|  |
| --- |
| JAVA CODING BEST PRACTICE |
| PART I |
|  |

Contents

[1.1. Database access 3](#_Toc450726646)

[1.1.1. N+1 selects 3](#_Toc450726647)

[1.1.2. DTO Usage 4](#_Toc450726648)

[1.2. Object initialization 5](#_Toc450726649)

[1.2.1. Use StringBuffer 5](#_Toc450726650)

[1.3. Collection Usage 5](#_Toc450726651)

[1.3.1. Avoid using HashTable and Vetor 5](#_Toc450726652)

[1.3.2. Use Iterator instead of indexing for loop 6](#_Toc450726653)

[1.3.3. List to Array 7](#_Toc450726654)

[1.3.4. Arrays.asList(…) 7](#_Toc450726655)

[1.3.5. Unnecessary check when iterating collections 8](#_Toc450726656)

[2.1. Resources allocation 9](#_Toc450726657)

[2.1.1. Use lazy initialization in memory-extensive applications 9](#_Toc450726658)

[2.1.2. Initialize size of collection type 9](#_Toc450726659)

[2.1.3. Should declare local variable just before use 10](#_Toc450726660)

[2.2. Resources de-allocation 11](#_Toc450726661)

[2.2.1. Must not forget to close the stream 11](#_Toc450726662)

[3.1. Object initialization 12](#_Toc450726663)

[3.1.1. Use double checked locking when lazy-initializing object in threaded context where resource is shared. 12](#_Toc450726664)

[3.2. Comparing objects 13](#_Toc450726665)

[3.2.1. Comparing two objects (“.equals” instead of “==”) 13](#_Toc450726666)

[3.2.2. Comparing strings 13](#_Toc450726667)

[4.1. Design – OOP 14](#_Toc450726668)

[4.1.1. OOP – Encapsulation: declare a data member is public 14](#_Toc450726669)

[4.1.2. OOP – Encapsulation: Do not visible things that should not be visible 14](#_Toc450726670)

[4.1.3. OOP – Use inner class in order to avoid using member attributes 15](#_Toc450726671)

[4.1.4. OOP – Isolation between technical handlings and business handlings 15](#_Toc450726672)

[4.1.5. OOP – Isolation between independent businesses 16](#_Toc450726673)

[4.1.6. OOP – Isolation between systems 16](#_Toc450726674)

[4.1.7. OOP – Isolation between the generic algorithm and detail 17](#_Toc450726675)

[4.1.8. OOP – Information expert, correct responsibility 17](#_Toc450726676)

[4.1.9. Getter/Setter and Overriding 17](#_Toc450726677)

[4.1.10. OOP - Decouple with real-implementation/coding to interface 19](#_Toc450726678)

[4.1.11. Member attributes access in multi-thread context 20](#_Toc450726679)

[4.2. Pitfalls when applying design patterns 21](#_Toc450726680)

[4.2.1. Do not overuse the Singleton pattern 21](#_Toc450726681)

[4.3. Others 21](#_Toc450726682)

[4.3.1. Statements modifying the structure of database (DDL – Data Definition Language) are auto-committed 21](#_Toc450726683)

[4.3.2. The interface java.io.Serialize must be understood properly 22](#_Toc450726684)

[5.1. Object initialization 23](#_Toc450726685)

[5.1.1. Declare private constructor for Helper, Utility classes 23](#_Toc450726686)

[5.1.2. Declare constants in classes instead of interface 23](#_Toc450726687)

[5.2. Maintainability 24](#_Toc450726688)

[5.2.1. Use template method pattern if repeated structuring of code happens 24](#_Toc450726689)

[5.2.2. Avoid using instanceof method 25](#_Toc450726690)

[5.2.3. Mark method, classes as final if they are not intended to be overridden 26](#_Toc450726691)

[5.2.4. Quote texts when logging 26](#_Toc450726692)

[5.2.5. Return empty list, array in public methods 27](#_Toc450726693)

[5.2.6. Do not use Assertion in place of business checking 28](#_Toc450726694)

[5.2.7. Override “equals” method 28](#_Toc450726695)

[5.2.8. Comparing enum type 29](#_Toc450726696)

[5.2.9. Must not forget to put try … catch in finally 30](#_Toc450726697)

[5.2.10. Abstraction or NOT Abstraction 31](#_Toc450726698)

[5.2.11. Documentation: comments in the source code 32](#_Toc450726699)

[5.2.12. The constants and configurable parameters are different 32](#_Toc450726700)

[5.2.13. Use enums instead of integer constants 33](#_Toc450726701)

[5.2.14. Never leave an empty catch block 33](#_Toc450726702)

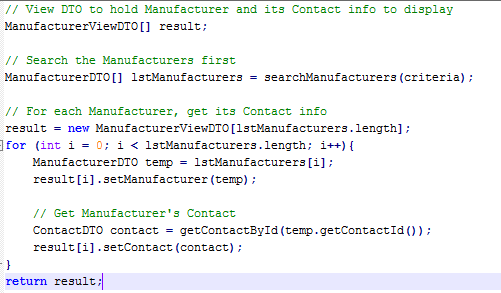
[5.2.15. Be specific in throw clause 34](#_Toc450726703)

1. Performance

## Database access

### N+1 selects

**Bad code/behavior**



SQL STATEMENT:

*SELECT \* FROM manufacturer WHERE location = 'HCMC';*

For each manufacturer found:

*SELECT \* FROM contact WHERE contact.id = :manufacturer\_found.contact\_id;*

Suppose we have a class Manufacturer with a many-to-one relationship with Contact. In other words, one contact can be related to many manufacturers (think, for example, of our contact as a reseller for one or more manufacturers, but with exclusive right to sell for the manufacturer).

