

# Guide - Java

- Javadoc
  - Commentaires de types
  - Commentaires d'attributs
  - Commentaires de méthodes
- Formatage, factorisation et conventions de code
  - Formatage de code
  - Warnings
  - Utilisation du mot-clé final
  - Eviter les imports static
  - Condition avec une seule ligne
  - Utilisation en tant qu'attribut privé d'une classe
  - Masquer les implémentations
  - Déléguer ou encapsuler plutôt qu'hériter
- String
  - Concaténation de chaîne de caractères
  - String en dur
- Exceptions
  - Conserver la stacktrace
- Constructeurs
  - Classe avec plusieurs constructeurs
  - Utilisation des setter dans le constructeur
- Méthodes
  - Pré-requis avec les objets
  - Utilisation du mot-clé static
- Comparaisons
  - Comparaison avec une constante
  - Comparaison d'enum
  - Comparaison d'objets
  - Comparaison du type (instanceof)
- Collections
  - EnumMap et EnumSet
  - Tester si une collection est vide
  - For Each
  - Switch sur des enum
  - Implémenter une collection FIFO/LIFO
- Tableaux
  - À la place d'une List
  - Copie de tableaux
- JDBC
  - Pas de point-virgule à la fin d'une requête
- Conversion
  - Tableau vers liste
  - Liste vers tableau
- Tests unitaires
  - Jeu de données
- SVN
  - Commit de code commenté
  - Commentaire de commit
- Normes (checkstyle, PMD)
- Autres bonnes pratiques...

## Javadoc

La documentation du code est essentielle pour la lisibilité et la maintenabilité de ce dernier. C'est pourquoi tout le code doit être commenté au format Javadoc : `/** */`.

La documentation fait partie du développement d'une fonctionnalité au même titre que les tests unitaires.

Eclipse pré-rempli les blocs de Javadoc avec les balises à renseigner.

## Commentaires de types

Tous les types doivent être commentés.

Si un type est paramétré, le paramètre doit être commenté avec la balise `@param`.

```
/**
 * This is a Javadoc comment.
 *
 * @param <T> My parameter type.
 */
public class MyClass<T> {
}
```

## Commentaires d'attributs

Tous les attributs doivent être commentés.

```
/**
 * This is a Javadoc comment.
 */
public static final DEFAULT_STATUS = 0;

/**
 * This is a Javadoc comment.
 */
private int status;

/**
 * This is a Javadoc comment.
 */
public String message;
```

Les attributs privés peuvent ne pas être explicitement documentés si leur signification est évidente.

## Commentaires de méthodes

Toutes les méthodes doivent être commentées.

Si une méthode a un type de retour, ce retour doit être commenté avec la balise @return.

Si une méthode a un ou des paramètres, ces paramètres doivent être commentés avec la balise @param.

Si une méthode peut renvoyer une ou des exceptions, ces exceptions doivent être commentées avec la balise @throws.

```
/**
 * This is a Javadoc comment.
 *
 * @param value My first argument.
 * @param value2 My second argument.
 *
 * @return A boolean.
 *
 * @throws MyException if something is wrong.
 */
public boolean doSomething(int value, String value2) throws MyException
{
}
```

Les méthodes redéfinies avec `@Override` peuvent simplement faire référence à la documentation de leur méthode parente en redéfinissant seulement ce qu'elles modifient :

```
/**
 * {@inheritDoc}
 * more Javadoc...
 */
@Override
public void myMethod() {
}
```

## Formatage, factorisation et conventions de code

### Formatage de code

Le code doit être formaté de façon uniforme en respectant les standards de Java.

Eclipse permet le formatage rapide d'un fichier avec le raccourci : *Ctrl+Shift+F*.

Une bonne pratique peut être de créer un formateur Eclipse pour un projet, de l'exporter (format XML) et de le commiter dans les ressources du projet afin que chaque développeur puisse l'importer sur son poste lorsqu'il récupère le code la première fois.

### Warnings

Les warnings relevés par les IDE ne sont pas anodins. Ils doivent alerter l'attention du développeur. Plus généralement, un code doit être commité sans aucun warning.

Un warning peut soit :

- être corrigé (en adaptant le code).
- être ignoré avec une annotation `@SuppressWarnings` (si et seulement si cela est contrôlé).

