Programmation 2 : deuxième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

17 ou 19 septembre 2018

Rappels sur les classes et les instances

Classe

Une **classe** (d'objet) définit des :

- **constructeurs** : des façons de construire/instancier les objets (**instances** de la classe)
- **attributs** (champs, propriétés ou données membres) : la structure des objets de la classe
- **méthodes** : le comportement des objets de la classe

Syntaxe de définition d'une classe

```
public NameOfTheClass{
    public final Type1 attribute1;
    private Type2 attribute2;

    public NameOfTheClass(...){
        // constructor code
    }

    public getAttribute2(){
        return this.attribute2;
    }
}
```

`public`/`private` : accessibilité en dehors de la classe

`final` : changement de valeur impossible après construction

Instances

- Chaque instance possède son propre état et donc ces propres valeurs de attributs.
- On peut copier les références vers des instances
- Une référence contient soit `null` soit la référence d'une instance compatible avec le type de la variable.

```
Counter counter1 = new Counter(12);
Counter counter2 = new Counter(24);
Counter counter3 = counter1;
Counter counter4 = null;

System.out.println(counter1.position); // 12
System.out.println(counter2.position); // 24
System.out.println(counter3.position); // 12
System.out.println(counter4.position);
// NullPointerException
```

Bonnes pratiques de programmation

Qu'est-ce que programmer proprement ?

Un programme propre :

- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous

En résumé

Un programme informatique est de bonne qualité s'il est facile pour un développeur externe de rajouter un nouvelle fonctionnalité.

Clarity is king

Un développeur professionnel se force à écrire du code compréhensible par le plus grand nombre.

Pourquoi programmer proprement ?

- Pour programmer les fonctionnalités les unes après les autres
- Pour ajouter des fonctionnalités à moindre coût
- Pour que les programmes soient utilisables plus longtemps
- Pour valoriser les codes écrits par les développeurs (car réutilisables)
- Pour effectuer des tests à toutes les étapes du développement
- Pour que les développeurs soient heureux de travailler

Une méthodologie pour bien nommer

Pourquoi bien nommer est important

Albert Camus (1944)

“Mal nommer un objet, c’est ajouter au malheur de ce monde”

Que veut dire le texte suivant ?

La L3 info : MIAGE ne dépend pas de la même UFR que la L2 info : MI, elle dépend de la FEG et non de la FS. Pour faire vos IA et IP, vous devez donc contacter la scol de Forbin et non celle de SCH.

Nommage (1/2)

Important

Les noms donnés aux variables/méthodes/attributs/classes sont essentiels pour la lisibilité du code.
Un code bien écrit passe d'abord par des noms bien choisis.

Exemple de code mal écrit

Êtes-vous capable de dire rapidement ce que fait cette classe ?

```
class Cream {
    private int donut = 0;
    public void p(){ donut++; }
    public int v(){ return donut; }
}
```

Nommage (2/2)

Le même programme avec un meilleur nommage des variables :

```
class Counter {
    private int count = 0;
    public void increment(){ count++; }
    public int value(){ return count; }
}
```

Important

Des noms donnés au hasard et en dépit des conventions rendent le code illisible en cachant l'intention du code.

Exemple de mauvais nommage

Autre exemple de programme mal écrit :

```
List<Rectangle> get(List<Rectangle> list){
    List<Rectangle> res = new ArrayList<Rectangle>();
    for(Rectangle r : list)
        if (r.w == r.h) res.add(r);
    return res;
}

class Rectangle{
    public int w, h;
}
```

Code corrigé

Programme légèrement refactoré :

```
List<Rectangle> findSquares(List<Rectangle> rectangles){
    List<Rectangle> squares = new ArrayList<Rectangle>();
    for(Rectangle rectangle : rectangles)
        if (rectangle.isSquare()) squares.add(rectangle);
    return squares;
}

class Rectangle{
    public int width, height;
    /* ... */
    boolean isSquare(){ return width == height; }
}
```

