

Une courte introduction aux  
**bonnes pratiques de  
programmation**

Amélie Cordier



# Objectifs du cours

---

Apprendre à **bien coder**

Comprendre l'importance de produire un  
**code propre**

Découvrir et mettre en application quelques  
**techniques** simples pour mieux coder

# Remarques

---

Les exemples de ce cours sont donnés en Java,  
mais les conseils sont valables pour  
**tous les langages de programmation !**

Pourquoi est-il important de  
**soigner** son code ?



# Que fait le code suivant ?

---

Attention, vous avez une minute pour répondre



```
int i, j, k, m, n, o = 0;
n = 9; o=13;
int [][] p = new int [n][o];
for(i=0;i<n;i++) {
p[i][0] = i;
m = i;
for(j=1;j<o;j++) {
for(k=0;k<j-1;k++) {m = m + p[i][k];}
p[i][j] = m;}}
System.out.println("----");
for(i=0;i<o;i++) {
for(j=0;j<n;j++) {
System.out.print("|" + p[j][i]);}
System.out.println("|");}}
```

# Résultat de l'exécution du code

---

```
-----  
|0|1|2|3|4|5|6|7|8|  
|0|1|2|3|4|5|6|7|8|  
|0|2|4|6|8|10|12|14|16|  
|0|4|8|12|16|20|24|28|32|  
|0|8|16|24|32|40|48|56|64|  
|0|16|32|48|64|80|96|112|128|  
|0|32|64|96|128|160|192|224|256|  
|0|64|128|192|256|320|384|448|512|  
|0|128|256|384|512|640|768|896|1024|  
|0|256|512|768|1024|1280|1536|1792|2048|  
|0|512|1024|1536|2048|2560|3072|3584|4096|  
|0|1024|2048|3072|4096|5120|6144|7168|8192|  
|0|2048|4096|6144|8192|10240|12288|14336|16384|
```

# Le même code, comme il se doit...

---

Observez le nommage des variables et l'indentation. Quel impact cela a-t-il sur la lisibilité du code ?



```

int i, j, k = 0; // les itérateurs pour les boucles
int sommeDebutColonne = 0;
int largeurMatrice = 9;
int hauteurMatrice = 13;
int [][] matrice = new int [largeurMatrice][hauteurMatrice];

    for(i=0;i<largeurMatrice;i++) {
        matrice[i][0] = i;
        sommeDebutColonne = i;
        for(j=1;j<hauteurMatrice;j++) {
            for(k=0;k<j-1;k++) {
                sommeDebutColonne = sommeDebutColonne + matrice[i][k];
            }
            matrice[i][j] = sommeDebutColonne;
        }
    }

System.out.println("Affichage de la matrice");
for(i=0;i<hauteurMatrice;i++) {
    for(j=0;j<largeurMatrice;j++) {
        System.out.print("|" + matrice[j][i]);
    }
    System.out.println("|");
}
}

```

# Qu'est-ce que du code « propre » ?

---

## Agréable à lire

Lorsque l'on programme, on passe plus de temps à lire du code qu'à l'écrire...

## Facile à comprendre et à réutiliser

... non seulement par celui qui a écrit le code mais aussi par les autres

## Logique

Le code doit être simple et efficace

## Explicite

Il ne faut pas cacher vos intentions... si vous cachez des choses dans le code, les bugs en profiteront pour se cacher aussi

## Soigné et robuste au temps qui passe

Pensez à entretenir votre code (supprimer ce qui est obsolète, documenter les ajouts, etc.)

# Exemple d'intention cachée

---

Dans l'exemple précédent, quelles sont les intentions cachées par le programmeur dans les lignes suivantes :

```
matrice[i][0] = i;  
sommeDebutColonne = i;
```

# Oui mais...

## ... coder proprement prend du temps !

---

### Non, ce n'est pas vrai !

On perd plus de temps à corriger du code  
« brouillon » qu'à écrire du **code propre**

Ne pas confondre **vitesse** et **précipitation**

Ne pas remettre à plus tard le soin que l'on apporte  
au code : adopter des **bonnes pratiques** dès le  
début !

