Contents

[Java 2](#_Toc476087524)

[Tag interface = marker interface. 2](#_Toc476087525)

[How to create custom Annotations ? 2](#_Toc476087526)

[Java Reflection 3](#_Toc476087527)

[Ways in which database connection can be made ? 4](#_Toc476087528)

[How to make a class immutable ? 5](#_Toc476087529)

[JDBC Savepoint : 6](#_Toc476087530)

[ArrayList Vs LinkedList 6](#_Toc476087531)

[What are all the Different Ways to Create an Object in Java? 7](#_Toc476087532)

[Cloning 7](#_Toc476087533)

[Why does Hashtable not take null key? 8](#_Toc476087534)

[Java static nested class 8](#_Toc476087535)

[Fail Fast vs Fail Safe 8](#_Toc476087536)

[How  Fail  Fast Iterator  come to know that the internal structure is modified ? 8](#_Toc476087537)

[CopyOnWriteArrayList 9](#_Toc476087538)

[Synchronized HashMap vs ConcurrentHashMap 9](#_Toc476087539)

[Copy constructor 10](#_Toc476087540)

[Diamond problem in java ? 10](#_Toc476087541)

[Mandatory condition for key Object in HashMap 10](#_Toc476087542)

[What is the difference between creating String as new() and literal? 10](#_Toc476087543)

[Double Checked Locking on Singleton Class in Java 10](#_Toc476087544)

[What will be the problem if you don't override hashcode() method ? 11](#_Toc476087545)

[What is Classloader in Java? 11](#_Toc476087546)

[AtomicInteger in java ? 11](#_Toc476087547)

[WebService 12](#_Toc476087548)

[WSDL : 12](#_Toc476087549)

[Soap WS vs Rest WS 14](#_Toc476087550)

[ Diffrence between soap & restful WS 14](#_Toc476087551)

[How to expose a web service ? 14](#_Toc476087552)

[Spring 16](#_Toc476087553)

[Spring Core: 16](#_Toc476087554)

[ Bean Scope: 16](#_Toc476087555)

[ApplicationContext vs BeanFactory 16](#_Toc476087556)

[Bean With same name & Id ? 17](#_Toc476087557)

[Singleton vs prototype 17](#_Toc476087558)

[lazy-initialized 17](#_Toc476087559)

[Bean LifeCycle: 18](#_Toc476087560)

[Bean Post Processor 19](#_Toc476087561)

[Spring JDBC: 21](#_Toc476087562)

[Spring MVC: 22](#_Toc476087563)

[Spring Transaction: 22](#_Toc476087564)

[ ACID : 22](#_Toc476087565)

[ Transaction Attributes 24](#_Toc476087566)

[Declarative Transaction Management 26](#_Toc476087567)

[Programmatic Transaction Management 27](#_Toc476087568)

[Programmatic Vs Declarative Transaction Management 28](#_Toc476087569)

# Java

Tag interface = marker interface.

**Marker interface in Java** is interfaces with no field or methods.Ex : Serializable, Clonnable. Marker interface indicate, signal or a command to Compiler or [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html).   
One can also develop his own marker interface. Marker interface is a good way to classify code. You can create marker interface to logically divide your code and if you have your own tool than you can perform some pre-processing operation on those classes

**How Annotations are better than Marker Interfaces?** - They let you achieve the same purpose of conveying metadata about the class to its consumers without creating a separate type for it. Annotations are more powerful, too, letting programmers pass more sophisticated information to classes that "consume" it.

How to create custom Annotations ?

[**Java Annotations**](http://docs.oracle.com/javase/tutorial/java/annotations/index.html) allow us to add metadata information into our source code, although they are not a part of the program itself. Annotations were added to the java from JDK 5.

**What’s the use of Annotations?**

**1) Instructions to the compiler**: There are three built-in annotations available in Java (@Deprecated, @Override & @SuppressWarnings) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.

**2) Compile-time instructors**: Annotations can provide compile-time instructions to the compiler that can be further used by sofware build tools for generating code, XML files etc.

**3) Runtime instructions**: We can define annotations to be available at runtime which we can access using [java reflection](http://docs.oracle.com/javase/tutorial/reflect/) and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

**Where we can use annotations?**  
Annotations can be applied to the classes, interfaces, methods and fields.

**Creating Custom Annotations**

* Annotations are created by using @interface, followed by annotation name as shown in the below example.
* An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.
* All annotations extends java.lang.annotation.Annotation interface. Annotations cannot include any extends clause.

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Inherited;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Documented

@Target(ElementType.METHOD)

@Inherited

@Retention(RetentionPolicy.RUNTIME)

public @interface MyCustomAnnotation{

int studentAge() default 18;

String studentName();

String stuAddress();

String stuStream() default "CSE";

}

@MyCustomAnnotation(

studentName="Chaitanya",

stuAddress="Agra, India"

)

public class MyClass {

...

}

**@Documented** annotation indicates that elements using this annotation should be documented by JavaDoc.

**@Target** It specifies where we can use the annotation. Some possible values are TYPE, METHOD, CONSTRUCTOR, FIELD etc. If Target meta-annotation is not present, then annotation can be used on any program element.

**@Inherited** annotation signals that a custom annotation used in a class should be inherited by all of its sub classes.

**@Retention** It indicates how long annotations with the annotated type are to be retained. It takes RetentionPolicy argument whose Possible values are SOURCE, CLASS and RUNTIME

We will use **Reflection** to parse java annotations from a class. Please note that Annotation Retention Policy should be *RUNTIME* otherwise it’s information will not be available at runtime and we wont be able to fetch any data from it.

Java Reflection

**Java Reflection** provides ability to inspect and modify the runtime behavior of application. Reflection in Java is one of the advance topic of core java. Using java reflection we can inspect a class, [interface](http://www.journaldev.com/1601/interface-in-java), [enum](http://www.journaldev.com/716/java-enum), get their structure, methods and fields information at runtime even though class is not accessible at compile time. We can also use reflection to instantiate an object, invoke it’s methods, change field values.

Reflection in Java is a very powerful concept and it’s of little use in normal programming but it’s the backbone for most of the Java, J2EE frameworks. Some of the frameworks that use java reflection are:

1. **JUnit** – uses reflection to parse @Test annotation to get the test methods and then invoke it.
2. **Spring** – dependency injection, read more at [Spring Dependency Injection](http://www.journaldev.com/2410/spring-dependency-injection)
3. **Tomcat** web container to forward the request to correct module by parsing their web.xml files and request URI.
4. **Eclipse** auto completion of method names
5. [**Struts**](http://www.journaldev.com/dev/struts-2)
6. **Hibernate**

The list is endless and they all use java reflection because all these frameworks have no knowledge and access of user defined classes, interfaces, their methods etc.

