Règles Standard Java

1. Nommage :

a. langue : Le code doit être en anglais.

Les noms de classes, d'attributs et de méthodes, de même que les commentaires, doivent être en anglais.

b. longueur : Les noms ne doivent pas dépasser 20 caractères.

c. caractères utilisés : "_", chiffres et lettres non accentuées.

Les noms de classes, d'attributs et de méthodes, doivent commencer par "_" ou une lettre; et ne doivent pas contenir des lettres accentuées.

2. Fichiers de code sources :

a. les fichiers sources java doivent avoir comme nom, le nom de la classe 'public' qu'ils contiennent.

b. Le paramétrage et les configurations sont contenu dans des fichiers YAML ou properties.

c. Chaque fichier source Java ne doit contenir qu'une seule classe ou interface de type public.

3. Packages :

a. Un package est composé de mots en lettres minuscules, séparés par des points.

Il est recommandé de composer un package de concaténation de mots courts, abréviations ou acronymes

b. L'arborescence d'un package correspond strictement à l'arborescence du système de fichier à partir de la racine des sources.

4. Classes et Interfaces :

a. Les noms de classes et d'interfaces doivent commencer par une majuscule et ne contenir que des lettres.

b. Les noms de classes et d'interfaces doivent être camelCase :

C'est à dire qu'ils sont composés de telle sorte que chaque mot ou abréviation au milieu commence par une majuscule, sans espace ni ponctuation.

c. Les noms de certaines classes peuvent êtres préfixés ou post-fixées, pour mettre en évidence le type ou un design pattern particulier.

Exemple :

```
// Type
public interface IEmployee ... //Interface
public abstract class AbstractEmployee ... //Classe Abstraite
public class EmployeeNotFoundException ... //Exception
public class EmployeeImpl ... //Implementation

// Design Pattern
public class EmployeeSingleton ... //Singleton
public class EmployeeFactory ... //Factory
```

d. Supprimer les imports inutiles.

e. Supprimer toute variable membre privée, variable locale, méthode privée et constructeur privé non utilisé.

f. Toute classe doit faire partie d'un package.

g. Une classe qui n'a que des constructeurs privés doit être déclarée final.

h. Une classe qui n'a que des méthodes statiques ne doit pas avoir de constructeur public.

5. Méthodes :

a. Un nom de méthode est composé uniquement de lettres et commence par une minuscule.

b. Un nom de méthode doit être camelCase.

c. Utilisation d'un verbe comme premier mot dans le nom d'une méthode.

Généralement, une méthode est une action, c'est pour celà qu'il est recommandé d'utiliser un verbe comme premier mot dans le nom de la méthode.

d. Les accesseurs d'un attribut d'une classe ont un préfixe normalisé :

- set (affectation),
- get (récupération),
- is (récupération lorsque la valeur retournée est du type primitif boolean).

e. Affecter la visibilité la plus restrictive à une méthode.

Il est important de limiter l'utilisation des visibilités public et protected, afin de réduire la dépendance avec d'autres classes, et donc garantir un code plus flexible.

f. Les méthodes et constructeurs ne doivent pas avoir plus que un certain nombre prédéfinie de paramètres.

par exemple 5 paramètres

g. La complexité cyclomatique des méthodes et constructeurs ne doit pas dépasser une certaine seuil prédéfinie.

La complexité cyclomatique mesure le nombre de chemins conditionnels (if, else, case, catch, ...) d'une méthode et donc indique le nombre de tests unitaires à réaliser pour la valider intégralement. Cette mesure est un bon indicateur de la maintenabilité d'un code. Il est important de la conserver dans une plage raisonnable, de l'ordre de 10 par exemple.

h. Dans les classes parentes, préférer les méthodes abstraites aux méthodes sans code

i. Toute classe qui redéfinit equals() doit également redéfinir hashcode() (et vice versa)

6. Variables :

a. Un nom de variable est composé de lettres, éventuellement de chiffres et commence par une minuscule.

b. Un nom de variable doit être camelCase.

c. Il est recommandé d'utiliser la forme typée pour la déclaration des variable de type Generic :

```
// à éviter
Map catCount = new HashMap();
catCount.put("CAT A", 51);

// à préférer
Map<String,Long> catCount = new HashMap<>();
catCount.put("CAT A", 51);
```

d. Eviter l'utilisation de même nom de variables et méthode dans une même classe.

e. Déclarer une seule variable par ligne.

f. Mettre les déclarations en début de bloc.

g. Toujours affecter la visibilité la plus restrictive à un attribut.

