Règles de codage Java

Version 0.1

03/10/2014

Dominique FRANCISCI

Règles de codage Java		

LISTE DE DIFFUSION				
Organisation	**Nom / Rôles**	**Validation**	**Information**	**Action**
USTV - FDS	Elisabeth MURISASCO		X	
USTV - FDS	Etudiants		X	

HISTORIQUE DES VERSIONS			
Version	**Date**	**Modification**	**Auteur**
0.1	03/10/2014	Création du document	Dominique FRANCISCI

Table des Matières

Index des tableaux

1 INTRODUCTION

1.1 OBJET DU DOCUMENT

Ce document décrit certaines règles de codage devant idéalement être respectées par les étudiants dans le cadre des projets Java.

Les objectifs du présent document sont :

- D'améliorer la qualité du code.
- De favoriser la collaboration.
- D'appliquer un standard.
- De faire détecter les défauts.
- De sensibiliser les développeurs.

1.2 GLOSSAIRE

Terme	Sigle	Définition
Application Programming Interface	API	Interface de Programmation Applicative
American Standard Code for Information Interchange	ASCII	Code américain normalisé pour l'échange d'information
Java Development Kit	JDK	Ensemble de bibliothèques logicielles de base du langage de programmation java
Java Virtual Machine	JVM	Machine Virtuelle Java
NullPointerException	NPE	Sans objet
Uniform Resource Locator	URL	Localisateur uniforme de ressource

2 REGLES

Ce chapitre présente l'ensemble des catégories de règles décrites dans la suite du document. Ces catégories sont résumées dans le tableau ci-dessous.

Catégorie de règles	Description
Collections	Ensemble de règles en lien avec la mise en œuvre de collections Java.
Conventions de nommage	Ensemble de règles en lien avec l'application de conventions de nommage.
Environnements concurrentiels	Ensemble de règles en lien avec les environnements concurrentiels.
Exceptions	Ensemble de règles en lien avec la mise en œuvre d'exceptions.
Journalisation	Ensemble de règles en lien avec la journalisation.
Performances	Ensemble de règles en lien avec les performances des applications.
Portée	Ensemble de règles en lien avec la notion de portée.
Structures de contrôle	Ensemble de règles en lien avec les structures de contrôle Java.

Tableau 1 : catégories de règles

2.1 COLLECTIONS

Cette section décrit l'ensemble des règles en lien avec la mise en œuvre de collections Java.

2.1.1 Classe concrète vs interface

➔ **Na pas** déclarer un attribut d'instance ou de classe, un paramètre formel ou encore un objet local comme instance d'une classe concrète.

➔ **Déclarer** plutôt cet élément via une interface.

➔ **Raisons** :

- Diminuer le couplage.
- Minimiser les dépendances.
- Réduire l'impact sur le code des changements de déclaration.
- Favoriser la réutilisabilité.

➔ **Exemple** :

```
private ArrayList<String> mList1;         // KO !

private List<String>      mList2;         // OK
```

```
private static HashMap<String, int> sMap1
        = new HashMap<String, int>();     // KO !

private static Map<String, int> sMap2     // OK
        = new HashMap<String, int>();
```

```
public void func1(HashSet<String> pSet) { // KO !
    // ...
}

public void func2(Set<String> pSet) {     // OK
    // ...
}
```

2.1.2 Type de retour des méthodes

➔ **Eviter** de définir une méthode par le biais d'une classe concrète comme type de retour.

➔ **Définir** plutôt cette méthode via une interface.

➔ **Raisons** :

- Diminuer le couplage.
- Minimiser les dépendances.
- Réduire l'impact sur le code des changements de déclaration.
- Favoriser la réutilisabilité.

➔ **Exemple** :

```
public ArrayList<String> getList1() { // KO !
    return list;
}

public List<String>      getList2() { // OK
    return list;
}
```

2.1.3 Collections « thread safe »

➔ **Eviter** de mettre en œuvre les classes `Vector` et `HashTable` en dehors d'un environnement concurrentiel.

➔ **Utiliser** plutôt leurs équivalences non « thread safe » : `ArrayList` et `HashMap`.