We should not use reflection in normal programming where we already have access to the classes and interfaces because of following drawbacks.

* **Poor Performance** – Since java reflection resolve the types dynamically, it involves processing like scanning the classpath to find the class to load, causing slow performance.
* **Security Restrictions** – Reflection requires runtime permissions that might not be available for system running under security manager. This can cause you application to fail at runtime because of security manager.
* **Security Issues** – Using reflection we can access part of code that we are not supposed to access, for example we can access private fields of a class and change it’s value. This can be a serious security threat and cause your application to behave abnormally.
* **High Maintenance** – Reflection code is hard to understand and debug, also any issues with the code can’t be found at compile time because the classes might not be available, making it less flexible and hard to maintain.

Ways in which database connection can be made ?

**Java(JDBC) :**

//STEP 1. Import required packages

import java.sql.\*;

static final String DB\_URL = "jdbc:mysql://localhost/EMP";

//STEP 2: Register JDBC driver

Class.forName("com.mysql.jdbc.Driver");

//STEP 3: Open a connection

conn = DriverManager.getConnection(DB\_URL,USER,PASS);

//STEP 4: Execute a query

stmt = conn.createStatement();

String sql = "SELECT id, first, last, age FROM Employees";

ResultSet rs = stmt.executeQuery(sql);

**Spring:**

<bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource" destroy-method="close">

<property name="URL" value="${jdbc.url}" />

<property name="user" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

<property name="connectionCachingEnabled" value="true"/>

</bean>

**Spring MVC in web.xml:**

<resource-ref id=*"ResourceRef\_1136024978153"*>

<res-ref-name>OracleDS</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

<res-sharing-scope>Shareable</res-sharing-scope>

</resource-ref>

**Spring Transaction:**

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

OR

*// construct an appropriate transaction manager*

DataSourceTransactionManager txManager = **new** DataSourceTransactionManager(getDataSource());

Spring JDBC:

**public** **void** setDataSource(DataSource dataSource) {

**this.jdbcTemplate = new JdbcTemplate(dataSource);**

}

How to make a class immutable ?

Immutable classes are Java classes whose objects cannot be modified once created. Any modifications in Immutable object result in new object. For example is String is immutable in Java. Mostly Immutable are also final in Java, in order to prevent sub class from overriding methods in Java which can compromise Immutability. You can achieve same functionality by :

* + Making class final prevents the class to be extended.
  + making member as non final but private prevents direct access to the member variables.
  + Not modifying members except in parameterized constructor so that every time a new object is required to modify the members.
  + Provide only getters but no setters.
  + If you have mutable fields in your class, like List, or Date, making them final won't suffice. You should return a defensive copy from their getters, so that their state isn't mutated by calling methods.

private final List<Integer> lst = new ArrayList<Integer>();

**public** List<Integer> getLst() {

lst.add(1);

lst.add(2);

lst.add(3);

**return** **new** ArrayList<Integer>(**this**.lst);

}

JDBC Savepoint :

Sometimes a transaction can be group of multiple statements and we would like to rollback to a particular point in the transaction. JDBC Savepoint helps us in creating checkpoints in a transaction and we can rollback to that particular checkpoint. Any savepoint created for a transaction is automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

Snippet:

*con*.setAutoCommit(**false**);

// if code reached here, means main work is done successfully

stmt.executeUpdate("update employee set salary = 10000 where id = 10");

//setting up a savepoint to ensure things get rolled back till this line and the

//queries run before this are committed

savepoint = *con*.setSavepoint("EmployeeSavepoint");

**catch** (ClassNotFoundException | SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

**try** {

**if** (savepoint == **null**) {

*con*.rollback();

System.***out***.println("txn rolled back");

}

**else**{

System.***out***.println("txn rolled back till commit point");

*con*.rollback(savepoint);

*con*.commit();

}

ArrayList Vs LinkedList

1. **Search**: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

**2**. **Deletion**: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element). LinkedList element deletion is faster compared to ArrayList.

**Reason**: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

**3**. **Inserts Performance**: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove

**4**. **Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

**When to use LinkedList and when to use ArrayList?**

1) As explained above **the insert and remove operations** give good performance (O(1)) in **LinkedList** compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) **Search (get method) operations** are fast in **Arraylist** (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

What are all the Different Ways to Create an Object in Java?

1. Using new keyword

2. Using Class.forName()

CrunchifyObj object2 = (CrunchifyObj) Class.forName("crunchify.com.tutorial.CrunchifyObj").newInstance();

3. Using clone()

CrunchifyObj secondObject = new CrunchifyObj();

CrunchifyObj object3 = (CrunchifyObj) secondObject.clone();

4. Using newInstance() method

Object object4 = CrunchifyObj.class.getClassLoader().loadClass("crunchify.com.tutorial.CrunchifyObj").newInstance();

5. Using Object Deserialization. Object [Deserialization](http://crunchify.com/how-to-serialize-deserialize-list-of-objects-in-java-java-serialization-example/) is nothing but creating an object from its serialized form.

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("crunchify.txt"));

CrunchifyObj object5 = (CrunchifyObj) ois.readObject();

6. use the Constructor class from the [java.lang.reflect](http://crunchify.com/in-java-how-to-find-list-of-all-class-names-from-inside-jar-file-jar-class-finder-utility/).

Class clazz = CrunchifyObj.class;

Constructor crunchifyCon = clazz.getDeclaredConstructors()[0];

CrunchifyObj obj = (CrunchifyObj) crunchifyCon.newInstance();

Cloning

**Shallow Cloning:**

The default version of clone() method creates the shallow copy of an object. The shallow copy of an object will have exact copy of all the fields of original object. If original object has any references to other objects as fields, then only references of those objects are copied into clone object, copy of those objects are not created. That means any changes made to those objects through clone object will be reflected in original object or vice-versa. Shallow copy is not 100% disjoint from original object. Shallow copy is not 100% independent of original object.

