



INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET

Romain MARIE

2019 – 2020

AVANT-PROPOS

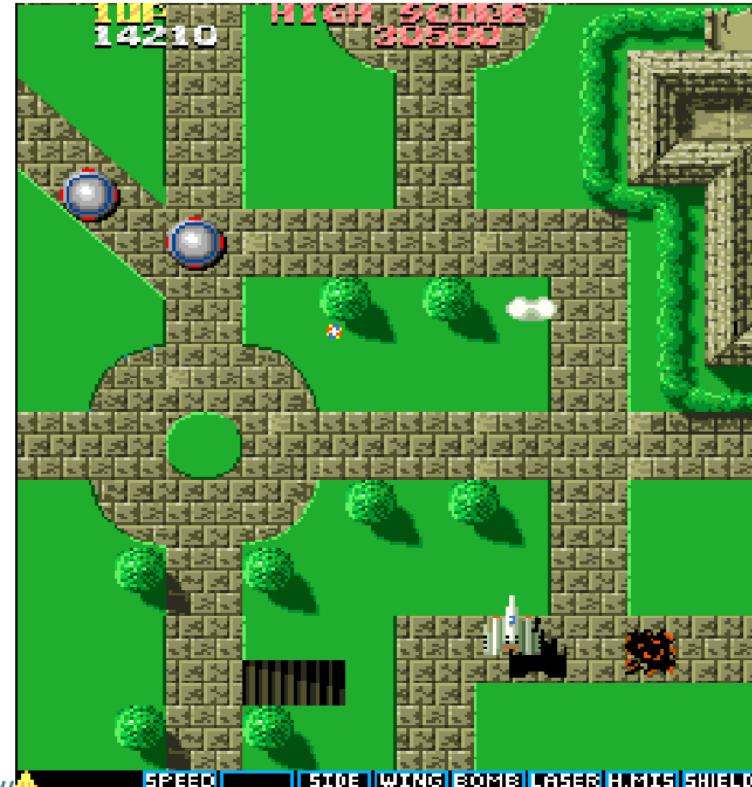
1 // [L'ENSEIGNEMENT]

- Contenu
 - Bases de la programmation *
 - Gestion entrées / sorties *
 - Les classes (attributs, méthodes, constructeurs) **
 - Héritage, polymorphisme, abstraction ***
 - Classes standards ***
 - Interfaces, packages **
- Déroulement
 - Cours : 12 séances
 - TD : 14 séances
 - TP : 20 séances (déoublées)
- Evaluation
 - 80% : 2 EI (sem 47 et 05)
 - 20% : 1 mini projet
 - N.E.P !!!

2 // LE MINI PROJET

- Déroulement
 - Libre service : 16 séances
- Sujet
 - Création d'un jeu vidéo
 - En utilisant une librairie existante : Slick 2D
 - Présentation de votre réalisation à la fin

- Soutenance : 1 séance (sem 5)



3 // [LES RESSOURCES]

- À 3iL
 - TOUSCOM (dossier POO de votre année)
 - Supports de l'enseignement (cours / TD / TP)
 - Correction des TD
 - Documents divers
 - Moi-même
 - Bureau 211 / romain.marie@3il.fr / 05 55 31 67 35
- Sur internet
 - Doc. officielle Java
 - Stackoverflow
 - Codingame
 - Sololearn

AVANT-PROPOS

- 1 / L'ENSEIGNEMENT
- 2 / LE MINI PROJET
- 3 / LES RESSOURCES

QUESTIONS ?

INTRODUCTION

1 // LE LANGAGE MACHINE

■ Ordinateur



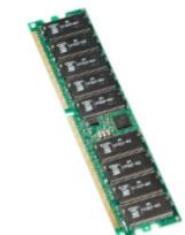
- Assemblage complexe de plusieurs composants
- Parmi les plus importants, on trouve la mémoire (**RAM**), et le processeur (**CPU**)

■ Le processeur

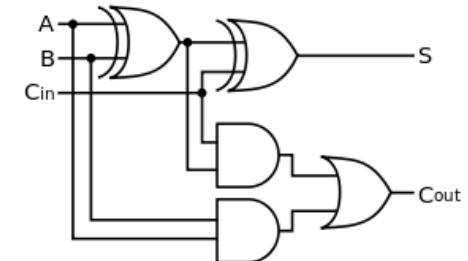
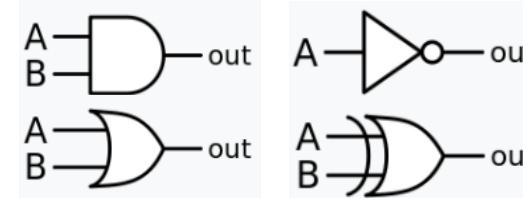


- Composé (notamment) de transistors
- Chargé d'exécuter des opérations binaires
- Comprend son **langage machine**

■ La mémoire



- Egalement formée (notamment) de transistors
- Contient un ensemble d'emplacements numérotés (adresses mémoires)
- Chargée de stocker / fournir des nombres binaires

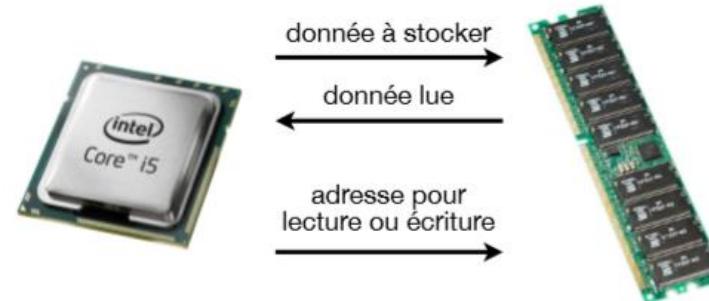


| | |
|----------|---------------|
| 00010111 | Emplacement 0 |
| 01000011 | Emplacement 1 |
| 01100011 | Emplacement 2 |
| 10000000 | |
| 00100010 | |
| 01000001 | |
| 01101111 | Emplacement 6 |

