

# ATYPON

DocumentDB

Atypon Final Project

Written by: Moayad AL-SALEH

Introduction .....	3
What is a Document? .....	3
How you implemented the DB .....	5
File System .....	5
Operation Read and Write:.....	7
Data Structure: .....	9
Index: .....	9
Distribution Models .....	10
Master-Slave Replication .....	11
Master Node: .....	12
Slave Node: .....	13
Eureka Server:.....	13
Clean Code .....	14
<b>Meaningful Names:</b> .....	14
<b>Functions</b> .....	15
<b>Error Handling</b> .....	17
SOLID principles .....	18
<b>Single Responsibility Principle</b> .....	18
<b>Open / Closed Principle</b> .....	18
<b>Liskov substitution principle</b> .....	18
<b>Interface Segregation Principle</b> .....	18
<b>Dependency Inversion Principle</b> .....	18
Effective Java .....	19
<b>CREATING AND DESTROYING OBJECTS</b> .....	19
<b>METHODS COMMON TO ALL OBJECTS</b> .....	22
<b>Classes and Interfaces</b> .....	23
<b>METHODS</b> .....	25
<b>GENERAL PROGRAMMING</b> .....	26
<b>EXCEPTIONS</b> .....	30

# Introduction

---

In this report I will explain how did I start thinking about building this project, how did I build it , what resources did I choose for studying , what obstacles did I face and how I managed to solve it.

## Diving into the database world

I started with reading, as it is always the first step to understand any topic, I chose two books on platform:

- NoSQL for Mere Mortals
- NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence

And after a long period of reading, I achieved a well based understanding that was good enough to enable me to start developing the system.

## What is a Document?

In NoSQL Distilled Book the definition of Document is

“Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable.”

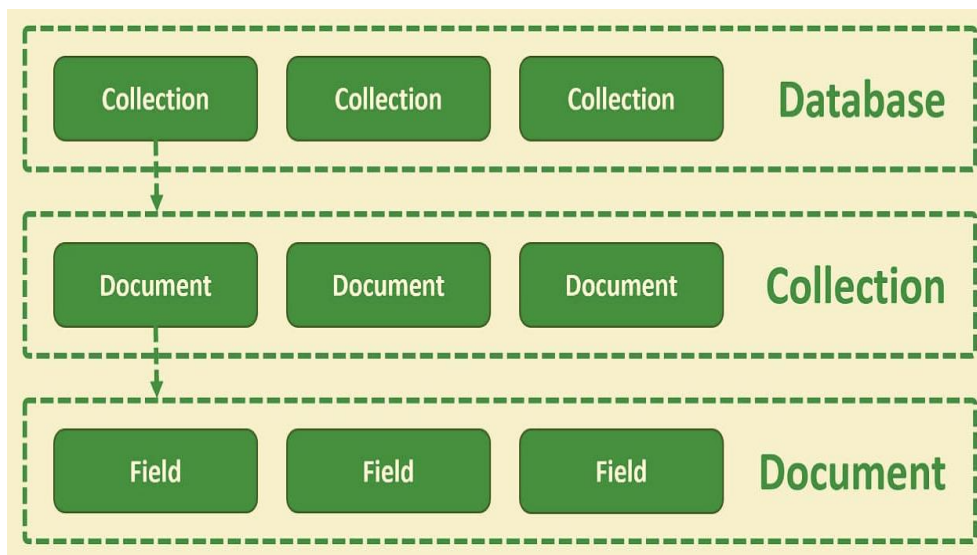
```
{ "firstname": "Martin",  
  "likes": [ "Biking",  
            "Photography" ],  
  "lastcity": "Boston",  
  "lastVisited":  
}
```

The above document can be considered a row in a traditional RDBMS.

Let's look at how terminology compares in RDBMS and UNRDBMS.

schema	database
table	collection
row	document
rowid	_id

This is the underlying structure of the database is:



And the real difference that makes document database special is the schemaless concept, so that a document does not necessarily have all the properties like the other documents, and this thing is what gives document database the ability to add data without restrictions, and use less memory as possible.

