91. Des normes de développement

Niveau : Elémentaire

Le but de ce chapitre est de proposer un ensemble de conventions et de règles pour faciliter la compréhension et donc la maintenance du code.

Ces règles ne sont pas à suivre explicitement à la lettre : elles sont uniquement présentées pour inciter les développeurs à définir et à utiliser des règles dans la réalisation du code surtout dans le cadre d'un travail en équipe. Les règles proposées sont celles couramment utilisées. Il n'existe cependant pas de règle absolue et chacun pourra utiliser tout ou partie des règles proposées.

La définition de conventions et de règles est importante pour plusieurs raisons :

- La majorité du temps passé à coder est consacrée à la maintenance évolutive et corrective d'une application
- Ce n'est pas toujours, voire rarement, l'auteur du code qui effectue ces maintenances
- ces règles facilitent la lisibilité et donc la compréhension du code

Le contenu de ce document est largement inspiré par les conventions de codage proposées par Sun à l'URL suivante : http://java.sun.com/docs/codeconv/index.html

Ce chapitre contient plusieurs sections :

- Les fichiers
- La documentation du code
- Les déclarations
- Les séparateurs
- Les traitements
- Les règles de programmation

91.1. Les fichiers

Java utilise des fichiers pour stocker les sources et le bytecode des classes.

91.1.1. Les packages

Les packages permettent de grouper les classes sous une forme hiérarchisée. Le choix des critères de regroupement est laissé aux développeurs.

Il est préférable de regrouper les classes par packages selon des critères fonctionnels.

Les fichiers inclus dans un package doivent être insérer dans une arborescence de répertoires équivalente.

91.1.2. Les noms de fichiers

Chaque fichier source ne doit contenir qu'une seule classe ou interface publique. Le nom du fichier doit être identique au nom de cette classe ou interface publique en respectant la casse.

Il faut éviter d'utiliser dans ce nom des caractères accentués qui ne sont pas toujours utilisables par tous les systèmes d'exploitation.

Les fichiers sources ont pour extension .java car le compilateur javac fourni avec le J.D.K. utilise cette extension

Exemple :

```
1. javac MaClasse.java
```

Les fichiers binaires contenant le bytecode ont pour extension .class car le compilateur génère un fichier avec cette extension à partir du fichier source .java correspondant. De plus, elle est obligatoire pour l'interpréteur Java qui l'ajoute automatiquement au nom du fichier fourni en paramètre.

Exemple :

```
1. java MaClasse
```

91.1.3. Le contenu des fichiers sources

Un fichier ne devrait pas contenir plus de 2 000 lignes de code.

Des interfaces ou classes privées ayant une relation avec la classe publique peuvent être rassemblées dans un même fichier. Dans ce cas, la classe publique doit être la première dans le fichier.

Chaque fichier source devrait contenir dans l'ordre :

1. un commentaire concernant le fichier
2. les clauses concernant la gestion des packages (la déclaration et les importations)
3. les déclarations de classes ou de l'interface

91.1.4. Les commentaires de début de fichier

Chaque fichier source devrait commencer par un commentaire multi-lignes contenant au minimum des informations sur le nom de la classe, la version, la date, éventuellement le copyright et tous les autres commentaires utiles :

Exemple :

```
01. /*
02.  * Nom de classe : MaClasse
03.  *
04.  * Description   : description de la classe et de son rôle
05.  *
06.  * Version       : 1.0
07.  *
08.  * Date          : 23/02/2001
09.  *
10.  * Copyright     : moi
11.  */
```

91.1.5. Les clauses concernant les packages.

La première ligne de code du fichier devrait être une clause package indiquant à quel paquetage appartient la classe. Le fichier source doit obligatoirement être inclus dans une arborescence correspondante au nom du package.

Il faut indiquer ensuite l'ensemble des paquetages à importer : ceux dont les classes vont être utilisées dans le code.

