Collège de Bois-de-Boulogne

Guide de style Java

Normes de codage à appliquer dans les travaux de programmation

Stéphane Lévesque
Version du 6 avril 2020

Table des matières

Introduction

Les directives suivantes portent sur la mise en forme du code source en Java. Ils correspondent aux règles standardisées de l'industrie proposées par Sun Microsystems (maintenant Oracle), avec des suppléments provenant d'autres sources présentées en bibliographie.

Quelques règles et pratiques recommandées ont été ajoutées en complément.

Un guide de style est un complément à la syntaxe d'un langage. Le non respect de ces conventions n'empêchera généralement pas votre programme de fonctionner, mais sa lecture par un programmeur sera plus ardue. La syntaxe d'un langage rend votre programme compréhensible par votre ordinateur. Le rôle d'un guide de style, c'est de rendre votre programme compréhensible par un humain.

Nomenclature

Règle générale : accents

Bien qu'officiellement supportés, l'encodage des accents crée généralement beaucoup de problèmes contre peu d'avantages.

Il est fortement déconseillé d'utiliser les caractères accentués à l'extérieur des commentaires ou dans le contenu des variables/constantes.

Pourquoi ? Les fichiers peuvent être encodés avec plusieurs jeux de caractères, et les codes des caractères accentués ne sont pas toujours les mêmes, ce qui cause des erreurs de compilation lorsqu'on change d'environnement.

Fichiers

Un fichier doit porter le nom de la classe qu'il contient et ne doit contenir qu'une seule classe, sauf dans le cas particulier des classes internes.

La classe `Tp2` sera donc dans le fichier source `Tp2.java`.

Pourquoi ? Avoir une classe par fichier simplifie la recherche des classes.

Classes

Nomenclature

La première lettre est toujours une majuscule, les suivantes sont en minuscules. Les séparations de mots se font à l'aide de majuscules. On ne devrait retrouver aucun caractère qui n'est pas une lettre.

Les noms des classes devraient être des noms communs ou propres plutôt que des verbes ou des adjectifs.

Conformes : `ClientCorporatif, Fournisseur.`

Incorrects : `Clientcorporatif, fournisseur, Client_Particulier, Clavier2, TraiterClient.`

Ordre des déclarations

Les déclarations devraient apparaître dans cet ordre :

1. package
2. import
3. classe/interface/énumération
4. constantes de classe
5. variables de classe
6. constantes d'instance
7. variables d'instance
8. constructeurs
9. accesseurs/mutateurs
10. autres méthodes
11. classes internes

Pourquoi ? Tous les éditeurs ne classent pas les déclarations dans une liste séparée. Un ordre précis facilite la recherche selon le type d'élément.

Constantes

Déclaration

Il ne devrait y avoir qu'une déclaration de variable par ligne.

Pourquoi ? Ça facilite le repérage et l'initialisation des variables, ça permet la documentation avec la Javadoc.

Lorsque possible, les constantes devraient être initialisées à leur déclaration.

Pourquoi ? Ça facilite le repérage de la valeur assignée à la constante.

Nomenclature

Les constantes devraient être en majuscules, en séparant les mots par des traits de soulignement. Le nom devrait indiquer très clairement la nature de la donnée qui s'y trouve.

Conforme : `public static final String NOM_COLLEGE = "Bois-de-Boulogne";`

Utilisation

Les constantes devraient être utilisées pour représenter toute donnée numérique non triviale.

Pourquoi ? Les nombres n'ont pas nécessairement un sens clair pour le lecteur. Le même nombre peut représenter plusieurs valeurs distinctes.

Conforme :

```
public static final double TAUX_TPS = 0.05;
[…]
montantTps = sousTotal * TAUX_TPS;
```

Inutile :

```
public static final int ZERO = 0;
public static final int DEUX = 2;
[…]
if (age % DEUX == ZERO) {
[…]
```

Variables et paramètres

Nomenclature

Les variables et paramètres devraient être en minuscules, avec des majuscules pour couper les mots. Dans le cas d'un acronyme, on ne fait pas d'exception à la règle des minuscules/majuscules.

Le nom devrait donner une idée claire de l'utilité de la variable ou du paramètre en indiquant la nature de la donnée qui s'y trouve.

Pourquoi ? Le nom de la variable est le principal indicateur de son utilisation. Un nom générique oblige la lecture complète du code manipulant la variable et peut entraîner des erreurs de compréhension.

Conformes : `nomFamille, heureDebut, ligne, colonne, requeteHttp.`

Incorrects : `temp, v1, unInt, montantTPS, h_deb.`

Déclaration

Il ne devrait y avoir qu'une déclaration de variable par ligne.

Pourquoi ? Ça facilite le repérage et l'initialisation des variables, ça permet la documentation avec la Javadoc.

