Rappels de base en Java

1) **Conventions d'écriture** pour normaliser la rédaction du code (lorsqu'on code à plusieurs)

Objectifs : Augmenter la lisibilité et la compréhension du code source.
Obtenir du code prédictible et facilement modifiable.
Que tous les développeurs d'un même projet puissent comprendre et appréhender le code des autres.

a. Le nom d'une classe commence toujours par une **MAJUSCULE**, le reste est en minuscule
b. Les noms d'une variable, d'une méthode, d'un type primitif (int, float, char …) **SONT TOUJOURS EN MINUSCULE** (même la première lettre)
c. Si un nom de variable ou de fonction **contient PLUSIEURS MOTS**, le début de chaque nouveau mot est en majuscule (pas de caractère souligné)
d. Les constantes sont toujours écrites **EN MAJUSCULES** avec un caractère souligné entre chaque mot
e. Les noms doivent être **TOUS** en anglais ou **TOUS** en français.
f. Les **caractères spéciaux** comme (TAB) et le saut de page (page break) doivent être **EVITES**
g. Quelques suggestions pour les noms de variables :
   - **Le préfixe « n » :** variables représentant un nombre d'objets (*ex : nPoints, nLignes*)
   - **Le préfixe « no » :** variables représentant un numéro d'entité (*ex : noEmploye)*
h. Utiliser les verbes suivants dans le nom des fonctions :
   - **obtenir/modifier, lire/écrire :** là où un attribut est accessible directement (*ex : employe.obtenirNom();*)
   - **calculer :** si des méthodes de calcul sont utilisées (*ex : matrice.calculerInverse();*)
   - **trouver :** méthodes de recherche (*ex : noeud.trouverCourtChemin(NoeudDestination);*)
   - **initialiser** : méthode d'initialisation (*ex : imprimante.initialiserStyle();*)
i. L'utilisation des variables globales devrait être **minimisée**
j. Les attributs de classes ne devraient **JAMAIS** être déclarés publics *(principe d'encapsulation)*

2) Les **classes** en Java

a. Tous les **CONSTRUCTEURS** d'une classe **PORTENT LE MÊME NOM que la classe** et ne retournent **AUCUNE valeur**. C'est dans le **CONSTRUCTEUR** qu'on **initialise** la valeur des variables membres et **PAS** au moment de leur déclaration !

b. Une **méthode** est caractérisée par sa **SIGNATURE**, c'est-à-dire son **nom**, le type de sa **valeur de retour** et le nombre et le type de ses **paramètres**. Les **instructions** définies après la signature, entre accolades, représentent le **CORPS** de la méthode

c. La **SURCHARGE** de méthode est un processus qui permet de donner le **même nom à des méthodes** d'une classe, à condition que leurs **paramètres diffèrent** : en type et/ou en nombre. Il est ainsi possible de définir une méthode réalisant la même opération sur des variables différentes en nombre ou en type, comme par exemple :

```
- int somme( int p1, int p2){ return (p1 + p2); }
- float somme( float p1, float p2){ return (p1 + p2);}
- float somme( float p1, float p2, float p3){return (p1 + p2 + p3);}
- int somme( float p1, int p2){ return (int(p1) + p2); }
```

d. Lorsqu'une **classe B hérite d'une classe A**, alors **B hérite de TOUTES les méthodes** de A, *c'est-à-dire* qu'on peut appeler n'importe quelle méthode définie dans **A** à partir de n'importe quel objet de type **B**. Lorsqu'on **REDEFINIT** une méthode dans la classe **B**, sa **signature** doit être **EXACTEMENT LA MÊME** que celle écrite dans la superclasse **A** (mêmes nombres et types d'arguments). De cette façon, la méthode originale (celle de **A**) est **ignorée** au profit de sa redéfinition (dans **B**).