Actions à entreprendre avec les warnings les plus courants :

- *serial*
  - Déclarer une serialid : `private static final long serialVersionUID = <random_long>` (ne pas utiliser 1L).
- *unused*
  - Supprimer l'élément non utilisé.
- *deprecation*
  - Modifier le code pour utiliser de nouvelles classes selon la préconisation de l'API.
- *rawtypes*
  - Paramétrer l'élément avec des generics.
- *unchecked*
  - Jouer avec les generics pour garantir un cast sûr.

### Utilisation du mot-clé final

Dès que possible, utilisez le mot clé final pour les attributs, les paramètres et les variables. Il y a deux avantages à cela : clarifier le code, mais surtout optimiser le code à la compilation et à l'exécution par la JVM.

Il est possible de configurer Eclipse pour qu'il ajoute le mot-clé final sur tous ces éléments lors de la sauvegarde d'un fichier : *Window > Preferences* puis *Java > Editor > Save Actions > Additional actions*.

```
public class Item {

    // Sur un attribut
    private final int value;

    public Item(final int value // Sur un paramètre) {
        // Sur une variable de bloc
        final int add = 50;
        this.value = value + add;
    }

}
```

## Eviter les imports static

Si possible, évitez les import static. Ils nuisent à la lisibilité du code et peuvent entrer en conflit avec des variables locales.

```
// à éviter
import static java.util.Math.PI;

...

final double aire = PI * rayon * rayon;
```

## Condition avec une seule ligne

Lorsque vous appliquez une condition if contenant seulement une instruction, privilégiez tout de même les accolades. Cela facilite la lecture du code (bloc défini) et permet d'ajouter rapidement une instruction dans la condition.

Préférez donc :

```
if (prix >= 0) {
    return 1;
}
```

à

```
if (prix >= 0) return 1;
```

## Utilisation en tant qu'attribut privé d'une classe

Autant que possible, utilisez des classes concrètes plutôt que des interfaces pour définir les attributs d'une classe.

Lorsque la JVM appelle une méthode sur une classe concrète, elle garde en mémoire un pointeur vers la méthode afin de l'appeler plus rapidement lors des passages suivants. Sur une interface, la JVM ne peut pas faire cela car l'implémentation de la méthode dépend de la classe réelle de l'instance. Du coup, même si vous ne changez jamais le type de la variable, la JVM va toujours faire l'indirection suivante pour appeler votre méthode : interface classe méthode.

```

public class MaClass {

    // à éviter
    private List<Integer> integers = new ArrayList<Integer>();

    // à préférer
    private ArrayList<Integer> integers = new ArrayList<Integer>();

}

```

## Masquer les implémentations

Il faut toujours masquer les implémentations concrètes pour l'extérieur et exposer des interface, voire des classes abstraites. Ceci pour pouvoir changer d'implémentation facilement sans refactoring.

```

public class Garage {

    private ArrayList<Voiture> voitures;

    ...

    // !\ à éviter :
    // ne pas exposer le type concret de la liste 'ArrayList' car il
    devient
    // difficile d'en changer car cette méthode peut être utilisée par
    d'autres
    // morceaux de code à l'extérieur
    public ArrayList<Voiture> getVoitures() {
        return voitures;
    }

    // à préférer :
    // utiliser l'interface car il devient possible de changer le type
    concret
    // de la liste sans impacter le code extérieur
    public List<Voiture> getVoitures() {
        return voitures;
    }

}

```

## Déléguer ou encapsuler plutôt qu'hériter

Dans la mesure du possible, il est préférable de déléguer ou d'encapsuler plutôt qu'hériter.

En effet, l'héritage est un choix conceptuel lourd et réfléchi. La délégation est moins lourde et plus maintenable.

Dans l'exemple suivant, l'héritage de JFrame est inutile et plus lourd.

```
public class SwingApplication extends JFrame {
    public SwingApplication() {
        super("Bad Swing");
        setVisible(true);
    }
}
```

Préférez une délégation :

```
public class SwingApplication {

    private final JFrame frame;

    public SwingApplication() {
        frame = new JFrame("Better Swing")
        frame.setVisible(true);
    }
}
```

## String

### Concaténation de chaîne de caractères

Pour la construction de chaînes de caractères avec variables, privilégiez la classe `StringBuilder`. Elle est plus performante que la concaténation par `+` et que la classe `StringBuffer` qui est synchronisée (thread-safe). Lorsqu'il y a un simple caractère à concaténer, utilisez plutôt les simples quotes : `'a'`.