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage

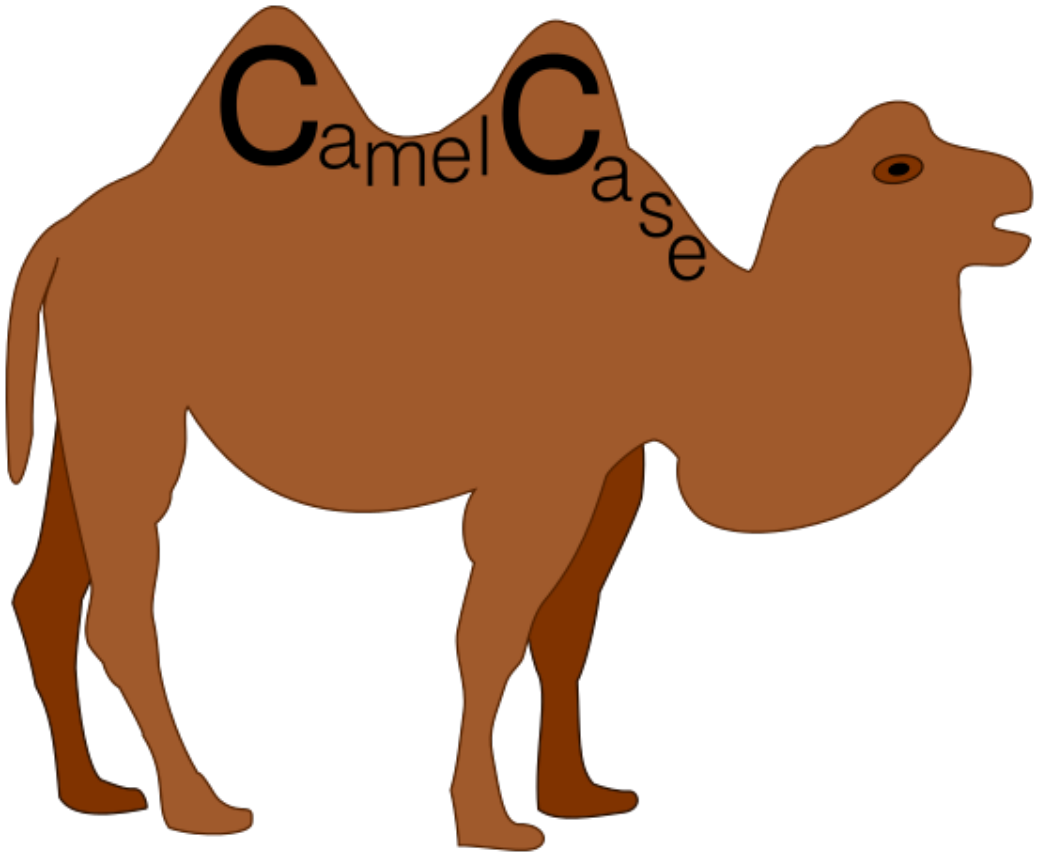
En anglais

- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards).
  $\Rightarrow$ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

En respectant les conventions

La plupart des langages (Java inclus) ont des conventions pour l'écriture des noms et la manière de coder les espaces.

Convention de nommage Java (1/2)



Manière d'écrire en Java

Utilisation du **Camel Case** (casse de chameau) dans la plupart des cas.

Convention de nommage Java (2/2)

- **UpperCamelCase** (commençant par une majuscule) : pour les noms de classes

  Exemples : `Collection`, `ArrayList`, `LinkedBlockingQueue`, `BeanContextServicesSupport`, ...

- **lowerCamelCase** (commençant par une minuscule) : pour les méthodes, variables, attributs

  Exemples : `add`, `length`, `addAll`, `modCount`, `lastIndexOf`, ...

- **ALL_CAPS** : pour les constantes

  Exemples : `PI`, `HALF_UP`, `DAYS_IN_WEEK`, ...

