JAVASTYLE

Guide francophone des conventions de codage pour la programmation en langage Java.

Hugo ETIEVANT
http://www.cyberzoide.net

AVANT-PROPOS

Date de dernière modification : 3 mai 2004.

Vérifiez sur le site de l'auteur si une mise à jour plus récente de ce document est disponible.

Ce document est téléchargeable à l'adresse suivante : http://www.cyberzoide.net/java/javastyle/.

Ce document est gratuit, il peut librement être utilisé par tout développeur. En revanche, aucune copie ni moyen de diffusion de se document ne peut être soumis à rétribution.

Ce document ne peut être modifié. Seules les copies conformes à l'original peuvent être diffusées.

Ce document reste l'entière propriété de son auteur : Hugo ETIEVANT.

HISTORIQUE

- 3 mai 2004 : annexe « JavaDoc » ajoutée
- 2 mai 2004 : annexes « mots réservés » et « qualifieurs » rajoutées
- 1er mai 2004 : création du document

REMERCIEMENTS

Merci à request et à Braim pour leurs corrections et suggestions.

SOMMAIRE

INTRODUCTION

DE L'UTILITE D'ADOPTER DES CONVENTIONS DE CODAGE

Dans le cycle de vie d'un produit logiciel, la phase de maintenance représente la majeur partie du temps (environ 80%). De plus, un logiciel est rarement développé par une seule personne, c'est une équipe entière qui réalise le projet de développement, avec toutes les contraintes de relecture et de compréhension que cela implique. Et les développeurs assurant la maintenance ne sont pas – en règle générale – ceux qui ont procédé à sa création, leur temps d'adaptation avant une pleine productivité est fortement dépendante de leur capacité à comprendre le code source et à assimiler la documentation relative au projet.

Ainsi, la réussite d'un projet logiciel, tant lors de la phase critique de développement que dans sa phase – tout aussi essentielle – de maintenance, dépend pour beaucoup des moyens mis en œuvre pour assurer une homogénéité dans le codage.

Cette homogénéité est assurée par la mise en œuvre de conventions strictes respectées par tous.

Bien que la plupart des éditeurs et outils de développement proposent des fonctionnalités permettant l'homogénéisation intuitive du code source (entre autre par l'auto indentation du code), il est des règles que ces outils ne peuvent imposer aux développeurs et que ces derniers doivent pouvoir appliquer même dans un environnement les plus simples (simple éditeur texte par exemple).

L'ORIGINE DES CONVENTIONS DECRITES

Le but de ce document est de présenter les conventions généralement admises par la communauté des développeurs en langage Java. Elles sont inspirées des règles proposées par Sun, des règles qui m'ont été inculquées à l'Université et de ma pratique du langage Java.

PRINCIPES GENERAUX

EQUILIBRE

Les différentes recommandations faites ici peuvent parfois entrer en conflit. Par exemple, la volonté d'avoir une structure esthétique d'un point de vue algorithmique et de faible complexité peut être un frein aux performances. Inversement, la course effrénée aux

performances (utilisation de la mémoire, vitesse d'exécution) peut amener à produire du code peu structuré et peu lisible. Il revient aux développeurs de trouver un équilibre entre ces recommandations en fonction des contraintes de leur projet. Il est évident qu'un projet de haute technique fortement contraint (typiquement : un logiciel embarqué) devra privilégier les performances .A l'inverse, un projet bureautique open source devra être fortement structuré.

BRIEVETE

Il faut être succinct ! Les structures alambiquées sont un frein à la lisibilité et à la compréhension ; compromettant d'autant la maintenance. Par contre, succinct ne veut pas dire laconique : les commentaires sont toujours les bienvenus.

UNIFORMITE

Il faut être homogène dans l'usage des recommandations faites ici. Certaines d'entre elles s'organisent harmonieusement en différents groupes. Ce sont ces groupes qui doivent être adoptés.

CONSISTANCE

Les recommandations que vous adopterez doivent être appliquées à l'intégralité de votre projet. Si les règles changent d'une classe à l'autre, la compréhension du code sera difficile. L'application uniforme d'un groupe de recommandations est un gage de maintenabilité.

SYSTEME DE FICHIERS

EXTENSIONS DES FICHIERS

Les noms des fichiers sources portent l'extension `.java` et le bytecode généré porte l'extension `.class`, les fichiers de configuration devraient porter l'extension `.properties`.