Bien choisir les noms des choses

# Bonnes pratiques pour le choix des noms

---

## Des noms significatifs pour soi et pour les autres

Exemple	Contre-Exemple	Justification
<code>adresseClient</code>	<code>a</code>	Non significatif
<code>nomOrdinateur</code>	<code>wally</code>	Significatif pour le programmeur
<code>largeurFenetre</code>	<code>ww</code>	Significatif, si l'on se souvient que <code>ww</code> signifie "windowWidth"... C'est donc à bannir

## Ne pas donner de fausse information

Exemple	Justification
<code>int Matrice = 8</code>	<b>Matrice</b> est un entier, le nom est donc mal choisi et donne une fausse information
<code>adresse = « Paris »</code>	On s'attend à ce que la variable <b>adresse</b> contienne toutes les informations relative à l'adresse, ce qui n'est pas le cas

# Bonnes pratiques pour le choix des noms

---

## Éviter les noms qui se ressemblent trop

Exemple	Justification
<code>maxEtuWithAccount</code> <code>maxEtuWithNoAccount</code>	Imaginez ces variables au milieu d'un code complexe...

## Donner des noms qui ont du sens !

Exemple	Justification
<code>int nom = 9;</code>	On s'attend à ce que <b>nom</b> soit une chaîne de caractères
<code>String plop;</code>	<b>plop</b> n'est pas un nom de variable « compréhensible »
<code>int a, b = 7;</code>	Si <b>a</b> et <b>b</b> définissent respectivement la hauteur et la largeur d'une matrice, il est préférable d'utiliser <b>hauteur</b> et <b>largeur</b> au lieu de <b>a</b> et <b>b</b> .

# Quelques conseils...

---

## Ne trichez pas !

N'utilisez pas **klass** sous prétexte que **class** est un mot réservé

## N'ajoutez pas de complexité inutile

Si **personne** est un objet, inutile de le nommer **personneObject**...

## Utilisez des noms prononçables pour faciliter la lecture et l'écriture du code

**rtc56xzc5** est un joli nom de variable, mais impossible à mémoriser et à écrire

## Ajoutez du contexte si nécessaire

**addrRue**, **addrVille**, **addrEtat** (ce qui évite «**etat** » qui est ambigu)



# ... et quelques remarques

---

## La longueur d'un nom n'est pas un problème

Pensez que vous avez des outils d'auto-complétion

## Pensez aux outils de recherche

Les variables `a` ou `i` ne peuvent pas facilement être trouvées, tandis que `indiceColonne` peut l'être

## Vous pouvez préfixer les variables par leur type

...mais grâce aux fonctionnalités offertes par les IDE, ce n'est plus forcément nécessaire

## Attention aux caractères pièges !

0 et 1 et o et l par exemple...

# Nommer les classes et les méthodes

---

**Nom classe** : noms

**Nom méthodes** : verbes

Les **accesseurs**, les **mutateurs** et les **prédicats** doivent être nommés en fonction de l'objet qu'il manipulent et préfixés par **get**, **set** ou **is**

**Attention** : toujours utiliser les mêmes mots pour nommer les même concepts

Exemple : ne pas mélanger **fetch**, **retrieve** et **get**

# Nommer les choses : résumé

---

Pensez **équipe**

Utiliser des noms que vous **partagez** avec d'autres programmeurs

N'hésitez pas à **renommer** pour faire mieux

N'oubliez pas tout ce que peuvent vous apporter les **outils modernes**

Posez vous la question « **français ou anglais ?** »

# Organiser les méthodes et les fonctions

# Rôle d'une fonction

---

Une fonction doit faire **une chose et une seule** !

Une fonction est réussie si, lorsqu'on la lit, on y trouve exactement ce que l'on attendait en voyant son nom

Une fonction peut :

- Soit **modifier** l'état d'un objet
- Soit **retourner** des informations sur un objet

# Organisation logique

---

**Dans une fonction, on doit s'efforcer de travailler à un niveau d'abstraction constant**

Exemple : ne pas mélanger le traitement des fichiers et celui des caractères

**Il ne faut pas cacher des choses dans une fonction**

Exemple : si une fonction s'appelle testMotDePasse mais qu'en pratique, elle modifie le mot de passe testé, cela pose problème

**Il faut éviter la redondance**

Si un même « morceau de code » est utilisé à plusieurs endroits, il est préférable de le placer dans une fonction, ce qui facilite notamment sa maintenance

# A propos du niveau d'abstraction

---

Afin de trouver le bon niveau d'abstraction,  
n'hésitez pas à faire des brouillons !