Exemple :

```
1. package monpackage;
```

```
2.
3. import java.util.*;
4. import java.text.*;
```

91.1.6. La déclaration des classes et des interfaces

Les différents éléments qui composent la définition de la classe ou de l'interface devraient être indiqués dans l'ordre suivant :

1. les commentaires au format javadoc de la classe ou de l'interface
2. la déclaration de la classe ou de l'interface
3. les variables de classes (déclarées avec le mot clé static) triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
4. les variables d'instances triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
5. le ou les constructeurs
6. les méthodes : elles seront regroupées par fonctionnalités plutôt que selon leur accessibilité

91.2. La documentation du code

Il existe deux types de commentaires en Java :

- les commentaires de documentation : ils permettent en respectant quelques règles d'utiliser l'outil javadoc fourni avec le J.D.K. qui formate une documentation des classes, indépendante de l'implémentation du code,
- les commentaires de traitements : ils fournissent un complément d'information dans le code lui-même.

Les commentaires ne doivent pas être entourés par de grands cadres dessinés avec des étoiles ou d'autres caractères.

Les commentaires ne devraient pas contenir de caractères spéciaux tels que le saut de page.

91.2.1. Les commentaires de documentation

Les commentaires de documentation utilisent une syntaxe particulière utilisée par l'outil javadoc de Sun pour produire une documentation standardisée des classes et interfaces au format HTML. La documentation de l'API du J.D.K. est le résultat de l'utilisation de cet outil de documentation

91.2.1.1. L'utilisation des commentaires de documentation

Cette documentation concerne les classes, les interfaces, les constructeurs, les méthodes et les champs.

La documentation est définie entre les caractères /** et */ selon le format suivant :

Exemple :

```
1. /**
2.  *  Description de la methode
3.  */
4. public void maMethode() {
```

La première ligne de commentaires ne doit contenir que /**

Les lignes de commentaires suivantes doivent obligatoirement commencer par un espace et une étoile. Toutes les premières étoiles doivent être alignées.

La dernière ligne de commentaires ne doit contenir que */ précédé d'un espace.

Un tel commentaire doit être défini pour chaque entité : une classe, une interface et chaque membre (variables et méthodes).

Javadoc définit un certain nombre de tags qu'il est possible d'utiliser pour apporter des précisions sur plusieurs informations.

Ces tags permettent de définir des caractéristiques normalisées. Il est possible d'inclure dans les commentaires des tags HTML de mise en forme (PRE, TT, EM ...) mais il n'est pas recommandé d'utiliser des tags HTML de structure tel que Hn, HR, TABLE ... qui sont utilisés par javadoc pour formater la documentation

Il faut obligatoirement faire précéder l'entité documentée par son commentaire car l'outil associe la documentation à la déclaration de l'entité qui la suit.

91.2.1.2. Les commentaires pour une classe ou une interface

Pour les classes ou interfaces, javadoc définit les tags suivants : @see, @version, @author, @copyright, @security, @date, @revision, @note

Les tags @copyright, @security, @date, @revision et @note ne sont pas traités par javadoc.

Exemple :

```
01. /**
02.  * NomClasse - description de la classe
03.  * explication supplémentaire si nécessaire
04.  *
05.  * @version1.0
06.  *
07.  * @see UneAutreClasse
08.  * @author Jean Michel D.
09.  * @copyright (C) moi 2001
10.  * @date 01/09/2000
11.  * @notes notes particulières sur la classe
12.  *
13.  * @revision référence
14.  *          date 15/11/2000
15.  *          author Michel M.
16.  *          raison  description
17.  *          description supplémentaire
18.  */
```

91.2.1.3. Les commentaires pour une méthode

Pour les méthodes, javadoc définit les tags suivants : @see, @param, @return, @exception, @author, @note

Le tag @note n'est pas traité par javadoc.

Exemple :

```
01. /**
02.  * nomMethode - description de la méthode
03.  *              explication supplémentaire si nécessaire
04.  *
05.  *     exemple d'appel de la methode
06.  * @return      description de la valeur de retour
07.  * @param       arg1 description du 1er argument
08.  *    :           :         :
09.  * @param       argN description du Neme argument
10.  * @exception   Exception1  description de la première exception
11.  *    :           :         :
12.  * @exception ExceptionN  description de la Neme exception
13.  *
14.  * @see UneAutreClasse#UneAutreMethode
15.  * @author   Jean Dupond
16.  * @date     12/02/2001
17.  * @note     notes particulières.
18.  */
```

Remarques :

- @return ne doit pas être utilisé avec les constructeurs et les méthodes sans valeur de retour (void)
- @param ne doit pas être utilisé s'il n'y a pas de paramètres
- @exception ne doit pas être utilisé s'il n'y pas d'exception propagée par la méthode
- @author doit être omis s'il est identique à celui du tag @author de la classe
- @note ne doit pas être utilisé s'il n'y a pas de note

91.2.2. Les commentaires de traitements

Ces commentaires doivent ajouter du sens et des précisions au code : ils ne doivent pas reprendre ce que le code exprime mais expliquer clairement son rôle.