FICHIERS COMMUNS

Les fichiers les plus fréquemment joints au code source sont le fichier de directive de compilation `makefile` (utilitaire make), le fichier `build.xml` (utilitaire Ant), le fichier de description du contenu d'un projet `README`.

ORGANISATION

Un projet de développement logiciel est organisé en de multiples répertoires qui peuvent judicieusement adopter la structure suivante :

```
PROJET/
       build/
       config/
       docs/
       generated/
       idl/
       ior/
       lib/
       log/
       orb.bd/
       src/
```

Où `src/` contient les sources du projet, `build/` les classes compilées, `docs/` la documentation générées via JavaDoc, `idl/` les interfaces IDL[1] si nécessaire, `generated/` toutes classes intermédiaires générées lors du processus de compilation (par exemple selon les interfaces IDL), `config/` des fichiers de configuration nécessaires à l'exécution du projet, `log/` pour les fichiers de log d'exécution du projet. Et autres répertoires `ior/`, `orb.db/` nécessaires au stockage des objets générés en cours d'exécution.

[1] IDL (*Interface Definition Language*) : Langage d'interfaçage des objets sous Corba.

Ainsi un projet peut se contenter de la hiérarchie minimaliste suivante :

```
PROJET/
       build/
       docs/
       src/
```

Les classes doivent être organisées en packages. Les packages peuvent être structurés. Exemple :

```
PROJET/
       build/
       docs/
       src/
            pack1/
                  pack11/
                  pack12/
            pack2/
```

TAILLE DES SOURCES

Il est recommandé de ne pas excéder 2000 lignes dans un fichier `.java` et de ne pas dépasser les 80 colonnes.

FORMATAGE

Le formatage du code consiste en son écriture aéré et homogène.

INDENTATION

TABULATION

Il ne faut pas utilisé le caractère de tabulation car son interprétation varie selon les éditeurs. Configurez votre éditeur pour que la tabulation écrive 8 caractères espace.

BLOCS

L'entrée dans un nouveau bloc fils impose le rajout d'un niveau d'indentation. Deux blocs de même niveau doivent débuter sur la même colonne (même niveau d'indentation).

TAILLE DES LIGNES

Une ligne ne doit pas excéder 80 colonnes (indentations comprises).

Pour permettre l'écriture de code, il peut alors s'avérer nécessaire de revenir à la ligne à la suite d'une virgule séparant différents paramètres d'une méthode, où à la précédence d'un opérateur par exemple, et à l'extérieur de toute parenthèse, de préférence.

Exemples :

```
setMyVarOfMyClass(myVeryLongVar1, MyVeryLongExpression1, myVeryLongVar2,
        myLonguestExpression2, myLonguestVariable3, myLonguestExpression3,
        myLonguestVariable4);
```

et

```
double longSize = myVar1 * (myLongVar2 + myVeryLongVar3)
        -  anOtherVeryLongVar4;
```

alors que ceci est à proscrire :

```
Double longSize = myVar1 * (myLongVar2 +
        myVeryLongVar3) -  anOtherVeryLongVar4;
```

De même, au sein de l'expression d'une structure de contrôle, il est nécessaire de revenir à la ligne en cours d'expression et d'augmenter l'indentation de deux niveaux (et pas d'un seul). Par contre, le niveau d'indentation du bloc des instructions internes à la structure de contrôle n'est pas affecté. Tout ceci afin que les instructions et les lignes de l'expression ne soient pas au même niveau.

Exemple :

```
if (!(condition1 || condition2)
                    && (condition3 || condition4)
                    && (condition5 || !condition6)) {
          doSomethingAboutIt();
}
```

et pas :

```
if (!(condition1 || condition2)
          && (condition3 || condition4)
          && (condition5 || !condition6)) {
          doSomethingAboutIt();
}
```

On revient à la ligne dans les conditions suivantes :

- après une virgule
- avant un opérateur
- de préférence au sein d'une parenthèse de haut niveau, plutôt que de petit niveau
- le niveau d'indentation doit être le même que celui de la ligne précédente, sauf si cela peut prêter à confusion, à ce moment là, on augmente le niveau d'indentation

LIGNES BLANCHES

Les lignes blanches doivent être utilisées pour séparer les méthodes, et des portions de code distinctes.