Préférez donc :

```
public String toString() {
    final StringBuilder sb = new StringBuilder();
    sb.append("Titre :");
    sb.append(book.titre);
    sb.append("\nAuteur");
    sb.append(book.auteur);
    return sb.toString();
}
```

à

```
public String toString() {
    return "Titre : " + book.titre + "\nAuteur : " + book.auteur;
}
```

Depuis Java 1.6, cette optimisation n'est plus nécessaire. Le précédent morceau de code est transformé par le compilateur en :

```
public String toString() {
    return new StringBuilder("Titre : ").
        append(book.titre).
        append("\nAuteur : ").
        append(book.auteur).
        toString();
}
```

## String en dur

Aucune chaîne de caractères en dur dans le code. Toutes chaînes doivent être externalisées (fichier de propriétés, i18n, etc.).

## Exceptions

### Conserver la stacktrace

Si une exception est renvoyée dans une clause catch, alors il faut encapsuler la première exception comme cause de la seconde pour conserver la stacktrace complète dans les logs.

```
try {
    ...
} catch (MyException e) {
    throw new MyException2("blabla"); // Stacktrace perdue !
}

try {
    ...
} catch (MyException e) {
    throw new MyException2("blabla", e); // Stacktrace préservée !
}
```

## Constructeurs

### Classe avec plusieurs constructeurs

Si votre classe offre plusieurs constructeurs, essayez de centraliser la logique d'instanciation dans un seul constructeur (souvent celui avec le plus de paramètres). Cela évite la redondance de code.

Utilisez les appels à `this(...)` dans les autres constructeurs.

```
// Constructeur 1
public Book () {
    this(null); // appel au constructeur 2
}

// Constructeur 2
public Book (String titre) {
    this(titre, null, null, 0); // appel au constructeur 3
}


// Constructeur 3
public Book (String titre, String auteur, String isbn, double price) {
    this.titre = titre != null ? titre : "no title";
    this.auteur = auteur;
    this.isbn = isbn;
    this.price = price;
}
```

## Utilisation des setter dans le constructeur

Si un des setter de votre classe a une logique plus complexe qu'une simple affectation, utilisez le dans le constructeur lors de l'instanciation de l'objet. Cela évite la redondance de code.

```
// Constructeur
public Book (String titre) {
    setTitre(titre);
}

// Setter
public void setTitre(String titre) {
    if(titre != null && !titre.equals("")) {
        this.titre = titre;
    } else {
        this.titre = "Pas de titre";
    }
}
```

 Evitez d'utiliser des méthodes publiques dans un constructeur. Créez plutôt une méthode privée commune au constructeur et à setTitre ou définissez setTitre en **final**. Utiliser une méthode publique dans un constructeur peut amener des effets de bords gênants lors de l'héritage de la classe.

## Méthodes

### Pré-requis avec les objets

Lorsque que vous créez des méthodes recevant des paramètres que vous ne contrôlez pas, il faut toujours vérifier la validité de ces paramètres avant de les manipuler pour éviter les exceptions.

Essayez de centraliser ces pré-conditions en début de méthode.



```

public double division (Double d1, Double d2) {

    // L'auto-unboxing plante pour des objets null
    if (d1 == null || d2 == null) {
        throw new IllegalArgumentException("L'un des paramètres de la
division est nul.");
    }

    // Division par zéro interdite
    if (d2 == 0) {
        throw new IllegalArgumentException("La division par zéro est
interdite.");
    }

    // Division sûre à partir d'ici

    return d1 / d2;

}

```

Voici l'exemple à privilégier :

```

(type) nomMethode(params) {
    [ Pré-conditions ]
    [ Traitement ]
    [ Retour ]
}

```

## Utilisation du mot-clé static

En fonction du traitement de la méthode que vous créez, pensez à l'utilisation du mot-clé static.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Ces méthodes peuvent être utilisées sans instancier un objet de la classe. Les méthodes ainsi définies peuvent être appelées avec la notation Classe.methode() au lieu de objet.methode().

## Comparaisons

### Comparaison avec une constante

Lors d'une comparaison avec une valeur constante, pensez à utiliser la fonction equals() sur la constante plutôt que sur la variable. Cela permet d'économiser un test avec null.

```

if (CONSTANT.equals(variable)) {
    return "OK";
}

```

### Comparaison d'enum

Les Enum peuvent être comparés directement.

Préférez donc :

```
if(value == MonEnum.VALUE) {  
    return "OK";  
}
```

à

```
if(MonEnum.VALUE.equals(value)) {  
    return "OK";  
}
```

## Comparaison d'objets

Lors de l'utilisation de comparaison, utiliser toujours la méthode equals pour les objets et l'opérateur == pour les types primitifs.

