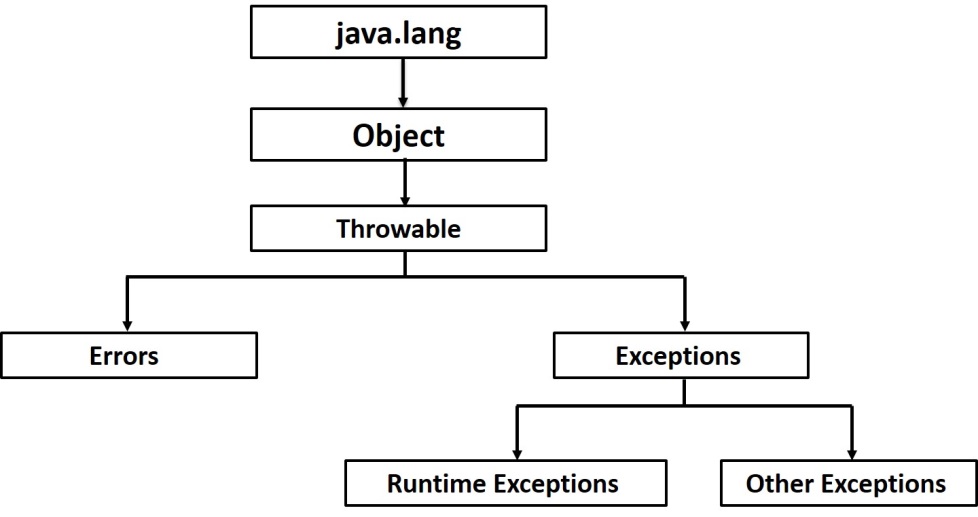
# ORACLE

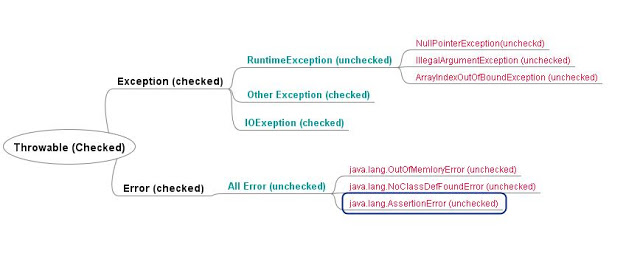
1. Different indexes, joins
2. Oracle Sequence and cache in sequence
3. Procedures vs functions

# JAVA

### Different type of Exceptions

* **Error** - OutOfMemoryError**,** VirtualMachineError
* **checked Exception (Compile time)** – IOException, [SQLException](http://javarevisited.blogspot.com/2012/01/javasqlsqlexception-invalid-column.html), NoSuchFieldException
* **unchecked Exception (runtime)** - NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException





### Comparator and Comparable

|  |  |  |
| --- | --- | --- |
| Parameter | Comparable | Comparator |
| Sorting logic | Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects | Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc. |
| Implementation | Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id | Class whose objects to be sorted do not need to implement this interface.Some other class can implement this interface. E.g.-CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id |
| Sorting method | int compareTo(Object o1) This method compares this object with o1 object and returns a integer.Its value has following meaning 1. positive – this object is greater than o1 2. zero – this object equals to o1 3. negative – this object is less than o1 | int compare(Object o1,Object o2) This method compares o1 and o2 objects. and returns a integer.Its value has following meaning. 1. positive – o1 is greater than o2 2. zero – o1 equals to o2 3. negative – o1 is less than o1 |
| Calling method | Collections.sort(List) Here objects will be sorted on the basis of CompareTo method | Collections.sort(List, Comparator) Here objects will be sorted on the basis of Compare method in Comparator |
| Package | Java.lang.Comparable | Java.util.Comparator |

### Different between arraylist and linked list

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| 1) ArrayList internally uses dynamic array to store the elements. | LinkedList internally uses doubly linked list to store the elements. |
| 2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| 4) ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |

### Why generics are useful

* Allows you to write code/use library methods which are type-safe, i.e. a List<string> is guaranteed to be a list of strings.
* As a result of generics being used the compiler can perform compile-time checks on code for type safety, i.e. are you trying to put an int into that list of strings?
* Using an ArrayList would cause that to be a less transparent runtime error.
* Faster than using objects as it either avoids boxing/unboxing or casting from

objects to the required reference type.

* Allows you to write code which is applicable to many types with the same underlying behaviour, i.e. a Dictionary<string, int> uses the same underlying code as a Dictionary<DateTime, double>; using generics, the framework team only had to write one piece of code to achieve both results with the aforementioned advantages too.