1 // LE LANGAGE MACHINE

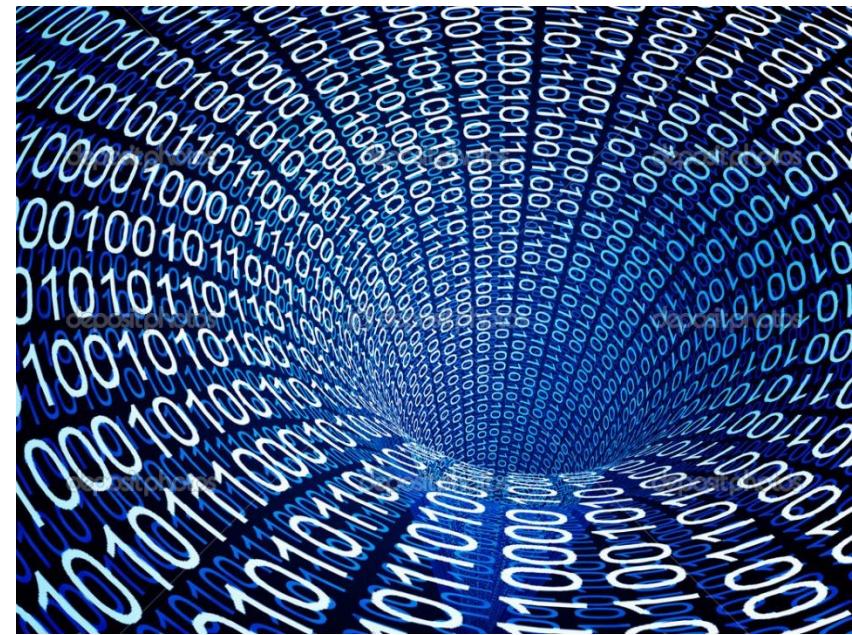
- Programme informatique = Ensemble d'instructions

- Ecrites en **langage machine**
- Stockées dans la **RAM**
- Exécutées par le **CPU**



- Des 0 et des 1 ...

- La mémoire stocke des nombres binaires
- Une instruction du langage machine est définie par un nombre binaire
- Le résultat d'une opération effectuée par le processeur est un nombre binaire



Problème : personne ne parle le binaire !

Compromis : les langages de programmation

2 // [LES LANGAGES DE PROGRAMMATION]

- Un langage de programmation doit être
 - Compréhensible par l'être humain
 - Possible à traduire sans ambiguïté en langage machine
- Un langage de programmation est composé de :
 - Types primitifs : pour décrire et modéliser les données
 - Règles syntaxiques : pour permettre la conversion en langage machine
 - Règles sémantiques : pour garantir que les instructions aient un sens

```
<?php  
  
echo "Hello World";  
?>
```

```
#include <stdio.h>  
  
int main()  
{  
    printf("hello world");  
    return 0;  
}
```

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

Les langages de programmation sont TRES stricts

| | | | | | | | | |
|------|------|------|------|------|------|------|------|------|
| 7f45 | 4c46 | 0201 | 0100 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 0200 | 3e00 | 0100 | 0000 | 3004 | 4000 | 0000 | 0000 | 0000 |
| 4000 | 0000 | 0000 | 0000 | e019 | 0000 | 0000 | 0000 | 0000 |
| 0000 | 0000 | 4000 | 3800 | 0900 | 4000 | 1f00 | 1c00 | |
| 0600 | 0000 | 0500 | 0000 | 4000 | 0000 | 0000 | 0000 | 0000 |
| 4000 | 4000 | 0000 | 0000 | 4000 | 4000 | 0000 | 0000 | 0000 |
| f801 | 0000 | 0000 | 0000 | f801 | 0000 | 0000 | 0000 | 0000 |
| 0800 | 0000 | 0000 | 0000 | 0300 | 0000 | 0400 | 0000 | 0000 |
| 3802 | 0000 | 0000 | 0000 | 3802 | 4000 | 0000 | 0000 | 0000 |
| 3802 | 4000 | 0000 | 0000 | 1c00 | 0000 | 0000 | 0000 | 0000 |
| 1c00 | 0000 | 0000 | 0000 | 0100 | 0000 | 0000 | 0000 | 0000 |
| 0100 | 0000 | 0500 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 0000 | 4000 | 0000 | 0000 | 0000 | 4000 | 0000 | 0000 | 0000 |
| fc06 | 0000 | 0000 | 0000 | fc06 | 0000 | 0000 | 0000 | 0000 |
| 0000 | 2000 | 0000 | 0000 | 0100 | 0000 | 0600 | 0000 | 0000 |
| 100e | 0000 | 0000 | 0000 | 100e | 6000 | 0000 | 0000 | 0000 |
| 100e | 6000 | 0000 | 0000 | 2802 | 0000 | 0000 | 0000 | 0000 |
| 3002 | 0000 | 0000 | 0000 | 0000 | 2000 | 0000 | 0000 | 0000 |
| 0200 | 0000 | 0600 | 0000 | 280e | 0000 | 0000 | 0000 | 0000 |
| 280e | 6000 | 0000 | 0000 | 280e | 6000 | 0000 | 0000 | 0000 |
| d001 | 0000 | 0000 | 0000 | d001 | 0000 | 0000 | 0000 | 0000 |
| 0800 | 0000 | 0000 | 0000 | 0400 | 0000 | 0400 | 0000 | 0000 |
| 5402 | 0000 | 0000 | 0000 | 5402 | 4000 | 0000 | 0000 | 0000 |
| 5402 | 4000 | 0000 | 0000 | 4400 | 0000 | 0000 | 0000 | 0000 |
| 4400 | 0000 | 0000 | 0000 | 0400 | 0000 | 0000 | 0000 | 0000 |

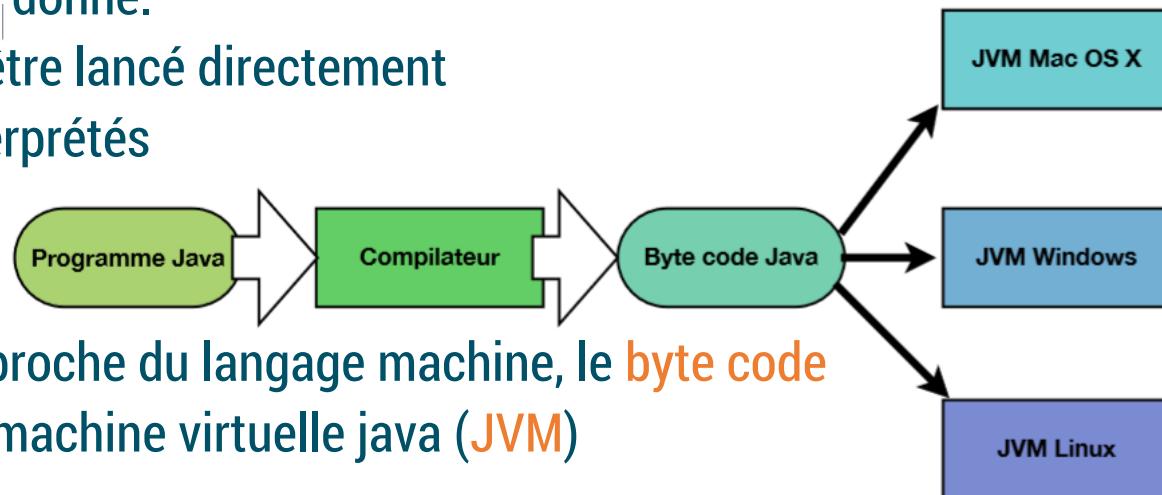
2 // [LES LANGAGES DE PROGRAMMATION]

