NORMES DE PROGRAMMATION AVEC LE LANGAGE JAVA

TABLE DES MATIÈRES

INTRODUCTION

De nos jours, les programmes d'une certaine taille ne sont plus conçus par une seule personne, mais plutôt par une équipe de programmeurs, d'analystes, de gestionnaires et de chercheurs travaillant tous en interaction. Il est donc très important que l'on trouve un seul style de programmation dans la version finale du programme.

La phase de conception d'un programme n'est jamais vraiment terminée. Car, tout au long de son développement et pendant sa durée de vie, de nouvelles contraintes ou de nouveaux besoins viennent modifier la structure des fonctions initialement écrites.

De nouvelles contraintes, des erreurs, de nouveaux besoins peuvent amener le programmeur, ou quelqu'un d'autre le remplaçant dans le projet, à effectuer certains changements dans le programme. C'est ce que l'on appelle la maintenance du programme. Pour que cela soit réalisable, il faut que le programme soit lisible et compréhensible. L'utilisation de normes de programmation améliore grandement la lisibilité d'un programme.

Les normes de programmation varient d'une entreprise à une autre. C'est parfaitement normal. Il y a souvent des guerres de clochers entre les programmeurs. Des normes particulières peuvent être sujettes à discussion, et il est souvent difficile de savoir qui a raison, qui a tort. Ce n'est pas important. Ce qui compte, c'est que tous suivent les mêmes normes, bonnes ou mauvaises. Il est souvent préférable d'appliquer des normes de programmation discutables, plutôt que de ne pas en avoir.

Voici les normes de programmation que nous utilisons dans les programmes Java, afin de les rendre lisibles, compréhensibles, et faciles à modifier. Un programme Java qui est codé en utilisant des normes de programmation, qui est bien documenté et qui découle d'une bonne approche de programmation, est nécessairement plus facile à mettre à jour.

COMMENTAIRES

Les commentaires sont utilisés pour inclure des explications dans le programme.

On doit commenter un fichier source pour indiquer des renseignements importants relatifs au fichier. Par exemple : le nom du ou des auteurs, le nom du fichier, les dates de création et de mises à jour, une description du fichier source, etc.

Chaque classe est commentée (avant sa définition) pour indiquer ce qu'elle représente.

Chaque méthode est commentée (avant sa définition) pour indiquer ce qu'elle accomplie, ce qu'elle retourne, et ce que font ses différents paramètres.

Chaque interface est commentée (avant sa définition) pour indiquer ce qu'elle représente.

Il faut commenter les déclarations de constantes, de variables d'instance, et de variables locales pour les expliquer. Un tel commentaire peut être placé à droite de l'énoncé de déclaration.

Des commentaires pertinents (des explications) doivent être insérés dans le code, là où il est nécessaire d'en faciliter la compréhension.

Il est préférable de mettre une ligne blanche avant et après un bloc de commentaires, afin d'aérer le tout.

NOMENCLATURE

Les noms des identificateurs doivent être significatifs. Ne pas utiliser les caractères accentués (pas toujours utilisables par tous les systèmes d'exploitation).

Le nom d'une classe, d'un constructeur ou d'une interface doit commencer par une lettre majuscule. Lorsque le nom contient plusieurs mots, chaque mot doit commencer par une lettre majuscule. Par exemple : `NomClasse`, `NomInterface`.

Le nom d'une méthode doit commencer par une lettre minuscule. Lorsque le nom contient plusieurs mots, chaque mot, à part le premier doit commencer par une lettre majuscule. Par exemple : `nomMethode`.

Le nom d'un champ, d'un paramètre ou d'une variable locale doit commencer par une lettre minuscule. Lorsque le nom contient plusieurs mots, chaque mot, à part le premier doit commencer par une lettre majuscule. Par exemple : `nomChamp`, `nomParametre`, `nomVariable`.