**Deep Copy:**

Deep copy of an object will have exact copy of all the fields of original object just like shallow copy. But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone() method on them. That means clone object and original object will be 100% disjoint. They will be 100% independent of each other. Any changes made to clone object will not be reflected in original object or vice-versa.

|  |  |
| --- | --- |
| Shallow Copy | Deep Copy |
| Cloned Object and original object are not 100% disjoint. | Cloned Object and original object are 100% disjoint. |
| Any changes made to cloned object will be reflected in original object or vice versa. | Any changes made to cloned object will not be reflected in original object or vice versa. |
| Default version of clone method creates the shallow copy of an object. | To create the deep copy of an object, you have to override clone method. |
| Shallow copy is preferred if an object has only primitive fields. | Deep copy is preferred if an object has references to other objects as fields. |
| Shallow copy is fast and also less expensive. | Deep copy is slow and very expensive. |

[Why does Hashtable not take null key?](http://stackoverflow.com/questions/7556357/why-does-hashtable-not-take-null-key)

From the Hashtable [JavaDoc](http://download.oracle.com/javase/6/docs/api/java/util/Hashtable.html):

To successfully store and retrieve objects from a hashtable, the objects used

as keys must implement the hashCode method and the equals method.

In a nutshell, since null isn't an object, you can't call .equals() or .hashCode() on it, so the Hashtable can't compute a hash to use it as a key.

[HashMap](http://download.oracle.com/javase/6/docs/api/java/util/HashMap.html) is newer, and has more advanced capabilities, which are basically just an improvement on the Hashtable functionality. As such, when HashMap was created, it was specifically designed to handle null values as keys and handles them as a special case.Specifically, the use of null as a key is handled like this when issuing a .get(key):

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* + It can access static data members of outer class including private.
  + Static nested class cannot access non-static (instance) data member or method.

Fail Fast vs Fail Safe

**Fail fast iterator:**

While iterating through the collection , instantly throws Concurrent Modification Exception if there is structural modification  of the collection .

**Fail-fast** iterator can throw ConcurrentModificationException in two scenarios :

1. **Single Threaded Environment**

After the creation of the iterator , structure is modified at any time by any method other than iterator's own remove method.

1. **Multiple Threaded Environment**

If one thread is modifying the structure of the collection while other thread is iterating ove

How  Fail  Fast Iterator  come to know that the internal structure is modified ?  
Iterator read internal data structure (object array) directly . The internal data structure(i.e object array) should not be modified while iterating through the collection. To ensure this it maintains an internal  flag "mods" .Iterator checks the "mods" flagwhenever it gets the next value (using hasNext() method and next() method). Value of mods flag changes whenever there is an structural modification. Thus indicating iterator to throw ConcurrentModificationException.

**Fail Safe Iterator :**  
  
Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.Any structural modification done to the iterator affects the copied data structure.  So , original data structure remains  structurally unchanged .Hence , no ConcurrentModificationException throws by the fail safe iterator.  
  
Two  issues associated with Fail Safe Iterator are :  
1. Overhead of maintaining the copied data structure i.e memory.  
2.  Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Fail Fast Iterator** | **Fail Safe Iterator** | |
| Throw ConcurrentModification Exception | Yes | No |
| Clone object | No | Yes |
| Memory Overhead | No | Yes |
| Examples | HashMap,Vector,ArrayList,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

CopyOnWriteArrayList

CopyOnWriteArrayList is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin ConcurrentHashMap in Java. CopyOnWriteArrayList implements List interface like ArrayList, Vector and LinkedList but its a thread-safe collection and it achieves its thread-safety in a slightly different way than Vector or other thread-safe collection class. As name suggest CopyOnWriteArrayList creates copy of underlying ArrayList with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient if you have a List where Iteration outnumber mutation

Synchronized HashMap vs ConcurrentHashMap

**Synchronized HashMap**：

1. Each method is synchronized using an object level lock. SO the get and put methods on synchMap acquire a lock
2. Locking the entire collection is a performance overhead. While one thread holds on to the lock, no other thread can use the collection.

**ConcurrentHashMap**was introduced in JDK 5.

1. There is no locking at the object level,The locking is at a much finer granularity. For a ConcurrentHashMap, the locks may be at a hashmap bucket level.
2. The effect of lower level locking is that you can have concurrent readers and writers which is not possible for synchronized collections. This leads to much more scalability.
3. ConcurrentHashMap does not throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.

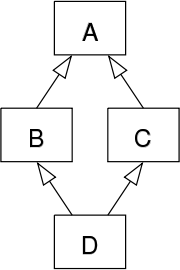
Copy constructor

There are many ways to copy the values of one object into another in java. They are:

* By constructor(Copy Constructor)
* By assigning the values of one object into another
* By clone() method of Object class

Diamond problem in java ?

The “diamond problem” is an ambiguity that can arise as a consequence of allowing multiple inheritance. It is a serious problem for languages (like C++) that allow for multiple inheritance of state. In Java, however, multiple inheritance is not allowed for classes, only for interfaces, and these do not contain state.



Mandatory condition for key Object in HashMap

In order to use any object as Key in HashMap, it must implements **equals** and **hashcode** method in Java.

What is the difference between creating String as new() and literal?

When we create string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

Double Checked Locking on Singleton Class in Java

Double checked locking of Singleton is a way to ensure only one instance of Singleton class is created through application life cycle. As name suggests, in double checked locking, code checks for an existing instance of Singleton class twice with and without locking to double ensure that no more than one instance of singleton gets created.

**Why you need Double checked Locking of Singleton Class?**

One of the common scenario, where a Singleton class breaks its contracts is multi-threading.  
This will bring us **to double checked locking pattern**, where only critical section of code is locked. Programmer call it double checked locking because there are two checks for \_instance == null, one without locking and other with locking (inside synchronized) block.  
  
Here is how double checked locking looks like in Java :

*public* *static* *Singleton* getInstanceDC() {

if (\_instance == **null**) { *// Single Checked*

*synchronized* (*Singleton*.class) {

if (\_instance == **null**) { *// Double checked*

\_instance = new *Singleton*();

}

}

}

return \_instance;

}

What will be the problem if you don't override hashcode() method ?

You will not be able to recover your object from hash Map if that is used as key in HashMap.  
Inheritance vs Composition

They are absolutely different. Inheritance is an *"is-a"* relationship. Composition is a *"has-a"*.

You do composition by having an instance of another class C as a field of your class, instead of extending C.

What is Classloader in Java?