Voici la bonne rédaction de la méthode equals :

```
@Override  
public boolean equals(Object obj) {  
    if(this == obj) {  
        return true;  
    }  
    if (!(obj instanceof MyClass)) {  
        return false;  
    }  
    final MyClass c = (MyClass) obj;  
    return [Condition du résultat];  
}
```

## Comparaison du type (instanceof)

Il n'est pas nécessaire de vérifier si une variable est nulle avant l'utilisation de l'instruction instanceof. null renvoie toujours false dans un test avec instanceof.

```
public static void main(String[] args) {  
    final String a = null;  
    final Object b = "b";  
    final Object c = 1;  
  
    // Affiche "false"  
    System.out.println(a instanceof String);  
  
    // Affiche "true"  
    System.out.println(b instanceof String);  
  
    // Affiche "false"  
    System.out.println(c instanceof String);  
}
```

# Collections

## EnumMap et EnumSet

EnumMap remplace avantageusement les HashMap pour les cas où la clef est de type Enum.

EnumSet est un Set spécialisé pour le stockage de valeurs provenant d'un Enum. Il est implémenté sous forme d'un flags binaires et est donc performant et peu gourmand en mémoire.

## Tester si une collection est vide

Pour tester si une collection est vide utilisez la méthode isEmpty() à la place de size() == 0.

Bien que l'implémentation par défaut de isEmpty() soit return size() == 0; certaines collections (comme ConcurrentLinkedQueue) l'implémentent de façon beaucoup plus performante.

## For Each

Pour parcourir un tableau ou une collection d'objets héritant de la classe Iterable, utilisez la notation for each :

```
for (final Book book : books) {  
    // faire quelque chose avec 'book'  
}
```

Cette notation est strictement équivalente à :

```
for (final Iterator<Book> it = books.iterator(); it.hasNext(); ){  
    final Book book = (Book) it.next();  
    // faire quelque chose avec 'book'  
}
```

⚠ Pensez à vérifier que votre liste n'est pas null sinon le "foreach" tombe en NullPointerException

## Switch sur des enum

Un enum est un objet et peut donc être null. Lors d'un switch sur une variable de type enum, il faut garantir que cette variable n'est pas null.

En effet, le code suivant plante. Une NullPointerException est levée à runtime.

```
enum MonEnum {  
    CAS1, CAS2;  
}  
  
MonEnum variable = null;  
  
switch (variable) {  
    case CAS1:  
        // quelque chose  
        break;  
    case CAS2:  
        // quelque chose  
        break;  
    default:  
        // quelque chose  
}
```

## Implémenter une collection FIFO/LIFO

Les opérations FIFO ou LIFO peuvent être réalisées à l'aide des classes implémentant les interfaces `Queue<E>` (Java 1.5) et `Deque<E>` (Java 1.6).

Pour les listes, privilégiez l'utilisation de la classe `ArrayDeque` (Java 1.6) ou `LinkedList` (Java 1.5)

Pour une map, utilisez la classe `LinkedHashMap` (Java 1.4) qui conserve l'ordre d'insertion pour les itérations.

## Tableaux

### À la place d'une List

Quand vous connaissez à l'avance la taille d'une liste pensez à utiliser un tableau à la place.

### Copie de tableaux

Quand vous devez copier un tableau, utilisez la méthode `Arrays.copyOf()` à la place d'une boucle. Cette méthode fait une copie de la zone mémoire plutôt que de copier un par un les valeurs.

## JDBC

### Pas de point-virgule à la fin d'une requête

En fonction de l'implémentation du driver JDBC que vous utilisez, l'ajout d'un ; à la fin d'une requête peut ne pas fonctionner (constaté avec le driver Oracle *ojdbc6 11.2.0.3.0*).