Le nom d'une constante doit être tout en majuscules. Lorsque le nom contient plusieurs mots, chaque mot doit être séparé d'un caractère de soulignement «_». Par exemple : `NOM_CONSTANTE`.

Le nom d'un package doit être tout en minuscules.

INDENTATION ET ESPACEMENT

L'indentation est le mode de présentation du code d'un programme qui met en évidence sa structure.

L'espacement est la façon d'ajouter des espaces pour aérer le code.

On décale de quatre espaces (par défaut dans Eclipse) vers la droite les énoncés d'un bloc d'énoncés mis entre accolades.

On aère judicieusement les énoncés, avec des lignes blanches, afin de bien délimiter les différents sujets.

Un fichier source (`.java`) imprimé ne doit pas contenir des lignes qui débordent. Après le formatage du code avec `Ctrl+Shift+F`, une ligne trop longue doit être coupée judicieusement afin de respecter l'indentation.

On ne met pas d'espace entre le dernier caractère d'un énoncé et le caractère «`;`».

On ne met pas d'espace entre le nom d'une méthode et la parenthèse ouvrante «`(`».

On ne met pas d'espace entre le nom d'un tableau et le crochet ouvrant «`[`».

On ne met pas d'espace entre le crochet fermant «`]`» et le crochet ouvrant «`[`» d'un tableau multidimensionnel. Par exemple :

```
rep = tab[0][4];
```

On ne met pas d'espace dans les parenthèses lors de l'appel d'une méthode qui ne contient aucun paramètre. Par exemple :

```
nombre = calculerNombre();
```

Un espace est mis au moins aux endroits suivants : après une virgule «`,`», et avant et après un opérateur (sauf pour les opérateurs qui s'appliquent à un seul opérande). Par exemple :

```
int nb1, nb2, rep;

rep = ( tab[nb1] * ++nb2 ) / 5;
```

La déclaration et l'initialisation d'un tableau sont disposées comme suit :

```
// Sur une seule ligne.

TypeÉlément[] nomTableau = { initialisation };

// Sur plus d'une ligne.

TypeÉlément[] nomTableau = {
    initialisation
};
```

Les énoncés qui font partie d'un énoncé `if`, `if... else`, `switch`, `while`, `do... while` ou `for`, **sont entourés d'accolades**, même s'il n'est pas obligatoire de le faire quand il n'y a qu'un seul énoncé. Ainsi, si l'on désire ajouter d'autres énoncés, les accolades qui sont alors nécessaires, seront déjà présentes. On évite ainsi des erreurs qui peuvent être difficiles à trouver.

L'énoncé `if` est disposé comme suit :

```
if ( condition ) {
    énoncés;
}
```

L'énoncé `if... else` est disposé comme suit :

```
if ( condition ) {
    énoncés1;
} else {
    énoncés2;
}
```

L'énoncé `if... else` concernant les choix multiples, est disposé comme suit :

```
if ( condition1 ) {
    énoncés1;
} else if ( condition2 ) {
    énoncés2;
.
.
.
} else if ( conditionn ) {
    énoncésn;
} else {
    énoncésn+1;
}
```

L'énoncé `switch` est disposé comme suit :

```
switch ( expression ) {
case valeur1:
    énoncés1;
    break;

case valeur2:
    énoncés2;
    break;
   .
   .
   .

case valeurn:
    énoncésn;
    break;

default:
    énoncésn+1;
    break;
}
```

L'énoncé `while` est disposé comme suit :

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

L'énoncé `do... while` est disposé comme suit :

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

L'énoncé `for` est disposé comme suit :

```
for ( initialisation; condition; expression ) {
    énoncés;
}
```

Une méthode est définie comme suit :

```
modificateursMéthode TypeRetour nomMéthode( paramètres ) {
    constantes locales;
    variables locales;

    énoncés;
}
```

Une méthode n'a qu'**un seul point de sortie**. Si la méthode retourne une valeur, il n'y a qu'**un seul énoncé `return`, à la fin de la méthode**.

