Programmation 2 : premier cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

10 ou 12 septembre 2018

Bienvenue dans le cours de programmation 2

Organisation du cours

Volume horaire

- 12 séances de cours d'1h30
- 12 séances de travaux dirigés d'1h30
- 12 séances de travaux pratique de 2h

Évaluation

- un examen (80% de la note)
- des rendus de travaux pratiques (20% de la note)

Objectifs du cours

- Apprendre à programmer avec des objets
  - adopter le “penser objet”
  - connaître et savoir mettre en œuvre les concepts fondamentaux de la programmation objet
- Apprendre à coder proprement
- Connaître le langage Java
- Préparer aux cours de **Projet : initiation génie logiciel** du semestre 4 et de **Programmation et conception** du semestre 5

Apprendre à programmer avec des objets

Notions de programmation objet

- Classes et instances
- Visibilité, composition et délégation
- Interface et polymorphisme
- Extension et redéfinition de méthode
- Types paramétrés et généricité
- ...

Qu'est-ce que programmer ?

1. **Analyser** les besoins
2. **Spécifier** les comportements du programme
3. **Choisir** et éventuellement **concevoir** les solutions techniques
4. **Implémenter** le programme (coder)
5. **Vérifier** que le programme a le comportement spécifié (tester)
6. **Déployer** le programme dans son environnement
7. **Maintenir** le programme (corriger les bugs, ajouter des fonctionnalités)

Qu'est-ce que programmer proprement ?

Un programme propre :

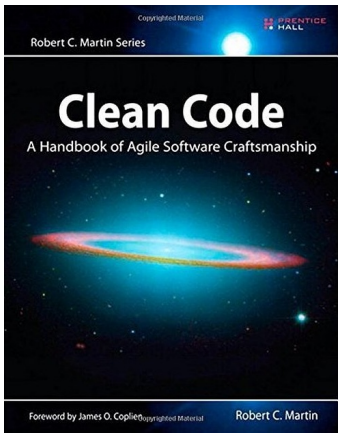
- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous

Comment programmer proprement ?

Pour programmer proprement dans un langage objet, il faut :

- nommer correctement les éléments du code
- écrire du code lisible (par un autre humain)
- relire et améliorer le code

Guide de bonnes pratiques disponible en ligne.



Pourquoi programmer proprement ?

- Pour rendre les programmes facilement compréhensibles par un être humain car l'essentiel du travail du programmeur consiste en la lecture de code.
- Du code sale est souvent sources d'erreurs et n'est pas maintenable.

Important

À partir de maintenant, vous devez écrire du code qui fonctionne et qui puisse être lu sans effort.
À l'examen, **un code ne respectant pas ces conditions sera considéré comme faux**.

Exemple de code mal écrit

```
struct Rec {
    int l, L;
}

typedef struct Rec Rec;

int compte(Rec** r, int l){
    int s = 0;
    for(int i = 0; i < l; i++){
        if (r[i]->l == r[i]->L) s++;
    }
    return s;
}
```

Après un petit peu de refactoring

```
#include<stdbool.h>
struct Rectangle {
    int width, height;
}
typedef struct Rectangle Rectangle;
bool isSquare(Rectangle* rectangle){
    return rectangle->width == rectangle->height;
}
int countSquare(Rectangle* rectangles[], int length){
    int nbSquares = 0;
    for(int index = 0; index < length; index++){
        if (isSquare(rectangles[i])) nbSquares++;
    }
    return nbSquares;
}
```

Le langage Java

Le langage Java :

- est un langage de programmation orienté objet
- créé par James Gosling et Patrick Naughton (Sun)
- présenté officiellement le 23 mai 1995

Les objectifs de Java :

- simple, orienté objet et familier
- robuste et sûr
- indépendant de la machine (machine virtuelle Java)
- multi-tâche et performant

Premier programme en Java

Le programme *HelloWorld.java* :