D'une manière générale il est préférable de ne pas ajouter de point-virgule.

```
public class BadSemiColon {
    public void execute() {
        final PreparedStatement st = new PreparedStatement("INSERT INTO t
VALUES ('a');");
        st.executeUpdate(); // Plante à cause du ; à la fin de la requête.
    }
}
```

## Conversion

### Tableau vers liste

```
ArrayList<Flag> flags = new ArrayList<>(); // Une liste...

Flag[] array = flags.toArray(new Flag[flags.size()]);
// new Flag[flags.size()] : un tableau de Flag de la taille de la liste
flags est créé.
// flags.toArray : crée un tableau à partir de la liste flags en
disposant les éléments dans le même ordre que dans la liste. Voici un
extrait de la documentation de « toArray » : Returns an array
containing all of the elements in this list in proper sequence
```

### Liste vers tableau

```
List<Flag> list = Arrays.asList(array); // Même méthode mais dans
l'autre sens. On crée une liste avec les éléments du tableau array et
la liste « list » contient cette liste. Extrait de la doc de asList :
Returns a fixed-size list backed by the specified array.

// La liste retournée étant une liste immutable, pour obtenir une liste
mutable avec cette méthode :
List<Flag> list = new ArrayList<Flag>(Arrays.asList(array));
```

## Tests unitaires

### Jeu de données

Pour créer un jeu de données de test, il est possible de procéder de 2 façons.

- Dynamiquement :
  - Créer le jeu de données en Java avant l'exécution des TU en utilisant annotation `@BeforeClass`.
  - Exécuter les TU sur ce jeu de données.
  - Supprimer le jeu de données après l'exécution des TU en utilisant annotation `@AfterClass`.
- Statiquement :
  - Insérer le jeu de données directement dans la base de données de test (avec pgAdmin par exemple).
  - Exécuter les TU pour vérifier que le jeu de données est valide.
  - Dumper ce jeu de données pour que l'exécution des TU fonctionne aussi sur un autre environnement :
    - En fonction de la base modifiée, aller dans le projet configuration correspondant.
    - Exécuter le script `maj_peupleur_tu.sh` dans le répertoire `peupleur`.
    - Commiter les scripts SQL mis à jour dans ce même répertoire.

## SVN

### Commit de code commenté

En général, il ne faut jamais commité du code commenté (dégrade la lisibilité).

Si vous souhaitez le faire pour "conserver du code qui pourra être réutilisé plus tard", alors le gestionnaire de versionning garantie les sauvegardes et un éventuel retour en arrière. Ce code doit donc être supprimé.

Le seul cas qui justifie le commit d'un code commenté est pendant le développement d'une fonctionnalité non stabilisée. Certains bouts de code peuvent donc être commentés pour que le code compile le temps que la fonctionnalité soit entièrement terminée.

### Commentaire de commit

Tous les commits doivent être accompagnés d'un commentaire respectant un formalisme uniforme.

Exemple mis en oeuvre sur le projet SGIC :

```
<matricule>/<Prénom NOM> - [<CHANTIER>] - <projets impactés> :
<message>.
// Exemple : x2012447/Tom MIETTE - [PERFS] - communs : correctif dans
l'intercepteur de chronométrage des méthodes de web services lors de
l'affichage des arguments nuls.
```

## Normes (checkstyle, PMD)

Voici une liste de règles de programmation Java utilisées chez Ixxi.

- La colonne « Nom de la règle » contient le nom de la règle dans l'outil de contrôle de code (Checkstyle ou PMD).
- La colonne « Description » donne une description et le code de la règle dans le document RATP « Normes de développement java /J2EE ».
- La colonne « Outil de contrôle » donne le nom de l'outil utilisé pour configurer la règle.