### Custom annotation

<http://www.mkyong.com/java/java-custom-annotations-example/>

This @interface tells Java this is a custom annotation. Later, you can annotate it on method level like this @Test(enable=false).

Test.java

package com.mkyong.test.core;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD) //can use in method only.

public @interface Test {

//should ignore this test?

public boolean enabled() default true;

}

Note

Method declarations must not have any parameters or a throws clause. Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types.

### Connection pooling advantages

* Performance. Connecting to the database is expensive and slow. Pooled connections can be left physically connected to the database, and

shared amongst the various components that need database access. That way the connection cost is paid for once and amortized across all the consuming components.

* Diagnostics. If you have one sub-system responsible for connecting to the database, it becomes easier to diagnose and analyze database connection usage.
* Maintainability. Again, if you have one sub-system responsible for handing out database connections, your code will be easier to maintain than if each component connected to the database itself.

### Marker interface (Can we create marker interface – Yes )

**Marker interface in Java** is interfaces with no field or methods or in simple word **empty interface in java is called marker interface**. Example of market interface is Serializable, Clonnable and Remote interface.

Yes We can create our own Marker interface..See following one...

interface Marker{ }

class MyException extends Exception {

public MyException(String s){

super(s);

}

}

class A {

void m1() throws MyException{

if((this instanceof Marker)){

System.out.println("successfull");

}

else {

throw new MyException("Must implement interface Marker ");

}

}

}

public class CustomMarkerInterfaceExample extends A implements Marker

{ // if this class will not implement Marker, throw exception

public static void main(String[] args) {

CustomMarkerInterfaceExample a= new CustomMarkerInterfaceExample();

try {

a.m1();

} catch (MyException e) {

System.out.println(e);

}

}

}

### Different way of creating new object in Java

There are four different ways to create objects in java:

A. **Using new keyword**  
This is the most common way to create an object in java. Almost 99% of objects are created in this way.

MyObject object = new MyObject();

B. **Using Class.forName()**  
If we know the name of the class & if it has a public default constructor we can create an object in this way.

MyObject object = (MyObject) Class.forName("subin.rnd.MyObject").newInstance();

C. **Using clone()**  
The clone() can be used to create a copy of an existing object.

MyObject anotherObject = new MyObject();

MyObject object = (MyObject) anotherObject.clone();

D. **Using object deserialization**  
Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream inStream = new ObjectInputStream(anInputStream );

MyObject object = (MyObject) inStream.readObject();

### Create Immutable object

<http://javarevisited.blogspot.in/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html>

1. State of immutable object can not be modified after construction, any modification should result in new immutable object.

2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. Object should be final in order to restrict sub-class for altering immutability of parent class.

public final class Contacts {

private final String name;

private final String mobile;

public Contacts(String name, String mobile) {

this.name = name;

this.mobile = mobile;

}

public String getName(){

return name;

}

public String getMobile(){

return mobile;

}

}

### Create Singleton class and implementation

<http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html#ixzz3hwhaSh58>

1.We can make constructor as private. So that We can not create an object outside of the class.

2.This property is useful to create singleton class in java.

3.Singleton pattern helps us to keep only one instance of a class at any time.

4.The purpose of singleton is to control object creation by keeping private constructor.

class Singleton

{ private volatile static Singleton \_instance;

private Singleton() { // preventing Singleton object instantiation from outside }

/\* \* 1st version: creates multiple instance if two thread access \* this method simultaneously \*/

public static Singleton getInstance()

{ if (\_instance == null) {

\_instance = new Singleton();

}

return \_instance;

/\* \* 2nd version : this definitely thread-safe and only \* creates one instance of Singleton on concurrent environment \* but unnecessarily expensive due to cost of synchronization \* at every call. \*/

public static synchronized Singleton getInstanceTS() {

if (\_instance == null) {

\_instance = new Singleton();

}

return \_instance;

}

/\* \* 3rd version : An implementation of double checked locking of Singleton. \* Intention is to minimize cost of synchronization and improve performance, \* by only locking critical section of code, the code which creates instance of Singleton class. \* By the way this is still broken, if we don't make \_instance volatile, as another thread can \* see a half initialized instance of Singleton. \*/

public static Singleton getInstanceDC() {

if (\_instance == null) {

synchronized (Singleton.class) {

if (\_instance == null)

{ \_instance = new Singleton(); }

}

}

return \_instance;

}

}

### Can we override static methods – No

Static methods can not be overridden because there is nothing to override, as they would be two different methods. For example

static class Class1 {

public static int Method1(){

return 0;

}

}

static class Class2 extends Class1 {

public static int Method1(){

return 1;

}

}

public static class Main {

public static void main(String[] args){

//Must explicitly chose Method1 from Class1 or Class2

Class1.Method1();

Class2.Method1();

}

}

And yes static methods can be overloaded just like any other method.