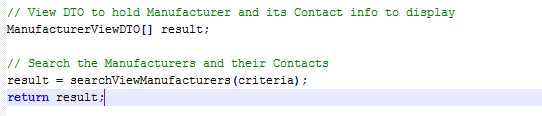
Let’s suppose we want to display all the names of the manufacturers in HCMC’s area together with their Contact’s names. It’s quite easy to make mistake like the above bad code (especially with OOP mind) by:

* First search all Manufacturers in HCMC => 1 select to DB.
* For each Manufacturer found, get its Contact => N selects.

**Solution**

The solution is to write 1 specific method to search for all Manufacturers in HCMC and their Contact info at once (one Select to DB by writing a join query). This solution even though is not perfect in terms of OOP design, but it is performant (remember we write program for machine to run, not for us).

**Good code/behavior**



SQL STATEMENT:

*SELECT \* FROM manufacturer AS man, contact AS con WHERE man.contact\_id = con.id AND man.location - 'HCMC';*

### DTO Usage

**Problem**

Method behavior granularity of “service interface”

**Bad code/behavior**

Suppose that we already have two DTOs;

public class DTOX

public class DTOY

…

DTOX x = loadDTOX();

doSomthingWithX(x);

DTOY y = loadDTOY();

doSomethingWithY(y);

**Description**

In the communication between layers, data are transferred (for example, client to EJB, EJB to Database, client to Database …) It is advised that we should try to build efficient DTOs to carry enough necessary information in order to reduce the calls via layers.

Laziness of building another object for wrapping DTOX and DTOY can be paid by bad performance when more calls are invoked via layers.

**Good code/behavior**

A new Bigger DTO

BiggerDTO biggerDTO = loadBiggerDTO();

doSomethingWithX(biggerDTO.getX());

doSomethingwithY(biggerDTO.getY());

* However, do not try to merge everything into a bigger object if they have nothing to do to each other and the performance is not a big issue in that case.

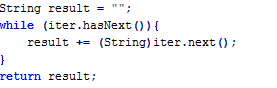
## Object initialization

### Use StringBuffer

**Problem**

Use StringBuffer (Java) or StringBuilder(C#)

**Bad code/behavior**

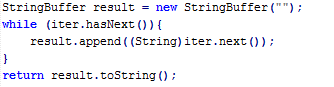


**Description**

Operation on String (immutable) objects creates a lot of short-live objects.

StringBuffer in Java or StringBuilder in C# provide an alternative.

**Good code/behavior**



## Collection Usage

### Avoid using HashTable and Vetor

**Problem**Avoid using Hastable and Vector.

**Bad code/behavior**

In un-threaded context

*Vetor m\_aCollection = new Vector();*

*m\_aCollection.get(i)*

In threaded context

*Hashtable m\_aCollection = new Hashtable();*

*If (!m\_aCollection.contains(someObject)){*

*m\_aCollection.add(someKey, someObject);*

*}*

**Description**

Accesses to Vector or Hashtable are synchronized => unthreaded context, this overhead is unnecessary.

In threaded context, m\_aCollection.contains(someObject) and m\_aCollection.add(someKey, someObject) should be in a single synchronized block. Therefore, the block of code

*if (!m\_aCollection.contains(someObject)){*

*m\_aCollection.add(someKey, someObject);*

*}*

is not correct although Hashtable already provides the synchronization facility.

**Good code/behavior**

Use ArrayList, LinkedList, HashMap … and provide synchronization block/method if necessary in threaded context

Unthreaded

*List m\_aCollection = new ArrayList();*

*m\_aCollection.get(i) ….*

Threaded

*HashMap m\_aCollection = new HashMap();*

*….*

*synchronized(this){*

*if (!m\_aCollection.contains(someObject)){*

*m\_aCollection.add(someKey, someObject);*

*}*

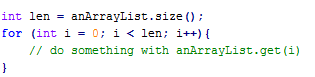
*}*

### Use Iterator instead of indexing for loop

**Problem**

Use Iterator instead of indexing for loop in Collections (List…)

**Bad code/behavior**

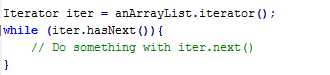
****

**Description**

Iterator is designed for best performance.

Using index is error-prone, so use alternatives if possible.

