,
- then the transactional advice executing,
- then the method on the advised object executing
- then the transaction committing (we'll assume a sunny day scenario here),
- and then finally the profiling aspect reporting (somehow) exactly how long the whole transactional method invocation took

Note

This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). the chapter entitled Chapter 6,*Aspect Oriented Programming with Spring* for detailed coverage of the vario pieces of the following AOP configuration (and AOP in general).

Here is the code for a simple profiling aspect. The ordering of advice is controlled via the `Ordered` interface. For full details on advice ordering, see Section 6.2.4.7, "Advice ordering".

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

    private int order;

    // allows us to control the ordering of advice
    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    // this method is the around advice
    public Object profile(ProceedingJoinPoint call) throws Throwable {
        Object returnValue;
        StopWatch clock = new StopWatch(getClass().getName());
        try {
            clock.start(call.toShortString());
            returnValue = call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
        return returnValue;
    }
}
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this is the aspect -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <tx:annotation-driven transaction-manager="txManager" order="200"/>

    <aop:config>
        <!-- this advice will execute around the transactional advice -->
        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                          expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>
    </aop:config>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>

    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. The configuration of any number of additional aspects is effected in a similar fashion.

Finally, find below some example configuration for effecting the same setup as above, but using the purely XML declarative approach.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>
```

```
    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>

        <aop:pointcut id="entryPointMethod" expression="execution(*
x.y..*Service.*(..))"/>

        <!-- will execute after the profiling advice (c.f. the order attribute) -->
        <aop:advisor
                advice-ref="txAdvice"
                pointcut-ref="entryPointMethod"
                order="2"/> <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a DataSource and a
PlatformTransactionManager here -->

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. If we wanted the profiling advice to execute *after* the transactional advice on the way in, and *before* the transactional advice on the way out, then we would simply swap the value of the profiling aspect bean's `'order'` property such that it was higher than the transactional advice's order value.

The configuration of any number of additional aspects is achieved in a similar fashion.

9.5.8. Using `@Transactional` with AspectJ

It is also possible to use the Spring Framework's `@Transactional` support outside of a Spring container by means of an AspectJ aspect. To use this support you must first annotate your classes (and optionally your classes' methods with the `@Transactional` annotation, and then you must link (weave) your application with the `org.springframework.transaction.aspectj.AnnotationTransactionAspect` defined in the `spring-aspects.jar` file. The aspect must also be configured with a transaction manager. You could of course use the Spring Framework's IoC container to take care of dependency injecting the aspect, but since we're focusing here on applications running outside of a Spring container, we'll show you how to do it programmatically.

Note

Prior to continuing, you may well want to read the previous sections entitled Section 9.5 @Transactional" and Chapter 6, *Aspect Oriented Programming with Spring* respectively.

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new
DataSourceTransactionManager(getDataSource());
```

```
// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

Note

When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

The `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any method in the class.

The `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Any method may be annotated, regardless of visibility.

To weave your applications with the `AnnotationTransactionAspect` you must either build your application with AspectJ (see the AspectJ Development Guide) or use load-time weaving. See the section entitled Section 6.8.4, "Using AspectJ Load-time weaving (LTW) with Spring applications" for a discussion of load-time weaving with AspectJ.

9.6. Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

- Using the `TransactionTemplate`.
- Using a `PlatformTransactionManager` implementation directly.

If you are going to use programmatic transaction management, the Spring team generally recommend, namely that of using the `TransactionTemplate`). The second approach is similar to using the JTA `UserTransaction` API (although exception handling is less cumbersome).

9.6.1. Using the `TransactionTemplate`

The `TransactionTemplate` adopts the same approach as other Spring *templates* such as the `JdbcTemplate`. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of transactional resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do.

Note

As you will immediately see in the examples that follow, using the `TransactionTemplate` absolutely cou
Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is s
your development needs is a decision that you will have to make yourself.

Application code that must execute in a transactional context, and that will use the `TransactionTemplate` explicitly, looks like this. You, as an application developer, will write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that will contain all of the code that you need to have execute in the context of a transaction. You will then pass an instance of your custom `TransactionCallback` to the `execute(..)` method exposed on the`TransactionTemplate`.

```
public class SimpleService implements Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private final TransactionTemplate transactionTemplate;

    // use constructor-injection to supply the PlatformTransactionManager
    public SimpleService(PlatformTransactionManager transactionManager) {
        Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
```

```
        return transactionTemplate.execute(new TransactionCallback() {

            // the code in this method executes in a transactional context
            public Object doInTransaction(TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

If there is no return value, use the convenient `TransactionCallbackWithoutResult` class via an anonymous class like so:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can roll the transaction back by calling the `setRollbackOnly()` method on the supplied `TransactionStatus` object.

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessExeption ex) {
            status.setRollbackOnly();
        }
    }
});
```

9.6.1.1. Specifying transaction settings

Transaction settings such as the propagation mode, the isolation level, the timeout, and so forth can be set on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the default transactional settings. Find below an example of programmatically customizing the transactional settings for a specific `TransactionTemplate`.

```
public class SimpleService implements Service {

    private final TransactionTemplate transactionTemplate;

    public SimpleService(PlatformTransactionManager transactionManager) {
         Assert.notNull(transactionManager, "The 'transactionManager' argument must not
be null.");
        this.transactionTemplate = new TransactionTemplate(transactionManager);

        // the transaction settings can be set here explicitly if so desired
         this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ
_UNCOMMITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
        // and so forth...
    }
}
```

Find below an example of defining a `TransactionTemplate` with some custom transactional settings, using Spring XML configuration. The '`sharedTransactionTemplate`' can then be injected into as many services as are required.

```
<bean id="sharedTransactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
```

```
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
    <property name="timeout" value="30"/>
</bean>"
```

Finally, instances of the `TransactionTemplate` class are threadsafe, in that instances do not maintain any conversational state. `TransactionTemplate` instances*do* however maintain configuration state, so while a number of classes may choose to share a single instance of a `TransactionTemplate`, if a class needed to use a`TransactionTemplate` with different settings (for example, a different isolation level), then two distinct `TransactionTemplate` instances would need to be created and used.