And now I will talk about how did I start applying these concepts that I got from reading, mentioning that reading and watching videos about the topic took about three weeks, I think this normal due to that I have never heard about this concept before.

# How you implemented the DB

---

First I start building read and write classes ,which are respectively responsible for reading and writing operations on file.

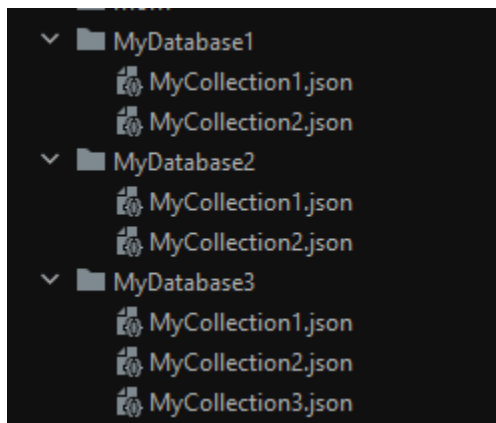
And here I faced a problem , which is how will I structure the file system.

## File System

And these are the proposed solutions for the file system :

1. To give the JSON file a field that has the name of the database and the collection.
2. To make the name of the JSON file consisting of a prefix (database name) and postfix (collection name).

But I found that the best method is to create a folder that can be thought as the database, that contains a JSON file named with the collection name ,which has all the documents in the collection.



Note : I found the buffer reader and writer are the best methods for reading from the file and writing on it.

```
BufferedReader br = new BufferedReader (new FileReader ( fileName: path + databasename + "/" + collectio
```

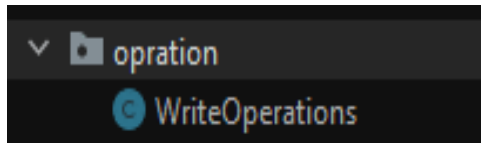
```
BufferedWriter bw = new BufferedWriter (file);
```

For reading and writing I found that the operation of reading from the file and writing on it, is heavily slow, so I had to find a way to fasten from these operations. The best method that I found is reading the files and storing its content in a data structure one time (in the start of reading) and applying any changes directly on the data structure, and when the writes are done, we apply these changes directly on the files.

## Operation Read and Write:

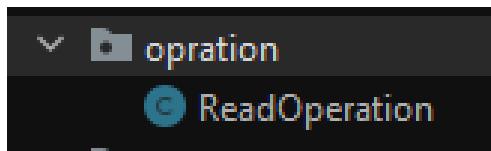
### Class Write:

This class is responsible for all write operations, such as creating ,adding , updating and deleting operations.

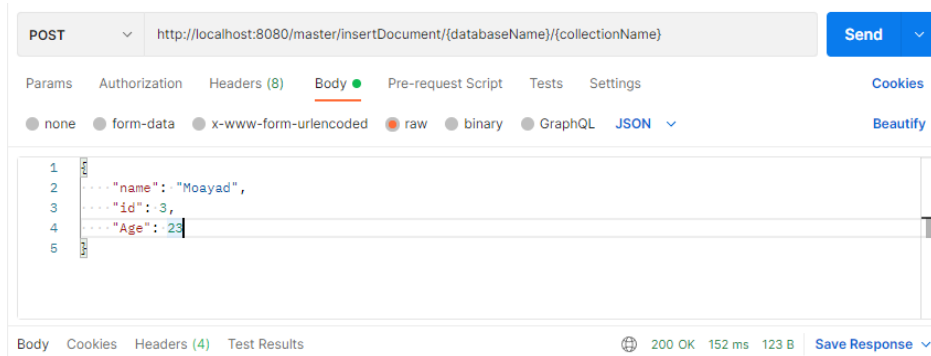


### Class Read:

This class is responsible for all reading operations form the databases.



Now I will show you a write operation, which is inserting a document:



The user inserts a new document and send it to the master node.