### Can abstract classes have constructors

Yes when we define a **class** to be an **Abstract Class** it cannot be instantiated but that **does** not mean an **Abstract class** cannot **have** a **constructor**. Each **abstract class** must **have** a concrete subclass which **will** implement the **abstract** methods of that **abstract class**

abstract class Product {

int multiplyBy;

public Product( int multiplyBy ) {

this.multiplyBy = multiplyBy;

}

public int mutiply(int val) {

return multiplyBy \* val;

}

}

class TimesTwo extends Product {

public TimesTwo() {

super(2);

}

}

class TimesWhat extends Product {

public TimesWhat(int what) {

super(what);

}

}

### How to inherit methods from two class in a third class

Short answer: You can't. Java only has multiple inheritance of interfaces.

Slightly longer answer: If you make sure the methods you care about are in interfaces, then you can have a class that implements the interfaces, and then delegates to instances of the "super classes":

interface Noisy {

void makeNoise();

}

interface Vehicle {

void go(int distance);

}

class Truck implements Vehicle {

...

}

class Siren implements Noisy {

...

}

class Ambulance extends Truck implements Noisy {

private Siren siren = new Siren();

public makeNoise() {

siren.makeNoise();

}

...

}

### Data structure used by arraylist

ArrayList uses an Array of Object to store the data internally.

When you initialize an arraylist, an array of size 10 (default capacity) is created and any element added to the arrayList is actually added to this array. 10 is the default size and it can be passed as a parameter while initializing the arrayList.

When adding a new element, if the array is full, then a new array of double the initial size is created and the last array is copied to this new array, so that now there is empty spaces for the new element to be added.

Since, the underlying data-structure used is an array, it is fairly easy to add a new element to the arrayList as it is added to the end of the list. When an element is to be added anywhere else, say the beginning, then all the elements shall have to move one position to the right to create an empty space at the beginning for the new element to be added. This process is time-consuming (linear-time). But the Advantage of ArrayList is that retrieving an element at any position is very fast (constant-time), as underlying it is simply using an array of objects.

### Final Keyword .. how final objects can be modified - <http://javarevisited.blogspot.in/2011/12/final-variable-method-class-java.html>

1. **Final keyword** can be applied to member variable, local variable, method or [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

2. **Final member variable** must be initialized at the time of declaration or inside constructor, failure to do so will result in compilation error.

3. You can not reassign value to *final variable in Java*.

4. **Local final variable** must be initializing during declaration.

5. Only final variable is accessible inside anonymous class in Java.

6. **Final method** can not be [overridden in Java](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html).

7. **Final class** can not be inheritable in Java.

8. **Final** is different than **finally** keyword which is used on [Exception handling in Java](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html).

9. Final should not be confused with finalize() method which is declared in object class and called before an object is garbage collected by JVM.

10. All variable declared inside java interface are implicitly final.

11. **Final and abstract** are two opposite keyword and a final class can not be [abstract in java](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html).

12. Final methods are bonded during compile time also called static binding.

13. *Final variables* which is not initialized during declaration are called blank final variable and must be initialized on all constructor either explicitly or by calling this(). Failure to do so compiler will complain as "*final variable (name) might not be initialized*".

14. Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.

15. As per Java code convention **final variables are treated as constant** and written in all Caps e.g.

private final int COUNT=10;

16. Making a collection reference variable final means only reference can not be changed but you can add, remove or change object inside collection. For example:

private final List Loans = new ArrayList();

list.add(“home loan”); **//valid**

list.add("personal loan"); **//valid**

loans = new Vector(); **//not valid**

### Transaction management in Java

<http://examples.javacodegeeks.com/core-java/sql/jdbc-transaction-management-example/>

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

**Atomicity** means either all successful or none.

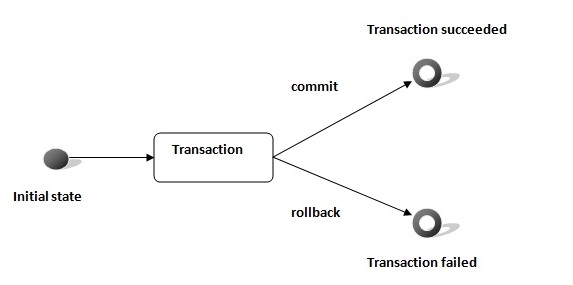
Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

**Advantage of Transaction Mangaement**

**fast performance** It makes the performance fast because database is hit at the time of commit.



In JDBC, **Connection interface** provides methods to manage transaction.

|  |  |
| --- | --- |
| **Method** | **Description** |
| void setAutoCommit(boolean status) | It is true bydefault means each transaction is committed bydefault. |
| void commit() | commits the transaction. |
| void rollback() | cancels the transaction. |

### Serialization UID

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members.