9.6.2. Using the `PlatformTransactionManager`

You can also use the `org.springframework.transaction.PlatformTransactionManager` directly to manage your transaction. Simply pass the implementation of the `PlatformTransactionManager` you're using to your bean via a bean reference. Then, using the `TransactionDefinition` and`TransactionStatus` objects you can initiate transactions, rollback and commit.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can only be done programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

9.7. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the`TransactionTemplate` *may* be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

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. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

9.8. Application server-specific integration

Spring's transaction abstraction is generally application server agnostic. Additionally, Spring's `JtaTransactionManager` class, which can optionally perform a JNDI lookup for the JTA `UserTransaction` and `TransactionManager` objects, can be set to autodetect the location for the latter object, which varies by application server. Having access to the `TransactionManager` instance does allow enhanced transaction semantics. Please see the `JtaTransactionManager` Javadocs for more details.

9.8.1. BEA WebLogic

In a WebLogic 7.0, 8.1 or higher environment, you will generally prefer to use `WebLogicJtaTransactionManager` instead of the stock`JtaTransactionManager` class. This special WebLogic-specific subclass of the normal `JtaTransactionManager`. It supports the full power of Spring's transaction definitions in a WebLogic managed transaction environment, beyond standard JTA semantics: features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.

9.8.2. IBM WebSphere

In a WebSphere 5.1, 5.0 and 4 environment, you may wish to use Spring's `WebSphereTransactionManagerFactoryBean` class. This is a factory bean which retrieves the JTA `TransactionManager` in a WebSphere environment, which is done via WebSphere's `static` access methods. (These methods are different for each version of WebSphere.) Once the JTA `TransactionManager` instance has been obtained via this factory bean, Spring's `JtaTransactionManager` may be configured with a reference to it, for enhanced transaction semantics over the use of only the JTA `UserTransaction` object. Please see the Javadocs for full details.

9.9. Solutions to common problems

9.9.1. Use of the wrong transaction manager for a specific `DataSource`

You should take care to use the correct *`PlatformTransactionManager`* implementation for their requirements. It is important to understand how the the Spring Framework's transaction abstraction works with JTA global transactions. Used properly, there is no conflict here: the Spring Framework merely provides a straightforward and portable abstraction. If you are using global transactions, you *must* use the`org.springframework.transaction.jta.JtaTransactionManager` class (or an application server-specific subclass of it) for all your transactional operations. Otherwise the transaction infrastructure will attempt to perform local transactions on resources such as container `DataSource` instances. Such local transactions don't make sense, and a good application server will treat them as errors.

Explain about RowCallbackHandler and why it is used?

Ans: RowCallbackHandler interface is used by JdbcTemplate for processing rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of processing each row but don't need to worry about exception handling. SQLExceptions will be caught and handled by the calling JdbcTemplate. RowCallbackHandler object is typically stateful: It keeps the result state within the object, to be available for later inspection.

RowCallbackHandler interface has one method :

void processRow(ResultSet rs) :- Implementations must implement this method to process each row of data in the ResultSet.

Explain about BatchPreparedStatementSetter?

Ans: BatchPreparedStatementSetter interface sets values on a PreparedStatement provided by the JdbcTemplate class for each of a number of updates in a batch using the same SQL. Implementations are responsible for setting any necessary parameters. SQL with placeholders will already have been supplied. Implementations of BatchPreparedStatementSetter do not need to concern themselves with SQLExceptions that may be thrown from operations they attempt. The JdbcTemplate class will catch and handle SQLExceptions appropriately.

BatchPreparedStatementSetter has two method:

* int getBatchSize() :- Return the size of the batch.
* void setValues(PreparedStatement ps, int i) :-Set values on the given PreparedStatement.

Explain about PreparedStatementCreator?

Ans: The PreparedStatementCreator interface is a callback interfaces used by the JdbcTemplate class. This interface creates a PreparedStatement given a connection, provided by the JdbcTemplate class. Implementations are responsible for providing SQL and any necessary parameters. A PreparedStatementCreator should also implement the SqlProvider interface if it is able to provide the SQL it uses for PreparedStatement creation. This allows for better contextual information in case of exceptions.

It has one method:

PreparedStatement createPreparedStatement(Connection con) throws SQLException

Create a statement in this connection. Allows implementations to use PreparedStatements. The JdbcTemplate will close the created statement.

How can you create a DataSource connection pool?

Ans: To create a DataSource connection pool :

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driver">
        <value>${db.driver}</value>
```

```
    </property>
    <property name="url">
         <value>${db.url}</value>
    </property>
    <property name="username">
        <value>${db.username}</value>
    </property>
    <property name="password">
       <value>${db.password}</value>
    </property>
</bean>
```

How can you configure JNDI instead of datasource in spring applicationcontext.xml?
Ans: You can configure JNDI instead of datasource in spring applicationcontext.xml using "org.springframework.jndi.JndiObjectFactoryBean". For Example:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiName">
      <value>java:comp/env/jdbc/appfuse</value>
   </property>
</bean>
```

How does Spring supports DAO in hibernate?
Ans: Spring's HibernateDaoSupport class is a convenient super class for Hibernate DAOs. It has handy methods you can call to get a Hibernate Session, or a SessionFactory. The most convenient method is getHibernateTemplate(), which returns a HibernateTemplate. This template wraps Hibernate checked exceptions with runtime exceptions, allowing your DAO interfaces to be Hibernate exception-free.

What classes are used to Control the database connection in Spring's JDBC API?
Ans: The classes that are used to Control the database connection in Spring's JDBC API are :
* DataSourceUtils
* SmartDataSource
* AbstractDataSource
* SingleConnectionDataSource
* DriverManagerDataSource
* TransactionAwareDataSourceProxy
* DataSourceTransactionManager

Name the JDBC Core classes to control basic JDBC processing and error handling in Spring's JDBC API?
Ans: the JDBC Core classes to control basic JDBC processing and error handling in Spring's JDBC API are :
* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate (Java5)
* DataSource
* SQLExceptionTranslator

Ques: 12 What is NamedParameterJdbcTemplate class used for in Spring's JDBC API?
Ans: The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters (as opposed to programming JDBC statements using only classic placeholder ('?') arguments. The NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work.