A checking is applied to check if the new document has an id or not.

```
public void insertDocument (String database, String collection, JSONObject jsonObject)
{
    if (collectionToBPlusTree.containsKey (database + collection))
    {
        if (isContainID (jsonObject, key: "id"))
        {
            if (isDocumentExistID (database, collection, jsonObject.get ("id").toString ()))
            {
                System.out.println ("the id is already exist");
                return;
            }
        }
    }
}
```

```

    } else
    {
        addRandomIntID (jsonObject);
    }

```

In case that ID is not existed, a new random ID is added, which is impossible to be generated twice.

```

public JSONObject addRandomIntID (JSONObject obj)
{
    // ...
    long idRandom = (long) (Math.random () * 10e18);
    String date = new SimpleDateFormat ( pattern: "yyyyMMddHHmmss").format (new Date ());
    //convert id to long
    String id = date + idRandom;
    obj.put ("id", id);
    return obj;
}

```

To ensure that every ID is unique , I chose appending the date to a random number from 1 to  $10^{18}$  .

```

collectionToBPlusTree.get (database + collection).insert (jsonObject.get (index).toString (), jsonObject);

```

After that, the document is added to the data structure.



# Data Structure:

For the data structures I used HashMap and B+tree.

I created a HashMap that has all the databases names and the collections names , which will fasten the operations of checking if certain databases or collections are existed or not , with a complexity of  $O(1)$ .

```
public HashMap<String, List<String>> databaseToCollection = new HashMap<> ();
```

I created a Hashmap that maps from collections to their respective B+Tree.

Note that reading from the B+tree has a complexity of  $O(\log(n))$ .

```
public HashMap<String, BPlusTree> collectionToBPlusTree = new HashMap<> ();
```

## Index:

I created an index inside the master so that the user can change the index whenever he pleases, all he has to do is to create a new index based on any field then the master will instantiate the B+Tree based on that field and not on the ID.

And now I will show you how the index is changed:

The screenshot shows a REST client interface with a PUT request to `http://localhost:8080/master/changeIndexCollection/{databaseName}/{collectionName}/{indexName}`. The 'Headers' tab is selected, showing 7 hidden headers. Below the headers, there is a table with 6 columns: KEY, VALUE, DESCRIPTION, Bulk Edit, and Presets. The first row contains 'Key' and 'Value'. The 'Body' tab is also visible at the bottom.

KEY	VALUE	DESCRIPTION	Bulk Edit	Presets
Key	Value	Description		

First the user asks the master to change the index using the database, collection and property field which the b+tree will be built on, and let's choose the name here .

```

public void changeIndexCollection (String dbName, String collectionName, String index)
{
    if (isCollectionExist (dbName, collectionName))
    {
        JSONArray jsonArray = collectionToBPlusTree.get (dbName + collectionName).getAllValues ();
        collectionToBPlusTree.remove (key: dbName + collectionName);
        collectionToBPlusTree.put (dbName + collectionName, new BPlusTree ());
        for (int i = 0; i < jsonArray.length (); i++)
        {
            ...
        }
        setIndexForCollection (dbName, collectionName, index);
        writeFile (dbName, collectionName, getAllDocument (dbName, collectionName));
        //for print all name of database and collection
    }
}

```

After that the master will receive the request and delete the old tree that belonged to the same collection. Then it will create another tree based on the new chosen field (such as the name).

The operation of destroying the old index and reconstructing a new one will take a time complexity of  $O(n \log(n))$ .

This will give the read operations which are based on the name, a time complexity of  $O(\log(n))$ .