■ Deux méthodes de conversion en langage machine

- L'interprétation (matlab, python, ...)
 - La traduction se fait à l'exécution, instruction par instruction
 - Généralement multi-plateforme
- La compilation (C, C++, ...)
 - la traduction se fait en amont, pour un système donné.
 - Aboutit à la création d'un exécutable, qui peut être lancé directement
 - Généralement plus rapide que les langages interprétés

■ Le langage java

- Combine compilation et interprétation
- Un compilateur produit un code intermédiaire, proche du langage machine, le **byte code**
- A l'exécution, le byte code est interprété par la machine virtuelle java (**JVM**)



3 // [PROGRAMMER]

■ Programme informatique = ensemble d'instructions

- Séquence d'instructions réalisant une tâche précise (**algorithme**)
- Ecrites dans un **langage de programmation**
- Manipulant des données (**variables**)
- Converties en **langage machine**
- Stockées dans la **RAM**
- Exécutées par le **CPU**

■ Exemple :

- Afficher «Entrez la somme en euros : »
- Saisir euros
- francs = euros x 6.559
- Afficher «La somme convertie est »,francs

```
01 package convertisseur;  
02  
03 import java.util.Scanner;  
04  
05 public class Convertisseur {  
06  
07     public static void main(String[] args) {  
08         double euros,francs;  
09  
10         System.out.print("Entrez la somme en euros : ");  
11         Scanner sc = new Scanner(System.in);  
12         euros = sc.nextDouble();  
13         francs = 6.559 * euros;  
14         System.out.print("La somme convertie est ");  
15         System.out.println(francs);  
16     }  
17 }
```

3 // [PROGRAMMER]

■ Ecrire un programme :

■ 1 - Analyse et conception :

- Définition des données de départ et du résultat attendu
- Modélisation des données du problème à partir des **types primitifs**
- Description des étapes permettant d'arriver au résultat (**algorithme**)

■ 2 - Codage :

- Traduction de l'algorithme dans le langage de programmation choisi (**code source**)

■ 3 - Mise au point :

- Compilation pour valider la syntaxe du code source
- Exécution du programme avec des jeux de tests pour identifier d'éventuels comportements non prévus (bugs)
- Corrections éventuelles : Un bug identifié nécessite souvent de repartir à l'étape 1

■ 4 – Maintenance :

- Un bug peut être identifié des années après la phase de codage, et donc nécessiter une mise à jour d'un code source totalement oublié → **BONNES PRATIQUES !**

3 // [PROGRAMMER]

```
01 import java.util.Scanner;  
02 public class Main {  
03     public static void main(String[] args) {  
04         Scanner sc=new Scanner(System.in);  
05         double e=sc.nextDouble();  
06         System.out.println(e*6.55957);}}}
```

```
01 import java.util.Scanner;  
02  
03 public class ConversionEurosFrancs {  
04  
05     public static void main(String[] args) {  
06         Scanner sc = new Scanner(System.in);  
07  
08         double euros;  
09         double francs;  
10  
11         System.out.print("Entrez le montant en euros à convertir : ");  
12         euros = sc.nextDouble();  
13         francs = euros * 6.55957;  
14  
15         System.out.println(euros + " euro(s) = " + francs + " franc(s));  
16     }  
17 }
```

INTRODUCTION

- 1 / LANGAGE MACHINE
- 2 / LANGAGES DE PROGRAMMATION
- 3 / PROGRAMMER

QUESTIONS ?

VARIABLES & EXPRESSIONS

1 // [LES VARIABLES]

- Outil de base pour manipuler des données et des valeurs

```
13 | francs = euros * 6.55957;
```

- Une variable se caractérise par :

- Un **nom** qui la désigne (pour le programmeur 
- Une **adresse mémoire** qui localise où elle est stockée (pour la machine 
- Un **type** qui indique comment la manipuler ( et comment la stocker (
- Une **portée** qui délimite où elle est utilisable ( 

1 // [LES VARIABLES]

- Avant d'être utilisée, une variable doit être déclarée :
 - On fournit à l'ordinateur un **type** et un **nom** pour qu'il y associe une **adresse mémoire** de la bonne taille
- Plusieurs raccourcis possibles :
 - Déclarer en même temps plusieurs variables du même type

```
double francs, euros;
```
 - Initialiser une variable au moment de sa déclaration

```
double francs=10;
```
 - Les deux à la fois ...

```
double francs=10, euros=6.55*francs;
```

| | |
|----|----------------|
| 08 | double euros; |
| 09 | double francs; |

2 // [NOMMER SES VARIABLES]

■ Plusieurs contraintes :

- Le premier caractère ne peut être un chiffre
~~int 3fois4;~~
- Aucun espace ne peut être utilisé (on utilise `_`)
~~int ma variable; int ma_variable;~~
- Majuscules ≠ Minuscules

`int mavariable; int MaVariable; int maVariable;`

- Les accents sont tolérés mais déconseillés
- Certains mots sont réservés par le langage
- De nombreux caractères sont interdits :

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| & | ~ | " | ' | (|) | { | } | [|] | - | |
| ` | \ | ^ | @ | = | % | * | ? | : | / | ! | < |
| > | ; | - | + | . | | | | | | | |

| | | | | |
|----------|----------|------------|-----------|--------------|
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| | const | native | super | while |

2 // [NOMMER SES VARIABLES]

■ Choisir un nom de variable :

- A part les contraintes, on fait ce qu'on veut ... **MAIS**
- Votre code source va être lu par plusieurs personnes :
 - Vous, moi, vos (futurs) collègues, votre vous du futur
 - Moralité : on fait tout sauf ce qu'on veut