Ques: 13 What is JdbcTemplate class used for in Spring's JDBC API?
Ans: The JdbcTemplate class is the central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions defined in the hierarchy of org.springframework.dao package. Code using the JdbcTemplate only need to implement callback interfaces, giving them a clearly defined contract.

The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The JdbcTemplate can be used within a DAO implementation via direct instantiation with a DataSource reference, or be configured in a Spring IOC container and given to DAOs as a bean reference.
Example:

```
JdbcTemplate  template = new JdbcTemplate(myDataSource);
```

A simple DAO class looks like this.

```
public class StudentDaoJdbc implements StudentDao {
        private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
   // more code here
}
```

List the Springs's DAO exception hierarchy?
Ans:
Springs's DAO exception hierarchy is :
* CleanupFailureAccessException
* DataAccessResourceFailureException
* DataIntegrityViolationException
* DataRetrievalFailureException
* DeadlockLoserDataAccessException
* IncorrectUpdateSemanticsDataAccessException
* InvalidDataAccessApiUsageException
* InvalidDataAccessResourceUsageException
* OptimisticLockingFailureException
* TypeMismatchDataAccessException
* UncategorizedDataAccessException

Ques: 15 Explain the Spring's DataAccessException?
Ans: Spring's org.springframework.dao.DataAccessException extends the extends NestedRuntimeException which in turn extends and RuntimeException. Hence DataAcccessException is a RuntimeException so there is no need to declare it in the method signature.

What is Metadata autoproxying?
Ans: Spring also supprts autoproxying driven by metadata. Metadata autoproxy configuration is determined by source level attributes and keeps AOP metadata with the source code that is being advised. This lets your code and configuration metadata in one place.
The most common use of metadata autoproxying is for declarative transaction support.

Ques: 18 Explain BeanNameAutoProxyCreator and DefaultAdvisorAutoProxyCreator classes?
Ans: BeanNameAutoProxyCreator generates proxies for beans that match a set of names. This name matching is similar to the NameMethodMatcherPointcut which allows for wildcard matching on both ends of the name. This is used to apply an aspect or a group of aspects uniformly across a set of beans that follow a similar naming conventions.
The DefaultAdvisorAutoProxyCreator is the more powerful autoproxy creator. Use this class to include it as a bean in your BeanFactory configuration. It implemnts the BeanPostProcessor interface. DefaultAdvisorAutoProxyCreator only works with advisors.

What is Autoproxying in Spring?
Ans: Spring has a facility of autoproxy that enables the container to generate proxies for us. We create autoproxy creator beans. Spring provides two classes to support this:
* BeanNameAutoProxyCreator
* DefaultAdvisorAutoProxyCreator

What is ProxyFactoryBean? Describe its properties?

Ans: ProxyFactoryBean creates proxied objects. Like other JavaBeans, it has properties that control its behaviour. ProxyFactoryBean properties are:
* target : The target bean of the proxy.
* proxyInterface : A list of interfaces that should be implemented by the proxy.
* interceptorNames : The bean names of the advice to be applied to the target.
* singleton : Whether the factory should return the same instance of the proxy for each getBean invocation.
* aopProxyFactory : The implemetation of the ProxyFactoryBean interface to be used.
* exposeProxy : Whether the target class should have access to the current proxy. This is done by calling AopContext.getCurrentProxy.
* frozen : Whether changes can be made to the proxy's advice once the factory is created.
* optimize : Whether to aggressively optimize generated proxies.
* proxyTargetClass : Whether to proxy the target class, rather than implementing an interface.

Ques: 21 How will you create Introductions in Spring?
Ans: The Introductions are little different from Spring advice. Advice are woven at different jointpoints surrounding a method invocation. Introductions affect an entire class by adding new methods and attributes to the adviced class.
Introductions allow you to build composite objects dynamically, affording you the same benefits as multiple inheritance.
Introductions are implemented through IntroductionMethodInterceptor, a subinterface of MethodInterceptor. This method adds one method:
boolean implementsInterface(Class intf);
This method returns true if the IntroductionMethodInterceptor is responsible for implementing the given interface.

Explain the Regular expression pointcuts?
Ans: The RegexpMethodPointcut lets you leverage the power of regular expressions to define your pointcuts. It uses Perl-style regular expressions to define the pattern that should match your intended methods.
Common regular expression symbol used are (.) matches any single character, (+) Matches the preceding character one or more times, (*) matches the preceding character zero or more, (\) Escapes any regular expression symbol.
For Example:
If we want to match all the setXxx methods, we need to use the pattern .*set.* (the first wildcard will match any preceding class name.

Explain the Around advice?
Ans: The MethodInterceptor provides the ability to do both before advice and after advice in one advice object:

```
public interface MethodInterceptor{
Object invoke(MethodInvocation invocation) throws Throwable;
}
```

There are two important difference between MethodInterceptor and the both before and after advice. First, the method interceptor implementation controls the target method invocation, and this invocation done by calling MethodInvocation.proceed(). Second, MethodInterceptor gives you control over what object is returned.

Explain the After returning advice?
Ans: We want to make sure that we thank our patrons after they make their purchase. To do this, we implement AfterReturningAdvice:

```
public interface AfterReturningAdvice{
  void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws
Throwable
  }
}
```

Like MethodBeforeAdvice, this advice gives you access to the method that was called, the arguments that were passed, and the target object.

Explian the Before advice?

Ans: Before the purchasing of our customer, we want to give them a warm greeting. For this, we need to add functionality before the method buySquishee() is executed. To accomplish this, we extend the MethodBeforeAdvice interface:

```
public interface MethodBeforeAdvice{
  void before(Method m, Object[] args, Object target) throws Throwable
}
```

This interface provides access to the target method, the arguments passed to this method, and the target object of the method invocation.

Name the interfaces that are used to create an Advice in Spring framework?
Ans: The interfaces that are used to create an Advice are:
* org.aopalliance.intercept.MethodInterceptor
* org.springframework.aop.BeforeAdvice
* org.springframework.aop.AfterReturningAdvice
* org.springframework.aop.ThrowsAdvice

Explain the Weaving process in Spring framework?
Ans: Weaving is the process of applying aspects to a to a target object to create a new proxied object. The aspects are woven into the target object at the specified joinpoints. The weaving can take place at several points in the target class's lifetime:
* Compile time : Aspects are woven in when the target class is compiled. This requires a special compiler.
* Classload time : Aspects are woven at the time of loading of the target class into the JVM. It is done by ClassLoader that enhances that target class's bytecode before the class is introduced into the application.
* Runtime : Sometimes aspects are woven in during the execution of the applicaion.