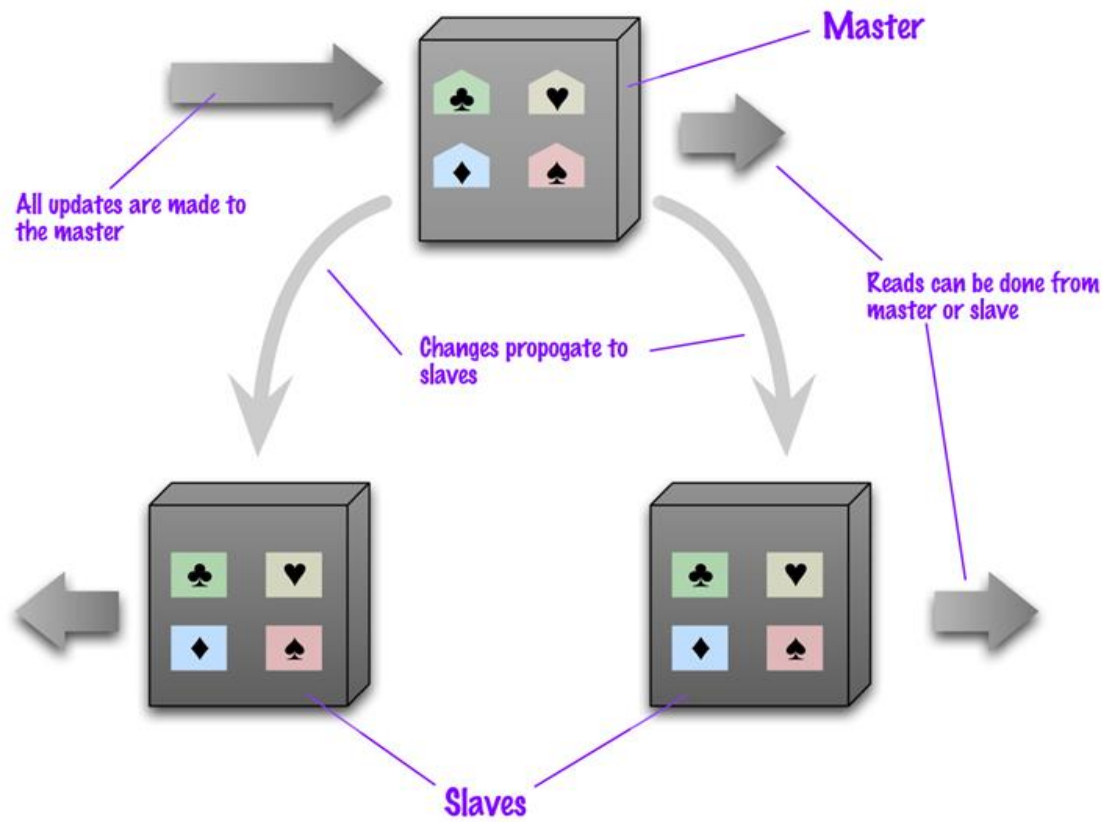
If the user tried to read a document based on another property which is not the same the b+tree is using, the system will handle this by reading directly from the files, which gives a complexity of  $O(n)$ .

## Distribution Models

---

Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages. These are often important benefits, but they come at a cost. Running over a cluster introduces complexity—so it's not something to do unless the benefits are compelling.

## Master-Slave Replication



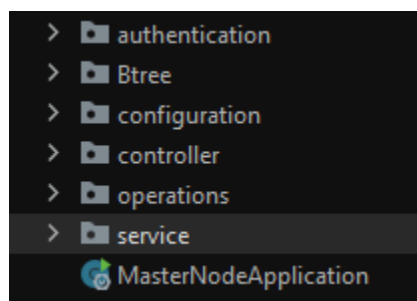
*Data is replicated from master to slaves. The master services all write.*

## Master Node:

It is the main node that controls the write operations on the databases, and responsible for any updates that are done on the database, in order to send them to the slaves.

It consists of three layers:

1. Controller
2. Service
3. Operation



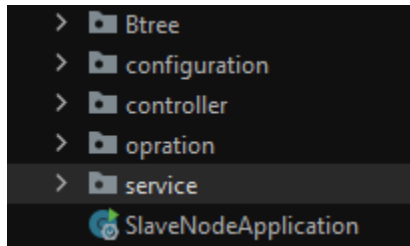
```
public void updataDataase ()
{
    ZipUtil.pack (new File ( pathname: "master-node/Data/DatabaseFile"), new File ( pathname: "master-node/Data/DatabaseFile.zip"));

    String fileName = "master-node/Data/DatabaseFile.zip";
    CloseableHttpClient client = HttpClientBuilder.create ().build ();
    //HttpPost post = new HttpPost( myFile_URL );
    String myFile_URL = "http://localhost:8082/slave/getDatabase";
    HttpPost post = new HttpPost (myFile_URL);
    File file = new File (fileName);
    FileBody fileBody = new FileBody (file, ContentType.DEFAULT_BINARY);
    MultipartEntityBuilder builder = MultipartEntityBuilder.create ();
    builder.setMode (HttpMultipartMode.BROWSER_COMPATIBLE);
    builder.addPart ( name: "file", fileBody);
    HttpEntity entity = builder.build ();
    post.setEntity (entity);}
    try
    {...} catch (Exception e)
    {...}
    restTemplate.postForObject ( url: "http://SLAVE-NODE/slave/updateDB", request: "Update The Database", String.class);
}
```

The database will be sent to the slaves after any change in the master.