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World !");
    }
}
```

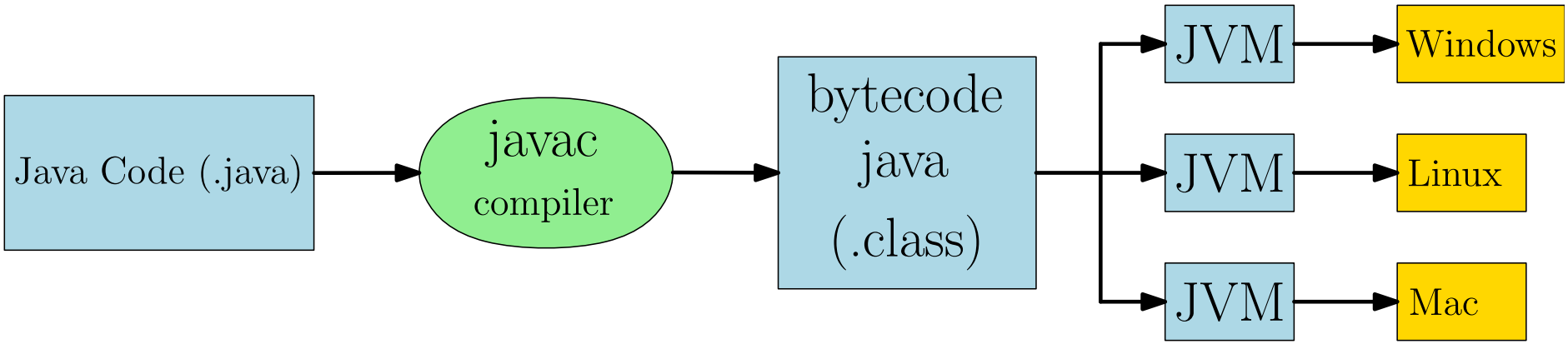
Compiler et exécuter

```
$ javac HelloWorld.java
$ ls
HelloWorld.java HelloWorld.class
$ java HelloWorld
Hello World !
```

Machine virtuelle

- Le compilateur java (`javac`) génère du **bytecode java**
- Le code java s'exécute dans une **Java Virtual Machine** (JVM)

$\Rightarrow$ il suffit de compiler le code une seule fois.



Instructions, variables et expressions

Comme pour le langage C, on manipule en Java les données à l'aide de variables et d'expressions :