**Good code/behavior**

****

### List to Array

**Bad code/behavior**

*MyClass[] result = (MyClass[])myList.toArray(new MyClass[0]);*

**Description**

At the first glance, the above code looks fine. However, the problem reveals when we look at the implementation /javadoc of the method toArray(). Here is a brief explanation of what it does:

* Check to see if the size of the input array matches the size of the list.
* If yes, copy the data of the list to that array.
* If no, create a new array of the same type (as the input array) and copy the data of the list to it.

Therefore, we can see that the object (new MyClass[0]) is used in a very short time and after that it is not referred anymore by anyone and ready for garbage collected. We call such kind of objects short-lived objects. These objects are a burden for garbage collector.

The better way to do is described below, we we prepare an array that has the exact size of the list. This way, no new object is created.

*MyClass[] result = (MyClass[])myList.toArray(new MyClass[****myList.size()****]);*

**Good code/behavior**

### Arrays.asList(…)

**Bad code/behavior**

*List list = new ArrayList(array.length);*

*for (int i = 0; i < array.length; ++i){*

*list.add(array[i]);*

*}*

**Description**

In order to convert an array to a list, we do not need to make the code ourselves. There is a method named asList() in the utilities class Arrays that does this for us.

The class Arrays also contains some other useful method for array type.

**Good code/behavior**

*List list = Arrays.asList(array);*

**Pitfall**

Convert an array to a List only when you really need a List. Do not forget that most utilities (search, sort) for Collections are also available for arrays in Arrays.

### Unnecessary check when iterating collections

**Bad code/behavior**

*if (!performances.isEmpty()){*

*for (Performance anotherPerformance : performances){*

*….*

*}*

*}*

**Description**

The above code performs an unnecessary check to see if the list is empty, instead of trying to iterate it immediately. The check is not needed because if the list is indeed empty, the body of ‘for’ loop will not be executed regardless.

**Good code/behavior**

*for (Performance anotherPerformance : performances){*

*….*

*}*

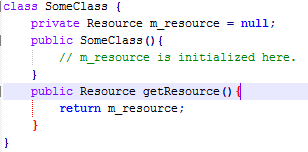
1. Resources management

## Resources allocation

### Use lazy initialization in memory-extensive applications

**Bad code/behavior**

Suppose m\_resource is very costly to initilize and is used sparsely.

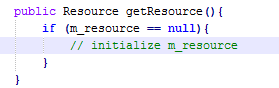
****

**Description**

On initialize m\_resource when necessary.

**Good code/behavior**

Initialize when needed:



In threaded context where the resource is concurrently accessed/ modified, use double-checked locking pattern.

### Initialize size of collection type

**Bad code/behavior**

*ArrayList anArrayList = new ArrayList();*

**Description**

Calling empty constructor creates a Collection of default size, which might be smaller than expected size. When the collection is about to be full, the size of the Collection will be extended together with copy operations, which is resource-consuming.

To avoid this, put an expected size in the constructor.

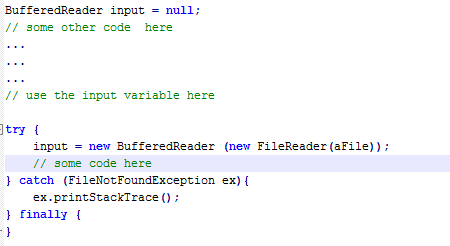
**Good code/behavior**

*ArrayList anArrayList = new ArrayList(expected\_size);*

Do not try to calculate precisely “expected\_size” if this calculation is complex. Better choose a best guess size, easy to calculate.

### Should declare local variable just before use

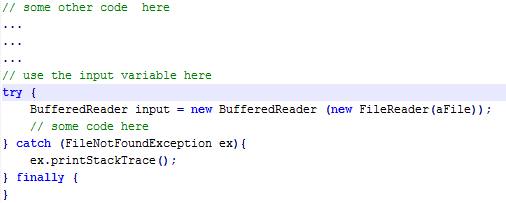
**Bad code/behavior**

****

**Description**

Declaring local variables without using them immediately will unnecessary increase their scope. This decreases legibility and increase the likelihood of error. It also hinders readability.

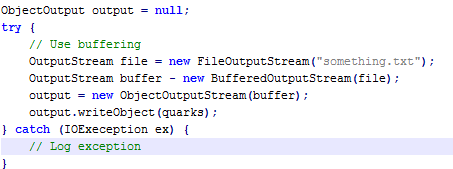
**Good code/behavior**

****

## Resources de-allocation

## Must not forget to close the stream

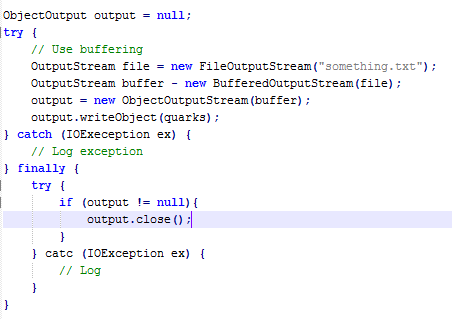
**Bad code/behavior**

****

**Description**

Streams represent resources which you must always clean up explicitly, by calling the close method.