## Slave Node:

It is the node that is responsible for all reading operations from the database.



Reading operations that are based on the same field that the tree is built on, will take  $O(\log(n))$ .

And reads based on other fields will take  $O(n)$ .

## Eureka Server:

---

Eureka Server adds nodes on the network to organize requests between master and slave

DS Replicas			
<a href="#">localhost</a>			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MASTER-NODE	n/a (1)	(1)	UP (1) - <a href="#">host.docker.internal:master-node:8080</a>
SLAVE-NODE	n/a (2)	(2)	UP (2) - <a href="#">host.docker.internal:slave-node:8084</a> , <a href="#">host.docker.internal:slave-node:8082</a>

*This is a picture of the interface for Eureka Server*

# Clean Code

---

Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code.

Here are the concepts that I applied while developing the system:

## Meaningful Names:

### Intention-Revealing Names

>> The name of a variable, function, or class should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, the name does not reveal its intent.

### Avoid Disinformation

>> Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning.

### Make Meaningful Distinctions

>> Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter.

### Use Searchable Name

>> Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

## One Word per Concept

>> Pick one word for one abstract concept and stick with it.

## Class Names

>> Classes and objects should have noun or noun phrase names.

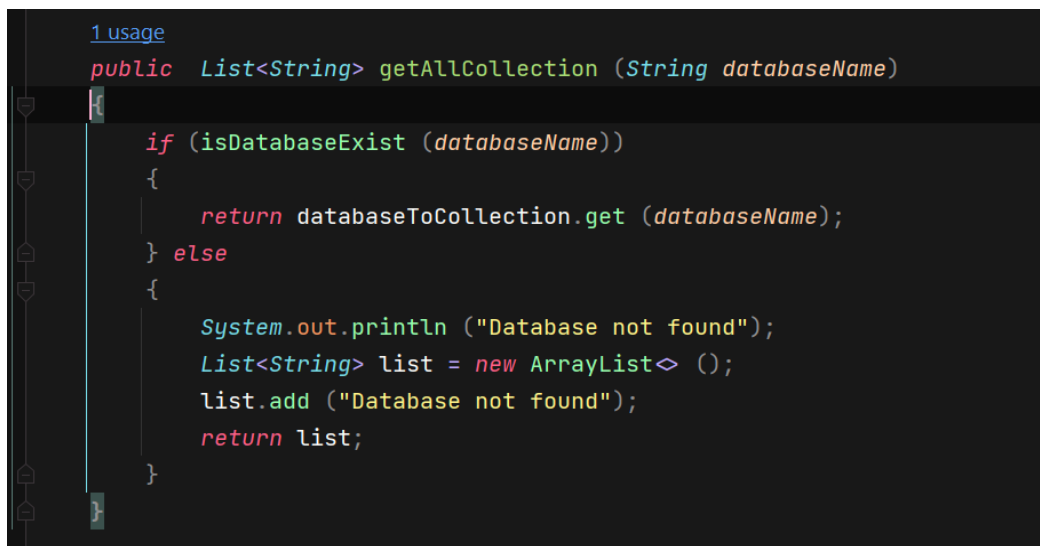
## Method Names

>> Methods should have verb or verb phrase names.

## Functions

### Function Arguments

>> The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn't be used anyway.