Define the AOP terminology?
Ans: Many terms used to define the AOP (Aspect Oriented Programming) features and they are part of the AOP language. These terms must be known to understand the AOP:
* Aspect : An aspect is the cross-cutting functionality you are implementing. It is the aspect, or area of your application you are modularizing. The most common example of aspect is logging. Logging is something that is required throughout an application.
* Joinpoint : A joinpoint is a point in the execution of the application where an aspect can be plugged in.
* Advice : Advice is the actual implementation of our aspect. It is advising your application of new behaviour.
* Pointcut : A pointcut defines at what joinpoints advice should be applied.
* Introduction : An introduction allows you to add new methods or attributes to existing classes.
* Target : A target is the class that is being advised.
* Proxy : A proxy is the object created after applying advice to the target object.
* Weaving : Weaving is the process of applying aspects to a target object to create a new proxied object.

Ques: 29 Define a Pointcut in Spring?
Ans: Spring defines pointcuts in terms of the class and method that is being advised. Advice is woven into the target class and its methods are based on their characteristics, such as class name and method signature. The core interface for Spring's pointcut framework is, naturally, the Pointcut interface.

Ques: 30 What is throws advice in Spring?
Ans: The ThrowsAdvice lets you definebehaviour should an exception occur. ThrowsAdvice is a marker interface and contains no method but need to be implemented. A class that implements this interface must have at least one method with either of the following two signatures:
* void afterThrowing(Throwable t)
* void afterThrowing(Method m, Object[] o, Object target, Throwable t)

What is an Advisor API in Spring
Ans: Advisor is an aspect that contains just a single advice object associated with a pointcut expression. A pointcut is something that defines at what joinpoints an advice should be applied.

Advices can be applied at any joinpoint that is supported by the AOP framework. These Pointcuts allow you to specify where the advice can be applied.
Any advisor can be used with any advice. The org.springframework.aop.support.DefaultPointcutAdvisor is the most commonly used advisor class. For example, it can be used with a MethodInterceptor, BeforeAdvice or ThrowsAdvice.

What is an Advice in Spring framework?
Ans: An Advice is a Spring bean. Advice is the implementation of an aspect. It is something like telling your application of a new behavior. Generally, and advice is inserted into an application at joinpoints.
An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to per-class or per-instance advice.
Advice types in Spring:
* Interception around advice
* Before advice
* Throws advice
* After Returning advice
* Introduction advice

Explain the Events and listener in Spring framework?
Ans:
All the Events are the subclasses of abstract class org.springframework.context.ApplicationEvent.
These are:
* ContextClosedEvent :- Published when the application context is closed.
* ContextRefereshedEvent :- Published when the application context is initialized or refereshed.
* RequestHandledEvent :- Published within a web application context when a request is handled.

If you want a bean to respond to application events, all you need to do is implement the org.springframework.context.ApplicationListener interface. This interface forces your bean to implement the onApplicationEvent() method, which is responsible for reaching to the application event:

```
public class RefreshListener implements ApplicaitonListener{
public void onApplicationEvent(ApplicationEvent e) {
  some code here//
}
```
You need to register it within the context:

```
<bean id="refreshListener"
    class="com.foo.RefreshListener"/>
```

Ques: 36 What are the various custom editors provided by the Spring Framework?
Ans:
The various custom editors provided by the Spring Framework are:
* PropertyEditor
* URLEditor
* ClassEditor
* CustomDateEditor
* FileEditor
* LocaleEditor
* StringArrayPropertyEditor
* StringTrimmerEditor

Ques: 37 How will you wire a string value to a property whose type is a non-string?
Ans: Yes. You can wire a string value to a property whose type is a non-string. The java.beans.PropertyEditor interface provides a means to customize how String values are mapped to non-String types. This interface is implemented by java.beans.PropertyEditorSupport that has two methods:
* getAsText() : returns the String representation of a property's value.
* setAsText(String value) : sets a bean property value from the String value passed in.

If you want to map the non-string property to a String value, the setAsText() method is called to perform the conversion.

How will you handle the ambiguities of autowiring in Spring?

Ans: When sutowiring using byType or constructor, it's possible that the container may find two or more beans whose type matches the property's type or the types of the constructor arguments. What will happen if there are ambiguous beans suitable for autowiring?
Spring can't sort out ambiguities and chooses to throw an exception raher than guess which bean you meant to wire in. If you encounter such ambiguities when autowiring, the best solution is not to autowire the bean.

What is autowiring in the Spring framework? Explain its type?
Ans:
You wire all of your bean's properties explicitly using the <property> element, However you can have Spring wire them automatically by setting the 'autowire' property on each <bean> that you want autowired:

```
<bean id="foo"
   class="com.Foo"
   autowire="autowireType"/>
```

There are four types of autowiring:
* byName :- Attempts to find a bean in the container whose name is the same as the name of the property being wired.
* byType :- Attempts to find a single bean in the container whose type matches the type of the property being wired. If no matching bean is found, the property will not be wired.
* constructor :- Tries to match up one or more beans in the container with the parameters of one of the constructors of the bean being wired.
* autodetect :- Attempts to autowire by constructor first and then using byType. Ambiguity is handled the same way as with constructor and byType wiring.

What is inner bean in Spring fremawork?
Ans:
You can embed a <bean> element directly in the <property> element. For Example:

```
<bean id="student"
 class="com.StudentImpl">
  <property name="course">
    <bean class="com.CourseImpl"/>
  </property>
</bean>
```

The drawback of wiring a bean reference in this manner is that you can't reuse the instance of StudentImpl anywhere else.