➔ **Raison :** ces collections « thread safe » sont de l'ordre de cinq fois plus coûteuses que leurs équivalences non « thread safe ». Elles continuent à être maintenues dans les JDK bien que dépréciées depuis longtemps. La raison : leur efficacité en environnements concurrentiels.

2.1.4 Collections de cardinalité constante

➔ **Eviter** de mettre en œuvre les collections implémentant l'interface `List` lorsque le nombre d'éléments qu'elles devront contenir est fixe.

➔ **Utiliser** plutôt un tableau « classique ».

➔ **Raison** : ces collections utilisent inutilement des ressources dès lors que la taille du tableau est constante.

➔ **Exemple :**

```
private static final STRINGS_COUNT = 100;
…
List<String> lList1
        = new ArrayList<String>(STRINGS_COUNT); // KO !

String[] lList2 = new String[STRINGS_COUNT];    // OK
```

2.2 CONVENTIONS DE NOMMAGE

Cette section décrit l'ensemble des règles en lien avec l'application de conventions de nommage.

2.2.1 Constantes chaînes de caractères

➔ **Eviter** l'emploi de constantes chaînes de caractères directement dans le code.

➔ **Définir** celles-ci en tant qu'attributs de classe non modifiables (c'est-à-dire préfixées des mots-clés « `static` » et « `final` »).

➔ **Raisons** :

- Permet d'éviter de déclarer dans le code plusieurs entités comportant la même valeur.
- Modifier le contenu d'une telle constante (« `static final` ») permet une « répercussion » de cette modification partout où cette constante est employée.

➔ **Exemple** :

```
public void f() {
    executeQuery("select * from tableX");          // KO !
}

public static final QUERY = "select * from tableX;";

public void f() {
    executeQuery(QUERY);                           // OK
}
```

2.2.2 Caractères non ASCII

➔ **Eviter** l'emploi de caractères non ASCII.

➔ **Raison** : ce type de caractères n'est pas affichable. Ceci peut donc provoquer des effets de bord gênants.

2.2.3 Conventions de nommage

➔ **Nommer** les identifiants :

- De manière explicite : leur nom doit dénoter le contenu de la fonction de l'objet nommé.

- De manière succincte : il est difficile d'utiliser un identifiant comportant beaucoup de caractères.
- De manière à ce que les acronymes apparaissent entièrement en majuscules.

2.2.3.1 Packages

➔ **Nommer** les packages :

- En minuscules.
- De manière à ce qu'ils soient uniques.
- Structurer par exemple le nommage par le nom de la société « `com.dcns` », suivi du nom du projet ou du composant, puis du sous-composant et ainsi de suite.
- De manière à ce que les acronymes apparaissent entièrement en majuscules.

2.2.3.2 Classes et interfaces

➔ **Nommer** les classes et les interfaces :

- En utilisant la pratique « CamelCase » (consiste à écrire un ensemble de mot en mettant en majuscule la première lettre des mots liées).
- Avec des identifiants les plus simples possibles et décrivant au mieux l'interface ou la classe concernée.
- En utilisant les préfixes « C », « A » et « `I` » respectivement pour classe, classe abstraite et interface. De plus une interface doit être suffixée de « `able` ».

2.2.3.3 Méthodes

➔ **Nommer** les méthodes :

- En faisant commencer l'identifiant par un verbe à l'infinitif.
- En utilisant la pratique « lowerCamelCase » identique à la pratique « CamelCase » avec la contrainte suivante : la première lettre de l'identifiant doit être une minuscule.

2.2.3.4 Attributs, paramètres et variables

➔ **Nommer** les attributs d'instance / de classe, les paramètres formels et les variables locales :

- En utilisant la pratique « lowerCamelCase » présentée précédemment.
- De la façon la plus simple possible, en décrivant au mieux l'élément associé.
- En évitant de faire contenir à l'identifiant, tout acronyme ou abréviation (excepté pour les abréviations et acronymes courants comme par exemple « API », « URL », …)
- En évitant de faire contenir le symbole « $ » ou de soulignement « _ » à l'identifiant.
- En ne précisant pas le type associé en préfixe ou en suffixe.
- A noter que les identifiants de collections d'objets doivent être au pluriel.