# Longueur d'une fonction

---

Faire court

Plus c'est court et plus c'est facile à tester

Si vos méthodes / fonctions sont trop longues,  
n'hésitez pas à les décomposer



# Présentation

---

**Blocs et indentation** : éviter de trop imbriquer les blocs, préférez le découpage

**Ordonner les méthodes dans le code** : elles doivent apparaître du niveau d'abstraction le plus élevé au niveau d'abstraction le moins élevé

# Les arguments d'une fonction

---

Limiter le nombre d'arguments d'une fonction : trop d'arguments rendent la manipulation et les tests complexes

Veiller à ce que tous les arguments soient au même niveau d'abstraction

Attention à l'ordre des arguments : il doit être logique

N'hésitez pas à passer des objets en argument :

```
Circle makeCircle(double x, double y, double radius);  
Circle makeCircle(Point center, double radius);
```

# Présentation des classes

---

## Appliquer les règles de mise en forme admises par l'équipe de développement

Ne pas hésiter à utiliser des outils pour cela

## Découper votre code en différentes classes, sans pour autant exagérer

Une méthode par classe n'a aucun sens...

## Espacer le code verticalement

Les méthodes doivent se détacher visuellement les unes des autres

**Les commentaires** ne doivent pas « polluer » le code et compliquer la lecture

**Rassembler** les choses qui doivent être rassemblées

## Ne pas succomber à la tentation de « simplifier les choses »

Par exemple, ne pas raccourcir l'écriture d'un bloc if, même si « c'est permis »

# Apprendre à lever des exceptions

# La gestion des exceptions

---

**La gestion des exceptions ne doit pas interférer  
avec le code métier**

Il faut donc :

- **Intercepter** les exceptions dans le code
- **Traiter** les exceptions dans une partie séparée

# Exemple : lever des exceptions

---

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

# Les commentaires

# Généralités sur les commentaires

---

Les commentaires sont **essentiels**

Il ne faut pas pour autant en **abuser**

Les commentaires doivent être **utiles**

Ils ne doivent pas **pallier** un **manque de clarté du code**



# Un bon commentaire ne fait pas tout

---

Si vous avez codé proprement, il devrait suffire de lire le code pour le comprendre !

Le commentaire ne doit pas « clarifier » le code... il doit **faciliter sa compréhension** en apportant des informations complémentaires

# Propriétés des bons commentaires

---

Ils sont bien structurés

Ils sont à jour

Ils sont efficaces

Ils n'expliquent pas l'algorithme, mais apportent des informations complémentaires

# Quels sont les commentaires utiles ?

---

Commentaires légaux (licences)

Commentaire donnant un exemple

Justification d'un choix de conception

Aide à la lecture (pour aller plus vite)

Todo

# Ce qu'il ne faut pas faire

---

## Trois lignes de commentaires pour une ligne de code

### Commentaires par obligation

```
// Additionne a et b et stocke le résultat dans c  
c =a+b
```

### Historique des révisions d'un fichier

Nous avons les outils de gestion de version pour cela

### Commentaire pour signaler une accolade de fin

Si vous avez besoin de ce type de commentaire, c'est que votre code est déjà trop long

### Commentaire obsolète qui n'a jamais été supprimé

# Ce que l'on verra plus tard

---

Apprendre à mieux gérer les erreurs

Faire des tests unitaires

Utiliser des design patterns

Appliquer les règles de mise en forme

Bien utiliser les outils d'aide à la programmation

Gérer des gros projets de développement

...

# Source

---

La plupart des idées de ce cours proviennent de l'ouvrage suivant :

*Coder proprement*. Robert C. Martin. Pearson Education France, 2009 - 457 pages