■ Un bon nom de variable :

- Renseigne sur le contenu de la variable
- Est raisonnable sur le nombre de caractères

~~double laValeurEnFrancsSaisieParLUtilisateur;~~

- Respecte une convention prédéterminée
 - Exemple : CamelCase

```
01 import java.util.Scanner;
02 public class Main {
03     public static void main(String[] args) {
04         Scanner sc=new Scanner(System.in);
05         double e=sc.nextDouble();
06         System.out.println(e*6.55957);}}
```

```
01 import java.util.Scanner;
02
03 public class ConversionEurosFrancs {
04
05     public static void main(String[] args) {
06         Scanner sc = new Scanner(System.in);
07
08         double euros;
09         double francs;
10
11         System.out.print("Entrez le montant en euros à convertir : ");
12         euros = sc.nextDouble();
13         francs = euros * 6.55957;
14
15         System.out.println(euros + " euro(s) = " + francs + " franc(s)");
16     }
17 }
```

2 // [NOMMER SES VARIABLES]

- L'auto-complétion, ou comment se faciliter la vie
 - Principe : on commence à taper, l'IDE finit
 - Gain de temps considérable
 - Confort de programmation évident
- Contrainte : savoir ce qu'on cherche !
 - D'où l'importance des noms de variables

A screenshot of an IDE interface illustrating code completion. On the left, there is a code editor window with the following Java-like pseudocode:

```
14  
15  
16  
17  
18 int maVariable1;  
19 int maVariable2;  
20 int maVariable3;  
21  
22  
23 }  
24  
25 }
```

The cursor is positioned at the start of the word "ma" in the line "maVariable1;". A code completion dropdown menu is open to the right, showing three suggestions:

- maVariable1 : int
- maVariable2 : int
- maVariable3 : int

Below the suggestions, a message reads: "Not what you're looking for? Discover new extensions to Code Completion". At the bottom of the IDE window, there are tabs for "Problems", "Javadoc", and "Declar", and a status bar with the text "Press 'Ctrl+Space' to show Template Proposals".

3 // [LES TYPES DE VARIABLES]

- 8 types primitifs (8 mots réservés)
 - Servent à modéliser l'information de manière adéquate
 - Type de données stockées, espace mémoire requis, plage de valeurs
 - Servent à définir des types plus complexes

| Type | Valeurs / Encodage | Taille en octets |
|---------|--|------------------|
| byte | de -128 à 127 | 1 |
| short | de -32768 à 32767 | 2 |
| int | de -2147483648 à 2147483647 | 4 |
| long | de -9223372036854775808 à -9223372036854775807 | 8 |
| float | Norme IEEE 754 | 4 |
| double | Norme IEEE 754 | 8 |
| boolean | true ou false | 1 |
| char | Encodage Unicode | 2 |

3 // [LES TYPES DE VARIABLES]

- Les dangers à connaître :

- Un type entier (byte, short, int, long) mal utilisé entraîne un résultat faux :

| | | | | | |
|------------|-------------|-------------|-------------|-------------|-------------|
| 2147483647 | +1 | +2 | +3 | +4 | +5 |
| Résultat | -2147483648 | -2147483647 | -2147483646 | -2147483645 | -2147483644 |

- Un type flottant (float, double) même bien utilisé peut engendrer un résultat inexact :
 - float = 6 chiffres exacts, double = 12 chiffres exacts

```
float a=1000000000, b=a+1;  
System.out.println(b);
```

1.0E9

- Certaines valeurs particulières sont susceptibles d'apparaître :
 - NaN (Not a Number) : obtenu lors d'une opération illégale (0.0/0.0 par exemple)
 - Infinity (ou -Infinity) : obtenu en divisant par 0.0 par exemple
 - -0.0 : Pas gênant, mais possible à obtenir pour un nombre négatif proche de 0

3 // [LES TYPES DE VARIABLES]

- Le type booléen
 - Permet de stocker une valeur logique (true / false ... 1 / 0)
 - Même si un seul bit est requis, un booléen occupe un octet (raisons pratiques)
- Le type char
 - Permet de stocker dans une variable un **unique** caractère codé sur 2 octets
 - Pour différencier une variable d'un caractère, on utilise les apostrophes :

```
char a = 'a';
```

- Pour stocker plusieurs caractères (mots, phrases, ...), on utilisera le type **String** (non primitif) qui sera présenté plus tard

4 // [LES OPÉRATIONS SUR VARIABLES]

- L'opération d'affectation `variable = expression;`
 - Permet de placer dans une **variable** le résultat d'une **expression**
 - Définie par le symbole `=`
 - Procède en deux temps :
 - 1 – Evaluation de l'**expression** à droite du `=`
 - 2 – Stockage de la valeur obtenue dans la **variable** située à gauche du `=`
- La notion d'**expression** `double variable = (12*n)%3/(1+Math.cos(2.0*Math.PI));`
 - Une **expression** est un **calcul générant une valeur** d'un type prédéfini
 - Peut être constituée :
 - de nombres
 - de variables
 - d'**opérateurs**
 - d'appels de méthodes (fonctions)

Seule la variable à gauche du `=` est modifiée !

4 // [LES OPÉRATIONS SUR VARIABLES]

■ Les opérateurs arithmétiques :

- Le groupe 2 est prioritaire
- A groupe égal, la gauche est prioritaire
- Les parenthèses permettent de modifier les priorités
- Attention, pas de [] ni de {}

| Symbole | Opération | Groupe de priorité | Résultat avec 11 et 4 comme opérandes |
|---------|--|--------------------|---------------------------------------|
| + | Addition | 1 | 15 |
| - | Soustraction | 1 | 7 |
| * | Multiplication | 2 | 44 |
| / | Division | 2 | 11/4 → 2 11.0/4.0 → 2.75 |
| % | Modulo (reste de la division entière) | 2 | 11%4 → 3 |

| Expression | Résultat | Commentaire |
|------------|----------|---|
| 2+3*10 | 32 | |
| (2+3)*10 | 50 | |
| b*b-4ac | | Erreur : si a et c sont des variables il manque des *; il faut écrire b*b-4*a*c |

4 // [LES OPÉRATIONS SUR VARIABLES]

- incrémenter / décrémenter : `a=a+1; a=a-1;`

- Processus en deux étapes :

