Conventions de code Java

Introduction

Pourquoi utiliser des conventions de code

Les conventions de code sont indispensables aux programmeurs pour plusieurs raisons :
- 80% du travail effectué sur un logiciel part en maintenance ;
- il est extrêmement rare que l'auteur initial maintienne son logiciel de bout en bout ;
- les conventions de code améliorent la lisibilité du logiciel, permettant aux ingénieurs de comprendre un nouveau code rapidement et correctement ;
- si vous distribuez votre code source, vous devez vous assurer qu'il est aussi proprement mis en forme qu'aucun autre produit.

Pour que ces conventions de code fonctionnent, il faut que tout le monde s'y conforme. Absolument tout le monde.

Auteurs

Ce document reprend les standard de code présentés dans [GJS96]. Les contributions majeures ont été faites par Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath et Scott Hommel.

Ce document est maintenu par Scott Hommel. Les commentaires peuvent être envoyés à `shommel@eng.sun.com`.

La traduction est distribuée sous la license originale de Sun Microsystems, Inc disponible en annexe B.

Table des matières

1 Noms de fichiers

1.1 Suffixes de fichiers

Type de fichier	Suffixe
Code source Java	`.java`
Bytecode Java	`.class`

1.2 Noms de fichiers courants

Quelques noms de fichiers utilisés couramment :

Nom de fichier	Utilisation
`GNUmakefile`	En cas d'utilisation de `gnumake` pour contrôler la compilation.
`README`	Le fichier résumant le contenu d'un répertoire.

2 Organisation d'un fichier

Un fichier est constitué de sections séparées par des lignes vides et des commentaires optionnels identifiant chaque section.

Des longueurs de fichier supérieures à deux mille lignes sont encombrantes et devraient être évitées.

Pour un exemple de programme Java convenablement formatté, voir l'annexe A en page 17.

2.1 Fichier code source Java

Tout fichier de code source Java contient une unique classe ou interface publique. Quand des classes ou interfaces privées lui sont associées, celles-ci peuvent être mises dans le même fichier de code source. La classe ou interface publique devrait être la première du fichier.

Les fichiers source Java ont la structure suivante :
- commentaires de début ;
- commandes `package` et `import` ;
- déclarations de classes et interfaces.

2.1.1 Commentaires de début

Tous les fichiers source devraient commencer par un commentaire de style C indiquant le nom de la classe, des informations de version, une date et une indication de copyright :

```
/*
 * Nom de la classe
 *
 * Version
 *
 * Date
 *
 * Indication de copyright
 */
```


2.1.2 Commandes `package` et `import`

Les premières lignes non commentées de la plupart des fichiers Java contiennet une commande `package`. Après cela, des commandes `import` peuvent suivre. Par exemple :

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

2.1.3 Déclarations de classes et interfaces

La table suivante décrit les différentes parties d'une déclaration de classe ou d'interface, dans l'ordre dans lequel elles devraient apparaître. Voir page 17 pour un exemple commenté.

	Portion de la déclaration	Remarques
1	Documentation `javadoc`	Voir « Documentation `javadoc` » page 8.
2	Déclaration `class` ou `interface`	
3	Commentaire sur l'implémentation de la classe ou interface si nécessaire	Ce commentaire devrait contenir toute information générale qui n'était pertinente dans une documentation `javadoc`.
4	Variables de classe (déclarées comme `static`)	D'abord les variables de classe `public`, puis `protected`, puis au niveau *package* (sans modificateur de portée), et enfin `private`.
5	Variables d'instance	D'abord `public`, puis `protected`, puis au niveau *package* et enfin `private`.
6	Constructeurs	D'abord le constructeur par défaut puis les autres.
7	Méthodes	Les méthodes devraient être groupées par fonctionnalités plutôt que par portée ou accessibilité. Par exemple une méthode privée de classe peut apparaître entre deux méthodes publiques d'instance. Le but est de rendre la lecture et la compréhension du code plus facile.

3 Indentation

L'unité d'indentation est de quatre (4) espaces. La construction exacte de l'indentation est laissée libre (tabulations ou espaces). Les tabulations doivent compter pour huit (8) espaces (et non quatre).

3.1 Longueur des lignes

Eviter les lignes de plus de quatre-vingt (80) caractères, car elles sont mal gérées par de nombreux terminaux et outils.

Remarque : Les exemples placés en documentation devraient avoir une longueur de ligne plus courte, généralement pas plus de soixante-dix (70) caractères.