**Good code/behavior**

****

**Comment**

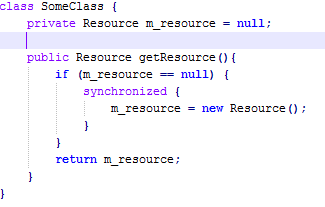
An additional best practice is to put the finally block code into a method of a utility class.

1. Integrity

## Object initialization

### Use double checked locking when lazy-initializing object in threaded context where resource is shared.

**Bad code/behavior**

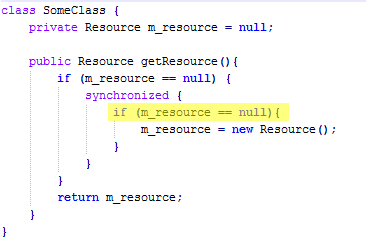


**Description**

Supposed more than one thread arrive at the getResource() method. All the threads found resource == null. All of them will take turn to initialize m\_resource, which is not desired.

A better way is to check m\_resource again to see if some other thread has already initialized it.

**Good code/behavior**

****

**Pitfall**

Double checked locking does not work with all JVM!! But at minimum, should do the single checked locking (i.e. put synchronized at method declaration)

## Comparing objects

### Comparing two objects (“.equals” instead of “==”)

**Bad code/behavior**

*if ((“abc” + “def”) == “abcdef”){*

*….*

*}*

**Description**

When we use the == operator, we are actually comparing two object references, to see if they point to the same object. We cannot compare, for example, two strings for equality, using the == operator. We muse instead use the “.equals” method, which is a method inherited by all classes from java.lang.Object.

**Good code/behavior**

*if ((“abc” + “def”).equals(“abcdef”)){*

*….*

*}*

### Comparing strings

**Problem**

Comparing variables with string constants

**Bad code/behavior**

*if (stringVariable.equals(“CONSTANT”)){*

*}*

**Description**

The code shown above may result in a NullPointerException being thrown if stringVariable to be null. Even if we know that it will be filled with a correct string, it is a good practice to compare strings as shown below, that doesn’t exists in an exception if the variable is indeed null.

**Good code/behavior**

*if (“CONSTANT”.equals(stringVariable))[*

*}*

1. Design related

## Design – OOP

### OOP – Encapsulation: declare a data member is public

**Problem**

Encapsulation: declare a data member is public

**Bad code/behavior**

*Public class ABC {*

*Public String m\_msg;*

*}*

**Description**

Declaring fields to be public is usually the wrong thing to do, since the caller is no longer protected from changes to class implementation. Declare fields as private. If fields need to be accessed by the caller, then provide necessary get and set methods.

**Good code/behavior**

*public class ABC {*

*private String m\_msg;*

*}*

### OOP – Encapsulation: Do not visible things that should not be visible

**Problem**

Encapsulation: do not make visible things that should be not visible. Not also attributes/methods in class but can also be inner classes.

**Bad code/behavior**

*public class ClassX {*

*private List m\_list;*

*public List getList();*

*public void setList(List list);*

*}*

**Description**

If m\_list is a private data that is updated inside class X, try to hide it instead of making it visible even via set/getter. The list will be got out via setter and updated outside at any time and is out of control of ClassX.

**Good code/behavior**

*public class ClassX(){*

*private List m\_list;*

*// try to support only necessary methods*

*public int size(){*

*}*

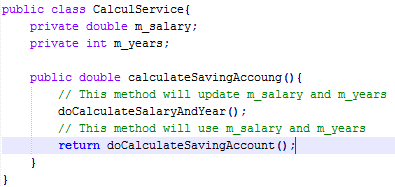
*public ConcreateClass get(int index){*

*}*

*}*

### OOP – Use inner class in order to avoid using member attributes

**Bad code/behavior**

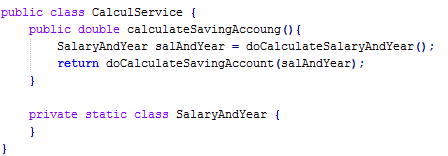


**Description**

This code is not good because we need to initiate CalculService for every call (in multi-thread). In this case, we should NOT create a new public class SalaryAndYear as this class is only used in this service.

This class is also static as it is technically independent on the CalculService, it is just placed inside CalculsService for being hidden from outside use.

**Good code/behavior**



### OOP – Isolation between technical handlings and business handlings

**Description**

Imagine that we have a service that is responsible for handling a business feature. And in order to handling that business, it has to do a lot of technical handlings (parsing file, string, calculate date, build xml content, …)

* Try to delegate responsibility of handling technical thing to another private method, or a utility class or another technical service.
* At least, a method should be kept as clean as we can see only either business or technical handlings inside it, and not see both.
* A class which is responsible for business handling should not have any public methods for technical handling.
* A class which is responsible for technical handling should not have any methods for business handling.
* …

### OOP – Isolation between independent businesses

**Description**

Try to isolate separate parts of code into separate methods, classes, packages, modules… (if two different businesses then at least two different classes/services. Not only two different methods).