2.2.3.5 Constantes

➔ **Nommer** les constantes :

- En utilisant que des majuscules.
- En utilisant uniquement le symbole « _ » comme séparateur.

2.3 ENVIRONNEMENTS CONCURRENTIELS

Cette section décrit l'ensemble des règles en lien avec les environnements concurrentiels.

2.3.1 Codes « synchronisés »

➔ **Eviter** les méthodes et les blocs « `synchronized` » autant que possible.

➔ **Eviter** également de synchroniser les méthodes d'écouteurs Swing car celles-ci seront forcément appelées dans l'EDT.

➔ **Déplacer** un maximum de code à l'extérieur de la portion de code synchronisée.

➔ **Préférer** l'emploi d'un objet « verrou » déclaré « `final` » (ceci afin de garantir qu'il ne sera pas modifié à son tour).

➔ **Raison** : améliorer les performances.

2.4 EXCEPTIONS

Cette section décrit l'ensemble des règles en lien avec la mise en œuvre d'exceptions.

2.4.1 Capturer les NPE

➔ **Eviter** autant que possible l'approche (palliative) consistant à tester les valeurs « `null` » afin d'éviter les exceptions de type « `NullPointerException` ». **Attention** : ne pas interpréter cette règle comme étant une invitation à ne pas tester les valeurs « `null` » ; l'interpréter plutôt comme une règle proposant de considérer, tous les cas de figure afin d'éviter autant que possible l'apparition de situations menant à la levée de cette exception.

➔ **Utiliser** autant que possible l'approche consistant à détecter en amont cette problématique :

- Ne pas utiliser le mot clé « `null` » pour initialiser un objet (le compilateur génèrera une erreur de type « variable x might not have been initalized ».
- Utiliser « `@NotNull` ».

➔ **Raisons** :

- En effet, éviter ce type d'erreur empêchera certes la levée d'une exception, mais surtout, masquera potentiellement un problème plus profond. En effet, une instance à « `null` » ne signifie pas forcément que cette instance n'a jamais été initialisée.
- Améliorer la robustesse du code.

2.4.2 Eviter de lever une NPE

➔ **Eviter** de lever une exception NPE.

➔ **Utiliser** plutôt une exception spécifique.

➔ **Raison** : les NPE doivent uniquement être levées par la JVM.

➔ **Exemple** :

```
throw new NullPointerException("Missing argument");     // KO !
throw new IllegalArgumentException("Missing argument"); // OK
```

2.4.3 Eviter de lever des exceptions de « bas niveau »

➔ **Eviter** de lever une exception de bas niveau comme `Throwable`, `Error` ou `RuntimeException`.

➔ **Lever** plutôt une exception dérivée de ces dernières.

➔ **Raison** : lever une exception de « bas niveau » donne que peu de renseignement sur l'erreur produite. A contrario, une exception dérivée permet de donner une raison explicite sur l'erreur, ne serait-ce que par l'intermédiaire du nom de la classe de cette exception.

➔ **Exemple** :

```
throw new RuntimeException("Cannot find conf. file : " + path); // KO !
throw new FileNotFoundException(path);                          // OK
```

2.4.4 Bloc « catch » vide

➔ **Eviter** les blocs « `catch` » vides.

➔ **Utiliser** plutôt un « logger » ou **lever** une exception.

➔ **Raison** : si un bloc « `catch` » est vide, l'exception est tout simplement ignorée.

2.4.5 Flux de contrôle vs exceptions

➔ **Eviter** d'utiliser la levée d'exceptions pour « casser » un flux d'exécution (par exemple capturer une NPE au lieu de tester une valeur à « `null` »).

➔ **Raison** : créer une instance d'exception est une opération coûteuse.

➔ **Exemple** :

```
try {
    service.doSomething();
} catch(NullPointerException e) {
    alternativeService.doSomethingElse(); // KO !
}
```

```
if(service != null) {
    service.doSomething();
} else {
    alternativeService.doSomethingElse(); // OK
}
```

2.4.6 Eviter l'appel à « printStackTrace »

➔ **Eviter** l'emploi systématique de la méthode « `printStackTrace()` ».