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
$\Rightarrow$ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet ⇒ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashcode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Nommage des méthodes

Méthodes de conversion $\Rightarrow$ utilisation du to

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les partie du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confier à d'autres méthodes

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

L'animal vertébré vivipare caractérisé par la présence de mamelles, qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives, et qui se multiplie avec une rapidité d'intensité forte, aux organes d'audition et d'équilibration à l'étendue supérieure à la moyenne dans le sens de la longueur, fait descendre par le gosier, postérieurement à une réduction en petites parcelles avec les dents, un être vivant appartenant au règne végétal cultivé pour l'usage culinaire, muni de pédicelles partant en faisceau depuis son pédoncule, et dont la partie souterraine permettant la fixation au sol est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

L'animal vertébré vivipare caractérisé par la présence de mamelles,
qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
rouge.

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le mammifère lagomorphe très prolifique aux longues oreilles avale après mâchage une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le mammifère lagomorphe très prolifique aux longues oreilles
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
mange
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
mange
une carotte.

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le lapin mange une carotte.

Ne pas mentir

```
class User {
    private boolean authenticated;
    private String password;

    public boolean checkPassword(String password) {
        if (password.equals(this.password)) {
            authenticated = true;
            return true;
        }
        return false;
    }
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom.

Des principes pour bien programmer

Principes de programmation

Il existe de nombreux principes de programmation permettant de guider l'écriture de code :

- DOT : Do One Thing
- DRY : Don't Repeat Yourself
- KISS : Keep It Simple Stupid
- SRP : Single Responsibility Principle
- ...

Autres principes

Les 5 principes SOLID (principes de conception objet) que vous verrez en détail en L3 dans le cours de **Programmation et conception**

Do One Thing (1/2)

Curly's Law/Do One Thing

Une variable/méthode/classe doit n'avoir qu'une seule signification.

Do One Thing (2/2)

Une variable/méthode/classe **ne** doit **pas** avoir :

- un sens dans une circonstance et autre sens dans un domaine différent,
- ou bien avoir deux sens en même temps.

Le principe **Do One thing** est fortement lié aux principes suivants :

- **Don't Repeat Yourself :** il ne faut pas se répéter car la répétition produit de l'inconsistance et des erreurs.
- **Once and Only Once :** chaque comportement ne doit être défini qu'une fois.

Do one Thing pour les méthodes

Une méthode ne doit avoir qu'une **responsabilité = raison de changer**.

Comment déceler une méthode ne respectant pas DOT

- Une méthode ayant un "and" dans le nom ⇒ `doOneThingAndAnotherThing`
- Une méthode peut réalisant une action complexe sans appeler d'autres méthodes.
- Une méthode trop longue (plusieurs dizaines de lignes de code)
- un niveau d'indentation élevé : par exemple un `if` dans un `for` dans un `for`

Exemple de méthode ayant trop de responsabilités

```
int countAndPrintSquares(List<Rectangle> rectangles) {
  int countSquare = 0;
  for (Rectangle rectangle : rectangles) {
    if (rectangle.width == rectangle.height) {
      System.out.println(rectangle);
      countSquare++;
    }
  }
  return countSquare;
}
```

La méthode doit :

- tester si un rectangle est un carré
- afficher les carrés
- compter les carrés

Chaque méthode a une responsabilité

```
boolean isSquare(){ return this.width == this.height; }
static int printSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) {
            System.out.println(rectangle);
        }
    }
}
static void countSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) squareCount++;
    }
    return squareCount;
}
```

Autre exemple

```
boolean removeFirstMinimumValue(List<Integer> list) {
    if (list.isEmpty()) { return false; }
    int position = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list.get(position) > list.get(i)) {
            position = i;
        }
    }
    list.remove(position);
    return true;
}
```

Correction (1/2)

```
int findMinimumValuePosition(List<Integer> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException();
    }
    int position = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list.get(position) > list.get(i)) {
            position = i;
        }
    }
    return position;
}
```

Correction (2/2)

