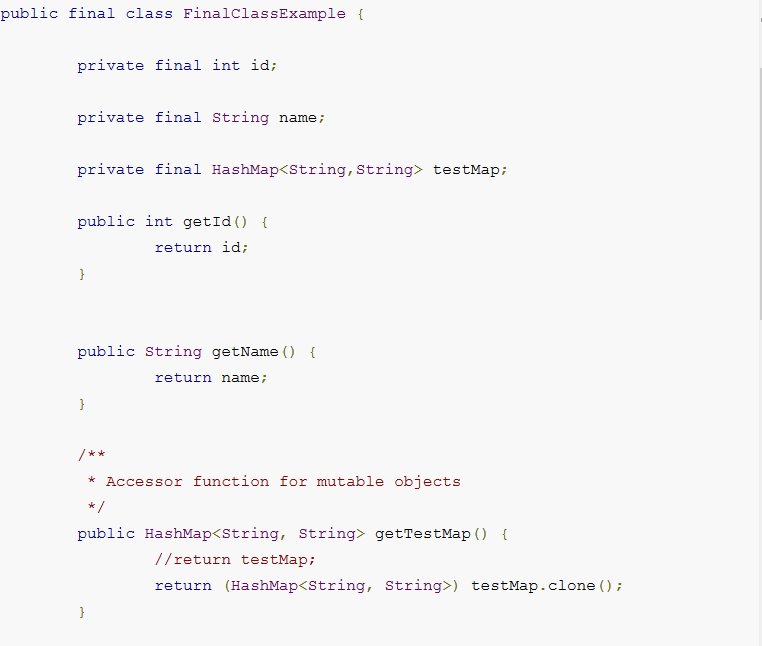
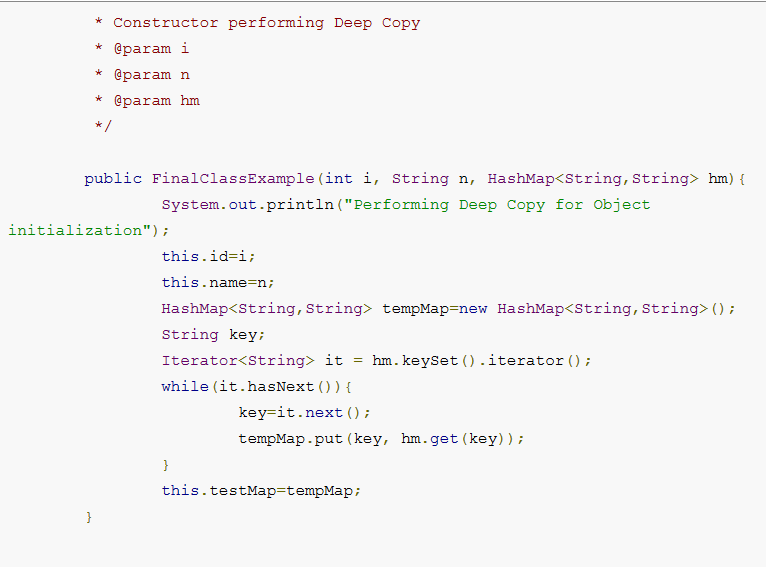
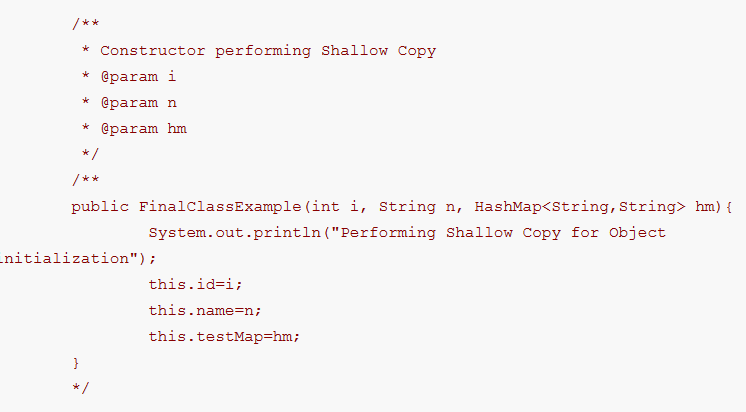
To create **immutable class** in java, you have to do following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables
4. Make all **mutable fields final** so that it’s value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference

Example







**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.

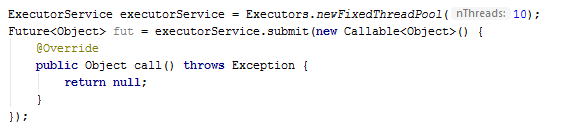
String s = new String("Test");  
   
does not  put the object in String pool , we need to call String.intern() method which is used to put  them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool.



**Difference between ExecutorService.submit() and ExecuterService.execute() method ?**

ExecutorService is interface inherited from Executor.

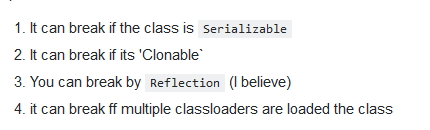
Submit can accept callable whereas execute can only accept Runnable. Callable can return Future through which we can return object back from thread, it also throws back the execption.