* An object is serializable only if its class or its superclass implements the Serializable (or Externalizable) interface.
* An object is serializable (itself implements the Serializable interface) even if its superclass is not. However, the first superclass in the hierarchy of the serializable class, that does not implements Serializable interface, MUST have a no-arg constructor. If this is violated, readObject() will produce a java.io.InvalidClassException in runtime.
* The no-arg contructor of every non-serializable superclass will run when an object is deserialized. However, the deserialized objects? constructor does not run when it is deserialized.
* The class must be visible at the point of serialization.
* All primitive types are serializable.
* Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
* Static fields (with static modifier) are Not serialized.
* If member vairiables of a serializable object reference to a non-serializable object, the code will compile but a RumtimeException will be thrown.

### What is encapsulation

**Encapsulation** is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, **encapsulation** is also referred to as data hiding.

### Different class loaders in java :

<http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

### Java Enum and Singleton Pattern

Following are some reasons which make sense to me for using Enum to implement Singleton pattern in Java. By the way If you like articles on design pattern than you can also check my post on [Builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) and [Decorator design pattern](http://javarevisited.blogspot.sg/2011/11/decorator-design-pattern-java-example.html) .

**1) Enum Singletons are easy to write**

This is by far biggest advantage, if you have been writing Singletons prior ot Java 5 than you know that even with double checked locking you can have more than one instances. though that issue is fixed with Java memory model improvement and gurantee provided by volatile variables from Java 5 onwards but it still tricky to write for many beginners. compared to double checked locking with synchronization Enum singletons are cake walk. If you don't believe than just compare below code for conventional singleton with double checked locking and Enum Singletons:

**Singleton using Enum in Java**

This is the way we generally declare Enum Singleton , it may contain instace variable and instance method but for sake of simplicity I haven’t used any, just beware that if you are using any instance method than you need to ensure thread-safety of that method if at all it affect the state of object. By default creation of Enum instance is thread safe but any other method on Enum is programmers responsibility.

***/\*\*  
\* Singleton pattern example using Java Enumj  
\*/***

**public** **enum** EasySingleton{  
    INSTANCE;  
}

You can acess it by EasySingleton.INSTANCE, much easier than calling getInstance() method on Singleton.

**Singleton example with double checked locking**

Below code is an example of double checked locking in Singleton pattern, here getInstance() method checks two times to see whether INSTANCE is null or not and that’s why it’s called double checked locking pattern, remember that double checked locking is broker before Java 5 but with the guranteed of [volatile variable in Java 5](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) memory model, it should work perfectly.

***/\*\*  
\* Singleton pattern example with Double checked Locking  
\*/***

**public** **class** DoubleCheckedLockingSingleton{  
     **private** **volatile** DoubleCheckedLockingSingleton INSTANCE;  
    
     **private** DoubleCheckedLockingSingleton(){}  
    
     **public** DoubleCheckedLockingSingleton getInstance(){  
         **if**(INSTANCE == **null**){  
            **synchronized**(DoubleCheckedLockingSingleton.**class**){  
                *//double checking Singleton instance*  
                **if**(INSTANCE == **null**){  
                    INSTANCE = **new** DoubleCheckedLockingSingleton();  
                }  
            }  
         }  
         **return** INSTANCE;  
     }  
}

You can call DoubleCheckedLockingSingleton.getInstance() to get access of this Singleton class.

Now Just look at amount of code needed to create a **lazy loaded thread-safe Singleton**. With Enum Singleton pattern you can have that in one line because creation of Enum instance is [thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) and guranteed by JVM.

People may argue that there are better way to write Singleton instead of Double checked locking approach but every approach has their own advantages and disadvantages like I mostly prefer static field Singleton intialized during classloading as shwon in below example, but keep in mind that is not a **lazy loaded Singleton**:

**Singleton pattern with static factory method**

This is one of my favorite method to impelemnt Singleton pattern in Java, Since Singleton instance is [static](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) and [final variable](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) it initialized when class is first loaded into memeory so creation of instance is inherently thread-safe.