```
boolean removeFirstMinimumValue(List<Integer> list) {
    if (list.isEmpty()) {
        return false;
    }
    int position = findMinimumValuePosition(list);
    list.remove(position);
    return true;
}
```

Single Responsability Principle (SRP)

Un des 5 principes SOLID : Do One Thing pour les classes

Principe SRP

Une classe ne doit avoir qu'une **responsabilité = raison de changer**

Les effets néfastes des responsabilités multiples

- Difficulté à nommer car la classe est trop complexe
- Difficulté à réutiliser le code car seulement une des responsabilités est réutilisable
- Difficulté à mettre à jour le code car changer l'implémentation d'une partie impacte les autres parties

Single Responsability Principle

Pourquoi SRP ?

- Facilite le nommage et documentation
- Facilite l'écriture des tests
- Facilite la réutilisation des classes

Comment détecter une violation de SRP

- La classe a beaucoup de méthodes
- Le code de la classe est long

Exemple de violation de SRP

```
public interface Modem
{
  public void dial(string phoneNumber);
  public void hangUp();
  public void send(char c);
  public char receive();
}
```

Deux responsabilités :

- gérer la connexion avec `dial` et `hangUp`.
- gérer la communication de données avec `send` et `receive`.

Don't Repeat Yourself

Andy Hunt et Dave Thomas

Dans un système, toute connaissance doit avoir une représentation unique, non-ambiguë, faisant autorité.

WET le contraire de DRY

- Write Everything Twice
- We Enjoy Typing
- Waste Everyone's Time

DRY : le pourquoi et le comment

Objectif

Éviter la redondance de code.

Pourquoi ?

- Cela facilite la maintenance d'un programme (un changement se fait en un seul endroit).
- **La duplication est source d'erreur :** si les mêmes lignes apparaissent de multiples fois, la probabilité qu'au moins une occurrence soit fausse est augmentée.

Comment ?

- En s'interdisant le copier/coller en programmant.
- En respectant DOT et SRP

Keep It Simple, Stupid

La simplicité avant tout

La complexité rend le code illisible.

Comment rester simple

- Essayer de toujours choisir la solution à un problème la plus simple
- Éviter les méthodes/classes trop longues
- Ne pas optimiser si ce n'est pas nécessaire et que cela crée de la complexité

Conseils et pièges à éviter pour bien programmer

Limiter le plus possible le nombre d'arguments des méthodes

```
void drawCircle(int x, int y, int radius){
    /* ... */
}
```

On peut regrouper les deux arguments `x` et `y` en seul concept : un `Point`.

```
void drawCircle(Point center, int radius){
    /* ... */
}
```

Éviter les drapeaux en tant qu'arguments

Code mal écrit

Code avec drapeau = **mauvais**

```
List<Integer> copy(List<Integer> list, boolean onlyEven){
    List<Integer> result = new ArrayList<Integer>();
    for (int i : list)
        if (!onlyEven || i%2 == 0)
            result.add(i);
    return result;
}
```

Éviter les drapeaux en tant qu'arguments

Il est préférable d'écrire deux méthodes :

```
List<Integer> copy(List<Integer> list){
    List<Integer> result = new ArrayList<Integer>();
    for (int element : list)
        result.add(element);
    return result;
}

List<Integer> copyOnlyEven(List<Integer> list){
    List<Integer> result = new ArrayList<Integer>();
    for (int element : list)
        if (element%2 == 0)
            result.add(element);
    return result;
}
```

Bien écrire les boucles

Code mal écrit : action et condition mélangées

```
boolean addToList(String[] source, int index,
                  List<String> destination) {
    if (index < 0 || index >= source.length)
        return false;
    destination.add(source[index]);
    return true;
}
public void copy(String[] source,
                 List<String> destination) {
    int index = 0;
    while (addToList(source, index, destination))
        index++;
}
```

Séparer condition et action de la boucle

Il faut essayer de ne pas mélanger condition de boucle et action.

L'action de la boucle :

```
void addToList(String[] source, int index,
               List<String> destination) {
    destination.add(source[index]);
}
```

Le test de la boucle :

```
boolean indexIsValid(String[] source, int index) {
    return index >= 0 && index < source.length;
}
```

Séparer condition et action de la boucle

```
public void copy(String[] source,
                 List<String> destination) {
    for (int index = 0;
         indexIsValid(source, index);
         index++)
        addToList(source, index, destination);
}
```

Évidemment dans ce cas, on peut écrire directement :