Tous les commentaires utiles à une meilleure compréhension du code et non inclus dans les commentaires de documentation seront insérés avec des commentaires de traitements. Il existe plusieurs styles de commentaires :

- les commentaires sur une ligne
- les commentaires sur une portion de ligne
- les commentaires multi-lignes

Il est conseillé de mettre un espace après le délimiteur de début de commentaires et avant le délimiteur de fin de commentaires lorsqu'il y en a un, afin d'améliorer sa lisibilité.

91.2.2.1. Les commentaires sur une ligne

Ces commentaires sont définis entre les caractères /* et */ sur une même ligne

Exemple :

```
1. if (i < 10) {
2.    /* commentaires utiles au code */
3.    ...
4. }
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

91.2.2.2. Les commentaires sur une portion de ligne

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Exemple :

```
1. i++;                    /* commentaires utiles au code */
```

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
1. i++;                    /* commentaires utiles au code */
2. j++;                    /* second commentaires utiles au code */
```

91.2.2.3. Les commentaires multi-lignes

Exemple :

```
1. /*
2.  * Commentaires utiles au code
3.  */
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

91.2.2.4. Les commentaires de fin de ligne

Ce type de commentaire peut délimiter un commentaire sur une ligne complète ou une fin de ligne.

Exemple :

```
1. i++;         // commentaires utiles au code
```

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
1. i++;                    // commentaires utiles au code
2. j++;                    // second commentaires utiles au code
```

L'usage de cette forme de commentaires est fortement recommandé car il est possible d'inclure celui-ci dans un autre de la forme /* */ et ainsi mettre en commentaire un morceau de code incluant déjà des commentaires.

91.3. Les déclarations

91.3.1. La déclaration des variables

Il n'est pas recommandé d'utiliser des caractères accentués dans les identifiants de variables, cela peut éventuellement poser des problèmes dans le cas où le code est édité sur des systèmes d'exploitation qui ne les gèrent pas correctement.

Il ne doit y avoir qu'une seule déclaration d'entité par ligne.

Exemple :

```
1. String nom;
2. String prenom;
```

Cet exemple est préférable à

Exemple :

```
1. String nom, prenom;          //ce type de déclaration n'est pas recommandée
```

Il faut éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

Exemple :

```
1. int age, notes[];            // ce type de déclaration est à éviter
```

Il est préférable d'aligner le type, l'identifiant de l'objet et les commentaires si plusieurs déclarations se suivent pour retrouver plus facilement les divers éléments.

Exemple :

```
1. String          nom          //nom de l'eleve
2. String          prenom       //prenom de l'eleve
3. int             notes[]      //notes de l'eleve
```

Il est fortement recommandé d'initialiser les variables au moment de leur déclaration.

Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc. (un bloc est un morceau de code entouré par des accolades).

La seule exception concerne la déclaration de la variable utilisée comme index dans une boucle.

Exemple :

```
1. for (int i = 0 ; i < 9 ; i++) { ... }
```

Il faut proscrire la déclaration d'une variable qui masque une variable définie dans un bloc parent afin de ne pas complexifier inutilement le code.

Exemple :

```
1. int taille;
2. ...
3. void maMethode() {
4.     int taille;
5. }
```

91.3.2. La déclaration des classes et des méthodes

Il ne doit pas y avoir d'espaces entre le nom d'une méthode et sa parenthèse ouvrante.

L'accolade ouvrante qui définit le début du bloc de code doit être à la fin de la ligne de déclaration.

L'accolade fermante doit être sur une ligne séparée dont le niveau d'indentation correspond à celui de la déclaration.

Une exception tolérée concerne un bloc de code vide : dans ce cas les deux accolades peuvent être sur la même ligne.

La déclaration d'une méthode est précédée d'une ligne blanche.

Exemple :

```
01. class MaClasse extends MaClasseMere {
02.    String nom;
03.    String prenom;
04.
05.    MaClasse(String nom, String prenom) {
06.       this.nom = nom;
07.       this.prenom = prenom;
08.    }
09.
10.    void neRienFaire() {}
11. }
```

Il faut éviter d'écrire des méthodes longues et compliquées : le traitement réalisé par une méthode doit être simple et fonctionnel. Cela permet d'écrire des méthodes réutilisables dans la classe et facilite la maintenance. Cela permet aussi d'éviter la redondance de code.