Une classe est définie comme suit :

```
modificateursClasse [abstract] class NomClasse  [extends NomSuperClasse]
                                                [implements NomInterface1,
                                                   NomInterface2, . . .] {
    membres de la classe;

    constructeurs de la classe;

    accesseurs de la classe;

    mutateurs de la classe;

    méthodes de la classe;
}
```

Une interface est définie comme suit :

```
modificateursInterface interface NomInterface {
    constantes de l'interface;

    méthodes abstraites de l'interface;
    méthodes par défaut de l'interface;
    méthodes statiques de l'interface;
}
```

OPÉRATEURS ++ et --

L'opérateur « **++** » ajoute la valeur 1 à son opérande.
L'opérateur « **--** » retranche la valeur 1 à son opérande.

Ces deux opérateurs modifient leurs opérandes.

L'instruction pour ajouter la valeur 1 à une variable est la suivante :

```
++nomVariable;
```
OU
```
nomVariable++;
```

Remarque : Il **ne faut pas** écrire

```
nomVariable = nomVariable + 1;
```
OU
```
nomVariable += 1;
```

L'instruction pour retrancher la valeur 1 à une variable est la suivante :

```
--nomVariable;
```
OU
```
nomVariable--;
```

Remarque : Il **ne faut pas** écrire

```
nomVariable = nomVariable - 1;
```
OU
```
nomVariable -= 1;
```

OPÉRATEURS D'AFFECTATION

Les opérateurs d'affectation affectent à leur premier opérande la valeur résultant du calcul de leur premier opérande, de l'opérateur associé à l'affectation et de leur second opérande.

Voici quelques opérateurs d'affectation en Java : +=, -=, *=, /=, %=.

L'instruction pour ajouter la valeur 2 à une variable est la suivante :

```
nomVariable += 2;
```

Remarque : Il **ne faut pas** écrire

```
nomVariable = nomVariable + 2;
```

L'instruction pour retrancher la valeur 2 à une variable est la suivante :

```
nomVariable -= 2;
```

Remarque : Il **ne faut pas** écrire

~~`nomVariable = nomVariable - 2;`~~

EXPRESSIONS LOGIQUES ET VARIABLES BOOLÉENNES

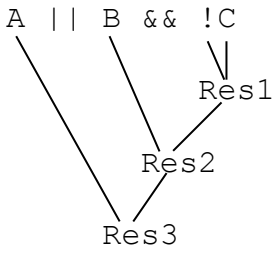
L'**algèbre de Boole** est très utile dans la construction d'expressions logiques avec les opérateurs `&&`, `||` et `!`.

Une expression logique (ou booléenne) est une expression dont la valeur appartient à l'ensemble `{false, true}`.

Ordre de priorité des opérateurs logiques pour évaluer une expression logique, lorsqu'il n'y a pas de parenthèses : `!`, `&&`, `||`.

Exemple

Supposons l'expression logique suivante : `A || B && !C`



l'expression `!C` sera évaluée en premier (`Res1`);

l'expression devient `A || B && Res1`;

l'expression `B && Res1` sers évaluée en deuxième (`Res2`);

l'expression devient `A || Res2`;

finalement l'expression `A || Res2` sera évaluée en dernier (`Res3`).