ESPACES

Les espaces doivent être utilisés à profusion, mais pas n'importe où !

L'espace est obligatoire dans les conditions suivantes :

- avant et après tout opérateur, sauf la parenthèse pour laquelle cela est optionnel
- après toute virgule
- après tout mot réservé du langage
- avant et après toute accolade

L'espace est autorisé :

- entre le nom d'une méthode et la parenthèse ouvrante listant ses paramètres

En revanche, il est proscrit :

- avant les point-virgules de fin d'instruction
- avant les crochets des tableaux
- entre une variable et les opérateurs de pré/post incrément

- avant et après un opérateur de transtypage

Exemples :

A faire	*A ne pas faire*

`int a = b + (c – d);`
ou
`int a = b + ( c – d );`
	`int a=b+(c-d);`
	ni
	`int a = b + ( c - d ) ;`

`while (true) {`	`while(true){`
`myMethod (a, b, c, d);`	`MyMethod (a,b,c,d);`

`for (i = 0; i < 100; i++) {`
ou
`for ( i = 0; i < 100; i++ ) {`
	`for (i=0;i<100;i++) {`

`++count;`	`++ count;`

`(MyClass)myVar.get(i);`	`(MyClass) myVar.get(i);`
	ni
	`( MyClass )myVar.get(i);`

`myTab[myInd] = myValue;`
ou
`myTab[ myInd ] = myValue;`
	`myTab [myInd] = myValue;`

NOMMAGE

Une remarque générale doit être faite sur tous les identifiants que vous utiliserez. Ils doivent être explicites, c'est-à-dire que leur nom doit dénoter le contenu et la fonction de l'objet nommé. Ils doivent également être succincts, il est très difficile d'utiliser un identifiant à rallonge : plus il est court, mieux c'est.

Les acronymes apparaissant dans les noms doivent être passés en minuscule (sauf l'initiale s'il n'est pas le premier mot).

Les identifiants doivent être en langue anglaise. Ceci assure une diffusion maximale des sources et une maintenabilité optimale par des équipes de développement internationales.

PACKAGE

Les noms de package doivent être en minuscule et ne pas reprendre des noms déjà utilisés dans le JDK employé. Ces noms doivent de préférence être le nom de l'entreprise, du département, du projet. Le tout premier mot du nom devrait être un TLD[2] (« *top-level domain* ») tel que défini par le standard ISO[3] 3166, 1981.

Exemples :

A faire	*A ne pas faire*
`package mypackage;`	`package MyPackage;`
`package com.mycom.mypackage;`	`package Com.MyCom.MyPackage;`

CLASSES ET INTERFACES

Le nom des classes doit être en minuscule, hormis les initiales des mots le composant.

Exemples :

[2] TDL (*Top -Level Domain*) : suffix des noms de domaine de l'Internet. Ils sont de deux types : catégorie (com, alt, edu, info…) ou pays (fr, uk, be…).

[3] ISO (*International Standard Organization*) : un réseau d'instituts nationaux de normalisation de 148 pays.

A faire	*A ne pas faire*

`class MyFavoritClass;`	`class myfavoritclass;`
	ni
	`class myFavoritClass;`
	ni
	`class myfavoritClass;`

METHODES

Les noms des méthodes doivent être en minuscule hormis les initiales des mots le composant (sauf le premier).

Exemples :

A faire	*A ne pas faire*

`public void myFavoritMethod() {`	`public void MyFavoritMethod() {`
	ni
	`public void myfavoritMethod() {`
	ni
	`public void myfavorithethod() {`

`public void closeHtmlBrowser() {`	`public void closeHTMLBrowser() {`

Les accesseurs directs (getters et setters) des attributs d'une classe doivent être préfixés d'un `get` pour la lecture et d'un `set` pour l'écriture. Le suffixe doit être le nom de l'attribut.

Exemples :

A faire	*A ne pas faire*
`public int getLevel() {`	`public int giveMeTheLevelValue() {`
`public void setLevel(int level) {`	`public void writeLevel(int level) {`

Le préfixe `is` doit être utilisé par les méthodes retournant un booléen.

Exemple :

A faire	*A ne pas faire*
`public boolean isVisible() {`	`public boolean canWeSeeThat() {`

Certains autres préfixes particuliers doivent être utilisés : `compute`, `find`, `initialize` (ou `init`), `delete`, `add`, `close`, etc. respectivement pour le calcul, la recherche, l'initialisation, la suppression, l'ajout, la fermeture d'objets.