➔ **Utiliser** plutôt un « logger ».

➔ **Raison** : les informations fournies par cette méthode sont à usage exclusif des développeurs et ne sont d'aucune utilité pour l'utilisateur final.

➔ **Exemple** :

```
try {
    Integer.parseInt(strValue);
} catch(NumberFormatException exception) {
    exception.printStackTrace(); // KO !
}
```

```
try {
    Integer.parseInt(strValue);
} catch(NumberFormatException exception) {
    if (LOGGER.isErrorEnabled()) {
        LOGGER.error("Cannot parse value.", exception); // OK
    }
}
```

2.4.7 L'exception RuntimeException

➔ L'exception `RuntimeException` **doit être capturée** dans des cas suivants, et tracée sans utiliser le niveau « FATAL » :

- Thread.
- DLRL listener.

- MANAGEMENT listener.
- EVENT slots.

➔ **Raison** : sans bloc « `catch` », l'erreur est perdue, et le thread est laissé dans un état inconnu.

➔ **Exemple :**

```
class UpdateTask implements Runnable {
    public void run() {
        // The whole method body should be wrapped with a try/catch block :
        // this pattern should be applied to DLRL listeners, MANAGEMENT listeners
        // and EVENT slot actions.
        try {
            doRun();
        } catch(RuntimeException e) {
            // do something :
            // you are allowed to use the ERROR level
            // for logging the error in this case
            LOGGER.error("Unexpected error", e);
        }
    }
}
```

2.4.8 Instruction "return" dans bloc "finally"

➔ **Eviter** d'utiliser l'instruction « `return` » dans un bloc « `finally` » .

➔ **Déplacer** le « `return` » en dehors du bloc « finally ».

➔ **Raison** : un « `return` » dans un bloc « `finally` » peut lever une exception, ce qui cachera toute autre exception levée dans le bloc « `try / catch` » lui-même.

➔ **Exemple :**

```
public class Bar {
    public String foo() {
        try {
            throw new Exception("My Exception");
        } catch (Exception e) {
            throw e;
        } finally {
            return stringNull.willHideError(); // KO !
            // If an error is raised with the return statement,
            // it will hides any errors from the upper catch block
        }
    }
}
```

2.4.9 Journalisation explicite

➔ **Eviter** d'utiliser les méthodes de journalisation avec un seul argument : l'exception.

➔ **Utiliser** plutôt les méthodes comportant deux paramètres :

- La description de l'exception.
- L'exception elle-même.

➔ **Raison** : le premier paramètre renseigne sur l'exception levée, ce qui rend celle-ci plus compréhensible.

➔ **Exemple :**

```
public class Main {
    private static final InternationalizingLog LOG =
       InternationalizingLogFactory.getInternationalizingLog(Main.class);


    void bar() {
        try {
            // ...
        } catch(Exception e) {
            LOG.error(e);                                          // KO !
        } catch(OtherException oe) {
            LOG.error("Some message to explain the error", oe); // OK
        }
    }
}
```

2.4.10 Tracer une exception

➔ **Eviter** de « tracer » une exception dans un bloc « `try / catch` » en utilisant le niveau « ERROR ».

➔ **Traiter** l'exception ou la propager via une autre exception.

➔ **Raison** : le niveau « ERROR » est réservé au erreurs inattendues.

➔ **Exemple :**

```
try {
    // do something...
} catch(NumberFormatException e) {
    LOGGER.error("Cannot parse value", e);                      // KO !
    LOGGER.debug("Cannot parse value: using default value", e); // OK
    value = DEFAULT_VALUE
    throw new IllegalArgumentException("Invalid int value", e); // OK
}
```

2.5 JOURNALISATION

Cette section décrit l'ensemble des règles en lien avec la journalisation.

2.5.1 Création de Logger

➔ **Créer** systématiquement un logger comme instance de la classe `InternationalizingLog` à partir de la brique PROGRAMMING_CORE.

➔ **Exemple :**

```
public class Foo {
    public static final InternationalizingLog LOGGER =
        InternationalizingLogFactory.getInternationalizingLog(Foo.class);
}
```

2.5.2 Appel de Logger

➔ **Eviter** d'appeler une méthode de journalisation d'un « logger », sans tester au préalable que cet appel est permis.