Quelques théorèmes utiles à connaître pour simplifier des expressions logiques

1. `!( !A )` → `A`

Exemple

`!( !( nb <= 4 ) )` → `!( nb > 4 )` → `nb <= 4`

2. `!( A || B )` → `!A && !B`

Exemple

`!( nb1 <= 4 || nb2 > 6 )` → `!(nb1 <= 4) && !(nb2 > 6)` → `(nb1 > 4) && (nb2 <= 6)`

3. `!( A && B )` → `!A || !B`

Exemple

`!( nb1 <= 4 && nb2 > 6 )` → `!(nb1 <= 4) || !(nb2 > 6)` → `(nb1 > 4) || (nb2 <= 6)`

Utilisation des variables booléennes

L'instruction pour vérifier qu'une variable booléenne contient la valeur `true` est la suivante :

```
if ( nomVariable ) {
     La variable nomVariable contient la valeur true
} else {
     La variable nomVariable contient la valeur false
}
```

Remarque : Il **ne faut pas** écrire

~~`if ( nomVariable == true ) {`~~
OU
~~`if ( nomVariable != false ) {`~~

L'instruction pour vérifier qu'une variable booléenne contient la valeur `false` est la suivante :

```
if ( !nomVariable ) {
     La variable nomVariable contient la valeur false
} else {
     La variable nomVariable contient la valeur true
}
```

Remarque : Il **ne faut pas** écrire

~~`if ( nomVariable == false ) {`~~
OU
~~`if ( nomVariable != true ) {`~~

Exemple

On veut vérifier s'il est possible de passer un après-midi de fin de semaine à la ronde ? On suppose que pour passer un après-midi de fin de semaine à la ronde, en plus d'être un après-midi de fin de semaine, il doit faire beau, on ne doit pas avoir de devoir à faire et on ne doit pas travailler.

On suppose que la variable `jour` contient un nombre entier (`0` pour dimanche, `1` pour lundi, `2` pour mardi, `3` pour mercredi, `4` pour jeudi, `5` pour vendredi et `6` pour samedi). La variable `jour` doit contenir la valeur `0` ou la valeur `6` pour vérifier si c'est la fin de semaine.

On suppose que la variable `heures` contient un nombre entier. La variable `heures` doit contenir une valeur entre `12` (inclusivement) et `18` (exclusivement) pour vérifier si c'est l'après-midi.

On suppose que la variable `beau` contient un booléen (`true` ou `false`). La variable `beau` doit contenir `true` pour vérifier s'il fait beau.

On suppose que la variable `devoir` contient un booléen (`true` ou `false`). La variable `devoir` doit contenir `false` pour vérifier qu'il n'y a pas de devoir à faire.

On suppose que la variable `travail` contient un booléen (`true` ou `false`). La variable `travail` doit contenir `false` pour vérifier qu'on ne doit pas travailler.

Voici donc l'expression logique simplifiée qui permet de résoudre ce problème.

```
// On suppose que les variables jour et heures sont de type int
// et qu'elles contiennent des valeurs.
// On suppose que les variables beau, devoir et travail sont de type boolean
// et qu'elles contiennent des valeurs.

if (  ( jour == 0 || jour == 6 ) && ( heures >= 12 && heures < 18 )
      && beau && !devoir && !travail ) {

      System.out.println( "Il est possible d'aller à la ronde." );

} else {
      System.out.println( "Il n'est pas possible d'aller à la ronde." );
}
```

L'instruction pour affecter la valeur `true` ou `false` à une variable booléenne selon une condition est la suivante :

```
nomVariable = condition;
```

Remarque : Il **ne faut pas** écrire

```
if ( condition ) {
     nomVariable = true;
} else {
     nomVariable = false;
}
```

OU

```
nomVariable = ( condition ) ? true : false;
```

Exemple

Supposons la variable booléenne suivante :

```
boolean pair;
```

Pour affecter la valeur `true` ou `false` à la variable `pair` selon si un nombre est pair, on peut écrire l'instruction suivante :

```
pair = ( ( nb % 2 ) == 0 );
```

Au lieu de :

```
if ( ( nb % 2 ) == 0 ) {
     pair = true;
} else {
     pair = false;
}
```

Ou de :

```
pair = ( ( nb % 2 ) == 0 ) ? true : false;
```

La façon suivante est acceptée :

```
boolean pair = false;

if ( ( nb % 2 ) == 0 ) {
     pair = true;
}
```