ATTRIBUTS, VARIABLES ET PARAMETRES

Les attributs de classe, les variables locales comme globales ainsi que les paramètres des méthodes doivent être en minuscule hormis les initiales des mots le composant (sauf le premier).

Le nom des variables doit être le plus explicite possible sur son contenu et être très court, à l'exception des variables temporaires. Typiquement les variables d'itération de boucle doivent porter un nom d'une seule lettre : `i`, `j`, `k`, `m`, et `n` pour les entiers ; `c`, `d`, et `e` pour les caractères ; `f`, `x`, `y` et `z` pour les flottants.

Le dollars (`$`) et le soulignement (`_`) sont proscrits.

Exemples :

A faire	*A ne pas faire*

`for (i = 0; i < 10; i++) {`	`for (myVeryLongCountVar = 0;`
	`myVeryLongCountVar < 10;`
	`myVeryLongCountVar++) {`

`Car nextCar = cars.get(this.id + 1);`	`Car a = cars.get(this.id + 1);`

`float myWidth = 145.5;`	`float w = 145.5;`
	ni
	`float my_Width = 145.5;`

Les collections d'objets doivent être nommées au pluriel.

Exemple :

A faire	*A ne pas faire*
`Vector accounts;`	`Vector account;`
`Collection Banks;`	`Collection Bank;`
`Object[] myObjs;`	`Object[] myObj;`

CONSTANTES

Les noms des constantes doivent être entièrement en majuscule. Le séparateur de mot est le caractère de soulignement (*underscore* : « _ »).

Exemples :

A faire	*A ne pas faire*

`static final int PROMPT_LOG = 1;`	`static final int PROMPTLOG = 1;`
	ni
	`static final int prompt_Log = 1;`
	ni
	`static final int Prompt_Log = 1;`
	ni
	`static final int Prompt_LOG = 1;`

COMMENTAIRES

Les commentaires sont essentiels au code source. Ils permettent d'inclure de la documentation à l'intérieur même du source en vue d'une génération automatique de documentation via JavaDoc, de faciliter la maintenance du projet, de permettre la distribution du code, d'inactiver certaines portions de code sans pour autant le supprimer (et d'avoir à le réécrire).

Il existe deux types de commentaires :

1. les commentaires mono ligne qui inactivent tout ce qui apparaît à la suite, sur la même ligne : `//`
2. les commentaires multi-lignes qui inactivent tout ce qui se trouve entre les deux délimiteurs, que ce soit sur une seule ligne ou sur plusieurs `/* */`

BLOC DE COMMENTAIRE

Exemple 1 : commenter une méthode, une classe, un attribut à l'aide d'un bloc de commentaire

```
/*
 * La classe MyClass fournis telles fonctionnalités…
 */
public class MyClass() {
```

COMMENTAIRE MONO LIGNE

Exemple 2 : insertion d'un commentaire afin d'expliquer le comportement du code

```
// Extraction de la fabrique ayant produit l'item
myFab = (Fabric)fabrics.get((int)item.getFabricId());
```

INACTIVATION D'UNE PORTION DE CODE

Exemple 3 : inactivation d'une portion de code pour débuggage

```
/*

// Extraction de la fabrique ayant produit l'item
myFab = (Fabric)fabrics.get((int)item.getFabricId());
```

```
// La fabrique supprime son dernier fils
myFab.deleteItem(myFab.getLastItem());
*/
```

PRIORITE ENTRE COMMENTAIRES

Il est important de ne réserver les commentaires multi-lignes qu'aux blocs utiles à JavaDoc et à l'inactivation de portions de code. Les commentaires mono-ligne permettant de commenter le reste, à savoir, toute information de documentation interne relatif aux lignes de code.

Ceci afin d'éviter des erreurs de compilation du aux imbrications des commentaires multi-lignes.

DECLARATIONS

VARIABLES

INDENTATION

Les variables doivent de préférence être déclarées lignes par lignes.

Exemple :

```
int level;
int frameWidth;
```

Sauf lorsqu'il s'agit de variables temporaires itératives pour lesquelles une déclaration globale sur une seule et même ligne est recommandée.

Exemple :

```
int i, j, k;
```

En revanche, il est formellement interdit de déclarer des variables de types différents sur la même ligne.