Nom de la règle	Description	Outil de contrôle
AbstractClassWithoutAbstractMethod	[JAVA-PROG-1] Une classe abstraite doit contenir au moins une méthode abstraite	Pmd
AddEmptyString	[JAVA-PERF-6] Il est interdit d'ajouter une String vide pour convertir un type à String	Pmd
AppendCharacterWithChar	[JAVA-PERF-3] Il est interdit de concaténer des caractères en tant que String avec StringBuffer.append	Pmd
ArrayIsStoredDirectly	[JAVA-PROG-25] Les constructeurs et les méthodes recevant des tableaux doivent cloner l'objet et stocker une copie	Pmd
AssignmentInOperand	[JAVA-FORM-41] Les affectations de variables à l'intérieur d'expressions sont interdites	Pmd
AvoidAccessibilityAlteration	[JAVA-PROG-29] La modification à temps d'exécution de la visibilité d'une méthode ou variable est interdite	Pmd
AvoidArrayLoops	[JAVA-PERF-7] Utiliser System.arrayCopy pour effectuer une copie de tableaux	Pmd
AvoidCallingFinalize	[JAVA-PROG-19] Il est interdit d'implémenter ou d'appeler explicitement la méthode finalize()	Pmd
AvoidCatchingNPE	[JAVA-PROG-22] Il est interdit d'intercepter 'NullPointerException' dans un bloc catch	Pmd
AvoidCatchingThrowable	[JAVA-PROG-21] Il est interdit d'intercepter 'Throwable' dans un bloc catch	Pmd
AvoidDecimalLiteralsInBigDecimalConstructor	[JAVA-PROG-9] Utiliser un String pour créer une instance de BigDecimal à partir d'un décimal	Pmd
AvoidDuplicateLiterals	[JAVA-FORM-43] Si la même chaîne de caractères est présente à plusieurs endroits du code, utiliser une constante à la place	Pmd
AvoidFieldNameMatchingMethodName	[JAVA-FORM-9] Il est interdit d'avoir un nom de champ identique à un nom de méthode	Pmd
AvoidFieldNameMatchingTypeName	[JAVA-FORM-8] Il est interdit d'avoir un nom de champ différencié du nom de la classe mère uniquement par les caractères minuscule/majuscule	Pmd
AvoidInstanceOfChecksInCatchClause	[JAVA-PROG-13] Chaque exception doit avoir son bloc catch	Pmd
AvoidInstantiatingObjectsInLoops	[JAVA-PERF-9] La création d'objets à l'intérieur d'une boucle est interdite	Pmd
AvoidMultipleUnaryOperators	[JAVA-FORM-44] Ne pas utiliser d'opérateurs unaires (agissant sur un seul argument) multiples	Pmd
AvoidPrintStackTrace	[JAVA-PROG-30] L'utilisation de printStackTrace() est interdite dans le code de production ; préférer l'appel à un logger	Pmd
AvoidReassigningParameters	[JAVA-FORM-42] Il est interdit de réaffecter les paramètres d'une méthode. Utiliser des variables temporaires	Pmd

AvoidRethrowingException	[JAVA-PROG-23] Il est interdit d'utiliser un bloc catch uniquement pour re-déclancher la même exception sans modifications	Pmd
AvoidStarImport	[JAVA-FORM-34] Le bloc import doit uniquement contenir les types utilisés par la classe du fichier	checkstyle
AvoidSynchronizedAtMethodLevel	[JAVA-PROG-34] La synchronisation (mot clé 'synchronized') ne doit pas se faire au niveau de la méthode mais du bloc de code	Pmd
AvoidThrowingRawExceptionTypes	[JAVA-PROG-15] Il est interdit de générer RuntimeException, Throwable, Exception ou Error	Pmd
ClassNamingConventions	[JAVA-FORM-4] Chaque fichier source doit contenir une et une seule classe publique ou interface, et son nom doit respecter le pattern <code>^[A-Z][a-zA-Z0-9]*\$</code>	Pmd
CloneMethodMustImplementCloneable	[JAVA-PROG-6] Toute classe redéfinissant clone() doit implémenter l'interface Cloneable	Pmd
ConstantName	[JAVA-FORM-7] Le nom des constantes doit être toujours en majuscules ; les champs sémantiques sont séparés par un underscore '_'. Pattern <code>^[A-Z]_[A-Z0-9]*\$</code>	checkstyle
ConstructorCallsOverridableMethod	[JAVA-PROG-35] Un constructeur ne doit pas appeler des méthodes surchargeables de sa classe ou de sa hiérarchie	Pmd
DeclarationOrder	[JAVA-FORM-1] Les différents éléments constituant une classe ou une interface doivent respecter l'ordre spécifié	checkstyle
DefaultComesLast	[JAVA-FORM-32] Le cas « default » doit être toujours présent et être toujours positionné en dernier dans les instructions switch	checkstyle
DoNotCallGarbageCollectionExplicitly	[JAVA-PROG-18] Il est interdit d'appeler explicitement le garbage collector	Pmd
DoubleCheckedLocking	[JAVA-PROG-20] Il est interdit d'utiliser la technique de « double check locking »	checkstyle
EmptyCatchBlock	[JAVA-PROG-11] Les blocs catch vides sont interdits	Pmd
EmptyFinallyBlock	[JAVA-PROG-12] Les blocs finally vides sont interdits	Pmd
EmptyTryBlock	[JAVA-PROG-10] Les blocs try vides sont interdits	Pmd
EqualsNull	[JAVA-PROG-5] Il est interdit d'utiliser l'instruction equals(null)	Pmd
ExceptionAsFlowControl	[JAVA-PROG-17] Il est interdit d'utiliser les exceptions comme des contrôleurs de flux	Pmd
FallThrough	[JAVA-PROG-32] Chaque case d'une commande switch doit avoir un point de sortie (break, return...). Le 'fall-through' est interdit	checkstyle
FileLength	[JAVA-FORM-38] Une classe java ne doit pas contenir plus de 1000 lignes de code	checkstyle
FileTabCharacter	[JAVA-FORM-22] L'unité d'indentation est constituée de 4 espaces, le caractère de tabulation est interdit	checkstyle