```
class Exemple{
    public static void main (String args[]){
        int variable1 = 10;
        double variable2 = 3.2;
        variable2 += variable1;
        variable1 = 3*variable1 - 4
        System.out.println(variable1); // 26
        System.out.println(variable2); // 13.2
    }
}
```

Les types primitifs

En java, en plus des objets que nous allons voir, il existe des types primitifs similaires voire identiques aux types du langage C :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	`false` ou `true`	`false`

Plan du cours

1. Introduction à la programmation objet et Java
2. Bonnes pratiques de programmation
3. Instances et classes
4. Interfaces et polymorphisme
5. Types génériques et paramétrés
6. Spécification, documentation et tests
7. Classes abstraites et extension
8. Exceptions
9. Classes anonymes, interfaces fonctionnelles, lambda
10. Structure d'un projet et paquets
11. Notions de conception objet
12. Session de live coding

Instances et classes

Exemple d'application

On souhaite réaliser une application permettant d'affecter des étudiants à des cours d'options :

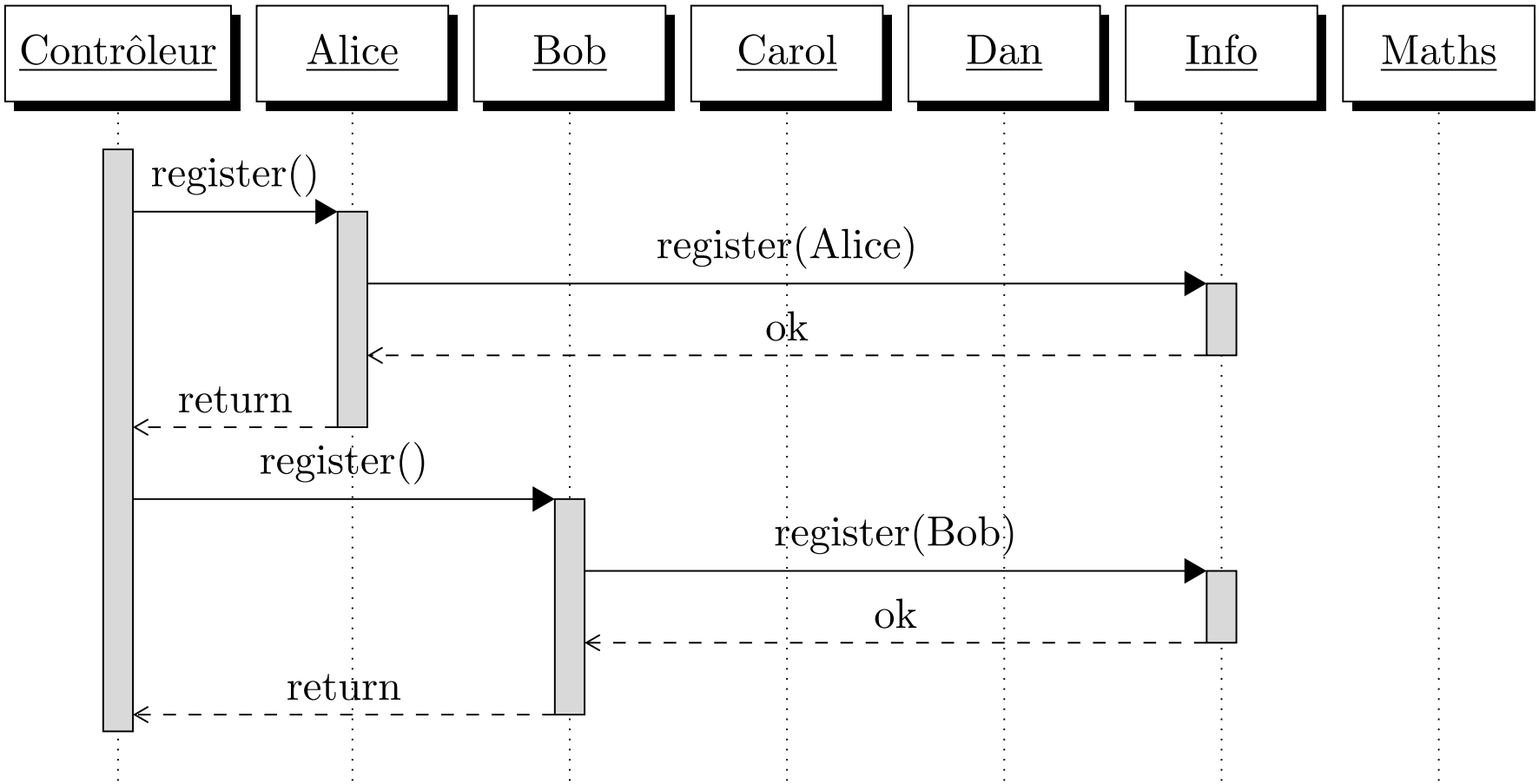
- chaque cours a une capacité maximale
- chaque étudiant a une liste de préférences sur les cours (liste ordonnée des cours)
- les listes de cours, d'étudiants et leurs préférences sont sous forme de fichiers
- il nous faut attribuer une option à chaque étudiant et produire un fichier avec ces informations

Les éléments de cette tâche

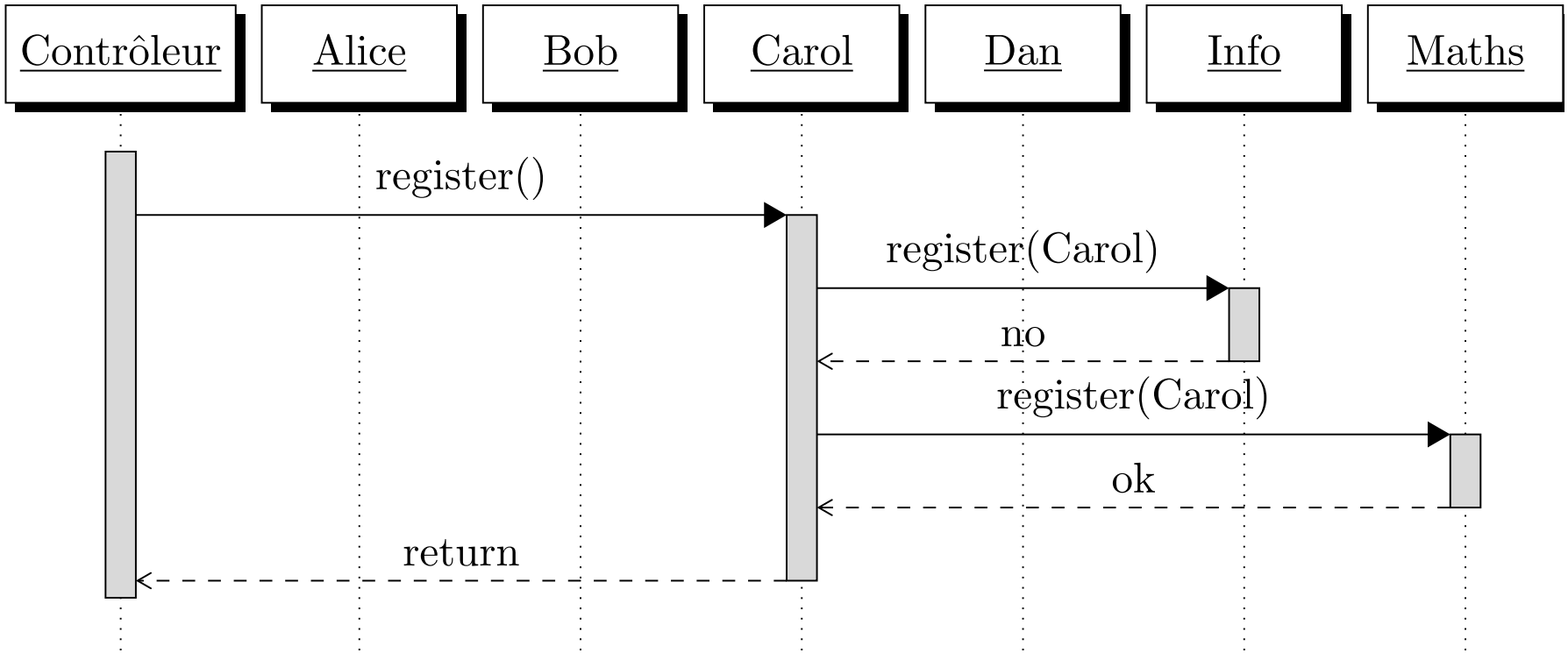
- des étudiants
- des cours
- des listes de préférences
- des listes d'inscrits
- des fichiers à lire/écrire

Chaque élément va être représenté par un objet.

Exemple de processus (1/2)



Exemple de processus (2/2)



Comment maîtriser cette complexité

En programmation objet :

- Le comportement du programme est le résultat de la **communication** des objets entre eux.
- Les interactions produites sont en général **trop complexes** pour être appréhendées par un être humain.

Mais on réfléchit uniquement au niveau de l'objet :

- Chaque objet doit avoir **un seul rôle, une seule responsabilité**.