Exemple :

A faire	*A ne pas faire*
`int level;` `int[] carCount;`	`int level, carCount[];`

L'indentation peut exceptionnellement être adaptée lors de la déclaration de variables afin d'en aligner les identificateurs.

Exemple :

```
int    level;
float  myWidth;
Car    niceCar;
String authorName;
```

Les types tableaux doivent être spécifiés sur le type et non sur la variable.

Exemple :

A faire	*A ne pas faire*
`int[] carCount;`	`int carCount[];`

INITIALISATION

L'initialisation des variables doit se faire lors de la déclaration lorsque cela est possible.

Exemple :

A faire	*A ne pas faire*
`int level = 10;`	`int level;` `level = 10;`

EMPLACEMENT

Les variables doivent être déclarées au plus tôt, sitôt après l'accolade ouvrante du bloc. Et non pas juste avant leur utilisation dans le code.

Exemple :

A faire	*A ne pas faire*

`void myMethod() {`
`  int level;`
	`void myMethod() {`
	`  println("foobar");`
	`  int level;`

Une seule exception à cette règle : la structure de contrôle itérative `for` en dehors de laquelle la variable d'itération n'est point utilisée.

Exemple :

```
for (int i = 1; i < 10; i++) {
```

Une variable ne peut porter le même nom qu'une autre située dans un bloc de niveau supérieur.

Exemple :

A faire	*A ne pas faire*

`int level;`
`...`
`void myMethod() {`
`  int otherLevel;`
	`int level;`
	`...`
	`void myMethod() {`
	`  int level;`

METHODES

Les noms des méthodes sont accolés à la parenthèse ouvrante listant les paramètres. Aucun espace ne doit y être inséré.

Exemple :

A faire	*A ne pas faire*
`void myMethod() {`	`void myMethod () {`

BLOCS

Tout bloc est délimité par des accolades. L'accolade ouvrante doit être placée en fin de ligne, à la suite d'un espace, après l'instruction/méthode/classe créant le bloc. L'accolade fermante doit être placée en début d'une ligne vierge à la suite de la dernière instruction du bloc.

Exemples :

A faire	*A ne pas faire*

`void myMethod() {`
`  int level;`
`  ...`
`}`
	`void myMethod()`
	`{`
	`  int level;`
	`  ...`
	`}`

`while (true) {`
`  println("foobar");`
`}`
	`while (true) { println("foobar"); }`

`for (i = 1; i < 10; i++) {`
`  println("%d\n", i);`
`}`
	`for (i = 1; i < 10; i++){`
	`  println("%d\n", i);`
	`}`

ORDRE

L'ordre de déclaration des entités du code source doit être le suivant :

1. les attributs de la classe
    a. en premier les statiques (**`static`**)
    b. en deuxième les publiques (**`public`**)
    c. ensuite les protégés (**`protected`**)
    d. et enfin les privés (**`private`**)
2. les méthodes
    a. en premier les statiques
    b. en deuxième les publiques
    c. ensuite les protégées
    d. et enfin les privées

INSTRUCTIONS

SIMPLES INSTRUCTIONS

Une ligne de code ne peut contenir qu'une seule instruction (ou partie de celle-ci, si elle est trop longue).

Exemple :

A faire	*A ne pas faire*

`count++;`
`i--;`
`println("foobar");`
	`count++; i--; println("foobar");`

INSTRUCTION DE RETOUR

L'instruction de retour `return` peut ne retourner aucune valeur.

Cette instruction n'utilise pas les parenthèses, sauf, si elles sont syntaxiquement indispensables.

Exemples :

A faire	*A ne pas faire*
`return 50;`	`return (50);`
`return box.getWidth();`	`return (box.getWidth());`
`return ((size < MIN_SIZE) ? size : MIN_SIZE);`	`return (size < MIN_SIZE) ? size : MIN_SIZE;`

STRUCTURES DE CONTROLE

EXPRESSION TERNAIRE

Il y a trois manières d'écrire l'expression ternaire.