***/\*\*  
\* Singleton pattern example with static factory method  
\*/***  
  
**public** **class** Singleton{  
    *//initailzed during class loading*  
    **private** **static** **final** Singleton INSTANCE = **new** Singleton();  
    
    *//to prevent creating another instance of Singleton*  
    **private** Singleton(){}  
  
    **public** **static** Singleton getSingleton(){  
        **return** INSTANCE;  
    }  
}

You can call Singleton.getSingleton() to get access of this class.

**2) Enum Singletons handled Serialization by themselves**

Another problem with conventional Singletons are that once you implement [serializable interface](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html) they are no longer remain Singleton because readObject() method always return a new instance just like constructor in Java. you can avoid that by using readResolve() method and discarding newly created instance by replacing with Singeton as shwon in below example :

*//readResolve to prevent another instance of Singleton*  
    **private** Object readResolve(){  
        **return** INSTANCE;  
    }

This can become even more complex if your Singleton Class maintains state, as you need to make them [transient](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html), but witn **Enum Singleton**, Serialization is guarnateed by JVM.

**3) Creation of Enum instance is thread-safe**

As stated in point 1 since creatino of Enum instance is thread-safe by default you don't need to worry about double checked locking.

In summary, given the **Serialzation and thraead-safety guaranteed** and with couple of line of code enum Singleton pattern is best way to create Singleton in Java 5 world. you can still use other popular methods if you feel so but I still have to find a *convincing reason not to use Enum as Singleto*n, let me know if you got any.

Read more: <http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html#ixzz47Trd1BQk>

# J2EE

### Servlets and JSP’s are thread safe

In general Servlet/JSP are not thread safe.

Following are the guidelines:

1. Application (ServletContext) attributes, session(HttpSession) attributes, Servlet Instance variables and Servlet class variables are "not" thread safe.

2. Local variables (variables defined in a servlet methods) and request (ServletRequest) attributes are thread safe.

3. If you implement SingleThreadModel interface in a servlet then you can make Servlet Instance variables also thread safe. (In a JSP you implement SingleThreadModel by specifying page directive <%@ page isThreadSafe="false" %>

Note: SingleThreadModel interface is deprecated in Servlet API 2.4

but if somebody ask u how u make ur servlet/jsp thred safe, not explicitly asking u context, session, request attibute, what will be ur answer.

again the page directive <%@page isThreadSafe="true"%> which is by default say that jsp is thread safe.

Yes, the default is true. But, that is assuming that you will take care of actually making it thread-safe. The most appropriate thing to do is synchronize on any process/logic that will make your servlets (including servlets resulting from JSP) NOT thread-safe. (i.e. any logic that can be accessed by more than one thread simultaneously)

### Session attributes are stored at server or client?

Session attributes are stored at server and we can get the value using :

session.setAttribute( "username", firstname );

Object username = (session.getAttribute("username"));

HttpSession session = request.getSession(false);

**if** (session == null){

response.sendRedirect("/dbproj/login.jsp");

} **else** {

response.sendRedirect("/dbproj/menu.jsp");

}

### Cookies **-**

<http://www.tutorialspoint.com/jsp/jsp_cookies_handling.htm>

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying returning users:

Server script sends a set of cookies to the browser. For example name, age, or identification number etc.

Browser stores this information on local machine for future use.

When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

### EJB transaction attributes and trasactions

<http://www.tutorialspoint.com/ejb/ejb_transactions.htm>

### EJB stateless and stateful beans

**Stateless Session Beans**

A session bean represents work performed by a single client. That work can be performed within a single method invocation, or it may span multiple method invocations. If the work does span more than one method, the object must retain the user’s object state across the method calls, and a stateful session bean would therefore be required.

Generally, stateless beans are intended to perform individual operations automatically and don’t maintain state across method invocations. They’re also amorphous, in that any client can use any instance of a stateless bean at any time at the container’s discretion. They are the lightest weight and easiest to manage of the various EJB component configurations.

**Stateful Session Beans**

Stateful session beans maintain state both within and between transactions. Each stateful session bean is therefore associated with a specific client. Containers are able to save and retrieve a bean’s state automatically while managing instance pools (as opposed to bean pools) of stateful session beans.

Stateful session beans maintain data consistency by updating their fields each time a transaction is committed. To keep informed of changes in transaction status, a stateful session bean implements the SessionSynchronization interface. The container calls methods of this interface while it initiates and completes transactions involving the bean.

Session beans, whether stateful or stateless, are not designed to be persistent. The data maintained by stateful session beans is intended to be transitional. It is used solely for a particular session with a particular client. A stateful session bean instance typically can’t survive system failures and other destructive events. While a session bean has a container-provided identity (called its handle), that identity passes when the client removes the session bean at the end of a session. If a client needs to revive a stateful session bean that has disappeared, it must provide its own means to reconstruct the bean’s state.