How can you reference a bean from another bean in Spring?
Ans:
We use the <property> element to set the properties that reference other beans. The <ref> subelement of <property> lets us do this:

```
<bean id="foo"
   class="com.Foo">
  <property name="bar">
    <ref bean="bar"/>
  </property>
</bean>
```

What are singleton beans and how can you create prototype beans?
Ans:
The singleton property of <bean> tells the context whether or not a bean is to be defined as a singleton. This attribute in bean tag named ‘singleton’ if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

```
<beans>
   <bean id="bar" class="com.act.Foo" singleton=”false”/>
</beans>
```

Prototyped beans are useful when you want the container to give a unique instance of a bean ezch time it is asked for, but you still want to configure one or more properties of that bean through Spring.

```
<bean id="student">
```

```
   class="com.StudentImpl"
   singleton="false">
```
A new instance of a prototype bean will be created each time getBean() is invoked with the bean's name.

What is Spring configuration file?
What is XMLBeanFactory in Spring framework?
Ans:
The org.springframework.beans.factory.xml.XmlBeanFactory is a class that implements the BeanFactory interface. To create an XmlBeanFactory, pass a java.io.InputStream to the constructor. The InputStream will provide the XML to the factory. For example, the following code snippet uses a java.io.FileInputStream to provide a bean definition XML file to XmlBeanFactory.

```
   BeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));
```

To retrieve the bean from a BeanFactory, call the getBean() method by passing the name of the bean you want to retrieve.

```
   MyBean myBean = (MyBean) factory.getBean("myBean");
```

How do add a bean in spring application?
Ans:
A bean can be added in spring application using :
```
<beans>
  <bean id="foo" class="com.Foo"/>
  <bean id="bar" class="com.Bar"/>
</beans>
```

in the Spring configuration file. In the bean tag the id attribute specifies the bean name and the class attribute specifies the fully qualified class name.

What do you mean by bean wiring in Spring container?
Ans: Piecing together beans within the Spring container is known as wiring. Wiring the beans means you are telling the Spring container what beans are needed and how the container should use dependency injection to tie them together. You should have some basic idea of XML for basic wiring of beans.

Ques: 47 What are the various steps involved in a Spring beans life cycle?
Ans: The bean factory performs several steps before a bean is ready is use:
 * Instantiate.
 * Populate properties.
 * The factory calls BeanNameAware's setBeanName() method.
 * The factory calls BeanFactoryAware's setBeanFactory() method.
 * PostProcessBeforeInitialization() method is called if BeanPostProcessor associated with the bean.
 * An init-method is called, if specified.
 * Finally, PostProcessAfterInitialization() is called if BeanPostProcessor associated with it.

What are features of Spring ?
Ans:
Features of Spring :
 * Lightweight :- Spring is lightweight in terms of size and overhead. The entire spring network can be distributed in a single JAR file that weights in at just over 1 MB.
 * Inverse of Control :- The basic concept of the Inversion of Control pattern 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 IOC container) is then responsible for looking it all.
 * Aspect-Oriented :- Spring supports the features of aspect oriented programmin approach that enables cohesive development.
 * Container :- Spring is a container because it manages the life cycle and configuration of application objects.

* 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.

What are the advantages of Spring framework?
Ans:
The advantages of Spring framework are:
* Spring is a layered architecture
* Spring Enables POJO (Plain Old Java Object) Programming. POJO programming enables continuous integration and testability.
* Dependency Injection and Inversion of Control Simplifies JDBC
* Spring is an open source framework available to all for commercial purpose.

Ques: 50 What is BeanFactory in Spring framework?
Ans: A BeanFactory is an implementation fo the factory design patterns. The BeanFactory is a root interface for accessing a Spring bean container. This is the basic client view of a bean container; further interfaces such as ListableBeanFactory and ConfigurableBeanFactory are available for specific purposes.
This interface is implemented by objects that hold a number of bean definitions, each uniquely identified by a String name. Depending on the bean definition, the factory will return either an independent instance of a contained object. The point of this approach is that the BeanFactory is a central registry of application components, and centralizes configuration of application components (no more do individual objects need to read properties files.
There are several implementations of BeanFactory in Spring. But the most useful is org.springframework.beans.factory.xml.XmlBeanFactory, which loads its bean based on the definitions contained in an XML file.

```
BeanFactory f = new XmlBeanFactory(new FileInputStream("beans.xml"));
```

Ques: 51 Compare Spring with Enterprise Java Beans?
Ans: As J2EE containers, both Spring and EJB offer the deevloper powerful features for developing applications. Comparing the features:
* Transaction management : EJB must use a JTA transaction manager and supports transactions that span remote method calls. Spring supports multiple transaction environment with JTA, Hibernate, JDO, JDBC, etc and does not support distributed transaction.
* Declarative Transaction support : EJB can define transaction in deployment descriptor and can't declaratively define rollback behaviour. Spring can define transaction in Spring confuguration file and can declaratively define rollback behaviour per method and per exception type.
* Persistence : EJB supports programmatic bean-managed persistence and declarative container managed persistence. Spring provides a framework for integrating with several persistence technologies, including JDBC, Hibernate, JDO, and iBATIS.
* Declarative security : EJB supports declarative security through users and roles and configured in the deployment descriptor. In Spring, no security implementation out of the box. Acegi provides the declarative security framework built on the top of Spring.
* Distributed computing : EJB provides container managed remote method calls. Spring provides proxying for remote calls via RMI, JAX-RPC, and web services.

What are the various approaches use by IoC pattern for decoupling of component in Spring framework?
Ans:
The IoC (Inversion of Control) pattern uses three different approaches in order to achieve decoupling of control of services from your components:
* Interface Injection : Your components explicitly conformed to a set of interfaces, with associated configuration metadata, in order to allow the framework to manage them correctly.
* Setter Injection : External metadata is used to configure how your components can be interacted with.
* Constructor Injection : Your components are registered with the framework, including the parameters to be used when the components are constructed, and the framework provides instances of the component with all of the specified facilities applied.

What is (AOP) aspect-oriented programming?

Ans: Aspect oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality.
AOP gives you aspects. Aspects enable modularization of concerns such as transaction management that cut across multiple types and objects.
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.
* To allow users to implement custom aspects, complementing their use of OOP with AOP.