- Chaque objet doit avoir un comportement simple, facilement **compréhensible par l'humain**.
- Chaque objet doit limiter ses interactions avec les autres objets au nécessaire pour remplir son rôle.

Un étudiant

Un étudiant possède:

- un nom,
- une liste de préférences de cours.

Et il peut :

- s'inscrire à un cours,
- se faire désinscrire d'un cours.

Un cours

Un cours possède :

- une capacité,
- une liste d'étudiants inscrits.

Et il peut :

- traiter l'inscription d'un étudiant,
- fournir la liste des étudiants inscrits.

Une liste de préférences

Une liste de préférences de cours possède :

- une séquence ordonnée de cours.

Et elle peut :

- fournir le prochain cours dans la liste,
- déclarer qu'il n'y a plus de cours sur la liste.

Les objets

Étudiants, cours et listes de préférences ont donc :

- des **attributs** (le nom de l'étudiant, la capacité du cours, la séquence de cours de la liste de préférences, . . .),
- des **comportements** (s'inscrire à un cours, fournir la liste des étudiants inscrits, fournir le prochain cours dans l'ordre de préférence).

Un **objet** est une entité composée de :

- **attributs** (ou champs, données membres, propriétés)
- de **méthodes** (ou **comportements**)

L'étudiant Paul Calcul

Attributs :

- **nom :** "Paul Calcul"
- **préférences :** Modélisation algorithmique, Introduction à l'informatique quantique, Introduction à la vérification, Apprentissage automatique

Comportements :

- **s'inscrire :** s'inscrit au premier cours de sa liste de préférences duquel elle n'a pas encore été rejetée, jusqu'à ce que son inscription soit acceptée (ou avoir épuisé tous les cours).
- **se faire rejeter d'un cours :** s'inscrire.

L'étudiante Marie Lis

Attributs :

- **nom :** "Marie Lis"
- **préférences :** Introduction à la vérification, Introduction à l'informatique quantique, Apprentissage automatique, Modélisation algorithmique

Comportements :

- **s'inscrire :** s'inscrit au premier cours de sa liste de préférences duquel elle n'a pas encore été rejetée, jusqu'à ce que son inscription soit acceptée (ou avoir épuisé tous les cours).
- **se faire rejeter d'un cours :** s'inscrire.