Exemple 1 :

```
resultExpr = (myVeryLongExpression) ? myFirstExpr : mySecondExpr;
```

Exemple 2 :

```
resultExpr = (myVeryLongExpression) ? myFirstExpr
                                    :  mySecondExpr;
```

Exemple 3 :

```
resultExpr = (myVeryLongExpression)
             ? myFirstExpr
             : mySecondExpr;
```

BOUCLE FOR

Syntaxes générales :

```
for (initialisation; condition; modification) {
    instructions;
}
for (initialisation; condition; modification);
```

BOUCLE WHILE

Syntaxes générales :

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

BOUCLE DO

Syntaxe générale :

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

CONDITION IF

Syntaxes générales :

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

```
    instructions;
}
```

CONDITION SWITCH

Les cas **`case`** d'un **`switch`** peuvent exceptionnellement ne pas incrémenter d'un niveau l'indentation de l'instruction **`switch`**. Cependant, par soucis d'homogénéité, il est recommandé d'augmenter l'indentation des cas puisqu'ils se trouvent à l'intérieur d'un nouveau bloc.

Syntaxe générale :

```
switch (condition) {
    case valeur1:
        instructions;
        // passe à travers

    case valeur2:
        instructions;
        break;

    case valeur3:
        instructions;
        break;

    default:
        instructions;
        break;
}
```

ou bien :

```
switch (condition) {
case valeur1:
    instructions;
    // passe à travers

case valeur2:
    instructions;
    break;

case valeur3:
    instructions;
    break;

default:
    instructions;
    break;
}
```

Les cas ne se terminant par un saut **`break`** doivent spécifier un commentaire rappelant que l'exécution se poursuit.

Le block d'instructions d'un cas n'a pas besoin d'être encadré par des accolades. Cependant, par soucis d'homogénéité, les accolades peuvent être rajoutées autour de ce bloc. Le saut sera alors inscrit à la suite de l'accolade fermante du bloc.

Syntaxe :

```
switch (condition) {
    case valeur1: {
```

```
        instructions;
    } // passe à travers

    case valeur2: {
        instructions;
    } break;

    case valeur3: {
        instructions;
    } break;

    default: {
        instructions;
    } break;
}
```

Toute instruction `switch` doit avoir un cas par défaut. Car il est rare de penser à tous les cas possibles y compris ceux erronés.

L'instruction de saut est obligatoire à la fin du cas par défaut. Cela est redondant mais protège d'une erreur en cas de rajout d'autres cas par la suite.

EXCEPTION TRY-CATCH

Les mots réservés `catch` et `finally` doivent être encadrés de l'accolade fermante du mot réservé qui le précède et de l'accolade ouvrante de son propre bloc.

Syntaxes générales :

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

```
try {
    instructions;
} catch (ExceptionClass1 e1) {
    instructions;
} catch (ExceptionClass2 e2) {
    instructions;
}
```

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

BONNES PRATIQUES DE DEVELOPPEMENT

Les « *best-pratices* » ou « *design-pattern* » sont en construction.

Vous les découvrirez à l'occasion de la future version de ce document.

LIENS

QUELQUES SITES UTILES

1. **Le CyberZoïde Qui Frétille**
   http://cyberzoide.developpez.com
   Webzine de vulgarisation des sciences et des nouvelles technologies.
2. **Developpez**
   http://www.developpez.com
   Club d'entraide des développeurs francophones
3. **Java Technology**
   http://java.sun.com

QUELQUES DOCUMENTS DE REFERENCE (EN ANGLAIS)

1. *"Code Conventions for the Java Programming Language"*, Sun
   http://java.sun.com/docs/codeconv/
2. *"Coding Standards for Java"*, Nejug
   http://www.nejug.org/standards.pdf

ANNEXE 1
LES MOTS RESEVES DU LANGAGE JAVA

Le tableau suivant présente tous les mots réservés du langage Java. Ils ne peuvent donc être utilisés en tant qu'identifiant.