**Stateful vs. Stateless Session Beans**

A stateful session bean will maintain a conversational state with a client. The state of the session is maintained for the duration of the conversation between the client and the stateful session bean. When the client removes the stateful session bean, its session ends and the state is destroyed. The transient nature of the state of the stateful session bean should not be problematic for either the client or the bean, because once the conversation between the client and the stateful session bean ends, neither the client nor the stateful session bean should have any use for the state.

A stateless session bean will not maintain conversational states for specific clients longer than the period of an individual method invocation. Instance variables used by a method of a stateless bean may have a state, but only for the duration of the method invocation. After a method has finished running either successfully or unsuccessfully, the states of all its instance variables are dropped. The transient nature of this state gives the stateless session bean beneficial attributes, such as the following:

Bean pooling Any stateless session bean method instance that is not currently invoked is equally available to be called by an EJB container or application server to service the request of a client. This allows the EJB container to pool stateless bean instances and increase performance.

Scalability Because stateless session beans are able to service multiple clients, they tend to be more scalable when applications have a large number of clients. When compared to stateful session beans, stateless session beans usually require less instantiation.

Performance An EJB container will never move a stateless session bean from RAM out to a secondary storage, which it may do with a stateful session bean; therefore, stateless session beans may offer greater performance than stateful session beans.

Since no explicit mapping exists between multiple clients and stateless bean instances, the EJB container is free to service any client’s request with any available instance. Even though the client calls the create() and remove() methods of the stateless session bean, making it appear that the client is controlling the lifecycle of an EJB, it is actually the EJB container that is handling the create() and remove() methods without necessarily instantiating or destroying an EJB instance

### EJB callbacks

- <http://www.tutorialspoint.com/ejb/ejb_callbacks.htm>

Callback is a mechanism by which life cycle of an enterprise bean can be intercepted. EJB 3.0 specification has specified callbacks for which callback handler methods are to be created. EJB Container calls these callbacks. We can define callback methods in the ejb class itself or in a separate class. EJB 3.0 has provided many annotations for callbacks

Following is the list of callback annotations for stateless bean.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@PostConstruct** | method is invoked when a bean is created for the first time |
| **@PreDestroy** | method is invoked when a bean is removed from the bean pool or is destroyed. |

Following is the list of callback annotations for stateful bean.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@PostConstruct** | method is invoked when a bean is created for the first time |
| **@PreDestroy** | method is invoked when a bean is removed from the bean pool or is destroyed. |
| **@PostActivate** | method is invoked when a bean is loaded to be used. |
| **@PrePassivate** | method is invoked when a bean is put back to bean pool. |

Following is the list of callback annotations for message driven bean.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@PostConstruct** | method is invoked when a bean is created for the first time |
| **@PreDestroy** | method is invoked when a bean is removed from the bean pool or is destroyed. |

Following is the list of callback annotations for entity bean.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@PrePersist** | method is invoked when an entity is created in database. |
| **@PostPersist** | method is invoked after an entity is created in database. |
| **@PreRemove** | method is invoked when an entity is deleted from the database. |
| **@PostRemove** | method is invoked after an entity is deleted from the database. |
| **@PreUpdate** | method is invoked before an entity is to be updated in the database. |
| **@PostLoad** | method is invoked when a record is fetched from database and loaded into the entity. |

### Durable queues in JMS

- <http://docs.oracle.com/cd/E13222_01/wls/docs103/jms/fund.html>

A durable subscription is one where the publications for a subscriber are stored by the messaging provider when that subscriber is not running. Once the subscriber becomes active, these stored messages will be delivered to that subscriber. For non-Durable subscribers will not receive any publications if they are not active.

With respect to Queues, the messages are held in the queue till someone receives them or they expire. The messages can be persistent meaning they will survive restart of messaging provider and non-persistent where the messages are lost when messaging provider goes down.

**Creating Durable Subscriptions**

To ensure that a pub/sub application receives all published messages, use PERSISTENT delivery mode for the publishers and durable subscriptions for the subscribers.

The Session.createConsumer method creates a nondurable subscriber if a topic is specified as the destination. A nondurable subscriber can receive only messages that are published while it is active.

At the cost of higher overhead, you can use the Session.createDurableSubscriber method to create a durable subscriber. A durable subscription can have only one active subscriber at a time.

A durable subscriber registers a durable subscription by specifying a unique identity that is retained by the JMS provider. Subsequent subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber. If a durable subscription has no active subscriber, the JMS provider retains the subscription’s messages until they are received by the subscription or until they expire.