Cours Modélisation algorithmique

Attributs :

- **intitulé :** “Modélisation algorithmique”
- **capacité :** 60 étudiants
- **étudiants inscrits :** initialement aucun

Comportements :

- **inscrire l’étudiant `student` :** si le nombre d’étudiants inscrits est égal à la capacité, rejeter la demande d’inscription. Sinon ajouter `student` aux étudiants inscrits.
- **fournir la liste des étudiants :** retourner la liste des étudiants.

Cours Introduction à la vérification

Attributs :

- **intitulé :** “Introduction à la vérification”
- **capacité :** 20 étudiants
- **étudiants inscrits :** initialement aucun

Comportements :

- **inscrire l’étudiant `student` :** si le nombre d’étudiants inscrits est inférieur à la capacité, ajouter `student` aux étudiants inscrits. Sinon, inscrire `student` et désinscrire un étudiant choisi par tirage au sort.
- **fournir la liste des étudiants :** retourner la liste des étudiants.

Classe

Classe `Student`

Paul Calcul et Marie Lis ont les mêmes comportements et les mêmes Attributs, seules les valeurs de leurs Attributs sont différentes. Ils appartiennent à la même classe.

Interface `Course`

“Modélisation algorithmique” et “Introduction à la vérification” ne se comportent pas de la même façon, ils ne sont donc pas dans la même classe. Cependant ils proposent la même liste de services, ils ont donc une interface commune.

Les objets et classes

Un **objet** :

- peut être **construit**
- est **structuré** : il est constitué d'un ensemble d'**attributs** (données de l'objet)
- possède un **état** : la valeur de ses attributs
- possède une **interface** : les opérations applicables appelées **méthodes**

Dans les langages orientés objet, une **classe** (d'objet) définit :

- une façon de construire les objets (**constructeurs**)
- la structure des objets de la classe (**attributs**)
- le comportement des objets de la classe (**méthodes**)
- l'interface des objets de la classe (**méthodes et attributs publiques**)
- un type "référence vers des objets de cette classe"

Classes et instances (1/2)

Le mot-clé `class` permet de définir une classe :