Par défaut, un attribut doit être de visibilité private, même s'il ne s'agit pas d'une propriété de javabean. Cela permet de respecter le principe d'encapsulation. Si cet attribut doit être accessible, créer des accesseurs get() et set(), comme pour les JavaBean. Cette pratique garantit une meilleure encapsulation et donc une meilleure maintenabilité.

h. Les variables locales, dont la valeur reste inchangée, doivent être déclarées final.

i. Toutes les chaînes de caractères autres que is la chaîne vide, doivent être placées dans des constantes.

7. Constantes :

a. Les noms des constantes (static final) sont écrites en majuscules.

b. Les mots qui composent le nom d'une constante sont séparés par "_".

8. Ordre des déclarations :

a. Les déclarations dans un fichier java suivent un ordre défini:

1. /* Commentaires de description de fichier avec copyright */
2. Déclaration du package
3. Liste des imports
4. /** Commentaires Javadoc de description de la classe ou interface */
5. Déclaration de la classe ou interface
6. /* Commentaires de description de l'implémentation de la classe */
7. Déclaration des constantes (statiques finales)
8. Déclaration des variables de classe (statiques)
9. Déclaration des variables d'instance
10. Constructeurs
11. Méthodes
12. Déclaration de classes imbriquées

b. Les classes privées sont définies à la fin du fichier, après la classe publique.

c. Les déclarations de constantes et variables sont classées par visibilité décroissante (public, protected, sans mot clé, puis enfin private).

d. Les annotations des classes et méthodes doivent être placées après la JavaDoc.

9. Longueurs :

a. Un fichier ne doit pas dépasser un certain nombre de lignes prédéfini :

par exemple 2000 lignes

b. Un contructeur ou une méthode ne doit pas dépasser un certain nombre de lignes prédéfini :

par exemple 100 lignes

c. Une ligne ne doit pas dépasser un certain nombre de caractères prédéfini :

par exemple 120 caractères

d. Il est aussi recommandé de respecter au maximum les métriques suivantes :

- Un package doit contenir au maximum 75 classes.
- Une classe doit contenir au maximum 30 méthodes.
- Une classe doit contenir au maximum 4 constructeurs.
- Une classe doit être composée d'au maximum 30 attributs.
- Une interface doit exposer au maximum 20 méthodes.

Le respect de ces métriques permet de limiter la complexité de vos composants Java.

10. Expressions :

a. Utiliser les parenthèses à chaque fois qu'une expression peut prêter à confusion, ou pour mettre en évidence l'ordre de traitement des opérateurs.

b. Une instruction return ne doit pas utiliser de parenthèses, sauf si l'expression est une instruction conditionnelle.

c. On accède à une variable ou une méthode static par le nom de classe ou d'interface et jamais via une instance.

d. Limiter au maximum le nombre de return par méthode (généralement un seul return est possible).

e. Dans les comparaisons, placer les constantes à gauche.

f. Ne pas utiliser l'opérateur == pour tester l'égalité de deux chaînes de caractères. Il faut utiliser la méthode equals().

11. Gestion d'erreurs et Exceptions :

a. Utiliser les exceptions au lieu de codes de retour.

b. Réutiliser les exceptions simples définies dans le JDK.

telques NumberFormatException, IllegalArgumentException, IllegalStateException, ...etc.

c. Les exceptions fonctionnelles doivent être dérivées de java.lang.Exception.

d. Les exceptions techniques irrémédiables doivent être dérivées de java.lang.RuntimeException.

e. Les blocs catch ne doivent jamais être vides. Ils doivent a minima contenir un message de log.

f. Dans le cas de clause catch rejetant une nouvelle exception, l'exception d'origine doit être référencée.

g. Catcher toujours le type le plus fin

h. Ordonner le catch des exceptions de la plus fine à la plus générique

i. Libérer les ressources (connexion à la base de données, fichiers, sockets, ...) dans le bloc finally.

12. Performance & Gestion de la Mémoire :

a. Sortir des boucles les invariants ou l'utilisation d'une méthode.

b. Favoriser l'utilisation de static.

c. Faire attention à l'utilisation de synchronized !

d. Favoriser l'utilisation de StringBuilder & StringBuffer à celui des String pour les manipulations de chaînes (concaténations, ...)

e. Remettre à null les références aux objets volumineux comme les tableaux.

f. Limiter l'accès indexé aux tableaux

```
// à éviter
int length = rectangles.length;
for (int i=0; i< length; i++) {
    rectangles[i].setX(rectangles[i].getX() + 10);
    rectangles[i].setY(rectangles[i].getY() + 10);
}

// à préférer
int length = rectangles.length;
for (int i=0; i< length; i++) {
    Rectangle rectangle = rectangles[i];
    rectangle.setX(rectangle.getX() + 10);
    rectangle.setY(rectangle.getY() + 10);
}
```