You establish the unique identity of a durable subscriber by setting the following:

A client ID for the connection

A topic and a subscription name for the subscriber

You set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console.

After using this connection factory to create the connection and the session, you call the createDurableSubscriber method with two arguments: the topic and a string that specifies the name of the subscription:

String subName = "MySub";

MessageConsumer topicSubscriber =

session.createDurableSubscriber(myTopic, subName);

The subscriber becomes active after you start the Connection or TopicConnection. Later, you might close the subscriber:

topicSubscriber.close();

The JMS provider stores the messages sent or published to the topic, as it would store messages sent to a queue. If the program or another application calls createDurableSubscriber using the same connection factory and its client ID, the same topic, and the same subscription name, then the subscription is reactivated and the JMS provider delivers any messages that were published while the subscriber was inactive.

To delete a durable subscription, first close the subscriber, then use the unsubscribe method with the subscription name as the argument:

topicSubscriber.close();

session.unsubscribe("MySub");

The unsubscribe method deletes the state the provider maintains for the subscriber.

### Acknowledgement in JMS

MDB’s are synchronous or asynchronous

**Controlling Message Acknowledgment**

Until a JMS message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

The client receives the message.

The client processes the message.

The message is acknowledged. Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see [Using JMS API Local Transactions](http://docs.oracle.com/javaee/6/tutorial/doc/bncfu.html#bncgh)), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depend on the value **specified as the second argument of the createSession method. The three possible argument values are as follows:**

**session = connection.createSession(true, 0);**

**session = connection.createSession(false,** AUTO\_ACKNOWLEDGE**);**

Session.AUTO\_ACKNOWLEDGE: The session automatically acknowledges a client’s receipt of a message either when the client has successfully returned from a call to receive or when the MessageListener it has called to process the message returns successfully.

A synchronous receive in an AUTO\_ACKNOWLEDGE session is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

Session.CLIENT\_ACKNOWLEDGE: A client acknowledges a message by calling the message’s acknowledge method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of **all** messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

**Note -** In the Java EE platform, a CLIENT\_ACKNOWLEDGE session can be used only in an application client, not in a web component or enterprise bean.

Session.DUPS\_OK\_ACKNOWLEDGE: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the JMS provider redelivers a message, it must set the value of the JMSRedelivered message header to true.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received from a queue but not acknowledged when a session terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages for a terminated session that has a durable TopicSubscriber. (See [Creating Durable Subscriptions](http://docs.oracle.com/javaee/6/tutorial/doc/bncfu.html#bncgd).) Unacknowledged messages for a nondurable TopicSubscriber are dropped when the session is closed.

If you use a queue or a durable subscription, you can use the Session.recover method to stop a nontransacted session and restart it with its first unacknowledged message. In effect, the session’s series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if messages have expired or if higher-priority messages have arrived. For a nondurable TopicSubscriber, the provider may drop unacknowledged messages when its session is recovered.

The sample program in [A Message Acknowledgment Example](http://docs.oracle.com/javaee/6/tutorial/doc/giwfh.html#bncfx) demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.

**Specifying Message Persistence**

The JMS API supports two delivery modes specifying whether messages are lost if the JMS provider fails. These delivery modes are fields of the DeliveryMode interface.

The PERSISTENT delivery mode, the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.

The NON\_PERSISTENT delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

You can use the setDeliveryMode method of the MessageProducer interface to set the delivery mode for all messages sent by that producer. For example, the following call sets the delivery mode to NON\_PERSISTENT for a producer:

**producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);**

You can use the long form of the send or the publish method to set the delivery mode for a specific message. The second argument sets the delivery mode. For example, the following send call sets the delivery mode for message to NON\_PERSISTENT:

**producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);**

The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode, the default is PERSISTENT. Using the NON\_PERSISTENT delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

**Setting Message Priority Levels**

You can use message priority levels to instruct the JMS provider to deliver urgent messages first. You can set the priority level in either of two ways.

You can use the setPriority method of the MessageProducer interface to set the priority level for all messages sent by that producer. For example, the following call sets a priority level of 7 for a producer:

producer.setPriority(7);

You can use the long form of the send or the publish method to set the priority level for a specific message. The third argument sets the priority level. For example, the following send call sets the priority level for message to 3:

producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);

The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A JMS provider tries to deliver higher-priority messages before lower-priority ones but does not have to deliver messages in exact order of priority.

**Allowing Messages to Expire**

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. You can do this in either of two ways.