```
public class Counter {
    /* Définitions des attributs et des méthodes */
}
```

Le mot-clé `public` sert à indiquer que la classe est accessible depuis l'extérieur.

On peut définir une variable de type “référence vers un `Counter`” :

```
Counter counter;
```

Classes et instances (2/2)

Une classe définit également un “moule” (**constructeur**) pour fabriquer des objets. La création d'un objet s'effectue avec le mot-clé `new` :

```
Counter counter = new Counter();
```

La variable `counter` contient alors une référence vers une instance de la classe `Counter`.

Instance

Un objet créé à partir d'une classe est une **instance** de cette classe et est du type de cette classe.

Attributs

Les *attributs* décrivent la structure de données de la classe :

```
class Counter {
    int position;
    int step;
}
```

On accède aux attributs avec l'opérateur . (point) :

```
Counter counter = new Counter();
counter.position = 12;
counter.step = 2;
counter.position += counter.step;
```

Attributs des instances

Chaque instance possède son propre état et donc ces propres valeurs de attributs.

```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
counter1.position = 10;
counter2.position = 20;

System.out.println("Counter 1 : " + counter1.position);
// Counter 1 : 10
System.out.println("Counter 2 : " + counter2.position);
// Counter 2 : 20
```

Références

On peut copier les références vers des instances :

```
Counter counter1 = new Counter();
Counter counter2 = counter1;
counter1.position = 10;

System.out.println("Counter 1 : " + counter1.position);
// Counter 1 : 10
System.out.println("Counter 2 : " + counter2.position);
// Counter 2 : 10
```

Important

Une référence contient soit `null` soit la référence d'une instance compatible avec le type de la variable.

Les méthodes (pour consulter l'état)

Méthodes permettant d'accéder à l'état de l'objet :

```
class Counter{
    int position, step;
    void getPosition(){ return position; }
}
```

Exemple d'utilisation :

```
Counter counter = new Counter();
counter.position = 10;
System.out.println("Counter : " + counter.getPosition());
// Counter : 10
```

Les méthodes (pour changer l'état)

Méthodes permettant de modifier l'état de l'objet :

```
class Counter{
    int position, step;
    void getPosition(){ return position; }
    void setPosition(int p){
        position = p;
    }
}
```

Exemple d'utilisation :

```
Counter counter = new Counter();
counter.setPosition(10);
System.out.println("Counter : " + counter.getPosition());

// Counter : 10
```

Les méthodes (avec retour et effet de bord)

Méthode qui modifie l'état de l'objet et retourne une valeur :

```
class Counter{
    // code des transparents précédents
    int count(){
        position += step;
        return position;
    }
}
```

Déclaration d'un constructeur

Les constructeurs permettent d'allouer et d'initialiser des instances.

```
class Counter{
    int position, step;

    Counter(int p, int s){
        position = p;
        step = s;
    }
}
```

Exemple d'utilisation :

```
Counter counter = new Counter(22, 5);
```

Déclaration de plusieurs constructeurs

Il est possible de déclarer plusieurs constructeurs (avec paramètres différents).

```
class Counter{
    int position, step;
    Counter(int p, int s){ position = p; step = s; }

    Counter(int p){ position = p; step = 1; }
}
```

Exemple d'utilisation :

```
Counter counter1 = new Counter(22, 5);
Counter counter2 = new Counter(22);
```

Constructeur par défaut

Si aucun constructeur n'est défini, la classe a un constructeur par défaut.

```
class Point{
    int x = 0;
    int y = 0;
}
```

Ce code est équivalent au code suivant :

```
class Point{
    int x, y;
    Point(){
        x = 0; y = 0;
    }
}
```

Le mot-clé this

Le mot-clé this fait référence à l'instance en construction ou à l'instance sur laquelle la méthode est appelée.

```
class Counter{
    int position, step;
    Counter(int position, int step) {
        this.position = position;
        this.step = step;
    }
    int count(){
        this.position += this.step;
        return this.position;
    }
}
```

this pour appeler un constructeur

Le mot-clé this permet également d'appeler un constructeur dans un constructeur :

```
class Counter{
    int position, step;
    Counter(int position, int step) {
        this.position = position;
        this.step = step;
    }
    Counter(int position) {
        this(position, 1);
    }
}
```

attributs et méthodes statiques

Il est possible de définir des méthodes directement associées à la classe avec le mot-clé `static`. Ces méthodes ne sont pas associées à une instance mais à la classe.