3.2 Lignes brisées

Quand une expression ne tient pas sur une simple ligne, il faut la briser selon les principes généraux suivants :
- couper après une virgule ;
- couper avant un opérateur ;
- préférer les coupures de haut niveau à celles de bas niveau ;

– aligner la nouvelle ligne avec le début de l'expression au même niveau de la ligne précédente ;
– si les règles précédentes produisent un code confus ou écrasé contre la marge droite, simplement indenter de huit (8) espaces à la place.

Voici quelques exemples dans des appels de méthodes :

```
uneMethode(expressionLongue1, expressionLongue2, expressionLongue3,
        expressionLongue4, expressionLongue5);

var = uneMethode1(expressionLongue1,
                uneMethode2(expressionLongue2,
                        expressionLongue3));
```

Deux exemples dans des expressions arithmétiques suivent. Le premier est préférable puisque de plus haut niveau.

```
nomLong1 = nomLong2 * (nomLong3 + nomLong4 - nomLong5)
           + 4 * nomLong6; // PRÉFÉRER

nomLong1 = nomLong2 * (nomLong3 + nomLong4
                       - nomLong5) + 4 * nomLong6; // ÉVITER
```

Voici à présent deux exemples de déclarations de méthodes. Le premier est le cas courant. Le second mettrait les deuxièmes et troisièmes lignes à l'extrême droite avec une indentation conventionnelle, et utilise plutôt une indentation de huit (8) espaces.

```
// INDENTATION CONVENTIONNELLE
uneMethode(int unArgument, Object unAutreArgument,
           String encoreUnAutreArgument, Object etEncoreUn) {
    ...
}

// INDENTER DE 8 ESPACES POUR ÉVITER DES INDENTATIONS TROP LONGUES
private static synchronized tresLongNomDeMethode(int unArgument,
        Object unAutreArgument, String encoreUnAutreArgument,
        Object etEncoreUn) {
    ...
}
```

Les retours à la ligne dans les `if` devraient généralement utiliser la règle des huit (8) espaces, puisque l'indentation conventionnelle rend l'identification du corps difficile :

```
// NE PAS UTILISER CETTE INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    faitQuelqueChose(); // CETTE LIGNE EST FACILEMENT RATÉE
}

// UTILISER CETTE INDENTATION À LA PLACE
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    faitQuelqueChose();
}

// OU ENCORE CELLE-CI
if ((condition1 && condition2) || (condition3 && condition4)
```



```
        ||!(condition5 && condition6)) {
    faitQuelqueChose();
}
```

Voici enfin trois formattages acceptables d'expressions ternaires :

```
alpha = (uneLongueExpressionBooleene)? beta: gamma;

alpha = (uneLongueExpressionBooleene)? beta
                                     : gamma;

alpha = (uneLongueExpressionBooleene)
        ? beta
        : gamma;
```

4 Commentaires

Les programmes Java ont deux types de commentaires : des commentaires d'implémentation et des commentaires de documentation. Les commentaires d'implémentation sont les mêmes qu'en C++, délimités par `/*...*/` et `//`. Les commentaires de documentation (aussi appelés commentaires `javadoc`) sont spécifiques à Java et sont délimités par `/**...*/`. Les commentaires `javadoc` peuvent être extraits vers des fichiers au format HTML grâce au programme `javadoc`.

Les commentaires d'implémentation servent à commenter le code ou une implémentation particulière. Les commentaires `javadoc` servent à décrire les spécifications du code d'un point de vue extérieur pour des développeurs qui n'auront pas forcément le code source à leur disposition.

Les commentaires devraient être utilisés pour abstraire le code et fournir des informations absentes du code lui-même. Les commentaires ne devraient contenir que des informations utiles à la lecture et à la compréhension du programme. Par exemple, comment construire le package ou dans quel répertoire il se situe n'a pas sa place dans le code.

Des explications sur des choix de développement sont appropriées, mais il faut éviter de dupliquer l'information déjà clairement présente dans le code. Les commentaires redondants sont en effet très facilement périmés. En général, les commentaires qui risquent d'être périmés suite aux évolutions du code sont à éviter.

Remarque : Quand vous sentez le besoin d'ajouter un commentaire, voyez aussi si le code ne pourrait pas être réécrit plus simplement et plus clairement.