* Try to delegate responsibility of handling technical thing to another private method, or a unit class or another technical service.
* At least, a method should be kept as clean as we can see only either business or technical handlings inside it, and not see both.
* A class which is responsible for business handling should not have any public methods for technical handling.
* A class which is responsible for technical handling should not have any methods for business handling.
* …

### OOP – Isolation between systems

**Description**

When two different systems, modules need to talk to each other, try to separate code for interfacing between two systems into separate classes. For example system A needs to interact to system B.

* Only few of classes of system A should depend on system B.
* Other parts of system A should not know much about system B.
* Modification on system B should only impact on the interfacing part of system A.

### OOP – Isolation between the generic algorithm and detail

**Description**

The generic algorithm and detail implementation

* Generic algorithm should be put in the abstract base class. For detail implementations that the base class cannot know how to do, we introduce abstract methods accordingly.
* A concrete class for a detail implementation will be responsible for fulfilling detail implementation by implementing abstract methods.

### OOP – Information expert, correct responsibility

**Description**

It is very important to have correct class be responsible for an implementation.

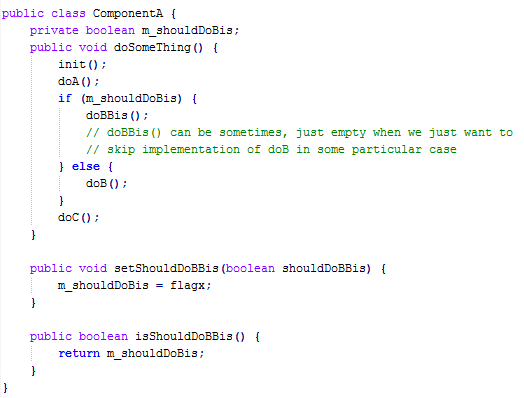
Hints to decide which class is proper in order to implement a method by asking ourselves these questions:

* Which class knows the best about the service that we are going to implement?
* If we put the implementation in this class, do we have a high cohesion to other parts of the class? If not, then it may not be the right class.
* If we put the implementation in this class, do we still have the loose coupling to other classes? If not, then it can be not the right class as this service requires so many other information that this class cannot provide.
* If we put this implementation in this class, are the name, the main objective and the comment about the class still valid and proper?

### Getter/Setter and Overriding

**Bad code/behavior**

Modify directly on ComponentA

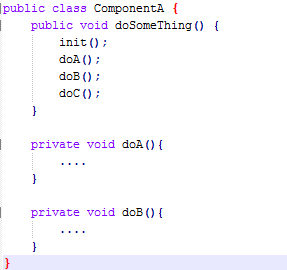


**Description**

Imagine that we have an existing class that we need to customize in order to support a new feature or to change the behavior.

The new behavior is not so common and really a kind of particular.

For example:



Now, we need to customize the implementation of doSomeThing a bit in some special cases in order to do some particular treatments.

Bad points:

* doBBis() can be, sometimes, specific and should not (or even cannot) be placed in a common component.
* Very confusing for client code. How to know that we should call setDoBBis before calling doSomeThing. What if client code forgets to call setDoBBis.

setDoBBis is introduced and is very rarely used except for our particular purpose.

**Good code/behavior**

As doB() can be probably customized, if it is private, it should be modified to protected.

*public class ComponentA{*

*protected void doB() {*

*}*

*}*

Introduce a new class to override doB

*public class ParticularComponentA extends ComponentA {*

*protected void doB() {*

*doBBis();*

*}*

*}*

### OOP - Decouple with real-implementation/coding to interface

**Bad code/behavior**

Take the typical case of DataAccess object (DAO). It is likely that the switch between datbase (Oracle to SQLServer for example) happens.

*OracleCustomerDAO myDAO = new OracleCustomerDAO();*

*myDAO.update(…);*

**Description**

Whennever a switch between database is required, the line *OracleDAO myDAO = new OracleDAO();*

has to be replaced everywhere it occurs.

To improve, code to interface. With that, changes in the real implementation do not affect the client code.

**Good code/behavior**

Declare an interface

*CustomerDAOInterface {*

*public void update(….);*

*}*

Following are example of possible DAOs with different database

*public class OracleCustomerDAO implements CustomerDAO{*

*public void update(…)[*

*// real implementation*

*}*

*}*

*public class SQLCustomerDAO implements CustomerDAO{*

*public void update(…)[*

*// real implementation*

*}*

*}*

Add a singleton factory

*public class DAOFactoy {*

*public CustomerDAO getCustomerDAO() [*

*// real from configuration file and produce*

*// correspoding DAO*

*// this method is responsible for creating*

*// DAO of specific database, according to*

*// configuration.*

*]*

*}*

In client code:

*CustomerDAO myDAO = DAOFactory.instance().getCustomerDAO();*

*myDAO.update(…);*

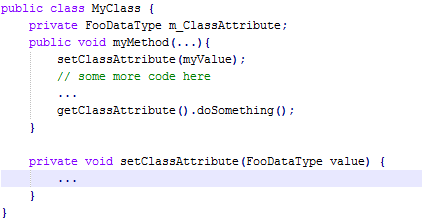
With this way, change in the interface only requires change in configuration file for the application to work.