Java Classloader is the program that loads byte code program into memory when we want to access any class. We can create our own classloader by extending ClassLoader class and overriding loadClass(String name) method. Learn more at [java classloader](http://www.journaldev.com/349/java-interview-questions-understanding-and-extending-java-classloader)

There are three types of built-in Class Loaders in Java:

1. **Bootstrap Class Loader** – It loads JDK internal classes, typically loads rt.jar and other core classes for example java.lang.\* package classes
2. **Extensions Class Loader** – It loads classes from the JDK extensions directory, usually $JAVA\_HOME/lib/ext directory.
3. **System Class Loader** – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.

AtomicInteger in java ?

Atomic operations are necessity in multi-threaded environment to avoid data inconsistency. In a counter operation you will notice that count value varies between 5,6,7,8. The reason is because **count++** is not an atomic operation. So by the time one threads read it's value and increment it by one, other thread has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using [Synchronization](http://www.journaldev.com/1061/thread-safety-in-java) but Java 5 java.util.concurrent.atomic provides wrapper classes for int and long that can be used to achieve this atomic operation without usage of Synchronization.

Here is the updated program that will always output count value as 8 because AtomicInteger method incrementAndGet() atomically increments the current value by one.

class ProcessingThread implements Runnable {

private AtomicInteger count = new AtomicInteger();

@Override

public void run() {

for (int i = 1; i < 5; i++) {

processSomething(i);

count.incrementAndGet();

}

}

Benefits of using Concurrency classes for atomic operation is that we don't need to worry about synchronization. This improves code readability and chance of errors are reduced. Also atomic operation concurrency classes are assumed to be more efficient that synchronization which involves locking resources.

# WebService

WSDL :

Figure : WSDL

XSD



Has

Has

Has

Points to

Defines

Has

Soap WS vs Rest WS

# Diffrence between soap & restful WS

|  |  |  |  |
| --- | --- | --- | --- |
|  | **SOAP** | **REST** | |
| 1) | SOAP is a **protocol**. | REST is an **architectural style**. |
| 2) | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| 3) | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| 4) | SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| 5) | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| 6) | SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |
| 7) | SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| 8) | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| 9) | SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| 10) | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |

How to expose a web service ?

** **

@WebService(endpointInterface = "com.iclnbi.iclnbiV200.wsdl.ICLNBIDataServicePortType",

targetNamespace = "http://www.ICLNBI.com/ICLNBI.xsd/WSDL",

serviceName = "ICL2NBIDataServiceService")

**Attributes of the javax.jws.WebService JWS Annotation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Data Type** | **Required?** |
| name | Name of the Web Service. Maps to the <wsdl:portType> element in the WSDL file.  Default value is the unqualified name of the Java class in the JWS file. | String | No. |
| targetNamespace | The XML namespace used for the WSDL and XML elements generated from this Web Service.  The default value is specified by the [JAX-RPC specification](http://java.sun.com/xml/jaxrpc/index.jsp). | String. | No. |
| serviceName | Service name of the Web Service. Maps to the <wsdl:service> element in the WSDL file.  Default value is the unqualified name of the Java class in the JWS file, appended with the string Service. | String | No. |
| wsdlLocation | Relative or absolute URL of a pre-defined WSDL file. If you specify this attribute, the jwsc Ant task does not generate a WSDL file, and returns an error if the JWS file is inconsistent with the port types and bindings in the WSDL file.   |  |  | | --- | --- | | **Note:** | The wsdlc Ant task uses this attribute when it generates the endpoint interface JWS file from a WSDL. Typically, users never use the attribute in their own JWS files. | | String. | No. |
| endpointInterface | Fully qualified name of an existing service endpoint interface file. If you specify this attribute, it is assumed that you have already created the endpoint interface file and it is in your CLASSPATH. | String. | No. |

The starting point for developing a JAX-WS web service is a Java class annotated with the javax.jws.WebService annotation. The WebService annotation defines the class as a web service endpoint.

A *service endpoint interface* (SEI)is a Java interface that declares the methods that a client can invoke on the service. An SEI is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines a SEI.

You may specify an explicit SEI by adding the **endpointInterface** element to the WebService annotation in the implementation class. You must then provide a SEI that defines the public methods made available in the endpoint implementation class.

* **Bottom Up** Approach : In the first, you start programming the classes and business logic as java code and then generate the **web service** contract (i.e. WSDL) from it.

**Top-down** means you start with a WSDL and then create all the necessary scaffolding in Java all the way down.

# Spring

Spring Core:

BeanFactory factory = **new** XmlBeanFactory(**new** FileSystemResource("spring.xml"));

ApplicationContext context = **new** ClassPathXmlApplicationContext("Spring.xml");

# Bean Scope:

|  |  |
| --- | --- |
| **Scope** | **Description** |
| singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| prototype | This scopes a single bean definition to have any number of object instances. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

ApplicationContext vs BeanFactory

The [ApplicationContext](http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html) builds on top of the BeanFactory (it's a subclass) and adds other functionality such as :

* easier 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 theWebApplicationContext, among other enhancements.
* The main usage scenario when you might prefer to use the BeanFactory is when memory usage is the greatest concern (such as in an applet where every last kilobyte counts), and you don't need all the features of the ApplicationContext.
* The [BeanFactory](http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html) is the actual *container* which instantiates, configures, and manages a number of beans.
* The location path or paths supplied to an ApplicationContext constructor are actually resource strings, and in simple form are treated appropriately to the specific context implementation (i.e. ClassPathXmlApplicationContext treats a simple location path as a classpath location), but may also be used with special prefixes to force loading of definitions from the classpath or a URL
* For many usage scenarios, user code will not have to instantiate the BeanFactory or ApplicationContext, since Spring Framework code will do it. For example, the web layer provides support code to load a Spring ApplicationContext automatically as part of the normal startup process of a J2EE web-app

Bean With same name & Id ?

If 2 bean have same name or id or id=name in xml then at runtime it throws exception as:

* + Name = name (Run Time)
  + Id = id(Compile time)
  + Name = id(Runtime)

Exception in thread "main" org.springframework.beans.factory.parsing.BeanDefinitionParsingException: Configuration problem: Bean name 'triangle' is already used in this <beans> element

Offending resource: class path resource [Spring.xml]

Singleton vs prototype

When a bean is a **singleton**, only one shared instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned. Beans are deployed in singleton mode by default, unless you specify otherwise.