Les commentaires ne devraient pas être entourés dans de grandes boîtes d'astérisques ou d'autres caractères. Ils ne devraient jamais inclure de caractères spéciaux comme *backspace*.

4.1 Commentaires d'implémentation

Les programmes peuvent utiliser quatre styles de commentaires d'implémentation : en bloc, en ligne, insérés ou en fin de ligne.

4.1.1 Commentaires en blocs

Les commentaires en blocs servent à décrire les fichiers, méthodes, structures de données et algorithmes. Ils peuvent être utilisés au début de chaque fichier et avant chaque méthode. Ils peuvent aussi être utilisés ailleurs, comme par exemple à l'intérieur de méthodes. Un commentaire en bloc placé dans une méthode devrait être indenté au même niveau que le code qu'il décrit.

Un commentaire en bloc devrait être précédé d'une ligne blanche pour le séparer du reste du code.

```
/*
 * Ceci est un commentaire en bloc.
 */
```

Les commentaires en bloc peuvent commencer par `/*-`, qui est reconnu par `indent`(1) comme le début d'un commentaire en bloc qui ne doit pas être reformaté. Par exemple :

```
/*-
 * Ceci est un commentaire en bloc avec une mise en page très spéciale
 * qu'indent(1) doit ignorer.
 *
 *    un
 *        deux
 *            trois
 */
```

Remarque : Si vous n'utilisez pas `indent`(1), vous n'avez pas besoin d'utiliser `/*-` dans votre code ni de faire d'autres concessions à la possibilité que quelqu'un d'autre puisse exécuter `indent`(1) sur votre code.

Voir aussi la « Documentation `javadoc` » page 8.

4.1.2 Commentaires sur une seule ligne

Des commentaires courts peuvent être placés sur une seule ligne indentée au niveau du code qui la suit. Si un commentaire ne peut pas être écrit sur une seule ligne, alors il devrait utiliser une mise en forme de bloc (voir section 4.1.1). Un commentaire en ligne devrait être précédé d'une ligne blanche. Voici un exemple dans du code Java (voir aussi la « Documentation `javadoc` » page 8) :

```
if (condition) {

  /* Traite la condition. */
  ...
}
```

4.1.3 Commentaires insérés

Des commentaires très courts peuvent être placés sur la même ligne que le code qu'ils décrivent, mais devraient être décalés suffisamment loin sur la droite pour ne pas pouvoir être confondus avec du code. Si plus d'un commentaire court apparaît dans une portion de code, alors ils devraient tous être indentés au même niveau.

Voici un exemple de code inséré en Java :

```
if (a == 2) {
    return TRUE;            /* case particulier */
} else {
    return estPremier(a);   /* ne fonctionne que si a est pair */
}
```

4.1.4 Commentaires de fin de ligne

Le délimiteur de commentaires `//` peut commenter une ligne complète ou juste une portion de ligne. Il ne devrait généralement pas être utilisé sur plusieurs lignes consécutives, mais il peut l'être pour commenter des portions de code. Des exemples pour ces trois situations suivent :

```
if (foo > 1) {

    // Faire un double saut périlleux.
    ...
}
else {
    return false;          // Expliquer pourquoi ici.
}
```



```
//if (bar > 1) {
//
//    // Faire un triple saut périlleux.
//    ...
//}
//else {
//    return false;
//}
```

4.2 Documentation `javadoc`

Remarque : Voir l'annexe A en page 17 pour des exemples des mises en formes présentées ici.

Pour plus de détails, lire [Sun] qui explique comment utiliser les balises `javadoc` (`@return`, `@param`, `@see`, ...). Le site officiel dédié à `javadoc` se trouve à l'adresse :

`http://java.sun.com/javadoc/`.

La documentation `javadoc` décrit les classes, interfaces, constructeurs, méthodes et champs Java. Chaque commentaire de documentation est encadré par les délimiteurs `/**...*/`, avec un commentaire par classe, interface ou membre. Ce commentaire doit apparaître juste avant chaque déclaration :

```
/**
 * La classe Exemple fournit...
 */
public class Exemple {
    ...
```

Remarquez que les classes et interfaces au plus haut niveau ne sont pas indentées, tandis que leurs membres le sont. La première ligne (`/**`) du commentaire `javadoc` des classes et interfaces n'est pas indentée ; les lignes suivantes du commentaire ont chacune un (1) espace d'indentation de manière à aligner verticalement les astérisques. Les membres, y compris les constructeurs, ont quatre (4) espaces d'indentation pour la première ligne et cinq (5) pour les lignes suivantes.