A screenshot of a code editor with a dark background. The code is in Java and defines a public method named `getAllCollection` that takes a single `String` argument named `databaseName`. The method body contains an `if` statement to check if the database exists. If it does, it returns the result of `databaseToCollection.get(databaseName)`. If not, it prints "Database not found" to the console, creates a new `ArrayList`, adds "Database not found" to it, and returns the list. The code is color-coded: keywords like `public`, `if`, `else`, `return`, and `System.out.println` are in red; types like `List<String>` and `String` are in green; and identifiers like `databaseName`, `databaseToCollection`, and `list` are in white. The editor has a sidebar on the left with icons for Explorer, Search, and Run and Debug.

```
1 usage
public List<String> getAllCollection (String databaseName)
{
    if (isDatabaseExist (databaseName))
    {
        return databaseToCollection.get (databaseName);
    } else
    {
        System.out.println ("Database not found");
        List<String> list = new ArrayList<> ();
        list.add ("Database not found");
        return list;
    }
}
```

*I tried to make the number of arguments as less as possible, but I could not do that in all the situations I faced.*

## Small, Do One Thing

>> The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

## Use Descriptive Names

>> Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name result in a favorable restructuring of the code.

## Flag Arguments

>> Flag arguments are ugly. Passing a Boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false.

## Don't Repeat Yourself (DRY)

>> Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it.

## Try/Catch Blocks

>> Try/catch blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So, it is better to extract the bodies of the Try and catch blocks out into functions of their own.

```
public String readFile (String databaseName, String collectionName)
{
    StringBuilder sb = new StringBuilder ();
    try
    {
        if (isCollectionExist (databaseName, collectionName))
        {
            ...
        }
        else
        {
            System.out.println ("File not found");
            return "File not found";
        }
    }
    catch (IOException e)
    {
        e.printStackTrace ();
    }
}
```



## Objects and Data Structures

>> There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

## Comments

>> The proper use of comments is to compensate for our failure to express ourselves in code. Note that I used the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

## Error Handling

### Don't Pass Null

Returning null from methods is bad but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

### Don't Return Null

I think that any discussion about error handling should include mention of the things we do that invite errors. The first on the list is returning null. I can't begin to count the number of applications I've seen in which nearly every other line was a check for null.

# SOLID principles

---

SOLID is one of the most popular sets of design principles in object-oriented software development. It's a mnemonic acronym for the following five design principles:

## Single Responsibility Principle

(Each class should have one responsibility, one single purpose)

I have applied this principle, so you will find that each package, class and method is responsible for only one thing .

## Open / Closed Principle

(Software entities (classes, modules, methods, etc.) should be open for extension, but closed for modification)

I have also worked on applying this principle as much as I can, so you will find that the classes and the methods can be expanded in the future.

## Liskov substitution principle

(Any subclass object should be substitutable for the superclass object from which it is derived)

I have been careful while use the principle of inheritance, so you will find that every Super Class does not extend to his child except the method that he needs.

## Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use

I tried to make all my interfaces responsible for small matters so that the other classes only inherit what they need.

## Dependency Inversion Principle

(High level modules should not depend on low level modules; both should depend on abstractions)

I have worked hard to reduce the reliability between the classes and make each class independent of the other, so abstraction. I increased the use of the Interface and

# Effective Java

---

## CREATING AND DESTROYING OBJECTS

### 1. Use STATIC FACTORY METHODS instead of constructors

#### **ADVANTAGES**

- Unlike constructors, they have names
- Unlike constructors, they are not required to create a new object each time they're invoked
- Unlike constructors, they can return an object of any subtype of their return type
- They reduce verbosity of creating parameterized type instances

#### **DISADVANTAGES**

- If providing only static factory methods, classes without public or protected constructors cannot be subclassed (encourage to use composition instead inheritance).
- They are not readily distinguishable from other static methods (Some common names (each with a different purpose) are: `valueOf`, `of`, `getInstance`, `newInstance`, `getType` and `newType`)

### 2. Use BUILDERS when faced with many constructors

Is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

Builder pattern simulates named optional parameters as in ADA and Python.

### 3. Enforce the singleton property with a private constructor or an Enum type:

- **Public final field**
- **Singleton with static factory**
- **Serialize a singleton**
- **Enum Singleton, the preferred approach (JAVA 1.5)**

### 4. Enforce noninstantiability with a private constructor

For classes that group static methods and static fields.