13. Divers :

a. Tout bloc de contrôle (if, for, while, do, switch,...) doit être placé entre accolades, même si il est monoligne, ou vide.

b. Mettre une instruction par ligne.

c. Lorsqu'une instruction ne tient pas sur une ligne, faire un saut de ligne après une virgule, ou avant un opérateur.

d. Si une méthode comporte un nombre important de paramètres qui ne tiennent pas tous sur une ligne, mettre un seul paramètre par ligne.

e. Utiliser, lorsque cela est possible, les valeurs prédéfinies Boolean.TRUE et Boolean.FALSE de manière à éviter la création d'instances inutiles de la classe java.lang.Boolean.

f. Ne pas utiliser de littéraux (nombres « magiques », chaînes en dur, ...) mais des constantes nommées.

g. Ne pas utiliser de méthodes, attributs ou classes dépréciés (deprecated).

h. Ne pas utiliser System.err, System.out, System.printStackTrace.

i. Utiliser l'annotation @Override lorsque vous surchargez une méthode.

14. Bonnes Pratiques introduites en JAVA 8 :

a. Préférer les expressions Lambdas aux classes anonymes

```
// à éviter
public enum Operation {
    PLUS {
    @Override
    public double apply(double x, double y) {
      return x + y;
    }
    },
    MINUS {
    @Override
    public double apply(double x, double y) {
      return x - y;
    }
    },
    TIMES {
    @Override
    public double apply(double x, double y) {
      return x * y;
    }
    },
    DIVIDE {
    @Override
    public double apply(double x, double y) {
      return x / y;
    }
    };

    public abstract double apply(double x, double y);
}

// à préférer
public enum Operation {
  PLUS ( (x, y) -> x + y),
  MINUS ( (x, y) -> x - y),
  TIMES ( (x, y) -> x * y),
  DIVIDE ( (x, y) -> x / y);
  private Operation(DoubleBinaryOperator op) {
    this.op = op;
  }
  private final DoubleBinaryOperator op;
  public double apply(double x, double y) {
    return op.applyAsDouble(x, y);
  }
}
```

b. Favoriser les références de methodes aux expressions Lambdas

Si l'expression est l'invocation d'une méthode, comme le montre l'exemple ci-dessous :

```
// à éviter
map.merge("MaClef",  1, (count, incr) -> count + incr);

// à préférer
map.merge("MaClef", 1, Integer::sum);
```

c. Favoriser l'utilisation des Interfaces Fonctionnelles Standards

Surtout pour les six interfaces génériques suivantes :

Interface	Signature de la fonction	Exemple
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

d. Eviter au maximum l'utilisation de Optional comme variable d'instance ou paramètres de méthode

e. Ne jamais utiliser Optional comme variable locale

f. Favoriser l'utilisation de orElse() de la classe Optional qui permet de passer une valeur par défaut

Pour récuperer la valeur d'une instance de la classe Optional, Il est préférable d'utiliser orElse() qui permet de passer une valeur par défaut plutôt que get() qui lève NoSuchElementException si pas de valeur.

g. Eviter d’utiliser Optional avec une collection ou un tableau

Utiliser une collection ou un tableau vide avec isEmpty() ou length()

h. Favoriser l'utilisation de la nouvelle librairie Date & Time

Java 8 à introdui une nouvelle librairie riche et complexe pour la gestion des dates et temps (package java.time.*). il est donc préférable d'utiliser cette librairie plutôt que java.util.Calendar et java.util.Date. Avec cette nouvelle librarie, il faut bien choisir le type à utiliser selon les données temporelles requises.

i. Faire attention à l'utilisation des Stream :

- Réflichir au fonctionnel avant de les utiliser
- Attention à l’ordre des opérations intermédiaires : filter() + sorted() vs sorted() + filter() => performances
- Attention au surcoût de l'utilisation des Stream parallèles

- Il faut être concient qu'avec les parallelStream, l'ordre de parcours des éléments du Stream est aléatoire !
- Utiliser Arrays.asList plutôt que les Stream pour convertir un tableau en liste
- Utiliser la méthode forEach() sur des collections sans utiliser un Stream si pas d'opération intermédiaire à invoquer
- Eviter d'utiliser les Stream juste pour trouver le plus grand élément (resp. le plus petit élément, le count d'éléments ou l'existance d'élément) dans une collection. Utiliser plutôt la méthode max() (resp. min(), size() ou contains()) de la classe Collections ou l'interface Collection.
- Il est préférable d'utiliser un IntStream, LongStream ou DoubleStream plutôt qu''effectuer des calculs sur des wrappers de type primitif numérique pour éviter des opérations d'autoboxing, améliorer les performances, réduire le nombre d'objets créés et réduire la quantité de code nécessaire.