```
public void copy(String[] source,
                 List<String> destination) {
    for (int index = 0; index < source.length; index++)
        destination.add(source[index]);
```

Séparer condition et action de la boucle

Code mal écrit : action et condition mélangées

```
boolean contains(int[] array, int element) {
    boolean found = false;
    for (int i = 0; !found && i < array.length; i++)
        if (array[i] == element) found = true;
    return found;
}
```

Séparer condition et action de la boucle

La variable `found` ne fait pas partie de la condition de boucle donc :

```
boolean contains(int[] array, int element) {
    for (int i = 0; i < array.length; i++)
        if (array[i] == element) return true;
    return false;
}
```

Remarques

- Avoir plusieurs retours dans le code d'une méthode n'est pas un problème
- Trop de variables dans un programme rend le code difficile à lire (charge cognitive)
- Il faut sortir de la méthode dès que possible (Early exit)

Séparer condition et action de la boucle

Code mal écrit : action et condition mélangées

```
void copyToStop(Node head, List<String> destination) {
    Node node = head;
    boolean ok = false;
    while (node != null && !ok) {
        String element = node.element;
        if (!element.equals("stop"))
            destination.add(element);
        else
            ok = true;
        node = node.next;
    }
}
```

Séparer condition et action de la boucle

Utilisation du break pour simplifier le code

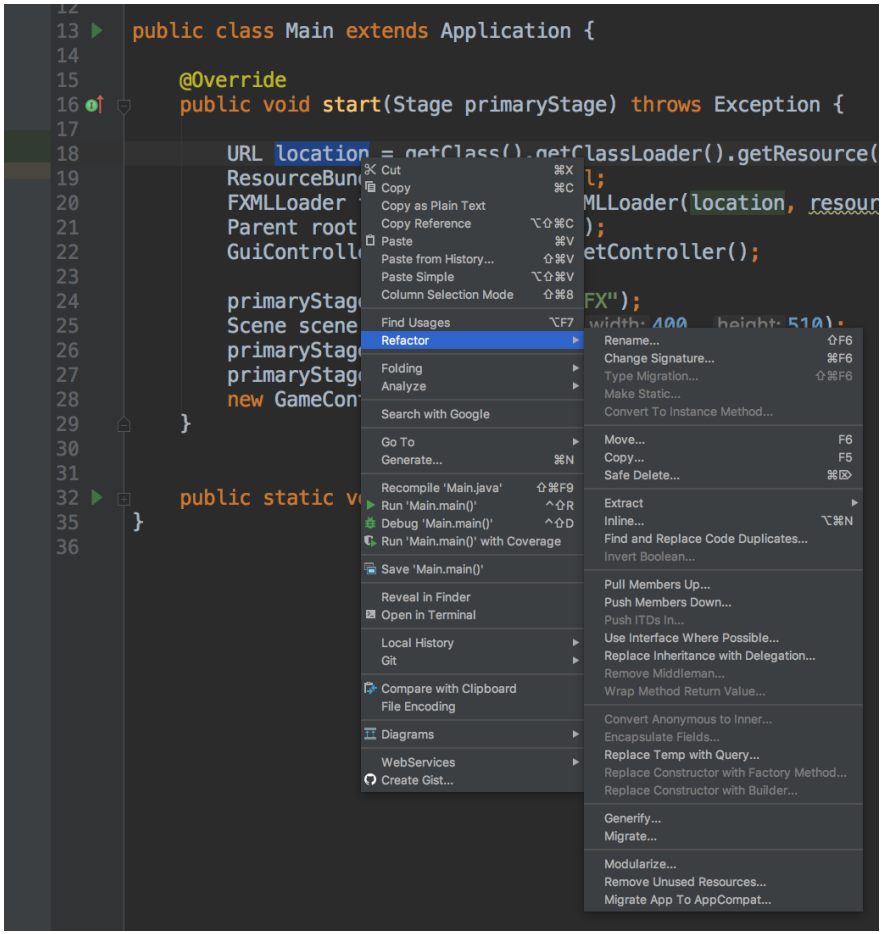
```
void copyToStop(Node head,
                List<String> destination) {
    for (Node node=head; node!=null; node=node.next) {
        String element = node.element;
        if (element.equals("stop"))
            break; // or return
        destination.add(element);
    }
}
```

Remarque

Si elle est bien écrite, une méthode courte avec plusieurs points de sortie est plus facile à lire qu'une méthode qui utilise des variables booléennes "artificielles" pour sortir des boucles.

En IntelliJ IDEA

- Auto-completion : flèche pour choisir et touche tabulation
- Ré-indentation du code : Alt+Ctrl+L
- Rennomage : shift + F6



Commentaires et documentation

Commentaires inutiles