HiddenField	[JAVA-FORM-45] Il est interdit d'avoir un nom de paramètre ou de variable locale identique à un nom de champ	checkstyle
IllegalImport	[JAVA-PROG-39] Il est interdit d'importer les packages <code>sun.*</code>	checkstyle
InefficientStringBuffering	[JAVA-PERF-2] Utiliser la fonction « <code>append</code> » pour concaténer des chaînes de caractères dans un appel au constructeur de <code>StringBuffer</code>	Pmd
JavadocMethod	[JAVA-FORM-19a] Un commentaire Javadoc doit être à minima défini pour chaque méthode, champ ou constructeur déclarés <code>public</code> , <code>package</code> ou <code>protected</code>	checkstyle
JavadocStyle	[JAVA-FORM-17-20] Les tags HTML de mise en forme utilisés dans les commentaires Javadoc doivent être fermés. Les tags <code>@return</code> , <code>@param</code> sont obligatoires	checkstyle
JavadocType	[JAVA-FORM-18] Le commentaire Javadoc de classe est obligatoire	checkstyle
JavadocVariable	[JAVA-FORM-19b] Un commentaire Javadoc doit être à minima défini pour chaque variable, champ ou constructeur déclarés <code>public</code> , <code>package</code> ou <code>protected</code>	checkstyle
JumbledIncrementer	[JAVA-PROG-36] Il est interdit de mélanger les increments des index de boucle	Pmd
LeftCurly	[JAVA-FORM-23] L'utilisation des accolades doit se faire en mode SUN	checkstyle
LineLength	[JAVA-FORM-39] Une ligne de code ne doit pas contenir plus de 120 caractères	checkstyle
LocalVariableName	[JAVA-FORM-6] Le nom des variables doit commencer par une minuscule, et chaque champ sémantique à l'intérieur du nom doit commencer par majuscule	checkstyle
MethodLength	[JAVA-FORM-40] Une méthode ne doit pas contenir plus de 100 lignes	checkstyle
MethodName	[JAVA-FORM-10] Les noms de méthodes doivent contenir un verbe, et la première lettre doit obligatoirement être une minuscule	checkstyle
MethodParamPad	[JAVA-FORM-29] Il est interdit d'utiliser d'espaces entre un nom de méthode et sa parenthèse ouvrante	checkstyle
MethodReturnsInternalArray	[JAVA-PROG-26] Il est interdit d'utiliser directement un tableau interne à une classe comme valeur de retour d'une méthode	Pmd
MissingSwitchDefault	[JAVA-FORM-32] Le cas « <code>default</code> » doit être toujours présent et être toujours positionné en dernier dans les instructions <code>switch</code>	checkstyle
ModifiedControlVariable	[JAVA-PROG-31] Il est interdit de modifier les index des boucles	checkstyle
ModifierOrder	[JAVA-FORM-2] Les modificateurs doivent apparaître dans l'ordre suivant : <code>public</code> , <code>protected</code> , <code>private</code> , <code>abstract</code> , <code>static</code> , <code>final</code> , <code>transient</code> , <code>volatile</code> , <code>synchronized</code> , <code>native</code> , <code>strictfp</code>	checkstyle
MultipleVariableDeclarations	[JAVA-FORM-35] Il ne doit y avoir qu'une seule déclaration de variable par ligne	checkstyle