e. Les classes **héritent TOUTES** de la **classe Object** (pas la peine de le préciser avec « extends Object »). Elles héritent ainsi de **TOUTES les méthodes** définies dans la classe **Object** qu'il faut généralement **redéfinir** pour obtenir le comportement spécifique voulu. Les méthodes suivantes sont généralement redéfinies :
   - `public String toString()` : qui doit retourner une **chaîne de caractère**, donc **PAS DE « System.out.println »** dans cette méthode !
   - `public boolean equals(Object obj)` : qui évalue si deux objets de la classe sont **équivalents**. Il faut commencer par vérifier que « `obj` » est bien une **instance de la classe** avec l'instruction « `instanceof` »

f. On n'appelle **JAMAIS** une méthode avec le **mot-clé « `new` »**, puisqu'il sert à **créer** (**construire**) un objet, en **faisant appel à un des CONSTRUCTEURS** définis dans la classe. Si aucun constructeur n'est défini dans la classe, un **constructeur par défaut sans paramètre** se charge d'attribuer une valeur par défaut à chaque variable membre (« `0` » pour les types primitifs, « `false` » pour les boolean et « `null` » autrement). *Exemples :*
   - ```
     public class A {
           private int a;   private char b;
           public A(int a) {this.a = a; }
           public A(char b) {this.b = b; }
           public A(int a, char b) {this.a = a;  this.b = b;}
     }
     ```
   - Si on écrit dans la méthode « `main` » :
     ```
     A obj1 = new A() ;        //appel du constructeur par défaut sans paramètre
     A obj2 = new A(2) ;       //appel du premier constructeur défini
     A obj3 = new A('f') ;     //appel du deuxième constructeur défini
     A obj4 = new A(2,'e') ;   //appel du troisième constructeur défini
     ```

g. Par défaut, il est **IMPOSSIBLE** d'ajouter, de soustraire, de multiplier, … **deux objets**, même s'ils sont de la **même classe** : il faut définir ces opérations !

h. Une **CLASSE ABSTRAITE** (définie avec le mot-clé **abstract** juste après le mot-clé **public**) ne **S'INSTANCIE JAMAIS**

3) Les **interfaces** en Java

a. Une **INTERFACE** en Java est proche d'une classe abstraite, elle est par défaut abstraite et publique mais cela est sous-entendu : les mots-clés **abstract et public** sont sous-entendus : on ne les écrit pas

b. Une **INTERFACE** en Java est composée **UNIQUEMENT** de constantes (les données non constantes sont **INTERDITES**) et de **déclarations de méthodes** (sans le corps). *Exemple :*
   - ```
     public interface Collection {
           int MAXIMUM = 500;          //constante
           void add(Object obj);       //déclarations de méthodes
           void delete(Object obj);    //terminées par un ;
           Object find(Object obj);
           int currentCount();
     }
     ```

c. Pour qu'une classe **A implémente une interface** (c'est-à-dire hérite du comportement définit dans l'interface), il faut :
   - rajouter à la déclaration de la classe **A** le **mot-clé « `implements` »** suivi du nom de l'interface
   - **définir** dans la classe **A TOUTES** les méthodes déclarées dans l'interface. Ces méthodes **DOIVENT AVOIR EXACTEMENT la même signature** que celle de l'interface et doivent avoir un **CORPS**.

Rappels de base en Java (2)

4) Les **tests** en Java avec JUnit
   a. Dans une classe de test **T**, définie pour **tester** une classe **A** : **AUCUNE** des variables membres de **A** ne peut être appelée directement dans **T** ! *Par exemple :*

```
▪ public class A {
        private int a;
        public void meth1() {…}
        …
  }
▪ public class T extends TestCase {
        public void testMeth1(){
              a = 2;  //INTERDIT !
        }
  }
```

   b. Dans une classe de test **T**, définie pour **tester** une classe **A** : si on utilise l'instruction « `assertEquals` » dans **T**, s'assurer que la méthode « `equals` » a été redéfinie dans **A**, sinon « `assertEquals` » ne retournera `true` que si les deux objets testés correspondent à la **même référence mémoire**.