```
class Counter{
    static int step;
    int position;

    Counter(int position) { this.position = position; }

    static void setStep(int step) {
        Counter.step = step;
    }

    void count() { position += step; }
}
```

Utilisation de attributs et méthodes statiques

On peut appeler ces méthodes sans utiliser d'instance.

```
public class test{
    public static void main(String[] arg) {
        Counter.setStep(3);
        Counter c1 = new Counter(2);
        Counter c2 = new Counter(3);
        c1.count(); c2.count();
        System.out.println(c1.position); // → 5
        System.out.println(c1.position); // → 6
        Counter.setStep(4); c1.count(); c2.count();
        System.out.println(c1.position); // → 9
        System.out.println(c1.position); // → 10
    }
}
```

Limites des méthodes statiques

Une méthode statique ne peut utiliser que :

- des attributs statiques de la classe
- des méthodes statiques de la classe

L'utilisation de `this` n'a aucun sens dans une méthode statique.

Il est préférable de ne pas utiliser le mot clé static sauf pour :

- la méthode `main`
- des constantes comme `Math.PI`
- des fontions de bibliothèque sur les types primitifs
- ...

Mot-clés private, public et final

```
public class Student {
    public final String name;
    private Course currentCourse;
}
```

Mot-clés utiles

- Les mot-clés `public` ou `private` indiquent si l'attribut ou la méthode est connue de tous les objets ou seulement ceux de la classe.
- Le mot-clé `final` indique que l'attribut ne change jamais de valeur après la construction de l'objet.

Classes utiles et types primitifs

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	`false` ou `true`	`false`

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Pour les objets, cela signifie passer une copie de la référence : il est donc possible de modifier l'état de l'objet.

Pour les types primitifs, cela signifie que l'argument est un copie uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Tableaux unidimensionnels

En Java, les tableaux sont des objets (et donc des références).

Déclaration d'une variable de type "référence vers un tableau" :

```
int[] arrayOfInt;
double[] arrayOfDouble;
```

Allocation d'un tableau :

```
arrayOfInt = new int[10]
arrayOfDouble = new double[3];
```

Utilisation d'un tableau :

```
arrayOfInt[0] = 5;
arrayOfInt[9] = 10;
arrayOfDouble[2] = arrayOfInt[0] / arrayOfInt[9];
system.out.println(arrayOfDouble.length) // 3
```

Tableaux multidimensionnels

Déclaration :

```
int[][] matrixOfInt;
```

Allocation :

```
matrixOfInt = new int[10][];
for(int row = 0; row < matrixOfInt.length; row++)
    matrixOfInt[row] = new int[5];
/* ou directement */
matrix = new int[10][5];
```

Matrices non rectangulaires :

```
matrixOfInt = new int[10][];
for(int row = 0; row < matrixOfInt.length; row++)
    matrixOfInt[row] = new int[row + 1];
```

Chaînes de caractères (1/2)

Trois classes permettent de gérer des chaînes de caractères :

- la classe `String` : chaîne invariable (**immutable**)
- la classe `StringBuilder` : chaîne modifiable (mono-thread)
- la classe `StringBuffer` : chaîne modifiable (multi-thread)

Déclaration et création :

```
String hello = "Hello";
String world = "World";
```

Concaténation :

```
String helloWorld = hello + " " + world + " ! ";
int integer = 13;
String helloWorld1213 = hello + " " + world + " "
                        + 12 + " " + integer;
```

Chaînes de caractères (2/2)

Affichage :

```
System.out.print(helloWorld);   // affiche "Hello World
System.out.println(helloWorld); // affiche "Hello World
                                // avec retour à la ligne
```

Comparaison :

```
String a1 = "a";
String a2 = "a";
String a3 = new String("a");
System.out.println(a1==a2);          // affiche "true"
System.out.println(a1==a3);          // affiche "false"
System.out.println(a1.equals(a3)); // affiche "true"
```

À retenir absolument

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

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