Ques: 54 Explain the IoC (Inversion of Control) in Spring framework?
Ans: Inversion of control is at the heart of the Spring framework. The org.springframework.beans and org.springframework.context packages provide the basis for the Spring Framework's IoC container. The basic concept of the Inversion of Control pattern (dependency injection) is that programmers don't need to create your objects but describe how they should be created. In a IOC scenario, the container creates all the objects, connects them together by setting the necessary properties, and determines when methods will be invoked. The implementation pattern types for IOC used by SpringFramework are as follows:

* Dependencies can be assigned through JavaBeans properties (setter methods).
* Dependencies are provided as constructor parameters and are not exposed as JavaBeans Setter properties.

Ques: 55 What are the various modules of Spring?
Ans: The Spring framework is made up of seven well defined modules. These modules give you everything you need to develop enterprise-ready applications.
All of Spring's modules of are built on top of the core container. The container defines how beans are created, configured and managed-more of the nuts and bolts of Spring.
* The core container
* Application context module
* Spring's AOP module
* JDBC abstraction and DAO module
* Object/relation mapping integration module
* Spring's web module
* The Spring MVC framework

Spring is a lightweight container, with wrappers that make it easy to use many different services and frameworks. Lightweight containers accept any JavaBean, instead of specific types of components. Spring Web Services aims to facilitate contract-first SOAP service development, allowing for the creation of flexible web services using one of the many ways to manipulate XML payloads.

Question1: What is IOC or inversion of control?

Ans: This question is first step towards spring and mostly interviewer starts from this question. As the name implies **inversion of control** means now we have inverted the control of creating the object from our own using new operator to container or framework
Now it's the responsibility of container to create object as required.
We maintain one xml file where we configure our components, services, all the classes and their property. We just need to mention which service is needed by which component and container will create the object for us. This concept is known as **dependency injection** because all object dependency (resources) is injected into it by framework.

Example:
```
<bean id="createNewStock" class="springexample.stockMarket.CreateNewStockAccont">
      <property name="newBid"/>
  </bean>
```
In this example CreateNewStockAccont class contain getter and setter for newBid and container will instantiate newBid and set the value automatically when it is used.

Question 2: Explain Bean-LifeCycle.
Ans: Spring framework is based on IOC so we call it as IOC container also So Beans reside inside the IOC container beans are nothing but Plain old java object (POJO).
Following steps explain their lifecycle inside container.
1. Container will look the bean definition inside configuration file (eg.bean.xml).

2 using reflection container will create the object and if any property is defined inside the bean definition then it will also be set.
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 before the properties for the Bean are set.
6. If an init-method is specified for the bean, it will be called.
7. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
8. If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.

Question 3: what is Bean Factory, have you used XMLBean factory?
Ans: BeanFactory is factory Pattern which is based on IoC.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.
org.springframework.beans.factory.xml.XmlBeanFactory is used to creat bean instance defined in our xml file.

```
BeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));
Or
ClassPathResource resorce = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(resorce);
```

Question 4: what are the difference between BeanFactory and ApplicationContext in spring?
Ans: This one is very popular spring interview question and often asks in entry level interview. ApplicationContext is preferred way of using spring because of functionality provided by it and interviewer wanted to check whether you are familiar with it or not.

ApplicationContext.	**BeanFactory**
Here we can have more than one config files possible	In this only one config file or .xml file
Application contexts can publish events to beans that are registered as listeners	Doesn't support.
Support internationalization (I18N) messages	It's not
Support application lifecycle events, and validation.	Doesn't support.
Support many enterprise services such JNDI access, EJB integration, remoting	Doesn't support.

Question 5: What are different modules in spring?

Ans: spring have seven core modules
1. The Core container module
2. Application context module
3. AOP module (Aspect Oriented Programming)
4. JDBC abstraction and DAO module
5. O/R mapping integration module (Object/Relational)
6. Web module
7. MVC framework module

Question 6: What is difference between singleton and prototype bean?

Ans: This is another popular *spring interview questions* and an important concept to understand. Basically a bean has scopes which defines their existence on theapplication
Singleton: means single bean definition to a single object instance per Spring IoC container.
Prototype: means a single bean definition to any number of object instances.
Whatever beans we defined in spring framework are singleton beans. There is an attribute in bean tag named 'singleton' if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

```
<bean id="createNewStock"
  class="springexample.stockMarket.CreateNewStockAccont"singleton=”false”>
      <property name="newBid"/>
  </bean>
```

Question 7: What type of transaction Management spring support?

Ans: This spring interview questions is little difficult as compared to previous questions just because **transaction management** is a complex concept and not every developer familiar with it. Transaction management is critical in any applications that will interact with the database. Theapplication has to ensure that the data is consistent and the integrity of the data is maintained. Two type of transaction management is supported by spring
1. Programmatic transaction mgt.
2. Declarative transaction mgt.

Question 8: What is AOP?

Ans: The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules.
AOP is a programming technique that allows developer to modularize crosscutting concerns, that cuts across the typical divisions of responsibility, such as **logging and transaction management.**Spring AOP, aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation

Question 9: Explain Advice?

Ans: It’s an implementation of aspect; advice is inserted into an application at join points. Different types of advice include “around,” “before” and “after” advice

Question 10: What is joint Point and point cut?

Ans: This is not really a spring interview questions I would say an AOP one. AOP is another popular programming concept which complements OOPS. Join point is an opportunity within code for which we can apply an aspect. In Spring AOP, a join point always represents a method execution.

Pointcut: a predicate that matches join points. A point cut is something that defines at what joinpoints an advice should be applied

1. 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.
2. 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

3. 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.

4. 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.

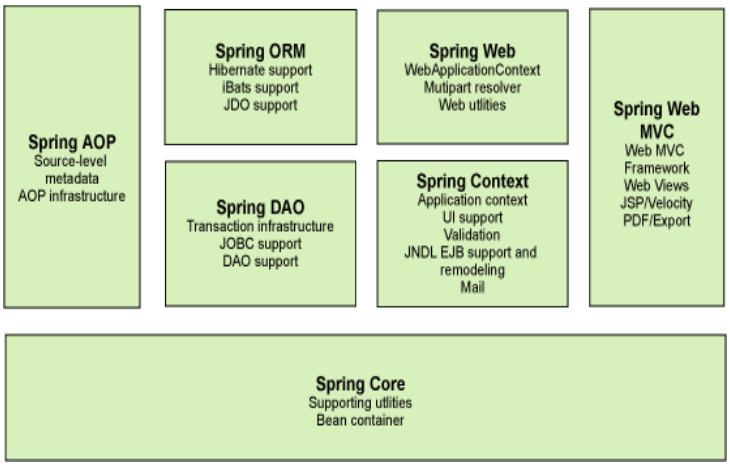
5. What are the advantages of Spring framework?

