Normes de programmation en JAVA

Le « listing » est une partie importante. Celui-ci représente de façon claire et compréhensible la résolution, dans un langage de programmation, du problème que vous venez de résoudre. Nous allons ici, nous préoccuper de la forme du « listing » et de la syntaxe du langage.

La forme et/ou la présentation du « listing » doivent être normalisées, pour **offrir une uniformisation des projets des différents programmeurs** dans une équipe de travail ou dans une entreprise. C'est pour cela que nous allons présenter une série de règles qui devront être obligatoirement suivies par tous les programmeur(se)s sous peine de pénalité.

Pourquoi ?

On définit des règles pour permettre d'écrire des programmes qui sont:

- *Corrects*
- *Avec un style consistant et cohérent*
- *Faciles à lire et à comprendre*
- *Exempts des types d'erreurs communes*
- *Faciles à maintenir par différents programmeurs*

Le programmeur doit donc suivre les règles décrites dans ce document dans le but d'éviter ces embûches :

- *Le langage Java est un langage facile mais très riche dans lequel la frontière entre une subtilité et un «bug» est très mince.*
- *Le programmeur a une très grande responsabilité et il doit se discipliner.*
- *Comme dans bien des langages, le Java permet d'écrire du code compact mais illisible.*

La syntaxe est un élément important dans la rédaction du code. Elle permet de dépersonnaliser le code et de le rendre plus homogène. Cette section comporte quelques parties:

- *les noms*
- *les compteurs*
- *les commentaires*
- *les indentations et les imbrications*
- *les énoncés de contrôle alternatifs*
- *les énoncés de contrôle itératifs*

Régles de forme et de syntaxe

RP-1:

Chaque fois qu'une règle est violée, la raison doit en être clairement documentée par un commentaire.

RP-2:

En Java, les fichiers d'implémentation ont l'extension « **.java** ».

```
UneClasse.java
Personne.java
Camion.java
```

RP-3:

Un fichier d'implémentation (d'extension « **.java** ») ne devrait contenir que la définition d'une seule classe et ses classes internes.

```
public class Personne
{
   private static final int NBRPERSONNE_DEFAUT = 0;
   private static final int VALEUR_NBRPERSONNE_MIN = 0;
   private static final int VALEUR_NBRPERSONNE_MAX = 100;

   private static final String NOM_DEFAUT = "Un nom";
   private static final int LONG_NOM_MIN = 1;
   private static final int LONG_NOM_MAX = 30;

   private static final int AGE_DEFAUT = 1;
   private static final int VALEUR_AGE_MIN = 1;
   private static final int VALEUR_AGE_MAX = 100;

   private int age = 0;
   private String nom = "";
   private static int nbrPersonne = Personne.NBRPERSONNE_DEFAUT;

   public Personne()
   {
      this(Personne.AGE_DEFAUT, Personne.NOM_DEFAUT);
   }

   public Personne(int paramAge,
                   String paramNom)
   {
      boolean ok = (setAge(paramAge) &&
                    setNom(paramNom));

      if (ok == false) //OU if (!ok)
      {
         setAge(Personne.AGE_DEFAUT);
         setNom(Personne.NOM_DEFAUT);
      }

    • • •

   public static boolean validerNbrPersonne(int paramNbrPersonne)
   {
      return ((paramNbrPersonne >= Personne.VALEUR_NBRPERSONNE_MIN) &&
            (paramNbrPersonne <= Personne.VALEUR_NBRPERSONNE_MAX));
   }
```

```
}
```

RP-4:

Tous les fichiers d'implémentation doivent avoir un en-tête décrivant le contenu du fichier. Voir les règles RP-21: et RP-22: pour la syntaxe.

RP-5:

Ne jamais définir des **attributs** d'objet qui ont comme visibilité « **public** ». Les déclarer avec la visibilité « **private** » et définir au besoin, des méthodes publiques d'**accès**, de **mutations** et de **validations** à cet attribut.

```
//Mauvaise façon
public class Date
{
    public int jour = 0;

    ...

}


//Bonne façon
public class Date
{
    private int jour = 0;

    ...

    public int getJour()
    {
       ...
    }

    public boolean setJour(int paramJour)
    {
       ...
    }

    public static boolean validerJour(int paramJour)
    {
       ...
    }

    ...

}
```

RP-6:

Ne jamais utiliser des nombres « **magiques** » ou « **littéraux** » dans le code, utiliser plutôt

une constante. Les constantes doivent toujours être définies avec « **static final** ». Elles peuvent avoir une visibilité publique ou privée.

```
//Bonne façon
public class MaClasse
{
   public static final float PI = 3.1415f;

   ...

   public void uneMethode()
   {
      ...
   }
}


//Mauvaise façon
public class MaClasse
{

   ...

   public void uneMethode()
   {
      if (niveau >= 3.567)
      {
         ...
      }
   }
}



//Bonne façon
public class MaClasse
{
   private static final float NIVEAU_CRITIQUE = 3.567f;

   ...

   public void uneMethode()
   {
      if (niveau >= MaClasse.NIVEAU_CRITIQUE)
      {
         ...
      }
   }
}
```

RP-7:

Définir les variables le plus **localement** possible de leur utilisation.

```
//Mauvaise façon
public class MaClasse
```

```
{
   //Cette variable n'est pas un attribut de la classe mais une
   //variable quelconque et elle est utile à un seul endroit.
   int uneVariableGlobale = 0;

   ...

   public void uneMethode(int paramVal)
   {
      if (uneVariableGlobale >= paramVal)
      {
         ...
      }
   }
}
```

```
//Bonne façon
public class MaClasse
{

   ...

   public void uneMethode(int paramVal)
   {
      int uneVariableLocaleTemp = 0;

      ...

      if (uneVariableLocaleTemp >= paramVal)
      {
         ...
      }
   }
}
```

RP-8:

Définir une **seule variable** par énoncé de déclaration.

```
//Mauvaise façon
public class MaClasse
{

   ...

   public void uneMethode(int paramVal)
   {

         int unEntier, unAutre;

      ...

   }
}
```

```
//Bonne façon
public class MaClasse
{

   ...

   public void uneMethode(int paramVal)
   {

      int unEntier = 0;
      int unAutre = 0;

      //OU

      int unEntier = 0,
          unAutre = 0;

      ...

   }
}
```

RP-9:

Une variable doit être initialisée avant d'être utilisée et si possible, préférer **l'initialisation** à **l'assignation**.

```
//Mauvaise façon
public class MaClasse
{

   ...

   public void uneMethode(int paramVal)
   {

      int unEntier;

      ... //plusieurs lignes de code

      unEntier = 10;
      unEntier = unEntier * paramVal;

      ...

   }
}
```

```
//Bonne façon
public class MaClasse
{

   ...
```

```
    public void uneMethode(int paramVal)
    {

        int unEntier = 10;

        ... //plusieurs lignes de code

        unEntier = unEntier * paramVal;

        ...

    }
}
```

RP-10:

Écrire une seule instruction par ligne, permet une meilleure lisibilité du code.

```
//Mauvaise façon
public class UneClasse
{

    ...

    public void faireUnSi(float paramVal)
    {
        float temp = 0f; temp++; temp = temp + paramVal;
        if (paramVal > 1) temp = paramVal; else temp = temp * paramVal;
    }

    ...

}



//Bonne façon
public class UneClasse
{

    ...

    public void faireUnSi(float paramVal)
    {
        float temp = 0f;

        temp++;
        temp = temp + paramVal;

        if (paramVal > 1)
        {
            temp = paramVal;
        }
        else
        {
            temp = temp * paramVal;
```

```
        }
    }

    ...

}
```

RP-11:

Le nom d'une classe doit commencer par **une lettre majuscule**. Si le nom de la classe est formé par des mots composés, alors la première lettre de chaque mot est aussi en majuscule.

```
//Soit la classe SectionCritique

//Mauvaise façon
public class Sectionncritique
{
    ...
}



//Bonne façon
public class SectionCritique
{
    ...
}
```

RP-12:

Le nom d'une méthode doit commencer par une lettre minuscule. Lorsqu'un nom de méthode est formé par des mots composés, la première lettre des autres mots est en majuscule. **Un nom de méthode doit débuter obligatoirement par un verbe**, s'il y a lieu, il est accompagné d'un complément et **doit exprimer clairement** ce que fait la méthode.

```
//Bonne façon
public class MaClasse
{

    ...

    public static String lireChaine(String paramMessage)
    {
        ...
    }

    public void dessinerDansPlan(Graphics paramPlan)
    {
        ...
    }

    private float convertirDateEnFloat()
    {
        ...
    }
```

```
    ...

}
```

RP-13:

Le nom de variable (attribut ou argument) doit commencer par une lettre minuscule et lorsqu'il est composé de plus d'un mot, chaque première lettre des autres mots est en majuscule. **Un nom de variable doit exprimer clairement** ce que contient la variable.

```
//Bonne façon
public class MaClasse
{
   private String nom = "";
   private float salaireAnnuel = 0f;

   ...

   public static String lireChaine(String paramMessage)
   {
      int valTemp = 0;
      boolean pasValide = true;

      ...

   }

   ...

}
```