Java propose deux syntaxes pour déclarer une méthode qui retourne un tableau : la première syntaxe est préférable.

Exemple :

```
1. public int[] notes() {     // utiliser cette forme
2. public int notes()[] {
```

Il est fortement recommandé de toujours initialiser les variables locales d'une méthode lors de leur déclaration car contrairement aux variables d'instances, elles ne sont pas implicitement initialisées avec une valeur par défaut selon leur type.

91.3.3. La déclaration des constructeurs

Elle suit les mêmes règles que celles utilisées pour les méthodes.

Il est préférable de définir explicitement le constructeur par défaut (le constructeur sans paramètre). Soit le constructeur par défaut est fourni par le compilateur et dans ce cas il serait préférable de le définir soit il existe d'autres constructeurs et dans ce cas le compilateur ne définit pas de constructeur par défaut.

Il est préférable de toujours initialiser les variables d'instance dans un constructeur soit avec les valeurs fournies en paramètres du constructeur soit avec des valeurs par défaut.

Exemple :

```
01. class Personne {
02.    String nom;
```

```
03.    String prenom;
04.    int    age;
05.
06.    Personne() {
07.       this( "Inconnu", "inconnu", -1 );
08.    }
09.
10.    Personne( String nom, String prenom, int age ) {
11.       this.name    = nom;
12.       this.address = prenom;
13.       this.age     = age;
14.    }
15. }
```

Il est possible d'appeler un constructeur dans un autre constructeur pour faciliter l'écriture.

Il est recommandé de toujours appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille grâce à l'utilisation du mot clé super.

Exemple :

```
01. class Employe extends Personne {
02.    int matricule;
03.    Employee() {
04.       super();
05.       matricule = -1;
06.    }
07.
08.    Employee(String nom, String prenom, int age, int matricule) {
09.       super(nom, prenom, age);
10.       this.matricule = matricule;
11.    }
12. }
```

Il est conseillé de ne mettre que du code d'initialisation des variables d'instances dans un constructeur et de mettre les traitements dans des méthodes qui seront appelées après la création de l'objet.

91.3.4. Les conventions de nommage des entités

Les conventions de nommage des entités permettent de rendre les programmes plus lisibles et plus faciles à comprendre. Ces conventions permettent notamment de déterminer rapidement quelle entité désigne un identifiant, une classe ou une méthode.

Entités	Règles	Exemple
Les packages	Toujours écrits tout en minuscules (norme Java 1.2)	com.entreprise.projet
Les classes, les interfaces et les constructeurs	La première lettre est en majuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_' Le nom d'une classe peut finir par impl pour la distinguer d'une interface qu'elle implémente. Les classes qui définissent des exceptions doivent finir par Exception.	MaClasse MonInterface MaClasse()
Les méthodes	Leur nom devrait contenir un verbe. La première lettre est obligatoirement une minuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule sans mettre de caractère underscore '_' Les méthodes pour obtenir la valeur d'un champ doivent commencer par get suivi du nom du champ. Les méthodes pour mettre à jour la valeur d'un champ doivent commencer par set suivi du nom du champ Les méthodes pour créer des objets (factory) devraient commencer par new ou create Les méthodes de conversion devraient commencer par to suivi par le nom de la classe renvoyée à la suite de la conversion	public float calculerMontant() {
	La première lettre est obligatoirement une minuscule et ne devrait pas être un caractère dollar '$' ou underscore '_' même si ceux-ci sont autorisés. Pour les variables d'instances non publiques, certains recommandent de commencer par un underscore pour éviter la confusion avec le nom d'une variable fournie en paramètre d'une méthode tel que le setter.	

Les variables	Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'. Les noms de variables composés d'un seul caractère doivent être évités sauf pour des variables provisoires (index d'une boucle). Les noms communs pour ces variables provisoires sont i, j, k, m et n pour les entiers et c, d et e pour les caractères.	String nomPersonne; Date dateDeNaissance; int i;
Les constantes	Toujours en majuscules, chaque mots est séparés par un underscore '_'. Ces variables doivent obligatoirement être initialisées lors de leur déclaration.	static final int VAL_MIN = 0; static final int VAL_MAX = 9;

91.4. Les séparateurs

L'usage des séparateurs tels que les retours à la ligne, les lignes blanches, les espaces, etc ... permet de rendre le code moins « dense » et donc plus lisibles.

