Optimize Java Code Performance

1. Avoid Writing Long Methods

The methods should not be too long and should be specific to perform single functionality. It is better for maintenance as well as performance since while class loading and during method call, the method is loaded in stack memory. If methods are large with too much processing they will consume memory as well as CPU cycles to execute.

Try to break the methods into smaller ones at suitable logical points.

2. **Avoid Multiple If-else Statements**

We use conditional statements in our code for decision-making. The conditional statements should not be overused. If we are using too many conditional if-else statements it will impact performance since **JVM** will have to compare the conditions. This can become worse if the same is used in looping statements like for, while, etc. If there are too many conditions in your business logic try to group the conditions and get the boolean outcome and use it in the if statement. Also, we can think of using a switch statement instead of multiple if-else if possible. Switch statement has a performance advantage over if – else. The sample is provided below as an illustration which is to be avoided as follows:

3. **Avoid Getting the Size of the Collection in the Loop**

While iterating through any collection get the size of the collection beforehand and never get it during iteration. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
List<String> objList = getData();

for (int i = 0; i < objList.size(); i++) { execute code ..}
```

Note: *Above sample is to be avoided and use this as follows:*

```
List<String> objList = getData();

int size = objList.size();

for (int i = 0; i < size; i++) { execute code ..}
```

4. Avoid Using String Objects For Concatenation

A string is an immutable class the object created by String cannot be reused. So if we need to create a large string in case of SQL queries etc it is bad practice to concatenate the String object using the '+' operator. This will lead to multiple objects of String created leading to more usage of heap memory. In this case, we can use StringBuilder or StringBuffer, the former is preferential over the latter since it has a performance advantage due to non-synchronized methods. The sample is provided below as an illustration which is to be avoided as follows:

```
String query = String1+String2+String3;
```

Note: *Above sample is to be avoided and use this as follows:*
StringBuilder strBuilder = new StringBuilder("");

strBuilder.append(String1).append(String2).append(String3);

String query = strBuilder.toString();

5. Use Primitive Types Wherever Possible

Usage of primitive types over objects is beneficial since the primitive type data is stored on stack memory and the objects are stored on heap memory. If possible, we can use primitive types instead of objects since data access from stack memory is faster than heap memory. So it is always beneficial to use int over Integer or double over Double.

Avoid Using BigDecimal Class

We know that BigDecimal class provides accurate precision for the decimal values. Over usage of this object hampers the performance drastically specifically when the same is used to calculate certain values in a loop. BigDecimal uses a lot of memory over long or double to perform calculations. If precision is not the constraint or if we are sure the range of the calculated value will not exceed long or double we can avoid using BigDecimal and use long or double with proper casting instead.

7. Avoid Creating Big Objects Often

There are certain classes that act as data holders within the application. These objects are heavy and their creation should be avoided multiple times. An example of such objects is the **DB connection objects** or **system configuration objects** or **session objects** for the user after login. These objects used a **lot of resources** while created. We should reuse these objects instead of creating them as creation will drastically hamper the application performance **due to more memory usage**. We should use the **Singleton pattern** wherever possible to create a single instance of the object and reuse it wherever **required or clone the object instead of creating a new one.**

8. **Use Stored Procedures Instead of Queries**

It is better to write stored procedures instead of complex and long queries and call them while processing. **Stored procedures are stored as objects in the database and pre-compiled**. The execution time of the stored procedure is less compared to the query with the same business logic as a **query is compiled and executed every time wherever it is called through the application.** Also, the stored procedure has an advantage in data transfer and network traffic since we are not **transferring the complex query for execution every time to the database server**

9. Using PreparedStatement instead of Statement

While executing the SQL query through the application we use **JDBC API** and classes for the same. PreparedStatement has an **advantage** over Statement for **parameterized query** execution since the preparedstatement object is **compiled once and executed multiple times**. Statement object on other hand is compiled and executed every time it is called. Also**,** the **prepared statement object is safe to avoid SQL injection attacks for Web Application security**.

10. Use of Unnecessary Log Statements and Incorrect Log Levels

Logging is an integral part of any application and needs to be implemented efficiently in order to **avoid performance hits** due to incorrect logging and log levels. We should avoid logging big objects into code. Logging should be limited to specific parameters we need to monitor and not the whole object. Also**,** the logging level should be kept to higher levels like **DEBUG, ERROR, and not INFO**. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
Logger.debug("User info : " + user.toString());

Logger.info("Method called for setting user data:" +
user.getData());
```

Note: *Above sample is to be avoided and use this as follows:*
Logger.debug(“User info : ” + user.getName() + ” : login ID : ” + user.getLoginId());

Logger.info(“Method called for setting user data”);

11. Select Required Columns in a Query

While getting the data from the database we use select queries to get the data. Avoid selecting columns that are not necessary for further processing. Select only those columns which we will be required for further processing or display on the front end. Selecting too many columns causes a delay in query execution at the database end. Also, it increases network traffic from database to application which should be avoided. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
select * from users where user_id = 100;
```

Note: *Above sample is to be avoided and use this as follows:*
select user_name, user_age, user_gender, user_occupation, user_address from users where user_id = 100;

12. **Fetch the Data Using Joins**

While getting the data from multiple tables it is necessary to use the joins properly on tables. **If the joins are not properly used or the tables are not normalized it will cause a delay in query execution** leading to a performance hit for the application. **Avoid using subqueries instead of joins as subqueries take more time for execution than joins**. Create an index on columns of the table which are frequently used for improving the performance of query execution and reducing the latency of the application.

.