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.

6. 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

7. How many modules are there in Spring? What are they?

(**Roll over to view the Image**)



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.

8. 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.

9. 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.

10. 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.

11. 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

org.springframework.transaction
Interface TransactionDefinition

`public interface` **`TransactionDefinition`**

Interface that defines Spring-compliant transaction properties. Based on the propagation behavior definitions analogous to EJB CMT attributes.

Note that isolation level and timeout settings will not get applied unless an actual new transaction gets started. As only `PROPAGATION_REQUIRED`,`PROPAGATION_REQUIRES_NEW` and `PROPAGATION_NESTED` can cause that, it usually doesn't make sense to specify those settings in other cases. Furthermore, be aware that not all transaction managers will support those advanced features and thus might throw corresponding exceptions when given non-default values.

The `read-only flag` applies to any transaction context, whether backed by an actual resource transaction or operating non-transactionally at the resource level. In the latter case, the flag will only apply to managed resources within the application, such as a Hibernate `Session`.

Field Summary

`static int`	`ISOLATION_DEFAULT` Use the default isolation level of the underlying datastore.
`static int`	`ISOLATION_READ_COMMITTED` Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
`static int`	`ISOLATION_READ_UNCOMMITTED` Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
`static int`	`ISOLATION_REPEATABLE_READ` Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
`static int`	`ISOLATION_SERIALIZABLE` Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.
`static int`	`PROPAGATION_MANDATORY` Support a current transaction; throw an exception if no current transaction exists.
`static int`	`PROPAGATION_NESTED` Execute within a nested transaction if a current transaction exists, behave like `PROPAGATION_REQUIRED` else.
`static int`	`PROPAGATION_NEVER` Do not support a current transaction; throw an exception if a current transaction exists.
`static int`	`PROPAGATION_NOT_SUPPORTED` Do not support a current transaction; rather always execute non-transactionally.
`static int`	`PROPAGATION_REQUIRED` Support a current transaction; create a new one if none exists.
`static int`	`PROPAGATION_REQUIRES_NEW`

Create a new transaction, suspending the current transaction if one exists.

`static int` PROPAGATION_SUPPORTS
Support a current transaction; execute non-transactionally if none exists.

`static int` TIMEOUT_DEFAULT
Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

Method Summary

`int` getIsolationLevel()
Return the isolation level.

`String` getName()
Return the name of this transaction.

`int` getPropagationBehavior()
Return the propagation behavior.

`int` getTimeout()
Return the transaction timeout.

`boolean` isReadOnly()
Return whether to optimize as a read-only transaction.

Field Detail

PROPAGATION_REQUIRED

`static final int` **PROPAGATION_REQUIRED**

Support a current transaction; create a new one if none exists. Analogous to the EJB transaction attribute of the same name.

This is typically the default setting of a transaction definition, and typically defines a transaction synchronization scope.

See Also:
Constant Field Values

PROPAGATION_SUPPORTS

`static final int` **PROPAGATION_SUPPORTS**

Support a current transaction; execute non-transactionally if none exists. Analogous to the EJB transaction attribute of the same name.

NOTE: For transaction managers with transaction synchronization, `PROPAGATION_SUPPORTS` is slightly different from no transaction at all, as it defines a transaction scope that synchronization might apply to. As a consequence, the same resources (a JDBC `Connection`, a Hibernate `Session`, etc) will be shared for the entire specified scope. Note that the exact behavior depends on the actual synchronization configuration of the transaction manager!

In general, use `PROPAGATION_SUPPORTS` with care! In particular, do not rely on `PROPAGATION_REQUIRED` or `PROPAGATION_REQUIRES_NEW` *within* a`PROPAGATION_SUPPORTS` scope (which may lead to synchronization conflicts at runtime). If such nesting is unavoidable, make sure to configure your transaction manager appropriately (typically switching to "synchronization on actual transaction").

See Also:
AbstractPlatformTransactionManager.setTransactionSynchronization(int),AbstractPlatformTransactionManager.SYNCHRONIZATION_ON_ACTUAL_TRANSACTION, Constant Field Values

PROPAGATION_MANDATORY

`static final int` **`PROPAGATION_MANDATORY`**

Support a current transaction; throw an exception if no current transaction exists. Analogous to the EJB transaction attribute of the same name.

Note that transaction synchronization within a `PROPAGATION_MANDATORY` scope will always be driven by the surrounding transaction.

See Also:
Constant Field Values

PROPAGATION_REQUIRES_NEW

`static final int` **`PROPAGATION_REQUIRES_NEW`**

Create a new transaction, suspending the current transaction if one exists. Analogous to the EJB transaction attribute of the same name.

NOTE: Actual transaction suspension will not work out-of-the-box on all transaction managers. This in particular applies to `JtaTransactionManager`, which requires the `javax.transaction.TransactionManager` to be made available it to it (which is server-specific in standard J2EE).

A `PROPAGATION_REQUIRES_NEW` scope always defines its own transaction synchronizations. Existing synchronizations will be suspended and resumed appropriately.

See Also:
`JtaTransactionManager.setTransactionManager(javax.transaction.TransactionManager)`, Constant Field Values

PROPAGATION_NOT_SUPPORTED

`static final int` **`PROPAGATION_NOT_SUPPORTED`**

Do not support a current transaction; rather always execute non-transactionally. Analogous to the EJB transaction attribute of the same name.

NOTE: Actual transaction suspension will not work out-of-the-box on all transaction managers. This in particular applies to `JtaTransactionManager`, which requires the `javax.transaction.TransactionManager` to be made available it to it (which is server-specific in standard J2EE).

Note that transaction synchronization is *not* available within a `PROPAGATION_NOT_SUPPORTED` scope. Existing synchronizations will be suspended and resumed appropriately.