RP-14:

Le nom de la classe doit obligatoirement être placé devant un nom de **méthode** « **static** » ou **d'attribut** « **static** ». Cette règle s'applique particulièrement pour les méthodes ou attributs « **static** » de la classe propriétaire.

```
//Bonne façon
public classe UneClasseExemple
{
   private static String type = "";
   private static float degreDeBase = 0.0f;
   private static int nbrInstances = 0;

   ...

   private static float getDegreDeBase()
   {
      return UneClasseExemple.degreDeBase
   }

   public static void setType(String paramType)
   {
      type = UneClasseExemple.degreDeBase;
   }

   public static void main(String[] args)
```

```
    {
        int nbrObjet = UneClasseExemple.nbrInstances;
        float degreTemp = 0f;

        UneClasseExemple.setType("Un Autre Type");
        degreTemp = UneClasseExemple.getDegreDeBase();

        ...
    }
}
```

RP-15:

Les noms de constantes sont toujours définis **en lettres majuscules**. Les mots différents la composant doivent être **séparés par des soulignés**. Les constantes devraient toujours être définies au début de la classe.

```
//Bonne façon
public class UneClasse
{
    public static final int MAXIMUM_VALIDE = 1;
    public static final double NIVEAU_CRITIQUE = 4.0;

    ...
}
```

RP-16:

Les variables utilisées comme compteurs de boucle peuvent être définies seulement au moment de leur utilisation.

```
//Bonne façon
public class UneClasse
{

    ...

    public void faireDeuxBoucle()
    {
        for (int i = 0; i < UneClasse.MAX_LIGNE; i++)
        {
            for (int j = 0; j < UneClasse.MAX_COLONNE; j++)
            {
                ...
            }
        }
    }

    ...

}
```

RP-17:

L'indentation et l'imbrication permettent de visualiser la structure d'un programme. Une indentation trop faible nuit à la lisibilité, tandis qu'une indentation trop grande décale de façon inutile tout le code vers la droite. La norme sera donc une indentation d'une tabulation

(quatre espaces). L'imbrication de début de bloc doit être placée à la ligne suivant la condition et doit être indentée.

```
public class UneClasse
{

   ...

   public static int trouverPlusPetit(int[] paramVecteur)
   {
      int plusPetit = paramVecteur [0];      //Valeur de comparaison

      //Boucle de recherche
      for (int i = 0; i < paramVecteur.length; i++)
      {
         //Determiner si j'ai plus petit que le plus petit que j'ai deja
         if (plusPetit > paramVecteur [i])
         {
            plusPetit = paramVecteur [i];
         }
      }

      //Retour du plus petit
      return plusPetit;
   }

   ...
}
```

RP-18:

Laisser **un espace** entre l'énoncé de contrôle comme « **if** », « **for** », « **while** », etc. et la condition qui le suit.

```
public class UneClasse
{

   ...

   public static int trouverPlusPetit(int[] paramVecteur)
   {
      int plusPetit = paramVecteur [0];      //Valeur de comparaison

      //Boucle de recherche
      for (int i = 0; i < paramVecteur.length; i++)
      {
         //Determiner si j'ai plus petit que le plus petit que j'ai deja
         if (plusPetit > paramVecteur [i])
         {
            plusPetit = paramVecteur [i];
         }
      }

      //Retour du plus petit
      return plusPetit;
   }
```

```
    ...
}
```

RP-19:

Les accolades sont toujours obligatoires même si on a une seule expression dans l'énoncé de contrôle. Elles sont placées au même niveau que l'énoncé de contrôle.

```
public class UneClasse
{

    ...

    public static int trouverPlusPetit(int[] paramVecteur)
    {
        int plusPetit = paramVecteur [0];      //Valeur de comparaison

        //Boucle de recherche
        for (int i = 0; i < paramVecteur.length; i++)
        {
            //Determiner si j'ai plus petit, que le plus petit que j'ai déjà
            if (plusPetit > paramVecteur [i])
            {
                plusPetit = paramVecteur [i];
            }
        }

        //Retour du plus petit
        return plusPetit;
    }

    ...
}
```

```
public class UneClasse
{

    ...

    public void faireUnSi(float paramVal)
    {
        float temp = 0f;

        temp++;
        temp = temp + paramVal;

        if (paramVal > 1)
        {
            temp = paramVal;
        }
        else
        {
            temp = temp * paramVal;
        }
```

```
    }

    ...

}
```

RP-20:

Dans l'énoncé « **switch** » les structurants « **case** » sont indentés d'une tabulation (4 espaces) vers la droite du « **switch** ». Le code qui suit un « **case** » est également indenté d'une tabulation (4 espaces) et on peut terminer chaque cas par un énoncé « **break** ». Un énoncé « **switch** » **doit obligatoirement avoir un énoncé** « **default** » pour traiter les cas non prévus.

```
public class UneClasse
{

   ...

   public static void main(String[] args)
   {

      int valeur = 0;

      ...

      switch(valeur)
      {
         case 1:
         {
            //Traitement pour le cas 1
            break;
         }
         case 2:
         {
            //Traitement pour le cas 2
            break;
         }
         default:
         {
            //Traitement pour les autres cas
            break;
         }
      }

      ...
   }

   ...
}
```

RP-21:

Le commentaire global du fichier d'implémentation « **.java** » sont sous la forme de **JavaDoc** (*voir le document à ce sujet dans les notes de cours sur le réseau*) et devrait contenir:

- *la description de la fonctionnalité générale de la classe.*

- *des notes aux développeurs (**TODO**) indiquant les particularités d'implantation s'il y a lieu.*
- *Version.*
- *date de création et nom de l'auteur en ordre chronologique.*
- *dates de modification, description et nom du programmeur.*
- *références à d'autres classes ou méthodes.*

```
/**
 * Cette classe gère le temps au format <b>24 heures</b>.
 *
 * Mettre tout autre détail de description des services de la classe.
 * <i><u><b>On peut rendre cela plus joli avec du HTML aussi.</u></b></i>
 *
 * TODO Il est aussi possible de mettre des notes qui ne seront pas dans la
 * JavaDoc, mais seulement disponible aux développeurs, par exemple: Pour
 * l'instant il n'y a que des fonctions de base mais, cette classe pourrait être
 * enrichie.
 *
 * @version 2.1 (19 septembre 2007)
 *
 * @author Robert Aubé (19 août 2007)
 * @author Julie Frève (12 février 2004)
 *
 * @see java.util.Date
 * @see java.text.DateFormat
 * @see java.util.Calendar
 * @see java.util.TimeZone
 *
 * @deprecated Cette classe est vue à être remplacée bientôt!
 */
```

RP-22:

Les en-têtes de méthodes devraient contenir :

- *Un sommaire ou description.*
- *Description du ou des paramètres d'entrée de la méthode.*
- *Description du type d'élément de sortie de la méthode.*
- *Note pour le programmeur s'il y a lieu (**facultative**).*
- *Cas d'utilisation (**facultatif**).*

```
/**
 * Définit une nouvelle valeur de temps en format <b>universel</b>. Vérifie la
 * validité des données.
 *
 * @param pHeure un entier, l'heure entre 0-24
 * @param pMinute un entier, les minutes entre 0-59
 *
 * @return float, l'objet courant une fois le temps mise à jour avec les
 *         valeurs reçus en argument.
 *
 * @throws RuntimeException dans le cas d'un temps non valide. i.e. que
 *             l'heure n'est pas entre 0-24 ou que les minutes transmises ne
 *             sont pas entre 0-59.
 *
```

```
 * TODO Il faut valider ceci et cela!
 *
 * @deprecated Cette méthode est vue à être remplacée bientôt!
 */
```

RP-23:

Il est très **IMPORTANT** de mettre des commentaires dans le programme. Placez des commentaires significatifs expliquant le pourquoi (explication d'implication, rôle joué dans le programme) d'une ligne ou d'un groupe de lignes de code. Placez les commentaires à la fin de la ligne **pour les déclarations** et **au-dessus de celle-ci pour le reste du code**.

```
public class UnePile
{

   ...

   public String depiler()
   {
      String valTemp = ""; //Chaine temporaire pour le retour d'info.

      //Ne pas dépiler si la pile est vide.
      if (!estVide())
      {
         //Prendre le dessus de la pile.
         valTemp = getDessus();
         //Supprimer la valeur du dessus pour mettre la pile à jour.
         v.removeElementAt(0);
      }

      //Retourner la valeur dépilée.
      return valTemp;
   }

   ...
}
```

RP-24:

Une méthode de classe ou d'instance doit avoir **un seul point de sortie**. Soit l'accolade « **}** »de fin de celle-ci ou l'instruction « **return** » aussi à la fin.

```
public class UnePile
{

   ...

   public String depiler()
   {
      String valTemp = ""; //Chaine temporaire pour le retour d'info.

      //Ne pas dépiler si la pile est vide.
      if (!estVide())
      {
         //Prendre le dessus de la pile.
         valTemp = getDessus();
         //Supprimer la valeur du dessus pour mettre la pile à jour.
         v.removeElementAt(0);
```

```
        }

        //Retourner la valeur dépilée.
        return valTemp;
    }

    ...
}
```