91.4.1. L'indentation

L'unité d'indentation est constituée de 4 espaces. Il n'est pas recommandé d'utiliser les tabulations pour l'indentation.

Il est préférable d'éviter les lignes contenant plus de 80 caractères.

91.4.2. Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Deux lignes blanches devraient toujours séparer deux sections d'un fichier source et les définitions des classes et des interfaces.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :

- avant la déclaration d'une méthode,
- entre les déclarations des variables locales et la première ligne de code,
- avant un commentaire d'une seule ligne,
- avant chaque section logique dans le code d'une méthode.

91.4.3. Les espaces

Un espace vide devrait toujours être utilisé dans les cas suivants :

- entre un mot clé et une parenthèse.

Exemple :

```
1. while (i < 10)
```

- après chaque virgule dans une liste d'argument
- tous les opérateurs binaires doivent avoir un blanc qui les précèdent et qui les suivent

Exemple :

```
1. a = (b + c) * d
```

- chaque expression dans une boucle for doit être séparée par un espace

Exemple :

```
1. for (int i; i < 10; i++)
```

- les conversions de type explicites (cast) doivent être suivies d'un espace

Exemple :

```
1. i = ((int) (valeur + 10));
```

Il ne faut pas mettre d'espace entre un nom de méthode et sa parenthèse ouvrante.

Il ne faut pas non plus mettre de blanc avant les opérateurs unaires tels que les opérateurs d'incrément '++' et de décrément '--'.

Exemple :

```
1. i++;
```

91.4.4. La coupure de lignes

Il arrive parfois qu'une ligne de code soit très longue (supérieure à 80 caractères).

Dans ce cas, il est recommandé de couper cette ligne en une ou plusieurs en respectant quelques règles :

- couper la ligne après une virgule ou avant un opérateur
- aligner le début de la nouvelle ligne au début de l'expression coupée

Exemple :

```
1. maMethode(parametre1, parametre2, parametre3,
2.         parametre4, parametre5);
```

91.5. Les traitements

Même s'il est possible de mettre plusieurs traitements sur une ligne, chaque ligne ne devrait contenir qu'un seul traitement

Exemple :

```
1. i = getSize();
2. i++;
```

91.5.1. Les instructions composées

Elles correspondent à des instructions qui utilisent des blocs de code.

Les instructions incluses dans ce bloc sont encadrées par des accolades et doivent être indentées.
L'accolade ouvrante doit se situer à la fin de la ligne qui contient l'instruction composée.
L'accolade fermante doit être sur une ligne séparée au même niveau d'indentation que l'instruction composée.

Un bloc de code doit être définit pour chaque traitement même si le traitement ne contient qu'une seule instruction. Cela facilite l'ajout d'instructions et évite des erreurs de programmation.

91.5.2. L'instruction return

Elle ne devrait pas utiliser de parenthèses sauf si celle-ci facilite la compréhension

Exemple :

```
1. return;
2. return valeur;
3. return (isHomme() ? 'M' : 'F');
```

91.5.3. L'instruction if

Elle devrait avoir une des formes suivantes :

Exemple :

```
01. if (condition) {
02.    traitements;
03. }
04.
05. if (condition) {
06.    traitements;
07. } else {
08.    traitements;
09. }
10.
11. if (condition) {
12.    traitements;
13. } else if (condition) {
14.    traitements;
15. } else {
16.    traitements;
17. }
```

Même si cette forme est syntaxiquement correcte, il est préférable de ne pas utiliser l'instruction if sans accolades :

Exemple :

```
1. if (i == 10) i = 0;  // cette forme ne doit pas être utilisée
```

91.5.4. L'instruction for

Elle devrait avoir la forme suivante :

Exemple :

```
1. for ( initialisation; condition; mise à jour) {
2.      traitements;
3. }
```

91.5.5. L'instruction while

Elle devrait avoir la forme suivante :

Exemple :

```
1. while (condition) {
2.      traitements;
3. }
```

S'il n'y a pas de traitements, la forme est la suivante :

```
while (condition);
```

91.5.6. L'instruction do-while

Elle devrait avoir la forme suivante :

Exemple :

```
1. do {
2.    traitements;
3. } while ( condition);
```

91.5.7. L'instruction switch

Elle devrait avoir la forme suivante :

Exemple :

```
01. switch (condition) {
02. case ABC:
03.    traitements;
04.    break;
05. case DEF:
06.    traitements;
07.    break;
08. case XYZ:
09.    traitements;
10.    break;
11. default:
12.    traitements;
13.    break;
14. }
```

Il est préférable de terminer les traitements de chaque cas avec une instruction break et de l'enlever au besoin plutôt que d'oublier une instruction break nécessaire.