Lorsque possible, les variables devraient être initialisées à leur déclaration.

Pourquoi ? Ça évite de chercher l'affectation plus loin dans le code.

Déclaration des tableaux

À la déclaration d'un tableau, les `[]` devraient être du côté du type plutôt que du côté du nom de la variable.

Pourquoi ? Les `[]` sont utilisés pour distinguer le type. Toute l'information sur le type devrait être regroupée.

La taille d'un tableau devrait être donnée par une constante.

Conforme : `int[] tab = new int[NOMBRE_ETUDIANTS];`

Incorrect : `int tab[] = new int[30];`

Déclaration de variables de types génériques

L'information sur le type générique devrait toujours être incluse.

Pourquoi ? Ça permet de détecter des incohérences de types à la compilation plutôt qu'à l'exécution.

Conformes :

```
ArrayList<String> liste = new ArrayList<String>();
ArrayList<String> liste = new ArrayList<>(); //Java 7
```

Incorrects :

```
ArrayList liste = new ArrayList();
ArrayList<String> liste = new ArrayList();
```

Utilisation

Les variables nommées `i`, `j` ou `k` ne devraient être utilisées que comme compteurs de boucles lorsqu'aucun nom plus significatif n'est apparent. Un compteur ne devrait pas être réutilisé d'une boucle à l'autre.

Pourquoi ? `i`, `j`, `k` ne sont généralement pas des noms significatifs dans d'autres contextes. Les compteurs réutilisés posent un risque accru d'erreur de réinitialisation et n'offrent généralement que des avantages négligeables.

Une variable ne devrait pas être réutilisée pour y stocker une donnée d'une autre nature. (Par exemple, dans une variable `prenom` qui ne sert plus, ne pas la réutiliser pour conserver une adresse.)

Pourquoi ? Le nom de la variable n'est généralement pas significatif lors de sa réutilisation et pose un risque de mauvaise compréhension du code.

Packages

Les packages devraient être en minuscule. Le début d'un package est le nom de domaine inversé de l'organisation, la suite devrait être une séquence de noms communs ou propres qui indiquent une classification.

Il est obligatoire d'utiliser un package dès que cette notion est abordée dans vos cours. Le package de référence à utiliser est `ca.qc.bdeb.[le sigle de votre cours].[…]`

Conforme : `package ca.qc.bdeb.inf202.exemple.nomenclature;`

Incorrect : `package tp1;`

import

Chaque import devrait figurer sur sa propre ligne. On préfère les retrouver en ordre alphabétique.

Pourquoi ? Ça rend le repérage plus facile.

Il est recommandé d'utiliser l'astérisque (*) seulement lorsque plus de cinq classes sont importées d'un même package.

Pourquoi ? Il devient plus facile de savoir quelles classes importées sont réellement utilisées. Certaines organisations n'utilisent pas du tout l'astérisque.

Méthodes

Nomenclature

Le nom d'une méthode devrait commencer par une minuscule. Les mots sont coupés par des majuscules.

Dans le cas d'une méthode qui retourne une valeur, le nom devrait refléter la valeur retournée. Typiquement, une telle méthode est nommée à l'aide d'un nom commun ou d'un verbe et d'un nom commun.

Dans le cas d'une méthode qui ne retourne aucune valeur, le nom devrait refléter le travail effectué. Typiquement, une telle méthode est nommée à l'aide d'un verbe.

Conforme :

```
public void afficherJeu(char[][] plateau) {[…]
public double montantTaxes(int sousTotal) {[…]
public double obtenirTaxes(int sousTotal) {[…]
```

Incorrect :

```
public void jeu() {[…]
public void rapport() {[…]
```

Les accesseurs (getters) devraient être nommés en préfixant la variable par `get` (`getPrenom` pour la variable `prenom`). Dans le cas de variables booléennes, on peut utiliser les préfixes `is` ou `has`.

Les mutateurs (setters) devraient être nommés en préfixant la variable par `set` (`setPrenom` pour la variable `prenom`). Dans le cas de variables booléennes, on peut utiliser les préfixes `is` ou `has` (le paramètre permet de faire la distinction avec l'accesseur dans ce cas.)

Conforme :

```
public int getAge() {[…]
public void setAge(int age) {[…]
public boolean hasPermisConduire() {[…]
public void hasPermisConduire(boolean permisConduire) {[…]
public boolean isVivant() {[…]
public void isVivant(boolean vivant) {[…]
```

Découpage

Une méthode ne devrait effectuer qu'une tâche.

Pourquoi ? Ça facilite la réutilisation des méthodes, rend leur code moins complexe, facilite les tests.