### Member attributes access in multi-thread context

**Problem**

Accessing member (static or not) attributes without synchronization is not thread safe.

**Bad code/behavior**

****

**Description**

Some classes are meant to be used in a multi-threaded environment. In this case, the above code is not thread safe; because the value mClassAttribute could have been modified by a different thread before it is recalled, causing the code not to work correctly.

This applies particularly for singleton classes but not only!

As a rule of thumb, whenever some data members of a class are accessed by methods in this class in a multi-threaded context, any access should be synchronized.

## Pitfalls when applying design patterns

### Do not overuse the Singleton pattern

**Description**

When all you have is a hammer, everything else looks like a snail. But if your hammer is the Singleton pattern, think twice before you use it. Singletons may well be an easy way out to a design problem, but more often than not they are not the optimal choice.

When abused, the generally lead to tightly coupled functionality and hamper code reuse and maintenance. They are the OOP equivalent of global variables in procedural languages. As a rule of thumb, use Singleton for horizontal scope functionality such as login or authentication, or to control access to a thread sensitive resource.

**Comment**

When in doubt if it may be appropriate to apply the Singleton pattern for a particular design problem, consult with an expert or someone with more experience.

## Others

### Statements modifying the structure of database (DDL – Data Definition Language) are auto-committed

**Bad code/behavior**

*CREATE VIEW ….*

*Rollback;*

**Description**

Rollback statement does not effect that change, so we have to handle drop view … for example

**Good code/behavior**

*CREATE VIEW ….*

*DROP VIEW ….*

**Comment**

Using DDL in a program is not really a good practice! Try to avoid this.

### The interface java.io.Serialize must be understood properly

**Bad code/behavior**

Please do not implement it for all data objects.

**Good code/behavior**

Only apply it:

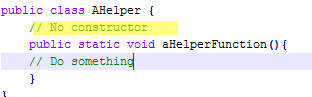
* Pass the objects via remote: RMI
* When using some cache libraries that can cache to disk.

1. Code Maintenance

## Object initialization

### Declare private constructor for Helper, Utility classes

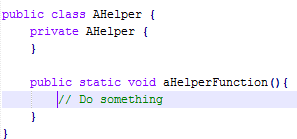
**Bad code/behavior**



**Description**

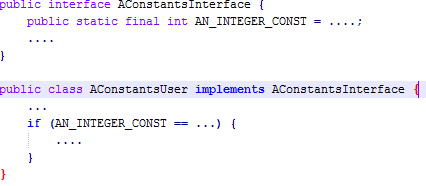
AHelper might be accidentally initialized. To avoid that, add a private constructor and the class cannot be initialized by invoking any constructor. The problem is that if AHelper was accidentally initialized, it creates short-lived objects then the Garbage Collector will collect for nothing.

**Good code/behavior**

****

### Declare constants in classes instead of interface

**Bad code/behavior**

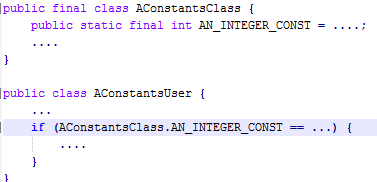


**Description**

Interfaces are not intended to be used like that.

Very unclear if we have a large amount of unqualified constant names.

**Good code/behavior**

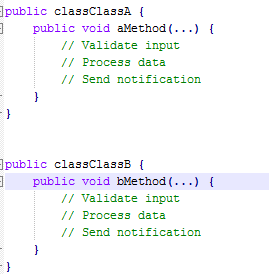
****

## Maintainability

### Use template method pattern if repeated structuring of code happens

**Bad code/behavior**

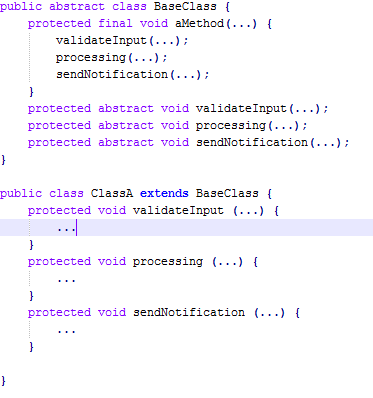
Suppose there are many similar processing with the same structures in many classes



**Description**

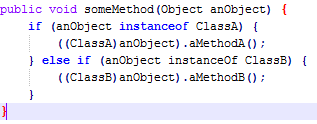
Repeating a common structure is error-prone. Code is not homogeneous.