The **non-singleton, prototype** mode of a bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is done. This is ideal for situations where for example each user needs an independent user object or something similar.

lazy-initialized

* The default behavior for ApplicationContext implementations is to eagerly pre-instantiate all singleton beans at startup i.e. an ApplicationContext will eagerly create and configure all of its [singleton](http://docs.spring.io/spring/docs/2.5.3/reference/beans.html#beans-factory-scopes-singleton) beans as part of its initialization process.
* If you do not want a singleton bean to be pre-instantiated when using an ApplicationContext, you can selectively control this by marking a bean definition as **lazy-initialized**.
* Assuming the bean is a singleton, and isn't configured for lazy initialisation, then it's created when the context is started up. getBean() just fishes it out.
* Lazy-init beans will only be initialised when first referenced, but this is not the default. Scoped beans (e.g. prototype-scoped) will also only be created when first referenced.

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" **lazy-init="true"**/>

* An **idref** element is simply a shorthand and error-proof way to set a property to the String *id* or *name* of another bean in the container.

<bean id="theTargetBean" class="..."/>

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

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

* It is used to set the value of the specified property to be a reference to another bean managed by the container
  + <ref bean="someBean"/>

The value of the bean attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean.

* + <ref local="someBean"/>

Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file

* + <ref parent="someBean"/>

Specifying the target bean by using the parent attribute allows a reference to be created to a bean which is in a parent BeanFactory (or ApplicationContext) of the current BeanFactory (or ApplicationContext).

Bean LifeCycle:

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

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



1. **InitializingBean and DisposableBean callback interfaces :**

The[org.springframework.beans.factory.InitializingBean](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/beans/factory/InitializingBean.html) interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The InitializingBean interface specifies a single method:

void afterPropertiesSet() throws Exception;

Similarly, implementing the [org.springframework.beans.factory.DisposableBean](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/beans/factory/DisposableBean.html) interface allows a bean to get a callback when the container containing it is destroyed.

void destroy() throws Exception;

1. **Other Aware interfaces for specific behavior**

Spring offers a range of Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean. Eg.

|  |  |  |
| --- | --- | --- |
| **AWARE INTERFACE** | **METHOD TO OVERRIDE** | **PURPOSE** |
| ApplicationContextAware | void setApplicationContext(ApplicationContext applicationContext) throws BeansException; | Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in. |
| ApplicationEventPublisherAware | void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher); | Set the ApplicationEventPublisher that this object runs in. |
| BeanClassLoaderAware | void setBeanClassLoader(ClassLoader classLoader); | Callback that supplies the bean class loader to a bean instance. |
| BeanFactoryAware | void setBeanFactory(BeanFactory beanFactory) throws BeansException; | Callback that supplies the owning factory to a bean instance. |
| BeanNameAware | void setBeanName(String name); | Set the name of the bean in the bean factory that created this bean. |

1. **Custom init() and destroy() methods in bean configuration file**

The default init and destroy methods in bean configuration file can be defined in two ways:

1. Bean local definition applicable to a single bean
2. Global definition applicable to all beans defined in beans context

**Local definition** is given as below.

|  |
| --- |
| <beans>      <bean id="demoBean" class="com.howtodoinjava.task.DemoBean" init-method="customInit" destroy-method="customDestroy"></bean>  </beans> |

Where **as global definition** is given as below. These methods will be invoked for all bean definitions given under tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

|  |
| --- |
| <beans default-init-method="customInit" default-destroy-method="customDestroy">          <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bean>  </beans> |

1. **@PostConstruct and @PreDestroy annotations**

Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.

1. @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object.
2. @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.

Bean Post Processor

* + A bean post-processor is a java class which implements the org.springframework.beans.factory.config.BeanPostProcessor interface, which consists of two callback methods(**postProcessBeforeInitialization , postProcessAfterInitialization**).
  + When such a class is registered as a post-processor with the BeanFactory, for each bean instance that is created by the BeanFactory, the post-processor will get a callback from the BeanFactory before any initialization methods (*afterPropertiesSet* and any declared init method) are called, and also afterwords.
  + An ApplicationContext will automatically detect any beans which are deployed into it which implement theBeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the factory on bean creation
  + The PropertyPlaceholderConfigurer, implemented as a bean factory post-processor, is used to externalize some property values from a BeanFactory definition, into another separate file in Java Properties format. This is useful to allow the person deploying an application to customize some key properties (for example database URLs, usernames and passwords), without the complexity or risk of modifying the main XML definition file or files for the BeanFactory.

Consider a fragment from a BeanFactory definition, where a DataSource with placeholder values is defined:

In the example below, a datasource is defined, and we will configure some properties from an external Properties file. At runtime, we will apply a PropertyPlaceholderConfigurer to the BeanFactory which will replace some properties of 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 actual values come from another file in Properties format:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>WEB-INF/properties/configuration.properties</value>

</list>

</property>

</bean>

* Having to specify all alias when the bean is actually defined is not always adequate however. It is sometimes desirable to introduce an alias for a bean which is define elsewhere. This may be done via a standalone alias element.

<alias name="fromName" alias="toName"/>

* Spring framework provide different ways through which we can provide post-initialization and pre-destroy methods in a spring bean life cycle.
  + By implementing **InitializingBean** and **DisposableBean** interfaces – Both these interfaces declare a single method where we can initialize/close resources in the bean. For post-initialization, we can implementInitializingBean interface and provide implementation of afterPropertiesSet() method. For pre-destroy, we can implement DisposableBean interface and provide implementation of destroy() method. These methods are the callback methods and similar to servlet listener implementations.
  + Providing **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

<bean name="myEmployeeService" class="com.journaldev.spring.service.MyEmployeeService"

init-method="init" destroy-method="destroy">

# Spring JDBC:

* The **JdbcTemplate** can be used within a DAO implementation through direct instantiation with a **DataSource** reference, or be configured in a Spring IoC container and given to DAOs as a bean reference.
* The **DataSource** should always be configured as a **bean** in the **Spring IoC container**. In the first case the bean is given to the service directly; in the second case it is given to the prepared template.
* Instances of the JdbcTemplate class are *threadsafe once configured*.
* This is important because it means that you can configure a single instance of a JdbcTemplateand then safely inject this *shared* reference into multiple DAOs (or repositories). The JdbcTemplate is stateful, in that it maintains a reference to a DataSource.
* A common practice when using the JdbcTemplate class is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes; the JdbcTemplate is created in the setter for the DataSource.