Si vous souhaitez donner une information à propos d'une classe, d'une interface, d'une variable ou d'une méthode qui ne trouverait pas sa place dans une documentation, utilisez un commentaire en bloc (section 4.1.1) ou un commentaire sur une seule ligne (section 4.1.2) juste *après* la déclaration. Par exemple, les détails de l'implémentation d'une classe devraient être écrits dans un commentaire en bloc placé *à la suite* de la déclaration de la classe, et non pas dans son commentaire `javadoc`.

Les commentaires `javadoc` ne doivent pas être placés dans le bloc de définition d'une méthode ou d'un constructeur puisque Java associe les commentaires `javadoc` à la première déclaration *après* le commentaire.

5 Déclarations

5.1 Nombre de déclarations par ligne

On recommande de ne mettre qu'une seule déclaration par ligne puisque cela encourage les commentaires. Par exemple :

```
int niveau;  // niveau d'indentation
int taille;  // taille de la table
```

est préféré à :

```
int niveau, taille;
```

Ne mettez pas différents types sur la même ligne. Par exemple :

```
int foo, tableauFoo[];  // MAUVAIS !
```

Remarque : Les exemples ci-dessus n'utilisent qu'un seul espace entre le type et l'identificateur. Une alternative acceptable est d'utiliser des tabulations, *e.g.* :

```
int     niveau;              // niveau d'indentation
int     taille;              // taille de la table
Object  entreeSelectionnee;  // l'entrée actuellement selectionnée dans la table
```

5.2 Initialisation

Essayez d'initialiser les variables locales au moment où elles sont déclarées. La seule raison de ne pas le faire est que sa valeur initiale dépend du résultat d'un calcul préliminaire.

5.3 Organisation

Les déclarations se font uniquement au début des blocs. (Un bloc est n'importe quelle portion de code entre accolades « { » et « } ».) N'attendez pas leur première utilisation pour déclarer vos variables : cela porte à confusion.

```
void maMethode() {
    int entier1 = 0;        // début du bloc de la méthode

    if (condition) {
        int entier2 = 0;    // début du bloc  « if »
        ...
    }
}
```

L'exception à cette règle est l'index dans les boucles `for`, qui en Java peut être déclaré dans l'appel à `for` :

```
for (int i = 0; i < maxBoucle; i++) {
    ....
}
```

Évitez les déclarations locales qui cachent des déclarations à des niveaux supérieurs. Par exemple, ne pas déclarer le même nom de variable dans un bloc interne :

```
int valeur;
...
maMethode() {
    if (condition) {
        double valeur;    // À ÉVITER !
        ...
    }
    ...
}
```

5.4 Déclarations de classes et interfaces

Les règles de mise en forme suivante devraient être suivies pour les classes et interfaces Java :

- pas d'espace entre le nom d'une méthode et la parenthèse ouvrante « (» commençant sa liste de paramètres ;
- l'accolade ouvrante « { » est placée à la fin de la ligne de déclaration ;
- l'accolade fermante « } » entame une nouvelle ligne indentée pour correspondre à la ligne de son ouverture, sauf quand le corps est vide, auquel cas la « } » devrait être placée immédiatement après la « { ».


```
class Exemple extends Object {
    int variableEntiere1;
    int variableEntiere2;

    Exemple (int i, int j) {
        variableEntiere1 = i;
        variableEntiere2 = j;
    }

    int methodeVide() {}

    ...
}
```

6 Instructions

6.1 Instructions simples

Chaque ligne ne devrait contenir qu'une unique instruction. Par exemple :

```
argv++;         // Correct
argc--;         // Correct
argv++; argc--; // À Éviter !
```

6.2 Instructions composées

On compose les instructions en les plaçant en liste entre accolades « { » et « } ». Les sections suivantes fournissent des exemples pour chaque type d'utilisation.

- Les instructions à l'intérieur du bloc doivent être indentées d'un niveau.
- L'accolade ouvrante devrait être placée à la fin de la ligne qui commence l'instruction composée ; l'accolade fermante quant à elle devrait l'être sur une nouvelle ligne avec une indentation identique à celle du début.
- Les accolades devraient toujours être écrites, même pour les instructions simples, quand celles-ci font partie d'une structure de contrôle comme une instruction `if-else` ou `for`. Cela évite d'oublier des accolades quand on ajoute de nouvelles instructions.