Paramétrage

Toutes les données requises par une méthode devraient lui être données par paramétrage, à l'exception :

- des constantes
- des variables de classe
- des variables d'instance

Pourquoi ? Ça permet de minimiser les risques d'erreurs par effets de bord (side effects).

Les paramètres ne devraient pas être modifiés. Lorsqu'il est nécessaire de les modifier, une mention explicite devrait être incluse dans le commentaire javadoc. Pour renforcer cet aspect, il est conseillé d'utiliser `final` à la déclaration des paramètres.

Pourquoi ? Un paramètre est une donnée considérée en lecture. Lorsque la donnée reçue est modifiée, il est préférable d'en avertir l'utilisateur de la méthode par l'omission du mot réservé `final` et une note dans la javadoc.

Instructions

Règle générale : disposition et indentation

Les instructions devraient être indentées en fonction de l'imbrication des blocs de code. L'indentation normale est de quatre espaces.

Pourquoi ? L'indentation facilite le repérage visuel des blocs de code.

Il ne devrait pas y avoir plus d'une instruction par ligne.

Des espaces devraient être placés de part et d'autre des opérateurs.

Une ligne blanche devrait être utilisée pour séparer les groupes de déclarations ou d'instructions qui ont un lien logique.

Pourquoi ? Favorise la lisibilité.

Règle générale : accolades

Les blocs de code devraient toujours être entre accolades (`{}`), même lorsqu'ils ne comprennent qu'une instruction. Il ne devrait jamais y avoir d'instruction sur une ligne comportant une accolade délimitant un bloc de code.

Pourquoi ? Réduit le risque d'erreurs lorsque le traitement à effectuer devient plus long qu'une ligne.

Conforme :

```
if (termine) {
    […]
}
```

Incorrect :

```
if (termine)
    […]
```

La pratique recommandée par Sun/Oracle est de placer les accolades fermantes à la fin de ligne. Il est acceptable, bien que plus rare, de placer l'accolade ouvrante sur une nouvelle ligne. Il est important de ne pas utiliser les deux conventions simultanément.

Règle générale : conditions

Une condition ne devrait pas effectuer de traitement.

Pourquoi ? Une condition est une vérification, et ne devrait pas changer l'état du système. Les changements devraient se trouver dans un bloc de code exécuté ou non selon le résultat de l'évaluation de la condition.

Conforme :

```
String ligne = f.readLine();
if (ligne == null) {[…]
```

Incorrect :

```
if (++i == 0) {[…]
while ((ligne = f.readLine()) != null) {[…]
```

Une condition ne devrait pas comporter de comparaisons avec des valeurs booléennes explicites.

Pourquoi ? Une valeur booléenne permet déjà d'évaluer la condition, il est logiquement inutile d'ajouter une comparaison avec une valeur booléenne explicite.

Conforme :

```
if (termine) {[…]
```

Incorrect :

```
if (!pasTermine != false) {[…]
```

Toutes sorties possibles devraient être contrôlées dans les conditions. Une boucle ne devrait jamais comporter d'instruction `continue` ou `break`.

Pourquoi ? Pour éviter de lire tout le code de la boucle pour trouver les conditions de répétition ou de sortie.

Conforme :

```
while (!termine) {
    […]
    termine = choix == '0';
}
while (!termine) {
    […]
    if (choix != '9') {
        choix = ' ';
    }
}
```

Incorrect :

```
while (true) {
    […]
    if (choix == '0') {
        break;
    }
}
while (true) {
    […]
    if (choix == '9') {
        continue;
    }
    choix = ' ';
}
```

Aucune clause d'une boucle `for` ne devrait être vide. La clause d'initialisation devrait initialiser le compteur, de préférence en le déclarant plutôt qu'en le réutilisant. La clause de contrôle devrait porter uniquement sur la valeur du compteur. La clause d'incrément ne devrait modifier que le compteur. Le compteur ne devrait être modifié que par la clause d'incrément.

Pourquoi ? L'idée d'une boucle for est d'effectuer un nombre prédéterminé d'itérations. Si le seul compteur n'est pas suffisant, il est préférable d'utiliser un autre type de boucle.

if…else if…else

Une instruction if ne devrait pas être utilisée lorsque la condition est suffisante pour affecter une variable booléenne ou retourner une valeur booléenne.

Pourquoi ? La valeur booléenne donnée par l'évaluation de la condition correspond à la valeur à affecter, le reste est redondant.

Conforme :

```
negatif = nombre < 0;

return choix == '0';
```

Incorrect :

```
if (nombre < 0) {
    negatif = true;
} else {
    negatif = false;
}

if (choix == '0') {
    return true;
} else {
    return false;
}
```

Sauf lorsqu'il est essentiel d'évaluer une condition très simple dans une expression, l'usage de l'opérateur ternaire `? :` est déconseillé.