**public** **class** JdbcCorporateEventDao **implements** CorporateEventDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this.jdbcTemplate = new JdbcTemplate(dataSource);**

}

*// JDBC-backed implementations of the methods on the CorporateEventDao follow...*

}

* The org.springframework.jdbc.datasource package contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a Java EE container.
* Here is an example of how to configure a DriverManagerDataSource in Java code:

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();

dataSource.setDriverClassName("org.hsqldb.jdbcDriver");

dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");

dataSource.setUsername("sa");

dataSource.setPassword("");

# Spring MVC:

* Upon initialization of a DispatcherServlet, Spring MVC looks for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.
* The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files.

# Spring Transaction:

# ACID :

* + Atomicity : Ensures that all operations in a Transaction should happen or none should happen.
  + Consistent : The system should be left ina consistent state even if the transaction succeeds or fails.
  + Isolation : Transactions should be isolated from each other thus preventing concurrent read and write.
  + Durability : After successful transaction the system should persist the data permanently.

Spring does not directly manage transaction.It delegates the role to TransactionManagers of a particular platform you are using.

Traditionally, Java EE developers have had two choices for transaction management: *global* or *local* transactions, both of which have profound limitations.

**Global transactions** enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which 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 you *also* need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

**Local transactions** are resource-specific, such as a transaction associated with a JDBC connection. 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.

Spring resolves the disadvantages of global and local transactions. 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. Most users prefer declarative transaction management, which is recommended in most cases.

If you know JDBC transaction management([JDBC Savepoint](#JDBCSavePoint)), you might argue that we can get do it easily by setting auto-commit to false for the connection and based on the result of all the statements, either commit or rollback the transaction. Obviously we can do it, but that will result in a lot of boiler-plate code just for transaction management. Also the same code will present in all the places where we are looking for transaction management, causing tightly coupled and non-maintainable code.

It is not sufficient to tell you simply to annotate your classes with the @Transactional annotation, add @EnableTransactionManagement to your configuration, and then expect you to understand how it all works. 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*](http://docs.spring.io/autorepo/docs/spring/4.2.x/spring-framework-reference/html/aop.html#aop-understanding-aop-proxies), and that the transactional advice is driven by *metadata* (currently XML- or annotation-based).

The key to the Spring transaction abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by theorg.springframework.transaction.PlatformTransactionManager interface:

**public** **interface** PlatformTransactionManager {

TransactionStatus getTransaction(

TransactionDefinition definition) **throws** TransactionException;

**void** commit(TransactionStatus status) **throws** TransactionException;

**void** rollback(TransactionStatus status) **throws** TransactionException;

}

The getTransaction(..) method returns a **TransactionStatus** object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new transaction, or can represent an existing transaction if a matching transaction exists in the current call stack.

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct **PlatformTransactionManager** implementation is absolutely essential. You typically define this implementation through dependency injection.

**PlatformTransactionManager** implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on. The following examples show how you can define a local PlatformTransactionManager implementation. (This example works with plain JDBC.)

You define a JDBC 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 then have a reference to the DataSource definition. It will look like this:

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

For JTA:

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

For Hibernate:

<bean id="txManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

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

</bean>

Platform Specific Transaction Manager

JtaTransactionManager

JpaTransactionManager

DataSourceTransactionManager

HibernateTransactionManager

Hibernate

JDBC

JPA

JTA

# Transaction Attributes

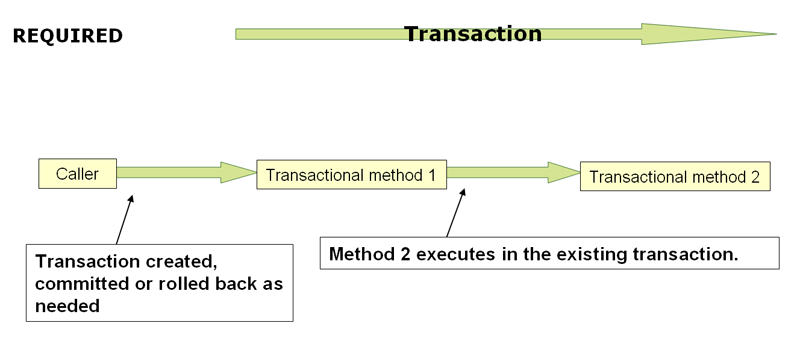
* + Propagation
  + Isolation
  + ReadOnly
  + Rollback Rules
  + Timeout
* **Propagation :**

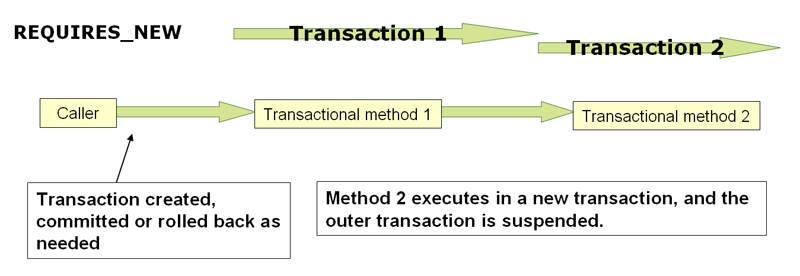
**Usage:**

@Transactional(propagation=Propagation.REQUIRED)

While dealing with Spring managed transactions the developer is able to specify how the transactions should behave in terms of propagation.Methods from distinct Spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions. This may lead to details like how does the inner transaction outcome result affects the outer transactions.

1. **Propagation\_Mandatory :** Method should run in a transaction and if nothing exists then throw an exception.
2. **Propagation\_NESTED :** The NESTED behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions.
3. **PROPAGATION\_NEVER :** Current method should not run in a transaction.If Transaction is there then it would throw an exception.
4. **PROPAGATION\_NOT\_SUPPORTED :** The NOT\_SUPPORTED behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.
5. **PROPAGATION\_REQUIRED :** Method should run in a transaction.If txn exists then it would run in that else create a new txn.
6. **PROPAGATION\_REQUIRES\_NEW:**Method should run in a new txn. Txn already exists thenit wold be suspended till the method finishes.
7. **PROPAGATION\_SUPPORTS :** The method need not run in a txn but if one exists then it supports the txn which is already in progress.





* **ISOLATION :**

**Usage:**

@Transactional(isolation=Isolation.READ\_COMMITTED)

Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions.