- 1) L'expression (droite du =) est évaluée avec la valeur courante de a
 - 2) Le résultat de l'opération écrase l'ancienne valeur de a

- Principe généralisable à n'importe quelle expression

`a=a*(a-1)+12*a;`

- Pour un incrément / décrément de 1, il existe des opérateurs abrégés

- incrément après l'opération : `a++;` `a--;`

- incrément avant l'opération : `++a;` `--a;`

- Exemple : `int a=12, b;`

`b=2*a--;`

24

11

`b=2*--a;`

22

11

4 // [LES OPÉRATIONS SUR VARIABLES]

■ Opérateurs combinés d'affectation

- Permettent de compresser certaines expressions

- Principe :

- Remplacer `<variable> = <variable> <opérateur> (<expression>)`
- Par `<variable> <opérateur> = <expression>`

| Symbol | Exemple | Équivalent à |
|-----------------|-----------------------|----------------------------|
| <code>+=</code> | <code>a += 5</code> | <code>a = a + 5</code> |
| <code>-=</code> | <code>a -= 5+k</code> | <code>a = a - (5+k)</code> |
| <code>*=</code> | <code>a *= 5</code> | <code>a = a * 5</code> |
| <code>/=</code> | <code>a /= 3*k</code> | <code>a = a / (3*k)</code> |
| <code>%=</code> | <code>a %= 4</code> | <code>a = a % 4</code> |

5 // [LES TRANSTYPAGES]

- Problématique :
 - Une opération doit être effectuée sur des variables de même type
 - Lorsque ce n'est pas le cas, il faut donc en convertir certaines !
 - Généralement, le type le plus fort est choisi
- Conversions implicites :
 - Réalisées automatiquement par le compilateur
 - Le java convertit implicitement les entiers en flottants, mais pas l'inverse
- Conversions explicites
 - Réalisées par le programmeur
 - Effectuées en mettant le type désiré entre parenthèses
 - Entraîne généralement une perte de donnée

```
int a=1;  
float b=1.2f;  
float c=a*b;
```

```
int a=1;  
float b=1.2f;  
int d=(int)(a*b);
```

6 // [LA GESTION DES ENTRÉES / SORTIES]

- Afficher du texte à l'écran (Sortie)
 - S'effectue grâce aux méthodes `System.out.print()` et `System.out.println()`
 - Pour écrire du texte, on utilise les guillemets « »
 - Pour écrire une variable, on met simplement son nom
 - Pour combiner plusieurs éléments, on utilise l'opérateur de concaténation +

```
01 public class cours {  
02     public static void main(String[] args) {  
03         double rayon = 2.5;  
04         double surface = 3.14159 * rayon * rayon;  
05  
06         System.out.print("La surface d'un disque de rayon "+rayon+  
07                         " est "+surface);  
08     }  
09 }
```



Affiche : La surface d'un disque de rayon 2.5 est 19.6349375

6 // [LA GESTION DES ENTRÉES / SORTIES]

- Saisir des éléments au clavier (Entrée)
 - 1) On initialise un objet de type Scanner
 - 2) On utilise cet objet pour récupérer dans une variable une information dont on connaît le type :
 - 3) On répète 2) autant de fois qu'on veut

```
01 import java.util.Scanner;
02
03 public class cours {
04     public static void main(String[] args) {
05         int a,b;
06         Scanner entree = new Scanner(System.in);
07
08         System.out.print("Entrez un entier : ");
09         a = entree.nextInt();
10         System.out.print("Entrez un autre entier : ");
11         b = entree.nextInt();
12         System.out.println("Vous avez entré : "+a+" et "+b);
13     }
14 }
```

VARIABLES & EXPRESSIONS

- 1 / LES VARIABLES
- 2 / NOMMER SES VARIABLES
- 3 / LES TYPES DE VARIABLES
- 4 / LES OPÉRATIONS SUR VARIABLES
- 5 / LES TRANSTYPAGES
- 6 / LA GESTION DES ENTRÉES / SORTIES

QUESTIONS ?

CONDITIONS & BOUCLES

1 // [LES COMMENTAIRES]

- Objectif : Ajouter du texte ignoré par l'ordinateur
- Permet de
 - Documenter le programme
 - Ignorer ponctuellement certains morceaux du code (debug)
- En java :
 - Le commentaire de fin de ligne //
 - Finit à la fin de la ligne
 - Le commentaire de bloc
 - Débute par /*
 - Finit par */
 - Autant de lignes que voulu

```
Programme.java
01 import java.util.Scanner;
02
03 public class cours {
04     public static void main(String[] args) {
05         int a=0,b=0;
06         Scanner entree = new Scanner(System.in);
07
08         // Début du programme ← Commentaire de fin de ligne
09         System.out.print("Entrez un entier : ");
10         a = entree.nextInt();
11         /*
12         System.out.print("Entrez un autre entier : ");
13         b = entree.nextInt();
14         */
15         System.out.println("Vous avez entré : "+a+ " et "+b);
16     }
17 }
```

2 // [LES BLOCS]

- Délimités par des accolades {}
- Permet de définir la portée de :
 - Certaines instructions
 - toutes les variables
- Plusieurs blocs peuvent être imbriqués
- Les blocs enfants ont accès aux variables de leurs parents
- Il est interdit pour un bloc enfant de redéclarer une variable d'un ancêtre

Programme java

```
01 public class cours {  
02     public static void main(String[] args) {  
03         int a = 10;  
04         {  
05             int b = a * 2;  
06             System.out.println("b = "+b);  
07         }  
08         System.out.println("b = "+b);  
09     }  
10 }  
11 }  
12 }
```