Mot réservé	***Catégorie***	***Description***
`abstract`	*Qualifieur*	Définie une classe ou méthode abstraite dont l'implémentation reste à effectuer au travers d'une classe fille.
`boolean`	*Type primitif*	Type énuméré dont les valeurs possibles sont : `true` (vrai) ou `false` (faux).
`break`	*Structure de contrôle*	Au sein d'une structure conditionnelle `switch` défini un saut vers la fin de la structure afin d'éviter le test des cas suivants. Au sein d'une structure répétitive telle que `while`, `do` ou `for`, termine directement la boucle. Au sein d'un bloc, sort immédiatement du bloc en cours d'exécution.
`byte`	*Type primitif*	Type entier dont la valeur est comprise entre 2^{7}-1 et -2^{7} : [-128,127].
`case`	*Structure de contrôle*	Au sein d'une structure conditionnelle `switch`, défini un cas de test.
`catch`	*Structure de contrôle*	Au sein d'une structure d'exception `try`, défini un cas d'exception.
`char`	*Type primitif*	Type caractère dont la valeur est celle parmi la table ASCII du système.
`class`		Définie une classe.
`const *`		Définie une constante. Non utilisé.
`continue`	*Structure de contrôle*	Au sein d'une structure répétitive `while`, `do` ou `for` passe directement à l'itération suivante.
`default`	*Structure de contrôle*	Au sein d'une structure conditionnelle `switch` définie le cas de test par défaut.
`do`	*Structure de contrôle*	Structure répétitive effectuant au moins un parcours. Répétition tant que la condition est vrai.
`double`	*Type primitif*	Type flottant dont la valeur est comprise entre $(2-2^{-52})\cdot 2^{1023}$ et 2^{-1074}.
`else`	*Structure de contrôle*	Au sein d'une structure conditionnelle `if`, définie une alternative à la condition initiale.
`extends`		Définition de l'héritage. Spécifie la classe mère.
`false`	*Valeur*	Une des valeurs possible du type `boolean`.

`final`	*Qualifieur*	Spécifie que la classe/méthode/variable ne peut changer ni être dérivée.
`finally`	*Structure de contrôle*	Au sein d'une structure `try`, spécifie un bloc d'instructions toujours exécutés même si l'exception n'est jamais levée.
`float`	*Type primitif*	Type flottant dont la valeur est comprise entre $(2-2^{-23})\cdot 2^{127}$ et 2^{-149}.
`for`	*Structure de contrôle*	Structure répétitive.
`goto *`	*Structure de contrôle*	Saut vers un label. Non utilisé.
`if`	*Structure de contrôle*	Structure conditionnelle.
`implements`		Spécifie l'interface que la classe implémente.
`import`		Spécifie le(s) package(s) requis par la classe.
`instanceof`		Réalise au cours de l'exécution un contrôle sur le type réel d'une variable.
`int`	*Type primitif*	Type entier dont la valeur est comprise entre 2^{31}-1 et -2^{31}.
`interface`		Défini une interface.
`long`	*Type primitif*	Type entier dont la valeur est comprise entre 2^{63}-1 et -2^{63}.
`native`	*Qualifieur*	Défini une méthode qui est implémentée dans une bibliothèque annexe propre à la plateforme de développement (et qui donc n'est pas portable).
`new`		Création d'une nouvelle instance d'une classe par appel au constructeur.
`null`	*Valeur*	Valeur par défaut lorsqu'un objet n'est pas instancié.
`package`		Spécifie le package de la classe.
`private`	*Qualifieur*	Rend privé (inaccessible depuis les autres classes, y compris fille) la classe, méthode, variable.
`protected`	*Qualifieur*	Rend protégée (accessible seulement depuis les autres classes non filles) la classe, méthode, variable.
`public`	*Qualifieur*	Rend public (accessible à tous) la classe, méthode, variable.
`return`		Retour d'une valeur depuis une méthode.
`short`	*Type primitif*	Type entier dont la valeur est comprise entre 2^{15}-1 et -2^{15}.
`static`	*Qualifieur*	Création d'un unique exemplaire de la méthode, variable.
`strictfp **`	*Qualifieur*	Défini une classe, interface, méthode strictement conforme au type défini à la compilation.
`super`		Fait référence au constructeur de la classe mère.
`switch`	*Structure de contrôle*	Structure conditionnelle.
`synchronized`	*Qualifieur*	Défini une méthode sur laquelle des verrous permettront la synchronisation des différents threads d'exécution.
`this`		Fait référence à l'objet instancié.
`throw`		Déclenche une exception.
`throws`		Spécifie la liste des exceptions qu'une méthode est

		susceptible de lever.
`transient`	*Qualifieur*	Défini un attribut transitoire à ne pas sauvegarder lors de la sérialisation d'un objet.
`true`	*Valeur*	L'une des valeur possible du type `boolean`.
`try`	*Structure de contrôle*	Structure de contrôle des exceptions.
`void`	*Type primitif*	Type indéfini.
`volatile`	*Qualifieur*	Défini un attribut dont l'accès par différents threads sera ordonné.
`while`	*Structure de contrôle*	Structure de contrôle répétitive s'exécutant 0 ou n fois.