15. Bonnes Pratiques JAVA 11 :

a. Avec l'utilisation du mot clé var, le bon choix d'un nom significatif d'une variable locale devient encore plus nécessaire.

b. Utiliser le mot clé var lorsque l'initialisation donne suffisamment d'information sur le type

c. Utiliser le mot clé var pour découper les Stream en plusieurs parties si trop de commandes s’enchaînent dans le Stream.

d. Utiliser le mot clé var pour découper les Stream en plusieurs parties

e. Évitez d'utiliser en même temps le mot clé var et l’opérateur <> de généricité (sans spécifier le type) !

Le fait de ne plus avoir de type explicite à droite comme à gauche de l'expression d'initialisation va faire que le type Object va être utilisé!

16. Bonnes Pratiques JAVA 17 :

a. Favoriser l'utilisation des Record pour les classes finaux et immuables. (depuis java 16)

b. Favoriser l'utilisation des Pattern Matching avec le instanceof. (depuis java 16)

c. Favoriser l'utilisation du blocs de textes à la place des concaténtion pour l'initialisation d'un littéral de chaine de caractère sur plusieurs lignes. (depuis java 13)

17. Bonnes Pratiques JAVA 21 :

a. Favoriser l'utilisation des Pattern Matching pour switch

```
// à éviter
public String getTypeDescription(Object obj) {
    if (obj instanceof Integer i) {
        return "Entier de valeur " + i;
    } else if (obj instanceof String s) {
        return "Chaîne de longueur " + s.length();
```

```
    } else if (obj instanceof Double d) {
        return "Nombre décimal: " + d;
    } else {
        return "Type non reconnu: " + obj.getClass().getName();
    }
}

// à préférer
public String getTypeDescription(Object obj) {
    return switch (obj) {
        case Integer i -> "Entier de valeur " + i;
        case String s -> "Chaîne de longueur " + s.length();
        case Double d -> "Nombre décimal: " + d;
        default -> "Type non reconnu: " + obj.getClass().getName();
    };
}
```

b. Utiliser les Unnamed Patterns et Variables quand la variable n'est pas utilisée

```
// à éviter
if (obj instanceof String s) {
    return true;
}

// à préférer
if (obj instanceof String _) {
    return true;
}
```

c. Adopter les Record Patterns pour décomposer les données

```
// à éviter
if (point instanceof Point p) {
    int x = p.x();
    int y = p.y();
    // utiliser x et y
}

// à préférer
if (point instanceof Point(int x, int y)) {
    // utiliser x et y directement
}
```

d. Utiliser les Virtual Threads pour les opérations bloquantes d'E/S

```
// à éviter
ExecutorService executor = Executors.newFixedThreadPool(100);
// soumission des tâches...

// à préférer
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
// soumission des tâches...
```

e. Favoriser les Séquenced Collections pour les opérations de premier/dernier élément

```
// à éviter
List<String> list = new ArrayList<>();
// ...
String first = list.isEmpty() ? null : list.get(0);
String last = list.isEmpty() ? null : list.get(list.size() - 1);

// à préférer
SequencedCollection<String> seq = new ArrayList<>();
// ...
String first = seq.isEmpty() ? null : seq.getFirst();
String last = seq.isEmpty() ? null : seq.getLast();
```

f. Utiliser String Templates pour la construction de chaînes complexes

```
// à éviter
String query = "SELECT * FROM " + tableName + " WHERE id = " + id + " AND active =
" + active;

// à préférer
String query = STR."SELECT * FROM \{tableName} WHERE id = \{id} AND active = \
{active}";
```

g. Privilégier les méthodes de traitement de chaînes améliorées

```
// à éviter
if (str != null && !str.isEmpty() && !str.isBlank())

// à préférer
if (str != null && str.stripLeading().length() > 0)
```

h. Utiliser les Scoped Values au lieu de ThreadLocal

```
// à éviter
private static final ThreadLocal<Context> contextThreadLocal = new ThreadLocal<>
();

// à préférer
private static final ScopedValue<Context> CONTEXT = ScopedValue.newInstance();
```

i. Adopter les nouveaux APIs pour les Collections

```
// à éviter
List<String> list = new ArrayList<>();
list.add("élément1");
list.add("élément2");
list.add("élément3");

// à préférer
List<String> list = List.of("élément1", "élément2", "élément3");
```

j. Favoriser les méthodes de Stream améliorées

```
// à éviter
stream.collect(Collectors.teeing(
    Collectors.summingInt(Person::age),
    Collectors.counting(),
    (sum, count) -> (double) sum / count));

// à préférer
stream.mapToInt(Person::age).average().orElse(0);
```