Pourquoi ? Cet opérateur diminue la lisibilité.

switch

Un switch ne devrait jamais être utilisé avec une expression booléenne.

Pourquoi ? Il est préférable d'utiliser un simple `if`.

Il est fortement recommandé d'utiliser systématiquement une clause `default`, ne serait-ce que pour signaler une erreur.

Pourquoi ? Favorise le repérage d'erreurs en cas d'oubli ou de changements.

Il est conseillé de diminuer le nombre de branches en normalisant l'expression avant d'effectuer le switch.

Pourquoi ? Diminue le nombre de comparaisons à effectuer (plus performant) et produit du code plus concis.

Conforme :

```
switch (Character.toUpperCase(choix)) //évite de traiter mes minuscules/majuscules
    case 'A' : […];
        break;
    […]
    default : […];
}
```

Incorrect :

```
switch termine {
    case true : […]
        break;
    case false : […]
        break;
}
```

for

La boucle `for` devrait être utilisée uniquement lorsque le nombre exact d'itérations est connu.

Une boucle `for` ne devrait jamais comporter une clause vide ou une condition de terminaison ne portant pas sur la valeur du compteur.

La clause d'incrément ne devrait affecter que le compteur. Le compteur ne devrait jamais être modifié à l'intérieur de la boucle.

Pourquoi ? Dans tous ces cas, il est préférable d'utiliser un autre type de boucle.

Le compteur ne devrait pas être déclaré avant la boucle à moins qu'il soit nécessaire de consulter sa valeur après la fin de la boucle. Un compteur ne devrait jamais être réutilisé.

Pourquoi ? La réutilisation des erreurs favorise l'introduction d'erreurs de réinitialisation et peut induire le lecteur en erreur.

Une boucle `for each` devrait être utilisée pour parcourir un tableau ou une collection du premier au dernier élément, en lecture.

Pourquoi ? Écriture plus simple.

Conforme :

```
for (int i = 0; i<10; i++) {[…]

for (int note : notes) {[…]
```

Incorrect :

```
int i = 0;
for (; i<10 || termine;) {
    […]
    termine = choix == '0';
    i++;
}

for (;;) {
    […]
    if (choix == '0') {
        break;
    }
    i++;
}
```

return

Dans la mesure où ça ne complique pas la structure de la méthode, il est conseillé de n'utiliser qu'une expression return.

Pourquoi ? Rend la compréhension du code plus aisée.

Conseillé :

```
int resultat = 0;
if (a < b) {
    resultat = -1;
} else if {
    resultat = 0;
} else {
    resultat = 1;
}
return resultat;
```

Déconseillé :

```
if (a < b) {
    return -1;
} else if {
```

```
    return 0;
} else {
    return 1;
}
```

Une instruction `return;` ne devrait jamais figurer dans une méthode sans type de retour.

Pourquoi ? Une méthode devrait se terminer à l'exécution de sa dernière instruction. Il est préférable d'utiliser un `if` dans ce genre de cas.

Conseillé :

```
public void afficher(int nombre) {
if (nombre >= 0) {
    System.out.print(nombre);
}
```

Incorrect :

```
public void afficher(int nombre) {
if (nombre < 0) {
    return;
} else {
    System.out.print(nombre);
}
```

Javadoc

Les déclarations de classe/interface/énumération devraient avoir un commentaire javadoc indiquant ce que la classe/interface/énumération représente. On devrait retrouver une balise `@author` identifiant l'auteur.

Les déclarations de constantes et de variables d'instance et de classe devraient être décrites par un commentaire javadoc.

Les déclarations des méthodes, à l'exception des accesseurs, mutateurs, constructeurs et des méthodes `equals`, `hashCode` et `toString` devraient avoir un commentaire javadoc expliquant ce que la méthode fait. Chacun des paramètres devrait être expliqué à l'aide d'une balise `@parameter`. La valeur de retour, si autre que `void`, devrait être expliquée par une balise `@return`. Les exceptions annoncées par une clause `throws` devraient être documentées par une balise `@throws` indiquant dans quelle circonstance l'exception est levée.

Pourquoi ? La javadoc aide le programmeur utilisant le code à comprendre ce qu'il fait et comment l'utiliser sans avoir à en interpréter les instructions.

Bibliographie

Ambler, S. (2000, Janvier 15). Consulté le Février 14, 2012, sur http://www.ambysoft.com/downloads/javaCodingStandards.pdf

Geotechnical Software Services. (2011, Janvier). *Java Programming Style Guidelines*. Consulté le Février 3, 2012, sur GeoSoft: http://geosoft.no/development/javastyle.html

Oracle Corporation. (1999, Avril 20). *Code Conventions for the Java TM Programming Language*. Consulté le Février 3, 2012, sur Oracle: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html