See Also:
`JtaTransactionManager.setTransactionManager(javax.transaction.TransactionManager)`, Constant Field Values

PROPAGATION_NEVER

`static final int` **`PROPAGATION_NEVER`**

Do not support a current transaction; throw an exception if a current transaction exists. Analogous to the EJB transaction attribute of the same name.

Note that transaction synchronization is *not* available within a `PROPAGATION_NEVER` scope.

See Also:
Constant Field Values

PROPAGATION_NESTED

`static final int` **PROPAGATION_NESTED**

Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else. There is no analogous feature in EJB.

NOTE: Actual creation of a nested transaction will only work on specific transaction managers. Out of the box, this only applies to the JDBCDataSourceTransactionManager when working on a JDBC 3.0 driver. Some JTA providers might support nested transactions as well.

See Also:
DataSourceTransactionManager, Constant Field Values

ISOLATION_DEFAULT

`static final int` **ISOLATION_DEFAULT**

Use the default isolation level of the underlying datastore. All other levels correspond to the JDBC isolation levels.

See Also:
Connection, Constant Field Values

ISOLATION_READ_UNCOMMITTED

`static final int` **ISOLATION_READ_UNCOMMITTED**

Indicates that dirty reads, non-repeatable reads and phantom reads can occur.

This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

See Also:
Connection.TRANSACTION_READ_UNCOMMITTED, Constant Field Values

---

ISOLATION_READ_COMMITTED

`static final int` **ISOLATION_READ_COMMITTED**

Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

This level only prohibits a transaction from reading a row with uncommitted changes in it.

See Also:
Connection.TRANSACTION_READ_COMMITTED, Constant Field Values

ISOLATION_REPEATABLE_READ

`static final int` **ISOLATION_REPEATABLE_READ**

Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction re-reads the row, getting different values the second time (a "non-repeatable read").

See Also:
Connection.TRANSACTION_REPEATABLE_READ, Constant Field Values

---

ISOLATION_SERIALIZABLE

`static final int` **ISOLATION_SERIALIZABLE**

Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

This level includes the prohibitions in ISOLATION_REPEATABLE_READ and further prohibits the situation where one transaction reads all rows that satisfy a WHERE condition, a second transaction inserts a row that satisfies that WHERE condition, and the first transaction re-reads for the same condition, retrieving the additional "phantom" row in the second read.

See Also:
Connection.TRANSACTION_SERIALIZABLE, Constant Field Values

TIMEOUT_DEFAULT

```
static final int TIMEOUT_DEFAULT
```

Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

See Also:
Constant Field Values

Method Detail

getPropagationBehavior

```
int getPropagationBehavior()
```

Return the propagation behavior.

Must return one of the PROPAGATION_XXX constants defined on this interface.

Returns:
the propagation behavior

See Also:
PROPAGATION_REQUIRED, TransactionSynchronizationManager.isActualTransactionActive()

---

getIsolationLevel

```
int getIsolationLevel()
```

Return the isolation level.

Must return one of the ISOLATION_XXX constants defined on this interface.

Only makes sense in combination with PROPAGATION_REQUIRED or PROPAGATION_REQUIRES_NEW.

Note that a transaction manager that does not support custom isolation levels will throw an exception when given any other level than ISOLATION_DEFAULT.

Returns:
the isolation level

---

getTimeout

```
int getTimeout()
```

Return the transaction timeout.

Must return a number of seconds, or TIMEOUT_DEFAULT.

Only makes sense in combination with PROPAGATION_REQUIRED or PROPAGATION_REQUIRES_NEW.

Note that a transaction manager that does not support timeouts will throw an exception when given any other timeout than TIMEOUT_DEFAULT.

Returns:
the transaction timeout

isReadOnly

`boolean` **`isReadOnly`**`()`

Return whether to optimize as a read-only transaction.

The read-only flag applies to any transaction context, whether backed by an actual resource transaction (`PROPAGATION_REQUIRED`/`PROPAGATION_REQUIRES_NEW`) or operating non-transactionally at the resource level (`PROPAGATION_SUPPORTS`). In the latter case, the flag will only apply to managed resources within the application, such as a Hibernate `Session`. << *

This just serves as a hint for the actual transaction subsystem; it will *not necessarily* cause failure of write access attempts. A transaction manager which cannot interpret the read-only hint will *not* throw an exception when asked for a read-only transaction.

Returns:
`true` if the transaction is to be optimized as read-only
See Also:
`TransactionSynchronization.beforeCommit(boolean)`, `TransactionSynchronizationManager.isCurrentTransactionReadOnly()`

---

getName

`String` **`getName`**`()`

Return the name of this transaction. Can be `null`.

This will be used as the transaction name to be shown in a transaction monitor, if applicable (for example, WebLogic's).

In case of Spring's declarative transactions, the exposed name will be the `fully-qualified class name + "." + method name` (by default).

Returns:
the name of this transaction
See Also:
`TransactionAspectSupport`, `TransactionSynchronizationManager.getCurrentTransactionName()`

Difference between Spring and Struts 1 :

*** Struts is a web framework only, Struts can be compare with the SpringMVC. And SpringMVC is subset of the Spring framework. So we can**

say that Sturts can be seen as the subset of the spring framework in functionality point of view.

*** Spring is a Application Framework while Struts is a Web Framework.**

*** Spring is a Layered architecture but Struts is not.**

*** Spring is Light weight Framework while Struts is Heavy weight Framework.**

*** Spring does not support Tag Library but Struts support Tag Library.**

*** Spring is loosely coupled while Struts is tightly coupled.**

*** Spring provides easy integrated with ORM technologies while in Struts we need to do coding manually.**

*** Struts easily integrate with other client side technologies. It is not easy in case of Spring.**

*** What Action class do in struts, Controller does in Spring. And action in Struts is a Abstract class but Controller in Spring is an interface, This**

is very good advantage of the spring.

*** Spring don't have any Action from, it bind the http form values directly into pojo.**

*** No ActionForm, binds directly to domain objects.**

*** ActionForward in struts is replace with the ModelAndView in Spring. Model component contain the business object to be displayed via view**

component.

*** Spring is a general purpose framework we can develop Console Application, Web Application, Enterprise Application etc but in Struts we**

can develop only Web Application.