```
String get(String[] source, int index) {
    // Teste si l'index est dans les limites du tableau.
    if (index < 0 || index >= source.length)
        return null;
    return source[index];
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```
boolean indexIsInBounds(String[] source, int index) {
    return index >= 0 && index < source.length;
}
String get(String[] source, int index) {
    if (!indexIsInBounds(source, index))
        return null;
    return source[index];
}
```

Commentaires inutiles

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return true;
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return false;
}
```

Commentaires inutiles = répétition

Commentaires inutiles

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
    /* le résultat sera stocké dans cette liste : */
    List<Rectangle> list2 = new ArrayList<Rectangle>();
    for (Rectangle x : list)
        if (x.w == x.h /* un carré ? */)
            list2.add(x);
    return list2;
}
class Rectangle {
    public int w; /* largeur */
    public int h; /* hauteur */
}
```

Commentaires inutiles

On peut se passer de commentaire en rajoutant une méthode et en nommant correctement les éléments du code.

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {
    List<Rectangle> squares = new ArrayList<Rectangle>();
    for (Rectangle rectangle : rectangles)
        if (rectangle.isSquare())
            squares.add(rectangle);
    return squares;
}
class Rectangle {
    private int width, height;
    boolean isSquare() {
        return width == height;
    }
}
```

Commentaires **utiles**

Des commentaires pour décrire les tâches à réaliser peuvent être utiles

```
void processElement(Stack<Formula> stack,
                    String element) {
    // TODO : prendre en compte les signes '-' et '/'
    switch (element) {
        case "+": processSum(stack); break;
        case "*": processProduct(stack); break;
        default : processInteger(stack, element);
        break;
    }
}
```

Commentaires utiles

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns <tt>true</tt> if this list contains the
 * specified element. More formally, returns
 * <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is
 * to be tested @return <tt>true</tt> if this list
 * contains the specified element
 */
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}
```

JavaDoc

JavaDoc permet de génèrer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, ...

La documentation contient :

- Une description des membres : attributs et méthodes (publics et protégés) des classes
- Une description des classes, interfaces, ...
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

Un bloc de commentaire Java commençant par /** deviendra un bloc de commentaire Javadoc qui sera inclus dans la documentation du code source.

JavaDoc

Tag	Description
@author	pour preéciser l'auteur de la fonctionnalité
@deprecated	indique que l'attribut, la méthode ou la classe est dépréciée
@return	pouyr décrire la valeur de retour
{@code `literal`}	Formate `literal` en code
{@link `reference`}	permet de créer un lien

Pour générer la javadoc en IntelliJ

Tools → generate Javadoc

JavaDoc

```
/**
 * The {@code Byte} class wraps a value of primitive
 * type {@code byte} in an object.  An object of type
 * {@code Byte} contains a single field whose type is
 * {@code byte}.
 *
 * <p>In addition, this class provides several methods
 * for converting a {@code byte} to a {@code String} and
 * a  {@code String} to a {@code byte}, as well as other
 * constants and methods useful when dealing with a
 * {@code byte}.
 *
 * @author  Nakul Saraiya
 * @author  Joseph D. Darcy
 */
```

À retenir

À retenir absolument

- Bien nommer les éléments du code est essentiel.
- Chaque élément du code ne doit avoir qu'un sens/raison de changer/responsabilité et un seul.
- Ne pas se répéter est important.
- Il faut toujours essayer de choisir la solution la plus simple à un problème.

Votre objectif

Écrire du code lisible