➔ **Englober** systématiquement un appel de méthode de journalisation dans un « `if` ».

➔ **Raisons** :

- La journalisation est une opération coûteuse (principalement l'instanciation de la chaîne de caractères passée en paramètre à la méthode de journalisation).
- La journalisation est également potentiellement polluante.

2.5.3 Niveau de journalisation « FATAL »

➔ **Eviter** d'utiliser la journalisation avec le niveau « FATAL ».

➔ **Utiliser** plutôt la journalisation avec le niveau « ERROR ».

➔ **Raison** : ce niveau est réservé aux erreurs inattendues.

2.5.4 La méthode System.println

➔ **Eviter** d'utiliser les méthodes `System.out.print*` ou `System.err.print*` pour produire des traces.

➔ **Utiliser** plutôt la journalisation avec le niveau « DEBUG ».

➔ **Raisons** :

- La journalisation est une approche formalisée, celle consistant à utilise les méthodes « `print` » non.
- Ce type d'approche ne permet à priori pas de conserver les traces.

2.6 PERFORMANCES

Cette section décrit l'ensemble des règles en lien avec les performances des applications.

2.6.1 Accesseurs complexes

➔ **Eviter** d'implémenter des « setters » et « getters » complexes, c'est-à-dire réalisant de nombreuses voire très nombreuses opérations, que ce soit de manière directe ou bien indirecte (c'est-à-dire en invoquant eux-mêmes des méthodes complexes).

➔ Dans l'absolu, de telles méthodes d'accès aux attributs de la classes doivent se limiter strictement aux deux fonctions suivantes : d'une part valuer l'attribut par le biais du « setter » associé et d'autre part, retourner cet attribut via le « getter » correspondant.

➔ **Raisons** :

- Il est fréquent que ces méthodes d'accès aux attributs soient invoquées de très nombreuses fois, par exemple dans des boucles, etc… Faire faire autre chose que l'accès strict aux attributs revient à dérouler dans ces boucles une grande quantité de code, provoquant ainsi une dégradation des performances de l'application.
- « Traditionnellement », ces méthodes sont préfixées respectivement par « set » pour les « setters » et par « get » pour les « getters ». Il n'est donc logique du point de vue des règles de nommage, de faire réaliser plus d'actions à ces méthodes, que ne le suggère leur nommage.

2.7 PORTEE

Cette section décrit l'ensemble des règles en lien avec la notion de portée.

2.7.1 Déclarer la visibilité des méthodes

➔ **Eviter** d'utiliser la visibilité de package pour une méthode.

➔ **Utiliser** plutôt la visibilité « `public` », « `private` » ou « `protected` ».

➔ **Raison** : la visibilité de package rend la méthode visible à toute classe du package courant, ce qui peut ne pas être attendu.

➔ **Exemple :**

```
void f1() {        // KO !
    // ...
}

public void f2() { // OK
    // ...
}
```

2.7.2 Portée globale vs portée locale

➔ **Eviter** de cacher la visibilité des attributs de classe ou d'instance, par la déclaration de variables ou d'objets locaux ayant les mêmes identificateurs.

➔ **Utiliser** des constantes ou encore des types énumérés.

➔ **Raison** : éliminer les ambiguïtés.

2.7.3 Visibilité des variables

➔ **Eviter** d'utiliser une visibilité différente de « `private` » ou « `protected` » pour un attribut d'instance ou pour un attribut de classe.

➔ **Eviter** d'utiliser la visibilité de package.

➔ **Raisons** :

- La visibilité « `public` » est inutile du fait que chaque attribut a normalement son « setter » et son « getter » associés.
- La visibilité de package permet une visibilité des attributs de la classe à toutes classes dans le package courant ce qui peut provoquer des problèmes.

➔ **Exemple** :

```
public int value1 = 100;          // KO !
int value2 = 200;                 // KO !
public static int value3 = 300;   // KO !
static int value4 = 400;          // KO !
```

2.8 STRUCTURES DE CONTROLE

Cette section décrit l'ensemble des règles en lien avec les structures de contrôle.