NeedBraces	[JAVA-FORM-31] Les instructions composées doivent être encadrées par des accolades, et ce même si le traitement n'est composé que d'une seule instruction	checkstyle
NonCaseLabelInSwitchStatement	[JAVA-FORM-33] Il est interdit d'utiliser des labels dans l'instruction switch	Pmd
NoWhitespaceAfter	[JAVA-FORM-30b] Il est interdit d'utiliser d'espaces avant et après les opérateurs unaires, tels que les opérateurs d'incrément '++' et de décrément '--'	checkstyle
NoWhitespaceBefore	[JAVA-FORM-30a] Il est interdit d'utiliser d'espaces avant et après les opérateurs unaires, tels que les opérateurs d'incrément '++' et de décrément '--'	checkstyle
OverrideBothEqualsAndHashCode	[JAVA-PROG-7] Si la méthode equals() est redéfinie, alors hashCode() doit l'être également	Pmd
ProperCloneImplementation	[JAVA-PROG-24] L'implémentation de la méthode clone() doit appeler super.clone()	Pmd
RedundantImport	[JAVA-FORM-34] Le bloc import doit uniquement contenir les types utilisés par la classe du fichier	checkstyle
ReturnEmptyArrayRatherThanNull	[JAVA-PROG-28] Les méthodes dont le type de retour est un tableau doivent retourner un tableau vide et non pas une valeur 'null'	Pmd
ReturnFromFinallyBlock	[JAVA-PROG-16] Il est interdit d'utiliser l'instruction return dans un bloc finally	Pmd
RightCurly	[JAVA-FORM-23] L'utilisation des accolades doit se faire en mode SUN	checkstyle
ShortMethodName	[JAVA-FORM-37] Un nom de méthode doit être constitué d'au moins trois caractères	Pmd
SignatureDeclareThrowsException	[JAVA-PROG-14] Il est interdit de déclarer des méthodes générant Exception, Throwable ou Error	Pmd
SimpleDateFormatNeedsLocale	[JAVA-PROG-8] L'objet « Locale » doit être fourni pour l'instanciation d'instance de SimpleDateFormat	Pmd
StringInstantiation	[JAVA-PERF-1] Ne pas utiliser « new String » pour créer une constante chaîne de caractères	Pmd
SuspiciousEqualsMethodName	[JAVA-FORM-11] Les noms de méthodes trop proche de l'API de java.lang.Object tel que hashCode() ou equals(Object) sont interdits	Pmd
SuspiciousHashCodeMethodName	[JAVA-FORM-11] Les noms de méthodes trop proche de l'API de java.lang.Object tel que hashCode() ou equals(Object) sont interdits	Pmd
UnconditionalIfStatement	[JAVA-PROG-4] Il est interdit d'utiliser des instructions « if » dont la condition retourne toujours vrai ou toujours faux	Pmd
UnnecessaryCaseChange	[JAVA-PERF-5] Pour les comparaisons non dépendantes de la casse, utiliser equalsIgnoreCase() à la place de toUpperCase().equals() ou toLowerCase().equals()	Pmd
UnnecessaryWrapperObjectCreation	[JAVA-PERF-8] Utiliser les méthodes parseInt, toString... plutôt que créer des nouveaux objets	Pmd

UnusedFormalParameter	[JAVA-PROG-38] Les paramètres de méthode non utilisés sont interdits	Pmd
UnusedImports	[JAVA-FORM-34] Le bloc import doit uniquement contenir les types utilisés par la classe du fichier	checkstyle
UnusedLocalVariable	[JAVA-PROG-2] Les variables locales et les variables d'instances private non utilisées doivent être supprimées	Pmd
UnusedPrivateMethod	[JAVA-PROG-3] Les méthodes privées non utilisées doivent être supprimées	Pmd
UseEqualsToCompareStrings	[JAVA-PROG-33] Ne pas utiliser '==' ou '!=' pour comparer deux strings : utiliser equals()	Pmd
UseIndexOfChar	[JAVA-PERF-4] Il est interdit d'utiliser des caractères comme String dans l'appel à la méthode String.indexOf	Pmd
UselessOperationOnImmutable	[JAVA-PROG-37] Il est interdit d'ignorer le résultat d'une opération sur un objet 'immutable'	Pmd
WhitespaceAfter	[JAVA-FORM-26-28] Un espace doit toujours être utilisé après chaque virgule dans une liste d'arguments et toujours suivre les conversions de type explicites (cast)	checkstyle
WhitespaceAround	[JAVA-FORM-25-27] Un espace doit être utilisé entre un mot clé et une parenthèse, opérateurs binaires, d'affectation, relationnels et logiques	checkstyle

Les fichiers de configuration des règles sont récupérables ci-dessous :

- [rules-checkstyle.xml](#)
- [rules-pmd.xml](#)
- [rules-java.xml](#)

## Autres bonnes pratiques...

- Java Practices : <http://www.javapractices.com/home/HomeAction.do>
- Normes SIT : [Normes de dev JAVA, SIT août 2012.doc](#)
- Normes RATP : [Normes de développement Java/J2EE \(RATP\)](#)
- Conventions SUN : [Convention de code Java \(Sun\)](#)