6.3 `return`

Une instruction `return` ne devrait pas utiliser de parenthèses, sauf si elles rendent la valeur de retour plus lisible. Par exemple :

```
return;

return monDisque.taille();

return (estVide()? TAILLE_PAR_DEFAUT: taille());
```

6.4 `if`, `if-else`, `if-else if`

Les instructions de type `if-else` devraient avoir la mise en forme suivante :

```
if (condition) {
    instructions;
}

if (condition) {
    instructions;
```

```
} else {
    instructions;
}

if (condition) {
    instructions;
} else if (condition) {
    instructions;
} else {
    instructions;
}
```

Remarque : Les instructions `if` utilisent toujours les accolades. La mise en forme suivante facilite les erreurs :

```
if (condition)  // À ÉVITER !
    instruction;
```

6.5 `for`

Une instruction `for` devrait avoir la forme suivante :

```
for (initialisation; condition; mise à jour) {
    instructions;
}
```

Une instruction `for` vide, dans laquelle tout le travail est effectué par l'initialisation, la condition et la mise à jour, est mise en forme comme ceci :

```
for (initialisation; condition; mise à jour);
```

Si vous utilisez l'opérateur virgule dans l'initialisation ou la mise à jour, veillez à ne pas utiliser plus de trois variables. Ajoutez au besoin des instructions avant la boucle pour l'initialisation ou en fin de boucle pour la mise à jour.

6.6 `while`

Une instruction `while` devrait utiliser la mise en forme suivante :

```
while (condition) {
    instructions;
}
```

Une instruction `while` vide aura la forme :

```
while (condition);
```

6.7 `do-while`

Une instruction `do-while` devrait être mise en forme ainsi :

```
do {
    instructions;
} while (condition);
```


6.8 `switch`

Une instruction `switch` a la forme suivante :

```
switch (condition) {
case ABC:
    instructions;
    /* passe au suivant */

case DEF:
    instructions;
    break;

case XYZ:
    instructions;
    break;

default:
    instructions;
    break;
}
```

Chaque fois qu'un `case` passe au suivant (c'est-à-dire ne comporte pas d'instruction `break`), ajoutez un commentaire là où l'instruction `break` aurait normalement été placée. C'est ce qui est montré dans l'exemple précédent avec le commentaire `/* passe au suivant */`.

Toute instruction `switch` devrait comporter un cas `default`. Le `break` dans le cas par défaut est redondant, mais permet d'éviter une erreur si par la suite un autre `case` est ajouté.

6.9 `try-catch`

Une instruction `try-catch` utilise la mise en forme suivante :

```
try {
    instructions;
} catch (ClasseDException e) {
    instructions;
}
```

Une instruction `try-catch` peut aussi être suivie par `finally`, qui exécute ses instructions sans se préoccuper de savoir si le bloc `try` a été exécuté en entier ou non.

```
try {
    instructions;
} catch (ClasseDException e) {
    instructions;
} finally {
    instructions;
}
```

7 Blancs

7.1 Lignes blanches

Les lignes blanches améliorent la lisibilité en séparant des sections de code.

Deux lignes blanches devraient toujours être utilisées dans les circonstances suivantes :

- entre des sections d'un fichier de code source ;
- entre des définitions de classes et d'interfaces.

Une ligne blanche devrait toujours être utilisée dans les circonstances suivantes :
- entre des méthodes ;
- entre les déclarations de variables locales d'une méthode et sa première instruction ;
- avant un commentaire en bloc (voir section 4.1.1) ou en ligne (voir section 4.1.2) ;
- entre les sections logiques à l'intérieur d'une méthode pour en améliorer la lisibilité.

7.2 Espaces blancs

Des espaces blancs devraient toujours être utilisés dans les circonstances suivantes :
- un mot clef suivi d'une parenthèse devrait en être séparé par un espace. Par exemple :
```
while(true) {
    ...
}
```
Remarquez qu'un espace blanc ne devrait pas être utilisé lors d'un appel de méthode entre le nom de la méthode et la parenthèse ouvrante, ceci afin de distinguer les mots clefs des appels de méthodes ;
- après une virgule dans une liste d'arguments ;
- tous les opérateurs binaires à l'exception de « . » devraient être séparés de leurs opérandes par des espaces. En revanche, il ne doit pas y avoir d'espaces blancs les séparant pour les opérateurs unaires comme moins, incrément « ++ » et décrément « - - ». Exemple :
```
a += c + d;
a = (a + b) / (-c * d);

while (d-- = s--) {
    n++;
}
imprimeTaille("La taille est " + foo + "\n");
```
- les clauses d'une instruction `for` devraient être séparées par des espaces blancs :
```
for (expr1; expr2; expr3)
```
- les `cast` devraient être suivis par un espace blanc :
```
maMethode((byte) unNombre, (Object) x);
maMethode((int) (cp + 5), ((int) (i + 3)) + 1);
```