**Good code/behavior**

****

### Avoid using instanceof method

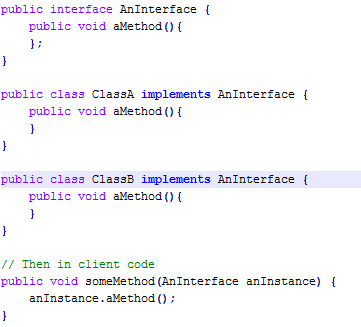
**Bad code/behavior**

****

**Description**

Polymorphism could provide much better code above.

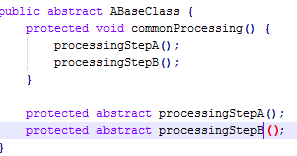
**Good code/behavior**

****

### Mark method, classes as final if they are not intended to be overridden

**Bad code/behavior**

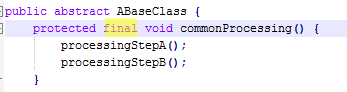
In method template where the sequence of method call is not intended to be changed



**Description**

Method commonProcessing() might be mistakenly changed in subclasses, which is against encapsulation and might cause undesired effect.

**Good code/behavior**

****

Note: marking a method/class as final hinders extensibility. In libraries or frameworks, be very careful because not all cases might have been foreseen. If it happens, it causes a lot of troubles since the final classes or methods cannot be extended.

**Pitfall**

Do not do that systematically, this is against the “open-close” principle of OOP! Do it only if you have serious reasons to think that people will misuse your class and make mistakes.

### Quote texts when logging

**Bad code/behavior**

*String m\_customerName;*

*…*

*S\_logger.debug(“Value of m\_customerName: “ + m\_customerName);*

**Description**

*customerName* might contain trailing blanks, which is not easy to see in the log file, which might be a source of confusion during debugging.

**Good code/behavior**

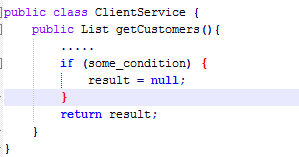
*String m\_customerName;*

*…*

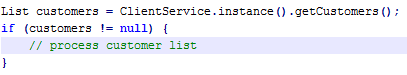
*S\_logger.debug(“Value of m\_customerName: \“ + m\_customerName + “\”);*

### Return empty list, array in public methods

**Bad code/behavior**

****

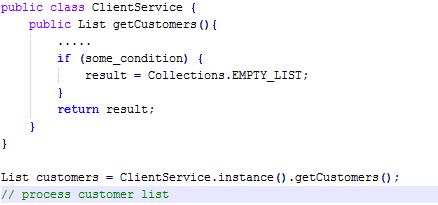
Client code fragment



**Description**

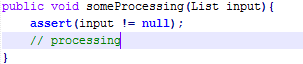
Make client code complex unnecessary.

**Good code/behavior**



### Do not use Assertion in place of business checking

**Bad code/behavior**

****

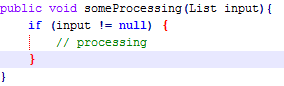
**Description**

Assertion is meant for programming error check and not for business code.

Assertion can be turned off -> Validation fails.

Assertion cannot throw specific Exception.

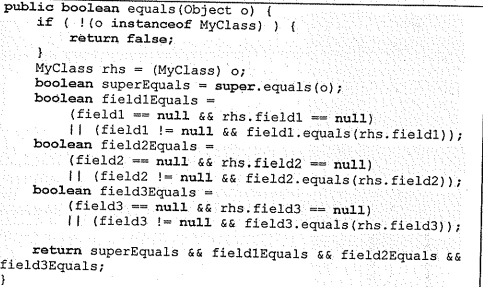
**Good code/behavior**

****

**Note:** however, if someProcessing(List input) is private, then the use of Assertion is acceptable, since in this case, normally the same developer will also write the caller of someProcessing and he should know the input condition of that method. Assertion in this case is used to prevent programming error.

### Override “equals” method

**Bad code/behavior**

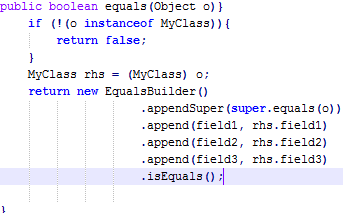
****

**Description**

It is not always easy to write correct and efficient equals method. The situation will become worse if the class contains array attributes.

Jakarta-Commons org.apache.commons.lang.builder.EqualsBuilder provids us a way to write the equals method easily and consistenly.

**Good code/behavior**

****

**Pitfall**

Create equals method only when needed.

If override equals method, must also override hashCode.

### Comparing enum type

**Bad code/behavior**

*TypeDossierState state = dossier.getState();*

*if (!(TypeDossierState.CLOSED.equals(state)){*

*….*

*}*

**Description**

The general idea of enum type is that we cannot create instance of that class other than the predefined ones. Thus we can actually use == to compare. It is more natural and increase readability.