Used for example to:

- **Group related methods on primitive values or arrays.**
- **Group static methods, including factory methods, for objects that implement a particular interface.**
- **Group methods on a final class instead of extending the class.**

### 5. Avoid creating unnecessary objects

- **REUSE IMMUTABLE OBJECTS**
- **Use static factory methods in preference to constructors**
- **REUSE MUTABLE OBJECTS THAT WON'T BE MODIFIED**
- **Prefer primitives to boxed primitives, and watch out for unintentional autoboxing**
- **Object pools are normally bad ideas**

## 6. Eliminate obsolete object references

- ***Null out references***

Nulling out objects references should be the exception not the norm. Do not overcompensate by nulling out every object.

Null out objects only in classes that manages its own memory.

- ***Memory leaks in cache***

Using *WeakHashMap* is useful when if desired lifetime of cache entries is determined by external references to the key, not the value.

Clean oldest entries in cache is a common practice. To accomplish this behaviors, it can be used: background threads, automatically delete older after a new insertion or the *LinkedHashMap* and its method *removeEldestEntry*.

- ***Memory leaks in listeners and callbacks***

If clients register callbacks, but never deregister them explicitly.

To solve it store only *weak references* to them, for example storing them as keys in a *WeakHashMap*.

- **Use a Heap Profiler from time to time to find unseen memory leaks**

## 7. Avoid finalizers

Finalizers are unpredictable, often dangerous and generally.

- **Never do anything time-critical in a finalizer.**

There is no guarantee they'll be executed promptly.

- ***Never depend on a finalizer to update critical persistent state.***

There is no guarantee they'll be executed at all.

Uncaught exceptions inside a finalizer won't even print a warning.

There is a severe performance penalty for using finalizers.

- **Possible Solution**

Provide an *explicit termination method* like the *close* on *InputStream*, *OutputStream*, *java.sql.Connection*...

Explicit termination methods are typically used in combination with the *try-finally* construct to ensure termination.

## METHODS COMMON TO ALL OBJECTS

### 8. Always override toString

providing a good toString implementation makes your class much more pleasant to use and makes systems using the class easier to debug

### 9. Consider implementing

By implementing Comparable, you allow your class to interoperate with all of the many generic algorithms and collection implementations that depend on this interface. You gain a tremendous amount of power for a small amount of effort

## Classes and Interfaces

### 10. Minimize the accessibility of classes and members

Information hiding is important for many reasons, most of which stem from the fact that it decouples the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation

### 11. In public classes, use accessor methods, not public fields.

### 12. Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed

### 13. Favor composition over inheritance

inheritance is powerful, but it is problematic because it violates encapsulation. It is appropriate only when a genuine subtype relationship exists between the subclass and the superclass. Even then, inheritance may lead to fragility if the subclass is in a different package from the superclass and the superclass is not designed for inheritance

## 14. Prefer interfaces to abstract classes

When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class. That a class implements an interface should therefore say something about what a client can do with instances of the class. It is inappropriate to define an interface for any other purpose.

## 15. Favor static member classes over nonstatic

4 types of nested classes:

- **static**
- **nonstatic**
- **anonymous**
- **local**

## 16. Limit source files to a single Top-level class

Never put multiple top-level classes or interfaces in a single source file



# METHODS

## 17. Check parameters for validity

Check parameters before execution as soon as possible.

Add in public methods *@throw*, and use *assertions* in non public methods

Do it also in constructors.

## 18. Design method signatures carefully

- Choose method names carefully.
- Don't go overboard in providing convenience methods. Don't add too many.
- Avoid long parameter list. Make a subset of methods, helper classes , or a builder instead.
- For parameter types, favor interfaces over classes No reason to write a method that takes a *HashMap* on input, use *Map* instead.
- Prefer two-element Enum types to *Boolean* parameters. 

```
public enum TemperatureScale {CELSIUS, FARENHEIT}
```

## 19. Return empty arrays or collections, not nulls

There is no reason ever to return null from an array- or collection-valued method instead of returning an empty array or collection

Return an immutable empty array instead of null.

## GENERAL PROGRAMMING

### 20. Minimize the scope of local variables

Declare local variable where it is first used. Most local variable declaration should contain an initializer. Prefer for loops to while loops. Keep methods small and focused.