You can use the setTimeToLive method of the MessageProducer interface to set a default expiration time for all messages sent by that producer. For example, the following call sets a time to live of one minute for a producer:

producer.setTimeToLive(60000);

You can use the long form of the send or the publish method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. For example, the following send call sets a time to live of 10 seconds:

producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);

If the specified timeToLive value is 0, the message never expires.

When the message is sent, the specified timeToLive is added to the current time to give the expiration time. Any message not delivered before the specified expiration time is destroyed. The destruction of obsolete messages conserves storage and computing resources.

### **Difference between SendRedirect() and Forward() in JSP Servlet**

Difference between SendRedirect and forward is one of classical interview questions asked during java web developer interview. This is not just applicable for servlet but also for JSP in which we can use forward action or call sendRedirect() method from scriptlet. Before examining difference on forward and SendRedirect let’s see what send Redirect method and forward method does.

**SendRedirect ():**

This method is declared in HttpServletResponse **Interface**.

**Signature**: void sendRedirect(String url)

This method is used to redirect client request to some other location for further processing **,**the new location is available on different server or different context.our web container handle this and transfer the request using browser ,and this request is **visible in browser as a new request**. Some time this is also called as client side redirect.

**Forward():**

This method is declared in RequestDispatcherInterface.

**Signature**: forward(ServletRequest request, ServletResponse response)

This method is used to pass the request to another resource for further processing **within the same server**, another resource could be any servlet, jsp page any kind of file.This process is taken care by web container when we call forward method request is sent to another resource **without the client being informed,** which resource will handle the request it has been mention on requestDispatcher object which we can get by two ways either using ServletContext or Request. This is also called server side redirect.

RequestDispatcher rd = request.getRequestDispatcher("pathToResource");  
rd.forward(request, response);

Or

RequestDispatcher rd = servletContext.getRequestDispatcher("/pathToResource");  
rd.forward(request, response);

Difference between SendRedirect and Forward

Now let’s see some difference between these two method of servlet API in tabular format.

|  |  |
| --- | --- |
| **Forward()** | **SendRediret()** |
| When we use forward method request is transfer to other resource within the same server for further processing. | In case of sendRedirect request is transfer to another resource to different domain or different server for futher processing. |
| In case of forward Web container handle all process internally and client or browser is not involved. | When you use SendRedirect container transfers the request to client or browser so url given inside the **sendRedirect** method is visible as a new request to the client. |
| When forward is called on **requestdispather** object we pass request and response object so our old request object is present on new resource which is going to process our request | In case of SendRedirect call old request and response object is lost because it’s treated as new request by the browser. |
| Visually we are not able to see the forwarded address, its is transparent | In address bar we are able to see the new redirected address it’s not transparent. |
| Using forward () method is faster then send redirect. | SendRedirect is slower because one extra round trip is required beasue completely new request is created and old request object is lost.Two browser request requird. |
| When we redirect using forward and we want to use same data in new resource we can use request.setAttribute () as we have request object available. | But in sendRedirect if we want to use we have to store the data in session or pass along with the URL. |

Example of forward and SendRedirect in JSP Servlet:

Any kind of online payment when we use merchant site will redirect us to net banking site which is completely new request it process our request and again redirect to merchant site?

In Banking Application when we do login normally we use forward method. In case of online banking we are asked for username and password if it’s a correct some another servlet or resource will handle the request other wise request has been forwarded to error page.

Which one is good?

Its depends upon the scenario that which method is more useful.

If you want control is transfer to new server or context and it is treated as completely new task then we go for Send Redirect.

Normally forward should be used if the operation can be safely repeated upon a browser reload of the web page will not affect the result.

SendRedirect and forward method are still very useful while programming or working on any web application project using **servlet jsp**. This is still a popular interview questions so don’t forget to revise **forward and sendRedirect** before appearing for any job interview.

### Iterator vs ListIterator

1) Iterator is used for traversing List and Set both.

We can use ListIterator to traverse List only, we cannot traverse Setusing ListIterator.

2) We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6) Methods of Iterator:

* hasNext()
* next()
* remove()
* Methods of ListIterator:
* add(E e)
* hasNext()
* hasPrevious()
* next()
* nextIndex()
* previous()
* previousIndex()
* remove()
* set(E e)

# WEB Services

1. SSL config for webservice
2. Passing parameters in Jersey rest webservice
3. How to secure rest webservice - <https://docs.oracle.com/middleware/1212/wls/RESTF/secure-restful-service.htm>

# Hibernate

1. Hibernate (Lazy initilization), Second Level Cache,
2. How to remove second level of caching in hibernate