2.8.1 Switch et constantes

➔ **Eviter** d'utiliser des constantes numériques dans un « `switch` ».

➔ **Utiliser** des constantes ou encore des types énumérés.

➔ **Raisons** :

- Augmenter la lisibilité.
- Augmenter la compréhensibilité.

2.8.2 Chaînes de caractères vides

➔ **Ne pas** utiliser la syntaxe suivante pour tester si une chaîne de caractère est vide :

```
String string = …;
…
if ((string != null) && («».equals(string) == true) {
    // …
}
```

➔ **Utiliser** plutôt cette syntaxe :

```
String string = ...
...
if (StringUtils.isEmpty(string) == true) {
    // ...
}
```

➔ **Raisons** :

- Le premier test est plus coûteux que le second.
- Le premier test est moins lisible que le second.

2.8.3 Boucle « for each »

➔ **Eviter** la mise en œuvre d'une boucle « `for each` » sur une liste, sans tester au préalable la « nullité » de cette liste.

➔ **Utiliser** un test avant d'itérer sur une telle liste, et ce de manière systématique.

➔ **Raison** : la boucle « `for each` » lève une exception de type « `NullPointerException` » si la liste est à « `null` ».

➔ **Exemple** :

```
List<String> list = null;
...
for(String s : list) { // KO !
    // ...
}

if (list != null) {    // OK
    for(String s : list) {
        // ...
    }
}
```

2.8.4 Création d'objets et boucles

➔ **Eviter** de créer des objets dans le corps des boucles.

➔ **Déporter** autant que possible en amont, l'allocation de ces objets.

➔ **Raison** : pour une boucle comportant n itérations, allouer un objet une fois au lieu de n fois est évidemment préférable pour une raison de performance : en effet, allouer n fois un objet, revient à dérouler n fois son constructeur ainsi que les constructeurs de ses classes de bases successives ce qui peut être très coûteux.

2.8.5 « Return » multiples

➔ **Eviter** la présence de plusieurs « `return` » au sein d'une même méthode.

➔ **Utiliser** évidemment un seul return par méthode.

➔ **Raisons** :

- Améliorer la visibilité.
- Ancienne pratique issue de « SESE » (par forcément adaptée à Java).

➔ **Exemple** :

```
public int factoriel(int n) {
    if (n < 0) {
        return -1; // -1 ⇔ error  // KO !
    } else if (n == 0) {
        return 1;                 // KO !
    } else if (n >= 0) {
        return n * factoriel(n-1), // KO !
    }
}
```

```
public int factoriel(int n) {
    int result = -1;


    if (n == 0) {
        result = 1;
    } else if (n >= 0) {
        result = n * factoriel(n-1),
    }
    return result; // OK
}
```

2.9 DIVERS

Cette section décrit l'ensemble des règles n'appartenant pas aux catégories décrites dans les sections précédentes.

2.9.1 Code « déprécié »

➔ **Eviter** l'implémentation de code « déprécié » (deprecated).

➔ **Mettre à jour** autant que possible le code concerné par ce que propose « nouvellement » le JDK en place.

➔ **Raison** : une fonctionnalité est généralement passée à « dépréciée » pour deux raisons :

- Soit celle-ci est non sécurisée. Dans ce cas, il n'y a pas de garantie qu'elle soit remplacée dans une future version de JDK. Elle est simplement passée à « dépréciée » pour alerter le développeur.
- Soit celle-ci est remplacée par une meilleure approche.

2.9.2 Les « MagicNumber »

➔ **Eviter** l'utilisation « en dur » de constantes numériques au sein du code. De telles constantes sont appelées « MagicNumber ».

➔ **Utiliser** autant que possible les expressions symboliques (c'est-à-dire les constantes déclarées « `static final` » dans une classe).

➔ **Raisons** :

- Facilite la maintenance.
- Facilite la visibilité.
- Facilite la compréhensibilité.

➔ **Exemple** :

```
public class Bar {
    private final static int DEFAULT_VALUE = 3; // OK
```

```
    public void foo() {
        int value = 87;                         // KO !
        value += DEFAULT_VALUE;                 // OK
    }
}
```