8 Conventions de nommage

Les conventions de nommages rendent les programmes plus aisément compréhensibles car plus lisibles. Elles donnent aussi des informations sur la fonction de l'identificateur ; par exemple, si il s'agit d'une constante, d'un *package* ou d'une classe, cer qui facilite grandement la compréhension du code.

Types	**Exemples**	**Règles de nommage**

packages	`com.sun.eng`
	`com.apple.quicktime.v2`
	`fr.esinsa.unice.quizz`
		Le préfixe d'un nom de *package* unique est toujours écrit en caractères ASCII minuscules et doit être l'un des noms des domaines primaires, actuellement com, edu, gov, mil, net et org, ou l'un des codes anglais de deux lettres identifiant les pays selon le standard ISO3166.
		Les composants suivants du nom varient selon les conventions internes à l'organisme, utilisant par exemple des noms de départements, projets, machines ou *logins*.

Classes et interfaces	`interface Liste`
	`class ListeSimple`
	`class ListeTrieeAbstraite`
		Les noms de classes et interfaces doivent être des noms avec les premières lettres de chaque mot interne en majuscules. Essayez de garder des noms de classes simples et descriptifs. Utilisez des mots entiers plutôt que des acronymes ou abbréviations, à moins que celles-ci ne soient habituellement utilisées à la place de leurs définitions, comme URI ou XHTML.

Méthodes	`cours();` `coursPlusVite();`	Les méthodes devraient commencer par des verbes conjugués, avec les premières lettres des mots à partir du second en majuscules.

Variables	`int i;`
	`char c;`
	`float maLargeur;`
		À l'exception des constantes, toutes les variables, de classe comme d'instance, ont la première lettre de chaque mot interne en majuscules excepté le premier mot. Les noms de variables ne devraient pas commencer par un blanc souligné « _ » ni un symbole dollar « $ ».
		Les noms de variables devraient être courts mais avoir un sens. Le choix du nom doit être mnémonique et indiquer au lecteur à quoi la variable sert. Les variables d'une seule lettre devraient être évitées sauf pour des variables temporaires jetables. Les noms courants pour de telles variables sont `i`, `j`, `k`, `m` et `n` pour les entiers, `c` pour les caractères, ...

Constantes	`static final int INDICE_MIN = 4;` `static final int INDICE_MAX = 9;`	Les noms des constantes devraient être intégralement en majuscules avec les mots internes séparés par des blancs soulignés « _ ».

9 Techniques de programmation

9.1 Accès aux variables d'instance et de classe

Ne rendez pas une variable d'instance ou de classe publique sans une bonne raison. Souvent, les variables d'instance n'ont pas besoin d'être directement affectées ou lues, mais plutôt accédées *via* des appels de méthodes.

Un exemple de variable d'instance publique appropriée est le cas où la classe n'est qu'une structure de donnée, sans comportement associé. En d'autres termes, là où vous auriez utilisé `struct` (à supposer que Java accepte `struct`), il est approprié de rendre les variables d'instance de la classe publiques.

9.2 Références aux variables et méthodes de classe

Évitez d'utiliser un objet pour accéder à une variable ou une méthode de classe (`static`). Utilisez plutôt le nom de la classe :

```
methodeDeClasse();           // correct
UneClasse.methodeDeClasse(); // correct
unObjet.methodeDeClasse();   // À ÉVITER !
```

9.3 Constantes

Les constantes numériques ne devraient pas apparaître directement dans le code, à l'exception de -1, 0 et 1 qui peuvent apparaître pour les compteurs dans les boucles.