### 21. Prefer for-each loops to traditional for loops

```
private void findAllFilesNames (String nameFolder)
{
    File root = new File ( pathname: path + nameFolder);
    for (File file : Objects.requireNonNull (root.listFiles ()))
    {
        databaseToCollection.get (nameFolder).add (file.getName ().split ( regex: "\\.[0]));
        putAllDocumentsInBPlusTree (nameFolder, file.getName ().split ( regex: "\\.[0]);
    }
}
```

### 22. Avoid float and double if exact answer are required

For monetary calculations use *int*(until 9 digits) or *long* (until 18 digits) taken you care of the decimal part and you don't care too much about the rounding. Use *BigDecimal* for numbers bigger than 18 digits and if you need full control of the rounding methods used.

## 23. Know and use libraries

By using a standard library:

- Advantage of the knowledge of the experts who wrote it and the experience of those who used it before you.
- Don't have to waste your time writing ad hoc solutions to problems that are only marginally related to your work.
- Their performance tends to improve over time
- Your code will be easily readable, maintainable, and reusable.

Numerous features are added to the libraries in every major release, and it pays to keep abreast of these additions

Every programmer should be familiar with:

- `java.lang`
- `java.util`
- `java.io`
- `java.util.concurrent`

## 24. Prefer primitive types to boxed primitives

Primitives: *int, double, boolean*

Boxed Primitives: *Integer, Double, Boolean*

Differences:

- Two boxed primitives could have the same value but different identity.
- Boxed primitives have one nonfunctional value: *null*.
- Primitives are more space and time efficient.

## 25. Avoid Strings where other types are more appropriate

- Strings are more cumbersome than other types.
- Strings are less flexible than other types.
- String are slower than other types.
- Strings are more error-prone than other types.
- Strings are poor substitutes for other value types.
- Strings are poor substitutes for *enum* types.
- Strings are poor substitutes for *aggregate* types.
- Strings are poor substitutes for *capabilities*.
- So, use String to represent text!

## 26. Use native methods judiciously

Historically, native methods have had three main uses.

- They provided access to platform-specific facilities.
- They provided access to libraries of legacy code.
- To write performance-critical parts

New Java versions make use of NDK rarely advisable for improve performance.

## 27. Optimize judiciously

Strive to write good programs rather than fast ones, speed will follow.

If a good program is not fast enough, its architecture will allow it to be optimized.

- More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.
- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
- We follow two rules in the matter of optimization:
  - Rule 1. Don't do it.
  - Rule 2 (for experts only). Don't do it yet — that is, not until you have a perfectly clear and unoptimized solution.

If you finally, do it **measure performance before and after each attempted optimization**, and focus firstly in the choice of algorithms rather than in low level optimizations.

## 28. Adhere to generally accepted naming conventions

# EXCEPTIONS

## 29. Use exceptions only for exceptional conditions

Exceptions are for exceptional conditions.

Never use or (expose in the API) exceptions for ordinary control flow.

## 30. Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

Throwables:

- checked exceptions: for conditions from which the caller can reasonably be expected to recover
- unchecked exceptions: shouldn't, be caught. recovery is impossible and continued execution would do more harm than good.
  - runtime exceptions: to indicate programming errors. The great majority indicate precondition violations.
  - errors : are reserved for use by the JVM. (as a convention)

Unchecked throwables that you implement should **always** subclass *RuntimeException*.

## 31. Avoid unnecessary use of checked exceptions

Use checked exceptions only if these 2 conditions happen:

- The exceptional condition cannot be prevented by proper use of the API
- The programmer using the API can take some useful action once confronted with the exception.

Refactor the checked exception into a unchecked exception to make the API more pleasant.

## 32. Favor the use of standard exceptions

## 33. Strive for failure atomicity

A failed method invocation should leave the object in the state that it was in prior to the invocation. Options to achieve this:

- Design immutable objects
- Order the computation so that any part that may fail takes place before any part that modifies the object.
- Write recovery code (Undo operation)
- Perform the operation on a temporary copy of the object, and replace it once is completed.

## 34. Don't ignore exceptions

Don't let catch blocks empty.