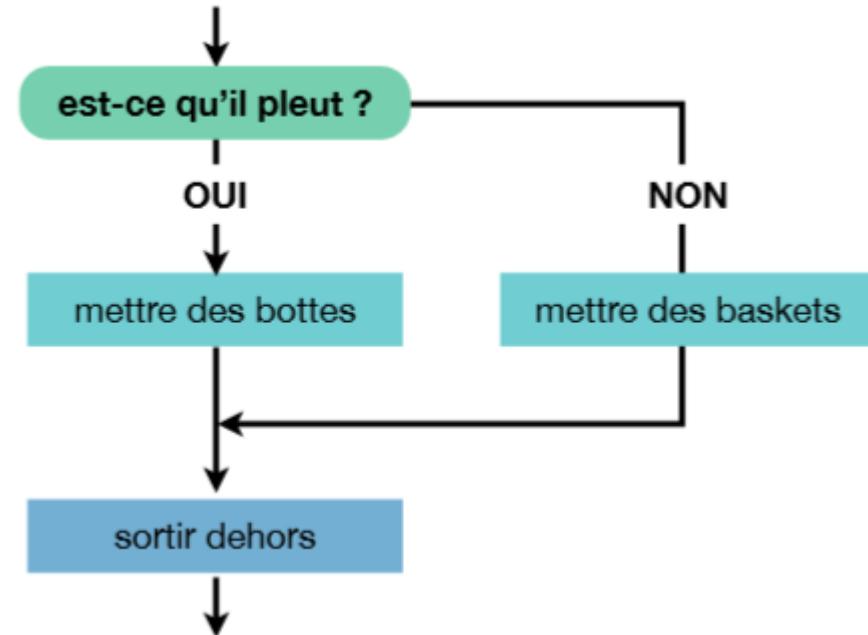
NE COMPILE PAS

La variable b n'existe que pour le bloc interne uniquement

La seule erreur de ce programme est dans cette tentative d'accès à une variable inconnue pour ce bloc.

3 // [INSTRUCTIONS CONDITIONNELLES]

- il est souvent utile de n'exécuter un morceau de code que sous certaines conditions
- Possible en java grâce aux instructions **if**, **else** et **else if**
- Principe : **Si condition, alors ... Sinon ...**
- Exemple :
 - Si il pleut,
 - Alors je mets des bottes
 - Sinon
 - Je mets des baskets



3 // [INSTRUCTIONS CONDITIONNELLES]

- il est souvent utile de n'exécuter un morceau de code que sous certaines conditions
 - Possible en java grâce aux instructions **if**, **else** et **else if**
 - Principe : **Si condition, alors ... Sinon**
-
- Exemple 2 :
 - Si la température est supérieure à 20°C,
 - Alors il fait bon pour la saison
 - Sinon
 - L'hiver arrive si vite

Programme java

```
01 import java.util.Scanner;
02
03 public class cours {
04     public static void main(String[] args) {
05         int temperature;
06         Scanner entree = new Scanner(System.in);
07
08         System.out.print("Entrez une température :");
09         temperature = entree.nextInt();
10
11         if(temperature>20){
12             System.out.println("Il fait bon pour la saison");
13         } else {
14             System.out.println("L'hiver arrive si vite");
15         }
16     }
17 }
```

3 // [INSTRUCTIONS CONDITIONNELLES]

- La condition s'écrit obligatoirement entre parenthèses
- L'expression dans le **if()** doit nécessairement avoir un type **booléen**
- Un **bloc {}** ou bien une **unique instruction** doit nécessairement suivre un **if()**
- La partie **else** est facultative, mais, si présente, doit être suivie d'un **bloc {}** ou d'une **unique instruction**

Fragment java

```
01 int a;  
02  
03 a = 4;  
04 if(a+2){  
05     System.out.println("Vrai");  
06 }
```

Résultat de type int et non boolean

NE COMPILE PAS

Fragment java

```
01 int a;  
02  
03 a = 4;  
04 if(a==4)  
05     System.out.println("Vrai");  
06     System.out.println("Vrai 2");  
07 else  
08     System.out.println("Faux");
```

Le if() s'arrête ligne 5, sinon il faut un bloc

NE COMPILE PAS

3 // [INSTRUCTIONS CONDITIONNELLES]

■ Opérateurs de comparaison

| Opérateur | Pour les valeurs numériques | Pour les caractères |
|--------------------|-----------------------------|-------------------------------------|
| <code>==</code> | égal | égal |
| <code><</code> | inférieur strictement | de codage UTF-16 plus petit |
| <code><=</code> | inférieur ou égal | de codage UTF-16 plus petit ou égal |
| <code>></code> | supérieur strictement | de codage UTF-16 plus grand |
| <code>>=</code> | supérieur ou égal | de codage UTF-16 plus grand ou égal |
| <code>!=</code> | différent | différent |

`==` : Test d'égalité

`=` : Opérateur d'affectation

3 // [INSTRUCTIONS CONDITIONNELLES]

- Exemple :
 - Pour une variable *note* donnée,
 - Si elle est supérieure à 10, afficher « Matière validée », et si en plus elle est supérieure à 15, afficher « avec mention Bien ».
 - Sinon, afficher « Matière non validée »

| Note | Affichage |
|------|------------------------------------|
| 8 | Matière non validée |
| 11 | Matière validéeMatière non validée |
| 16 | Matière validée avec mention Bien |

Message contradictoire !!!

Fragment java

```
01 if(note>=10)
02     System.out.print("Matière validée");
03     if(note>=15)
04         System.out.println(" avec mention Bien");
05 else
06     System.out.println("Matière non validée");
```

BOUGÉ

3 // [INSTRUCTIONS CONDITIONNELLES]

- Exemple :
 - Pour une variable *note* donnée,
 - Si elle est supérieure à 10, afficher « Matière validée », et si en plus elle est supérieure à 15, afficher « avec mention Bien ».
 - Sinon, afficher « Matière non validée »

| Note | Affichage |
|------|-----------------------------------|
| 8 | Matière non validée |
| 11 | Matière validée |
| 16 | Matière validée avec mention Bien |

Fragment java

```
01 if(note>=10) { ← Ajout des accolades
02     System.out.print("Matière validée");
03     if(note>=15)
04         System.out.println(" avec mention Bien");
05 } else
06     System.out.println("Matière non validée");
```

3 // [INSTRUCTIONS CONDITIONNELLES]

■ Opérateurs logiques

- Permettent de combiner des opérateurs de comparaison
- Attention à bien utiliser les parenthèses pour identifier la portée des opérateurs

| Opérateur | Signification | Effet |
|-----------|----------------------|---|
| ! | NON logique (unaire) | Inverse le résultat d'une condition |
| && | ET logique (binaire) | Vrai seulement si les deux opérandes sont vraies simultanément. |
| | OU Logique (binaire) | Vrai si au moins une des deux opérandes est vraie. |

| a | !a |
|-------|-------|
| true | false |
| false | true |

| a | b | a && b |
|-------|-------|--------|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

| a | b | a b |
|-------|-------|--------|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

3 // [INSTRUCTIONS CONDITIONNELLES]

