ENSICAEN
ÉCOLE NATIONALE SUPÉRIEURE D'INGÉNIEURS DE CAEN
& CENTRE DE RECHERCHE

Code Propre en Java
Quelques règles de codage

1. Nommer explicitement les identificateurs

Mauvaise pratique

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : _theList) {
    if (x[0] == 4) {
      list1.add(x);
    }
  }
  return list1;
}
```

Bonne pratique

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : _gameBoard) {
    if (cell.isFlagged()) {
      flaggedCells.add(cell);
    }
  }
  return flaggedCells;
}
```

2. Éviter les caractères ambigus dans les noms

Mauvaise pratique

```
int a = l;
if (O == l) {
  a = 1;
} else {
  l = 0;
}
```

3. Utiliser des noms prononçables, mnémotechniques et partageables

Mauvaise pratique

```
public final class DtaRcrd102 {
  private Date _genymdhms;
  private Date _modymdhms;
  private final String _pszqint = "102";
  /* ... */
}
```

Bonne pratique

```
public final class Customer {
```

```
  private Date _generationTimestamp;
  private Date _modificationTimestamp;
  private final String recordId = "102";
  /* ... */
}
```

4. Utiliser des noms recherchables

Mauvaise pratique

Ici 't' ne peut pas facilement être recherché dans le texte.

```
for (int j = 0; j < 34; j++) {
  s += (t[j] * 4) / 5;
}
```

Bonne pratique

```
int _realDaysPerIdealDay = 4;
final int WORK_DAYS_PER_WEEK = 5;
int _sum = 0;

for (int j = 0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

5. Éviter les noms difficiles à différencier

Mauvaise pratique

```
XYZControllerForEfficientHandlingOfStrings
XYZControllerForEfficientStorageOfStrings
```

6. Éviter les noms avec encodage

Mauvaise pratique

La notation hongroise a été introduite lorsque les compilateurs ne faisaient pas de vérification d'équivalence entre les types. Aujourd'hui, elle rend le code illisible et non adapté à la maintenance.

```
unsigned int UIGetData() {
      …
}
```

Bonne pratique

```
unsigned int getData() {
      …
}
```

Mauvaise pratique

Les IDE n'aident pas à faire évoluer le nom de la variable avec la maintenance du code. Ici, le type de la variable `phoneNumber` est passé de `String` à `int[]` sans que le nom de la variable ait changé.

```
int[10] SPhoneNumber;
```

7. Nommer les attributs en respectant une convention

Les noms des attributs de classes doivent être privés et préfixés par le caractère '_'. Ceci permet de distinguer immédiatement un paramètre d'un attribut par son nom, d'éviter les problèmes de masquage de paramètres et d'éliminer les lourdeurs introduites par l'utilisation du mot clé `this` (`this.attribut`). La première lettre doit être une minuscule.

Mauvaise pratique

```
private String nom;
private int volume;
```

Bonne pratique

```
private String _nom;
private int _volume;
```

8. Nommer les méthodes en respectant une convention

Les méthodes sont nommées avec un verbe ou une phrase intentionnelle (`postPayment()`, `deletePage()`, ou `save()`). Le nom doit commencer par une lettre en minuscule. Utilisez les standards `get`, `set`, et `is` pour les mutateurs et accesseurs.

Bonne pratique

```
Artist artist = song.getArtist();
if (artist.isEmpty()) {
  song.setTag("Unknown artist");
}
```

9. Nommer les paquets en respectant une convention

On utilise l'adresse Web de l'équipe de développement à l'envers. Puis, on y ajoute les noms de paquets du projet.

Bonne pratique

```
org.eclipse.swt.graphics
fr.ensicaen.ecole.projet.model
fr.ensicaen.ecole.projet.view
```

10. Indenter le code

Indentation à la Unix

```
int unix( int p ) {
  if (test) {
  } else {
  }
}
```

avec sa variante pour la déclaration des méthodes (p. ex. projet Linux) :

```
int unix( int p )
{
  if (test) {
  } else {
  }
}
```

11. Bannir l'appel des méthodes et de attributs avec le mot clé 'this'

Cette pratique introduit une lourdeur dans l'écriture.

Mauvaise pratique

```
this.x1 = (-this.b + sqrt(this.b * this.b - 4 * this.a * this.c))
          / 2 *this.a;
```

Bonne pratique

```
_x1 = (-_b + sqrt(_b*_b - 4*_a*_c)) / 2*a;
```

12. Éliminer les instructions naïves

Mauvaise pratique

```
if (boolean1 == true) {
} else if (boolean2 == false) {
}
```

Bonne pratique

```
if (boolean1) {
} else if (!boolean2) {
}
```

13. Encadrer les instructions imbriquées par des accolades même s'il n'y a qu'une ligne

Mauvaise pratique

```
for (int i = 0; i < 10; i++)
  tab[i] = i;
```

Bonne pratique

```
for (int i = 0; i < 10; i++) {
  tab[i] = i;
}
```

14. Remplacer les commentaires par du code

Utilisez des variables ou des méthodes explicatives plutôt qu'un commentaire.

Mauvaise pratique

```
// does the module from the global list <module> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(module.getSubSystem())) {}
```

Bonne pratique

Le commentaire est donné par le nom de la variable :

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = module.getSubSystem();
if (moduleDependees.contains(ourSubSystem)) { }
```

Mauvaise pratique

```
// Check to see if the employee is eligible for full benefits
if ((_employee.flags & HOURLY_FLAG) && (_employee.age > 65)) {
```

```
  /* ... */
}
```

Bonne pratique

Le commentaire est donné par le nom de la méthode :

```
if (isEligibleForFullBenefits()) {
  /* ... */
}
final boolean isEligibleForFullBenefits() {
  return (_employee.flags & HOURLY_FLAG) && (_employee.age > 65)
}
```

15. Remplacer les conditions complexes par une variable explicative

Vous avez une expression de condition compliquée.

Mauvaise pratique

```
if (platform.indexOf("MAC") > -1
    && browser.indexOf("IE") > -1
    && wasInitialized() && resize > 0) {
  // do something
}
```

Bonne pratique

```
boolean isMacOs = platform.indexOf("MAC") > -1;
final boolean isIEBrowser = browser.indexOf("IE") > -1;
final boolean wasResized = resize > 0;
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
  // do something
}
```

16. Ne pas faire d'optimisation dans l'écriture du corps d'une méthode que le compilateur peut faire

Mauvaise pratique

```
perimeter = 6.28 * radius; // 2.PI.R
```

Bonne pratique

```
perimeter = 2 * PI * radius;
```

Mauvaise pratique

```
int x = y << 1; // x = y * 2
```

Bonne pratique

```
int x = y * 2;
```

17. Utiliser une ligne par déclaration de variable

Mauvaise pratique

```
int a, b;
```

Bonne pratique

```
int a;
int b;
```

18. Remplacer un nombre magique par une constante symbolique

Mauvaise pratique

```
double potentialEnergy( double mass, double height ) {
  return mass * 9.81 * height;
}
```

Bonne pratique

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
double potentialEnergy( double mass, double height ) {
  return mass * GRAVITATIONAL_CONSTANT * height;
}
```

19. Supprimer les doubles négations

Mauvaise pratique

```
if (!item.isNotFound()) { }
```

Bonne pratique

```
if (item.isFound()) { }
```