Toutes les instructions switch devrait avoir un cas 'default' en fin d'instruction : le traitement de tous les cas est une bonne pratique de programmation.

Même si elle est redondante, une instruction break devrait être incluse en fin des traitements du cas 'default' afin de généraliser la première recommandation.

91.5.8. Les instructions try-catch

Elle devrait avoir la forme suivante :

Exemple :

```
01. try {
02.    traitements;
03. } catch (Exception1 e1) {
04.    traitements;
05. } catch (Exception2 e2) {
06.    traitements;
07. } finally {
08.    traitements;
09. }
```

91.6. Les règles de programmation

91.6.1. Le respect des règles d'encapsulation

Il ne faut pas déclarer de variables d'instances ou de classes publiques sans raison valable.

Il est préférable de restreindre l'accès à la variable avec un modificateur d'accès protected ou private et de déclarer des méthodes respectant les conventions instaurées dans les javaBeans : getXxx() ou isXxx() pour obtenir la valeur et setXxx() pour mettre à jour la valeur.

La création de méthodes sur des variables private ou protected permet d'assurer une protection lors de l'accès à la variable (déclaration des méthodes d'accès synchronized) et éventuellement un contrôle lors de la mise à jour de la valeur.

91.6.2. Les références aux variables et méthodes de classes.

Il n'est pas recommandé d'utiliser des variables ou des méthodes de classes à partir d'un objet instancié : il ne faut pas utiliser objet.methode() mais classe.methode().

Exemple à ne pas utiliser si afficher() est une méthode de classe :

```
1. MaClasse maClasse = new MaClasse();
2. maClasse.afficher();
```

Exemple à utiliser si afficher() est une méthode de classe :

```
1. MaClasse.afficher();
```

91.6.3. Les constantes

Il est préférable de ne pas utiliser des constantes numériques codées en dur dans le code mais de déclarer des constantes avec des noms explicites. Une exception concerne les valeurs -1, 0 et 1 dans les boucles for.

91.6.4. L'assignement des variables

Il n'est pas recommandé d'assigner la même valeur à plusieurs variables sur la même ligne :

Exemple :

```
1. i = j = k;                //cette forme n'est pas recommandée
```

Il ne faut pas utiliser l'opérateur d'assignement imbriqué.

Exemple à proscrire :

```
1. valeur = (i = j + k ) + m;
```

Exemple :

```
1. i = j + k;
2. valeur = i + m;
```

Il n'est pas recommandé d'utiliser l'opérateur d'assignation = dans une instruction if ou while afin d'éviter toute confusion.

91.6.5. L'usage des parenthèses

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes liés à la priorité des opérateurs.

Exemple :

```
1. if (i == j && m == n)                // à éviter
2. if ( (i == j) && (m == n ))          // à utiliser
```

91.6.6. La valeur de retour

Il est préférable de minimiser le nombre d'instructions return dans un bloc de code.

Exemple à eviter :

```
1. if (isValide()){
2.    return true;
3. } else {
4.   return false;
5. }
```

Exemple :

```
1. return isValide();
```

Exemple :

```
1. if (isValide()) {
2.    return x;
3. } else return y;
```

Exemple à utiliser :

```
1. return (isValide() ? x : y)
```

91.6.7. La codification de la condition dans l'opérateur ternaire ? :

Si la condition dans un opérateur ternaire ? : contient un opérateur binaire, cette condition doit être mise entre parenthèses.

Exemple :

```
1. ( i >= 0 ) ? i : -i;
```

91.6.8. La déclaration d'un tableau

Java permet de déclarer les tableaux de deux façons :

Exemple :

```
1. public int[] tableau = new int[10];
2. public int tableau[] = new int[10];
```

L'usage de la première forme est recommandé.