■ Quelques exemples

| Algorithmiquement | Condition Java |
|---|--|
| $a \in [5;10]$ | $(a \geq 5) \ \&\& \ (a \leq 10)$ |
| $a \in [5;10] \cup [16;30[$ | $((a \geq 5) \ \&\& \ (a \leq 10)) \ \parallel \ ((a \geq 16) \ \&\& \ a < 30)$ |
| $a \notin [5;10]$ | $!((a \geq 5) \ \&\& \ (a \leq 10))$ |
| Chiffre des unités de a est 3 ou 8 | $(a \% 10 == 3) \ \parallel \ (a \% 10 == 8)$ |
| Un triangle ABC est isocèle | $(\text{dist}(A,B) == \text{dist}(B,C)) \ \parallel \ (\text{dist}(A,B) == \text{dist}(A,C)) \ \parallel \ (\text{dist}(B,C) == \text{dist}(A,C))$ (en supposant d'avoir écrit ce que fait <code>dist()</code> qui n'existe pas en Java) |
| a est un multiple de 7 mais pas de 5 | $(a \% 7 == 0) \ \&\& \ !(a \% 5 == 0)$ |
| Pierre ou Paul doit être majeur mais pas les deux | $((\text{agePierre} \geq 18) \ \parallel \ (\text{agePaul} \geq 18)) \ \&\& \ !((\text{agePierre} \geq 18) \ \&\& \ (\text{agePaul} \geq 18))$ |

3 // [INSTRUCTIONS CONDITIONNELLES]

■ Branchements multiples

- Quand plusieurs **if** portent sur différentes valeurs d'une même expression, on peut utiliser le **switch / case**
- Ne fonctionne pas pour les nombres flottants

Fragment java

```
01 if(a==1){  
02     System.out.print("unique");  
03 }  
04 if(a==2){  
05     System.out.print("paire");  
06 }  
07 if(a==3){  
08     System.out.print("triplet");  
09 }
```

Fragment java

```
01 switch(a){  
02     case 1:  
03         System.out.print("unique");  
04         break;  
05     case 2:  
06         System.out.print("paire");  
07         break;  
08     case 3:  
09         System.out.print("triplet");  
10         break;  
11 }
```

3 // [INSTRUCTIONS CONDITIONNELLES]

- `switch()` accepte n'importe quelle expression de type byte, short, int, long ou String
- Les expressions sont interdites dans les `case`
- Une branche `default` peut être ajoutée si aucun cas ne correspond

The diagram illustrates two Java code snippets related to conditional instructions.

Fragment java

```
01 switch(a*2-5*b){ ← Expression autorisée
02     case 1:
03         System.out.print("unique");
04         break;
05     case 2:
06         System.out.print("paire");
07         break;
08     case 3:
09         System.out.print("triplet");
10         break;
11     default:
12         System.out.print("Je ne sais pas");
13 }
```

A callout box labeled "Expression autorisée" points to the expression `a*2-5*b`.

A callout box labeled "Sera affiché quand le résultat ne vaut ni 1, ni 2, ni 3" points to the `System.out.print("Je ne sais pas")` line.

Fragment java

```
01 switch(a){
02     case b+1: ← Interdit dans un case
03         System.out.print("unique");
04         break;
05     case 2:
06         System.out.print("paire");
07         break;
08     case 3:
09         System.out.print("triplet");
10         break;
11 }
```

A callout box labeled "Interdit dans un case" points to the `case b+1:` line.

An orange diagonal banner with the text "NE COMPILE PAS" points to the entire invalid code block.

3 // [INSTRUCTIONS CONDITIONNELLES]

■ Le mot-clé break

- Permet de sortir du bloc en cours (ici switch, mais pas que)
- Permet donc de savoir quelle(s) instruction(s) doivent être exécutées dans chaque cas

Fragment java

```
01 switch(a){  
02     case 1:  
03         b++;  
04         break;  
05     case 2:  
06         a++; ← Exécuté uniquement pour a = 2  
07     case 3:  
08     case 4:  
09         b--; ← Exécuté pour a = 2 ou a = 3 ou a = 4  
10         break;  
11 }  
12 System.out.println("A = "+a+" B = "+b);
```

| Avant switch() | a=1 b=2 | a=2 b=2 | a=3 b=2 | a=4 b=2 |
|----------------|---------|---------|---------|---------|
| Affichage | A=1 B=3 | A=3 B=1 | A=3 B=1 | A=4 B=1 |

4 // [LES BOUCLES]

- Permettent de répéter un morceau du code source soit
 - Un nombre prédéterminé de fois
 - Tant qu'une condition est respectée
 - Jusqu'à ce qu'une condition ne soit plus respectée
- 3 types de boucles répondant chacune à un de ces besoins

```
while(condition){  
...  
}
```

```
do {  
...  
}while(condition)
```

```
for(...;...;...){  
...  
}
```

4 // [LES BOUCLES]

■ La boucle **while** (Tant que)

- Exécute en boucle un morceau de code **tant que la condition est vraie**
- Peut donc ne **jamais** s'exécuter !
- Le morceau de code exécuté **DOIT** influer sur la condition (sinon boucle infinie)

Programme Java : Recherche la plus grande somme d'entiers consécutifs contenue

```
01 import java.util.Scanner;
02
03 public class cours {
04     public static void main(String[] args) {
05         int nb=0,somme=0,limite; ← nb parcourra les entiers à partir de 0
06         Scanner entree = new Scanner(System.in);
07
08         System.out.print("Entrez un nombre : ");
09         limite = entree.nextInt();
10
11         while(limite-somme>nb){ ← Tant que la distance entre la limite et
12             nb++;
13             somme += nb;
14         }
15         System.out.println("Résultat : "+nb+" somme :" +somme);
16     }
17 }
```

| limite | affichage |
|--------|-------------------------|
| 0 | Résultat : 0 somme :0 |
| 1 | Résultat : 1 somme :1 |
| 5 | Résultat : 2 somme :3 |
| 10 | Résultat : 4 somme :10 |
| 100 | Résultat : 13 somme :91 |

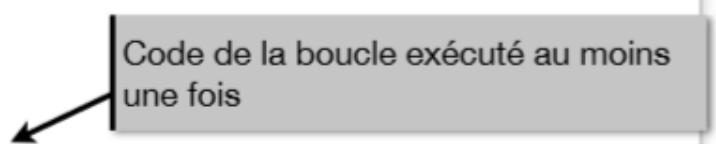
4 // [LES BOUCLES]