Possible scenarios which are avoided by using ISOLATIOn:

1. **Dirty Reads:** Occours when a transaction reads an uncommitted data.
2. **Non-Repeatable Reads :**Occours when a txn reads a record multiple times and gets different value each timedue to updation by some other thread.
3. **Phantom Read :** occours when two txn work on the same row where one updates and other reads.Reading txn gets a new data.

**Attributes:**

1. **ISOLATION\_DEFAULT :** Use the default isolation level of the underlying datastore.
2. **ISOLATION\_** [**READ\_COMMITTED**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#READ_COMMITTED) **:** A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
3. **ISOLATION\_** [**READ\_UNCOMMITTED**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#READ_UNCOMMITTED) **:** A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur.
4. **ISOLATION\_** [**REPEATABLE\_READ**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#REPEATABLE_READ)**:** A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5. **ISOLATION\_** [**SERIALIZABLE**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#SERIALIZABLE) **:** A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Dirty Reads** | **Non-Repeatable Reads** | **Phantom Read** |
| **DEFAULT** | Not Prevented | Not Prevented | Not Prevented |
| [READ\_UNCOMMITTED](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#READ_UNCOMMITTED) | Not Prevented | Not Prevented | Not Prevented |
| [READ\_COMMITTED](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#READ_COMMITTED) | Prevented | Not Prevented | Not Prevented |
| [REPEATABLE\_READ](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#REPEATABLE_READ) | Prevented | Prevented | Not Prevented |
| [SERIALIZABLE](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Isolation.html#SERIALIZABLE) | Prevented | Prevented | Prevented |

* **READ-ONLY:**

**B**y applying readonly to a transaction,the underlying datastore will apply some performance optimization to render data more faster.

* **TIMEOUT:**

By applying transaction timeout we can ensure that the txn times out after certain number of seconds.

* **Rollback:**

Rollback tells txnmgr to rollback a txn when an exception occours.

By default a txn gets rolled back when a runtime exeption occours.

Declarative Transaction Management

Links for ref : <http://www.tutorialspoint.com/spring/programmatic_management.htm>

Declarative transaction management approach allows you to manage the transaction with the help of configuration instead of hard coding in your source code. This means that you can separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions. The bean configuration will specify the methods to be transactional. Here are the steps associated with declarative transaction:

* We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a **pointcut** that matches all methods we wish to make transactional and reference the transactional advice.
* If a method name has been included in the transactional configuration then created advice will begin the transaction before calling the method.
* Target method will be executed in a *try / catch* block.
* If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.

<!-- We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a **pointcut** that matches all methods we wish to make transactional and reference the transactional advice -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="create"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut id="createOperation"

expression="execution(\* com.tutorialspoint.StudentJDBCTemplate.create(..))"/>

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

</aop:config>

<!-- Initialization for TransactionManager -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

* ***Advice***: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.
* ***Pointcut***: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Programmatic Transaction Management

Links for ref : <http://www.tutorialspoint.com/spring/declarative_management.htm>

Programmatic transaction management approach allows you to manage the transaction with the help of **programming in your source code**. That gives you extreme flexibility, but it is difficult to maintain.

We use ***PlatformTransactionManage****r* directly to implement programmatic approach to implement transactions.

To start a new transaction you need to have a instance of *TransactionDefinition* with the appropriate transaction attributes. For this example we will simply create an instance of*DefaultTransactionDefinition* to use the default transaction attributes.

Once the TransactionDefinition is created, you can start your transaction by calling *getTransaction()* method, which returns an instance of*TransactionStatus*. The *TransactionStatus* objects helps in tracking the current status of the transaction and finally, if everything goes fine, you can use*commit()* method of *PlatformTransactionManager* to commit the transaction, otherwise you can use *rollback()* to rollback the complete operation.

<!-- Initialization for TransactionManager -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

Programmatic Vs 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 theTransactionTemplate *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.

# Hibernate

Mapping

One-to-One Mapping :

4 ways to create one-to-one relationships between two entities

1. **First technique** is most widely used and uses a foreign key column in one to table.\

@OneToOne

@JoinColumn(name="ACCOUNT\_ID")

private AccountEntity account;

*To declare a side as not responsible for the relationship, the attribute* [**mappedBy**](http://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html#mappedBy%28%29) *is used*. mappedBy refers to the property name of the association on the owner side.

@OneToOne(mappedBy="account")

private EmployeeEntity employee;

Running above code creates desired schema in database and run these SQL queries.

Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)

Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?)

1. **Second technique** uses a rather known solution of having a third table to store mapping between first two tables.

In this technique, main annotation to be used is [**@JoinTable**](http://docs.oracle.com/javaee/5/api/javax/persistence/JoinTable.html). **This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables**. Lets see how it is used:

@OneToOne(cascade = CascadeType.ALL)

@JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),

inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))

private AccountEntity account;

Above code generates the below query:

Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)

Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into EMPLOYEE\_ACCCOUNT (ACCOUNT\_ID, EMPLOYEE\_ID) values (?, ?)

1. **Third technique** is something new which uses a common primary key value in both the tables. In this technique, hibernate will ensure that it will use a common primary key value in both the tables

@OneToOne(cascade = CascadeType.ALL)

@PrimaryKeyJoinColumn

private AccountEntity account;

Above code generates the below query:

Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)

Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?)

1. **Fourth technique** is by defining in xml**.**

File : Stock.hbm.xml

<one-to-one name="stockDetail" class="com.mkyong.stock.StockDetail" cascade="save-update"></one-to-one>

File : StockDetail.hbm.xml

<id name="stockId" type="java.lang.Integer">

<column name="STOCK\_ID" />

<generator class="**foreign**">

<param name="property">stock</param>

</generator>

</id>

<one-to-one name="stock" class="com.mkyong.stock.Stock"

constrained="true"></one-to-one>

The main difficulty in this **one-to-one** relationship is ensuring both are assigned the same primary key. In StockDetail.hbm.xml, a special **foreign identifier generator** is declared, it will know get the primary key value from STOCK table. With constrained=”true”, it ensure the Stock must exists.

One-to-Many Mapping

We have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee

This problem can be solved in two different ways

* 1. One is to have a foreign key column in account table i.e. EMPLOYEE\_ID. This column will refer to primary key of Employee table. This way no two accounts can be associated with multiple employees.

**EmployeeEntity.java**

@OneToMany(cascade=CascadeType.ALL)

@JoinColumn(name="EMPLOYEE\_ID")

private Set<AccountEntity> accounts;

**AccountEntity.java**

@ManyToOne

private EmployeeEntity employee;

Output:

Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)

Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)

Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)

Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?

Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?

Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?

* 1. Another approach is to have a common join table let’s say EMPLOYEE\_ACCOUNT. This table will have two column i.e. EMP\_ID which will be foreign key referring to primary key in EMPLOYEE table and similarly ACCOUNT\_ID which will be foreign key referring to primary key of ACCOUNT table.

**EmployeeEntity.java**

@OneToMany(cascade=CascadeType.ALL)

@JoinTable(name="EMPLOYEE\_ACCOUNT", joinColumns={@JoinColumn(name="EMPLOYEE\_ID", referencedColumnName="ID")}

, inverseJoinColumns={@JoinColumn(name="ACCOUNT\_ID", referencedColumnName="ID")})

private Set<AccountEntity> accounts;

Output:

Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)

Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)

Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?)

Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?)

* 1. XML Based:

This is a one-to-many relationship table design, a STOCK table has many occurrences STOCK\_DAILY\_RECORD table

Stock.hbm.xml

<set name="stockDailyRecords" table="stock\_daily\_record"

inverse="true" lazy="true" fetch="select">

<key>

<column name="STOCK\_ID" not-null="true" />

</key>

<one-to-many class="com.mkyong.stock.StockDailyRecord" />

</set>

StockDailyRecord.hbm.xml

<many-to-one name="stock" class="com.mkyong.stock.Stock" fetch="select">

<column name="STOCK\_ID" not-null="true" />

</many-to-one>

Many-to-Many Mapping

1. To demonstrate many to many mapping, we will associate two entities i.e. ReaderEntity and SubscriptionEntity. Owner entity is the entity which is responsible make making the association and maintaining it. In our case, I am making ReaderEntity the owner entity. @JoinTable annotation has been used to make this association.

**ReaderEntity.java**

**@**ManyToMany(cascade=CascadeType.ALL)

@JoinTable(name="READER\_SUBSCRIPTIONS",joinColumns={@JoinColumn(referencedColumnName="ID")}, inverseJoinColumns={@JoinColumn(referencedColumnName="ID")})

private Set<SubscriptionEntity> subscriptions;

Our mapped entity is SubscriptionEntity which is mapped to ReaderEntity using “**mappedBy**” attribute.

**SubscriptionEntity.java**

@ManyToMany(mappedBy="subscriptions")

private Set<ReaderEntity> readers;

Output:

Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)

Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)

Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)

Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)

Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)

Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)

Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)

1. XML Based:

This is a many-to-many relationship table design, a STOCK table has more than one CATEGORY, and CATEGORY can belong to more than one STOCK, the relationship is linked with a third table called STOCK\_CATEGORY.

Stock.hbm.xml

<set name="categories" table="stock\_category"

inverse="false" lazy="true" fetch="select" cascade="all" >

<key>

<column name="STOCK\_ID" not-null="true" />

</key>

<many-to-many entity-name="com.mkyong.stock.Category">

<column name="CATEGORY\_ID" not-null="true" />

</many-to-many>

</set>

Category.hbm.xml

<set name="stocks" table="stock\_category" inverse="true" lazy="true" fetch="select">

<key>

<column name="CATEGORY\_ID" not-null="true" />

</key>

<many-to-many entity-name="com.mkyong.stock.Stock">

<column name="STOCK\_ID" not-null="true" />

</many-to-many>

</set>

### Mapping collection in mapping file

There are many subelements of **<class>** elements to map the collection. They are **<list>**, **<bag>**, **<set>** and **<map>**.

**<list>**

**T**here are three subelements used in the list:

* **<key>** element is used to define the foreign key in this table based on the Question class identifier.
* **<index>** element is used to identify the type. List and Map are indexed collection.
* **<element>** is used to define the element of the collection
* But if collection stores entity reference (another class objects), we need to define **<one-to-many>** or **<many-to-many>** element.

**<bag>**,

The bag is just like List but it doesn't require index element.

**<set>**

The set element doesn't require index element. The one difference between List and Set is that, it stores only unique values.

**<map>**.

As we know, list and map are index-based collections. In case of map, index column works as the key and element column works as the value.

Hibernate Caching

1. **First Level Cache**: Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.

Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

* Hibernate First Level cache is enabled by default, there are no configurations needed for this.
* Hibernate first level cache is session specific, that’s why when we are getting the same data in same session there is no query fired whereas in other session query is fired to load the data.
* Hibernate first level cache can have old values, as you can see above that I have put my program to sleep for 10 seconds and in that time I updated the value (name from Pankaj to PankajK) in database but it didn’t get reflected in the same session. But in other session, we got the updated value.
* We can use session evict() method to remove a single object from the hibernate first level cache.
* We can use session clear() method to clear the cache i.e delete all the objects from the cache.
* We can use session contains() method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false.
* Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it’s necessary to clear the cache at certain intervals to avoid memory issues.

1. **Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.

Different strategies for caching an object.:

* **Read Only**: This caching strategy should be used for persistent objects that will always read but never updated. It’s good for reading and caching application configuration and other static data that are never updated.
* **Read Write**: It’s good for persistent objects that can be updated by the hibernate application. However if the data is updated either through backend or other applications, then there is no way hibernate will know about it and data might be stale. So while using this strategy, make sure you are using Hibernate API for updating the data.
* **Nonrestricted Read Write**: If the application only occasionally needs to update data and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.
* **Transactional**: The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.

For hibernate second level cache, we would need to add **ehcache-core** and **hibernate-ehcache** dependencies in our application.

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<!-- enable second level cache and query cache -->

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

<property name="hibernate.cache.use\_query\_cache">true</property>

<property name="net.sf.ehcache.configurationResourceName">/myehcache.xml</property>

1. **hibernate.cache.region.factory\_class** is used to define the Factory class for Second level caching, I am using org.hibernate.cache.ehcache.EhCacheRegionFactory for this. If you want the factory class to be singleton, you should use org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory class.

If you are using Hibernate 3, corresponding classes will be net.sf.ehcache.hibernate.EhCacheRegionFactory and net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory.

1. **hibernate.cache.use\_second\_level\_cache** is used to enable the second level cache.
2. **hibernate.cache.use\_query\_cache** is used to enable the query cache, without it HQL queries results will not be cached.
3. **net.sf.ehcache.configurationResourceName** is used to define the EHCache configuration file location, it’s an optional parameter and if it’s not present EHCache will try to locate ehcache.xml file in the application classpath.
4. **Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

.