**Good code/behavior**

*TypeDossierState state = dossier.getState();*

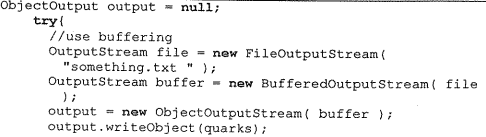
*if (state != TypeDossierState.CLOSED){*

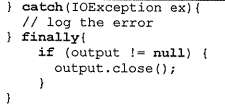
*….*

*}*

### Must not forget to put try … catch in finally

**Bad code/behavior**

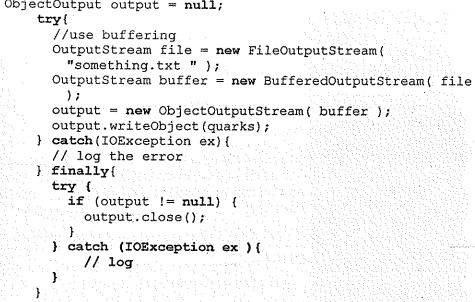
****



**Description**

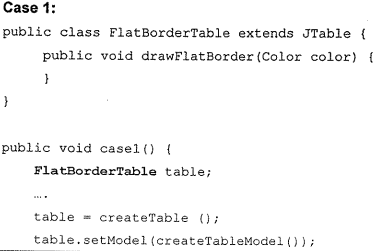
Many finally clauses in turn contain a try … catch block. This is because many of the methods called in a finally clause can themselves throw an exception.

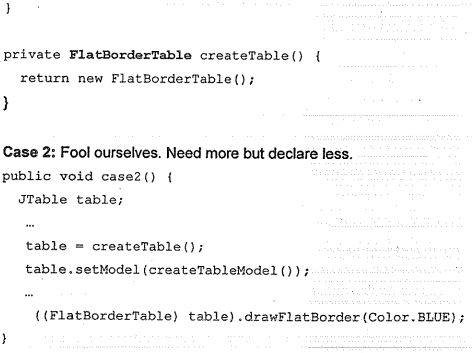
**Good code/behavior**

****

### Abstraction or NOT Abstraction

**Bad code/behavior**

****

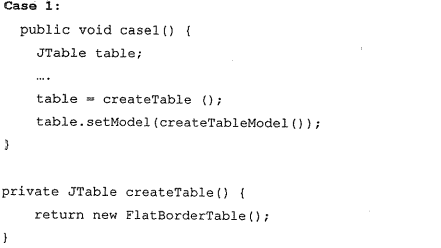
****

**Description**

It is advised that when we declare a reference, we should try to declare as abstract as possible. For example, in declaration, we should use List, Map, Set, Collection, … instead of ArrayList, HashMap, HashSet …

Someimes, this is not used, and sometimes, this is over-used.

**Good code/behavior**

****

### Documentation: comments in the source code

**Bad code/behavior**

There is a method: createUser, and its comments are: create User.

* It is useless comment.

**Description**

The code must be documented properly. It must be respected by ALL developer. Do not put the non-valuable comment. When you change the code, please check the comment if it is still valid. Make sure that the comment is matched with your code.

Tip: think carefully what the code will be developed. Then write them down: All the things in your mind (i.e. the logic or business or algorithm …)

**Good code/behavior**

Try to think about business of method:

This method will create a new user for the system. It will be used to logon to the system. When the user is created, the password must be changed at the first time of logon.

### The constants and configurable parameters are different

**Description**

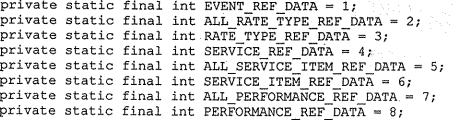
If you think the value will be never changed, or if we change it, we have to rebuild the system. Then put it as constants.

Else make it as configurable parameters. It means it will be configured when we install it in different environment (TEST, PROD).

But if you put too much parameter for the configuration, then it is difficult for the installation.

### Use enums instead of integer constants

**Bad code/behavior**

****

**Description**

Whenever you find yourself writing the above code to represent internal states or flags of a module, consider using enums instead. Enums are a better approach than integer constants because they are forced to fit within the range of allowed values, whereas using an int variable it can contain the whole range of integers even if only a few of them are valid states.

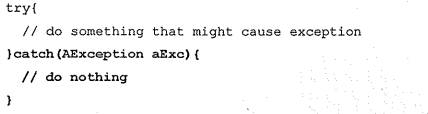
Also, enums ease debugging and login because you can print the human readable value by directly invoking the method toString(), while when using integers you would have to write a special purpose function to provide this functionality.

**Good code/behavior**



### Never leave an empty catch block

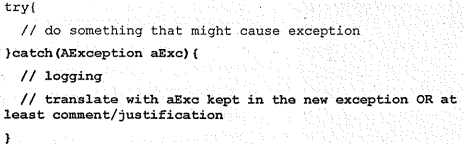
**Bad code/behavior**

****

**Description**

There is no easy way to trace back to original source of error.

**Good code/behavior**

****

### Be specific in throw clause

**Bad code/behavior**

****

**Description**

Difficult to figure out what specific error has occurred.

**Good code/behavior**

****