* : mot réservé non utilisé

** : mot réservé introduit en Java2

ANNEXE 2
LES QUALIFIEURS JAVA

Les qualifieurs sont des mots réservés du langage qui permettent de spécifier la nature des entités (classe, interface, méthode, attribut) qu'ils qualifient. Ils se positionnent sur la même ligne mais avant la définition de l'entité.

Le tableau suivant énumère les qualifieurs Java, leur portée et les décrit succinctement.

Qualifieurs	***Classe***	***Interface***	***Méthode***	***Attribut***	***Description***
`abstract`	X	X	X		Entité abstraite dont l'implémentation reste à effectuer au travers d'une classe fille.
`final`	X		X	X	Entité ne pouvant changer ni être dérivée.
`native`			X		Entité implémentée dans une bibliothèque annexe propre à la plateforme de développement (et qui donc n'est pas portable).
`private`	X	X	X	X	Entité privée (inaccessible depuis les autres classes, y compris fille).
`protected`	X	X	X	X	Entité protégée (accessible seulement depuis les autres classes non filles.
`public`	X	X	X	X	Entité publique (accessible à tous).
`static`			X	X	Création d'un unique exemplaire de l'entité.
`strictfp`	X	X	X		Entité strictement conforme au type défini à la compilation.
`synchonized`			X		Entité sur laquelle des verrous permettront la synchronisation des différents threads d'exécution.
`transient`				X	Entité transitoire à ne pas sauvegarder lors de la sérialisation d'un objet.
`volatile`				X	Entité dont l'accès par différents threads sera ordonné.

ANNEXE 3
LES COMMENTAIRES JAVADOC

JavaDoc est l'outil de génération automatique de documentation le plus répandu. Cette annexe présente l'ensemble des clauses JavaDoc ainsi que leur portée.

Clause	***Package***	***Classe et interface***	***Constructeur et méthode***	***Attribut***	***Description***
`@author`	X	X			Nom de l'auteur
`@deprecated`		X	X	X	Commentaire d'explication sur la péremption d'une entité
`@exception`			X		Synonyme de `@throws`
`@param`			X		Description d'un paramètre de la méthode
`@return`			X		Description de la valeur retournée
`@see`	X	X	X	X	Référence vers un concept, une URL, une autre entité de la documentation
`@serial`	X	X		X	Description d'une entité sérialisable
`@serialData`			X		Description d'un attribut sérialisable
`@serialField`				X	Description d'un attribut sérialisable
`@since`	X	X	X	X	Version du projet à laquelle a été introduite l'entité
`@throws`			X		Description d'une exception susceptible d'être levée
`@version`	X	X			Version de l'entité
`{@docRoot}`	X	X	X	X	Référence à la racine du répertoire d'export de la documentation générée
`{@inheritDoc}`			X		Référence à l'entité de laquelle on hérite
`{@link}`	X	X	X	X	Référence vers une autre entité de la documentation
`{@linkplain}`	X	X	X	X	Référence non active vers une autre entité de la documentation
`{@value}`				X	Valeur possible de l'attribut

Exemple de fichier source commenté avec JavaDoc :

```
package com.eliane;

import java.util.*;


/**
 * This class provide a persistant Musclor object.
 * @author Hugo ETIEVANT
 * @see ElianeCDC.doc
 * @see <a href="{@docRoot}/spec.html">Eliane Spec</a>
 * @since 1.3
 * @version 2.7
 */
class persistantMusclorImpl() extend Musclor implement persistantMusclor {


        /**
         * Level of force of the Musclor object
         * @serialField This field is serializable
         */
        private int level;


        /**
         * Temporary id of log file
         */
        static public transient String logId;


        /**
         * @throws IOException If an input or output exception occurred
         * @deprecated Replaced by {@link #Musclor(int)}
         */
        public persistantMusclorImpl() throws IOException {
                super();
        }


        /**
         * @param level Level of force
         * @throws IOException If an input or output exception occurred
         * @see #Musclor()
         * @serialData level Level is serializable
         */
        public persistantMusclorImpl(int level) throws IOException {
                super();
                this.level = level;
        }


        ...

}
```

INDEX