9.4 Affectations de variables

Évitez d'effectuer plusieurs affectations de variables à la même valeur en une seule instruction : c'est difficile à lire. Par exemple :

```
fooBar.premierCaractere = barFoo.dernierCaractere = 'c'; // À ÉVITER !
```

N'utilisez pas l'opérateur d'affectation à des endroits où il pourrait facileemnt être confondu avec l'opérateur d'égalité :

```
if (c++ = d++) { // À ÉVITER ! (Java l'empêche)
    ...
}
```

devrait être écrit comme :

```
if ((c++ = d++) != 0) {
    ...
}
```

N'utilisez pas d'affectations imbriquées dans l'espoir d'améliorer les performances. C'est le travail du compilateur.

```
d = (a = b + c) + r; // À ÉVITER !
```

devrait être écrit comme :

```
a = b + c;
d = a + r;
```


9.5 Divers

9.5.1 Parenthèses

C'est généralement une bonne idée d'utiliser largement les parenthèses dans des expressions mêlant plusieurs opérateurs pour éviter les problèmes de précédence. Même si la précédence vous paraît claire, elle peut ne pas l'être pour d'autres.

```
if (a == b && c == d)     // À ÉVITER !
if ((a == b) && (c == d)) // correct
```

9.5.2 Valeurs de retour

Essayez de faire coïncider la structure de votre programme et votre objectif. Par exemple :

```
if (expressionBooleenne) {
    return true;
} else {
    return false;
}
```

devrait plutôt être écrit comme :

```
return expressionBooleenne;
```

De même,

```
if (condition) {
    return x;
}
return y;
```

devrait plutôt être écrit comme :

```
return (condition? x: y);
```

9.5.3 Expressions avant le « ? » conditionnel

Si une expression contentant un opérateur binaire apparaît avant le « ? » de l'opérateur ternaire `? :` , alors elle devrait être parenthésée :

```
(x >= 0)? x: -x;
```

9.5.4 Commentaires spéciaux

Utilisez un `XXX` dans un commentaire pour indiquer quelque chose des boggé mais qui fonctionne. Utilisez `FIXME` pour indiquer quelque chose de boggé et qui ne fonctionne pas.

A Exemple de code source Java

```
/*
 * @(#)Blah.java        1.82 18/03/03
 */

package fr.unice.esinsa.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Description de la classe.
 *
 * @version     1.82 18 mars 2003
 * @author      Nom Prénom
 * @invar       Invariant de classe
 */
public class Blah extends UneClasse {
    /* Un commentaire spécifique à l'implémentation. */

    /** Documentation de la variable de classe variableDeClasse1. */
    public static int variableDeClasse1;

    /**
     * Documentation pour variableDeClasse2 qui se trouve être plus
     * long qu'une ligne de 70 caractères.
     */
    private static Object variableDeClasse2;

    /** Commentaire pour le champ variableDInstance1. */
    public Object variableDInstance1;

    /** Commentaire pour le champ variableDInstance2. */
    protected int variableDInstance2;

    /** Commentaire pour le champ variableDInstance3. */
    private Object[] variableDInstance3;

    /**
     * Documentation du constructeur Blah() par défaut.
     * @cons   Conséquent de la construction.
     */
    public Blah() {
        // ... implémentation
    }

    /**
     * Documentation de la méthode faitQuelqueChose.
     * @ante   Antécendant global de la méthode
     * @throws Exception et pourquoi
     * @return Détaille la valeur de retour
     * @cons   Conséquent global de la méthode
     * @compl  Complexité algorithmique du traitement
     */
    public Object faitQuelqueChose() throws Exception {
        // ... implémentation
    }

    /**
     * Documentation de la méthode faitQuelqueChoseDAutre.
     * @ante   Antécendant global de la méthode
     * @param  unParametre Description du paramètre
     * @param  unEntier    Description du paramètre
     * @cons   Conséquent global de la méthode
     * @see    variableDeClasse1
     */
    public void faitQuelqueChoseDAutre(Object unParametre, int unEntier) {
        // ... implémentation
    }
}
```


B License de diffusion

Références

[GJS96] Gosling James, Joy Bill et Steele Guy. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, first edition, 1996, `http://java.sun.com/docs/books/jls/`.

[GJSB00] Gosling James, Joy Bill, Steele Guy et Bracha Gilad. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, second edition, 2000, `http://java.sun.com/docs/books/jls/`.

[Hom99] Hommel Scott. *Java Code Conventions*. Sun Microsystems, Inc, April 1999, `http://java.sun.com/docs/codeconv/`.

[Sun] Sun Microsystems, Inc. *How to Write Doc Comments for the Javadoc™ Tool*, `http://java.sun.com/j2se/javadoc/writingdoccomments/`.