**Singleton (best way)**



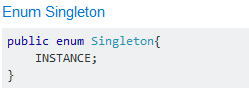
**Ways to break normal singleton**



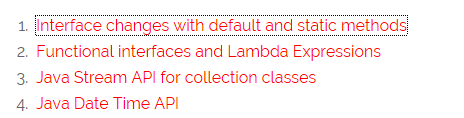
**What is difference between CyclicBarrier and CountDownLatch in Java**

Relatively newer Java tricky question, only been introduced form Java 5. Main difference between both of them is that you can reuse CyclicBarrier even if Barrier is broken but you can not reuse CountDownLatch in Java. See CyclicBarrier vs CountDownLatch in Java for more differences.

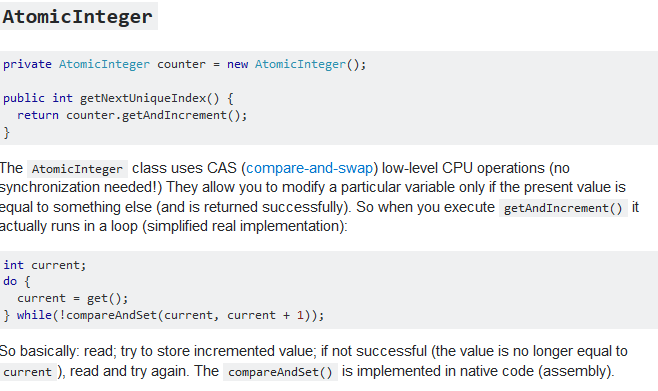




**Java 8 Features**



**https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method**



**Statement v/s Prepared statement vs callable statement**

**Creating Statement Object**

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

* **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
* **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
* **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

## Creating PreparedStatement Object

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

## Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure −

CREATE OR REPLACE PROCEDURE getEmpName

(EMP\_ID IN NUMBER, EMP\_FIRST OUT VARCHAR) AS

BEGIN

SELECT first INTO EMP\_FIRST

FROM Employees

WHERE ID = EMP\_ID;

END;

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each −

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| IN | A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods. |
| OUT | A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods. |
| INOUT | A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods. |

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure −

CallableStatement cstmt = null;

try {

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

**Advantages of prepares stmt over stmt**

Most relational databases handles a JDBC / SQL query in four steps:

1. Parse the incoming SQL query
2. Compile the SQL query
3. Plan/optimize the data acquisition path
4. Execute the optimized query / acquire and return data

A Statement will always proceed through the four steps above for each SQL query sent to the database. A Prepared Statement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a Prepared Statement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time.

 Precompilation and DB-side caching of the SQL statement leads to overall faster execution and the ability to reuse the same SQL statement in [batches](https://stackoverflow.com/questions/2467125/reusing-a-preparedstatement-multiple-times).

 Automatic prevention of [SQL injection](http://en.wikipedia.org/wiki/SQL_injection) [attacks](http://unixwiz.net/techtips/sql-injection.html) by builtin escaping of quotes and other special characters. Note that this requires that you use any of the PreparedStatement setXxx() methods to set the values

**How to analyze long running queries**

# he EXPLAIN QUERY PLAN Command

**Warning: The data returned by the EXPLAIN QUERY PLAN command is intended for interactive debugging only. The output format may change between SQLite releases. Applications should not depend on the output format of the EXPLAIN QUERY PLAN command.**

The [EXPLAIN QUERY PLAN](https://www.sqlite.org/lang_explain.html) SQL command is used to obtain a high-level description of the strategy or plan that SQLite uses to implement a specific SQL query. Most significantly, EXPLAIN QUERY PLAN reports on the way in which the query uses database indices.

For each table read by the query, the output of EXPLAIN QUERY PLAN includes a record for which the value in the "detail" column begins with either "SCAN" or "SEARCH". "SCAN" is used for a full-table scan, including cases where SQLite iterates through all records in a table in an order defined by an index. "SEARCH" indicates that only a subset of the table rows are visited. Each SCAN or SEARCH record includes the following information:

* The name of the table data is read from.
* Whether or not an index or [automatic index](https://www.sqlite.org/optoverview.html#autoindex) is used.
* Whether or not the [covering index](https://www.sqlite.org/queryplanner.html#covidx) optimization applies.
* Which terms of the WHERE clause are used for indexing.

**Queries to check locked and long running queries**



**How to test REST services**

1. POSTMAN ( google chrome extension also available)
2. HttpRequester (Firefox
3. RESTClient
4. REST Easy

**Rest Client code**

RestTemplate *restTemplate = new* RestTemplate();

**public <T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables) throws RestClientException**

eg:

String valueEncoded = *restTemplate*.getForObject(*urlWithoutDefaultValue*, String.**class**, params);

**public <T> T postForObject(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables) throws RestClientException**

eg:

String schema (request)

response = *restTemplate*.postForObject(addURI.toString(), schema, String.**class**, params);

**Spring Security**

https://learninglms.jpmchase.net/eLMS/app/management/LMS\_ActDetails.aspx?UserMode=0&showPortalMenu=true&pqrpqr=eyJ1c2VyTE9CQXBwTmFtZSI6IkNPUlBPUkFURSJ9&ActivityId=439868

**Loading property file using spring**

<**context:property-placeholder location="classpath\*:system.properties,classpath\*:common.properties,classpath\*:db.properties"**/>

<**bean id="dataSource" class="com.jpmc.ctryrisk.perseus.datasource.CompassDbcpDatasource" destroy-method="close"**>  
 <**constructor-arg name="tnsAdminDirectory" value="${tns.admin.directory}"** />