■ La boucle **do ... while** (Faire ... tant que)

- Exécute en boucle un morceau de code jusqu'à ce que la condition soit fausse
- S'exécute au moins une fois
- Le morceau de code exécuté DOIT influer sur la condition (sinon boucle infinie)

Programme Java : Saisie vérifiée d'un nombre

```
01 import java.util.Scanner;
02
03 public class cours {
04     public static void main(String[] args) {
05         int nombre;
06         Scanner entree=new Scanner(System.in);
07
08         do {
09             System.out.print("Entrez un nombre entre 5 et 10 :");
10             nombre = entree.nextInt();
11         } while(nombre<5 || nombre>10);
12         System.out.println("nombre = "+nombre);
13     }
14 }
```



Code de la boucle exécuté au moins une fois

4 // [LES BOUCLES]

■ La boucle For (Pour)

- Destiné à parcourir un certain nombre de valeurs déterminé à l'avance
- La syntaxe comporte 3 éléments ordonnés !
 - Une initialisation (exemple `i=0`)
 - Une condition (exemple `i<quantite`)
 - Une opération de fin de parcours (exemple `i++`)
- Peut ne jamais s'exécuter

```
for(i=0;i<quantite;i++)  
    Condition de type while  
    Pas d'incrémentation  
Initialisation de l'indice de boucle
```

Programme Java : Affiche les n premières puissances de 2

```
01 import java.util.Scanner;  
02  
03 public class cours {  
04     public static void main(String[] args) {  
05         int quantite,i,puissance;  
06         Scanner entree = new Scanner(System.in);  
07  
08         System.out.print("Entrez la quantité de puissance de 2 :");  
09         quantite = entree.nextInt();  
10  
11         puissance = 1;  
12         for(i=0;i<quantite;i++){  
13             puissance *= 2;  
14             System.out.println((i+1)+" : "+puissance);  
15         }  
16     }  
17 }
```

Avant de commencer la boucle, on connaît le nombre de répétitions

4 // [LES BOUCLES]

- Chaque élément de la déclaration peut être plus ou moins complexe
- Le point de départ n'est pas forcément 0
- Le sens de progression n'est pas forcément positif
- Le pas de progression n'est pas forcément 1
- Il est possible de déclarer l'indice de boucle dans l'initialisation
- Rien n'impose un indice entier

| boucle | valeur initiale | valeur finale | Pas d'incrémentation | nombre de passages | valeurs prise par i |
|----------------------|-----------------|---------------|----------------------|--------------------|---------------------|
| for(i=0;i<5;i++) | 0 | 4 | 1 | 5 | 0,1,2,3,4 |
| for(i=4;i<=12;i=i+2) | 4 | 12 | 2 | 5 | 4,6,8,10,12 |
| for(i=5;i>0;i=i-1) | 5 | 1 | -1 | 5 | 5,4,3,2,1 |

4 // [LES BOUCLES]

■ Equivalence for / while :

Fragment Java

```
01 puissance = 1;  
02 for(i=0;i<quantite;i++){  
03     puissance *= 2;  
04     System.out.println((i+1)+" : "+puissance);  
05 }
```

Fragment Java

```
01 puissance = 1;  
02 i=0; ← L'initialisation vient avant la boucle  
03 while(i<quantite){  
04     puissance *= 2;  
05     System.out.println((i+1)+" : "+puissance);  
06     i++; ← L'incrémentation est la dernière action  
07 } de la boucle
```

5 // [LES IMBRICATIONS]

- Toutes les imbrications de blocs (conditionnelles, boucles, ou autre) sont possibles, sans restriction de profondeur
 - Exemple : Une boucle peut contenir des instructions conditionnelles contenant d'autres boucles, ...
- Plus il y a de boucles imbriquées, plus la complexité du programme est grande (temps de calcul)
- Il faut absolument faire attention aux accolades pour bien délimiter les blocs (et aux indentations !)

Programme Java : Recherche la plus grande somme d'entiers consécutifs contenue

```
01 import java.util.Scanner;
02 public class cours {
03     public static void main(String[] args) {
04         int nb=0,somme=0,limite,multi3=0;
05         Scanner entree = new Scanner(System.in);
06         System.out.print("Entrez un nombre : ");
07         limite = entree.nextInt();
08         while(limite-somme>nb){
09             nb++;
10             somme += nb;
11             if(nb%3==0) multi3++;
12         }
13         System.out.println("Résultat : "+nb+" somme :"+somme);
14         System.out.println("Avec "+multi3+" multiple(s) de 3");
15     }
16 }
17 }
```

Pour compter en plus les multiples de
3

CONDITIONS & BOUCLES

- 1 / LES COMMENTAIRES**
- 2 / LES BLOCS**
- 3 / LES INSTRUCTIONS CONDITIONNELLES**
- 4 / LES BOUCLES**
- 5 / LES IMBRICATIONS**

QUESTIONS ?

PROGRAMMATION ORIENTÉE OBJET

1 ///// [/REPRÉSENTER UNE DATE]

- Solution 1 : 3 entiers
 - int jour = 2;
 - int mois = 10;
 - int annee = 2018;
- Avantages :
 - Simple à mettre en place
 - Simple à manipuler
 - Chaque donnée porte un nom explicite
- Inconvénients :
 - Les 3 variables sont indépendantes
 - Gérer plusieurs dates nécessite une gestion complexe des noms de variable

1 // [REPRÉSENTER UNE DATE]

- Solution 2 : 1 chaîne de caractères
 - String date = «2/10/2018»;
- Avantages :
 - Simple à mettre en place
 - Tout est contenu dans une unique variable
- Inconvénients :
 - Difficile d'accéder aux données de la date
 - Difficile d'effectuer des opérations sur la date

1 ///// [/REPRÉSENTER UNE DATE]

- Solution 3 : 1 tableau
 - `int date[] = {2,10,2018};`
- Avantages :
 - Simple à mettre en place
 - Accès aux données simple
 - Tout est contenu dans une unique variable
- Inconvénients :
 - Les données ne sont pas explicitement nommées
 - Toutes les données doivent être du même type

1 ///// [/REPRÉSENTER UNE DATE]

- Bilan, il manque un outil qui permette de
 - Regrouper plusieurs données de types différents en une seule variable
 - Autorise un nommage explicite des données
 - Soit simple à mettre en place
 - Soit simple à manipuler
- Solution : les classes
 - 1 classe = description d'une entité complexe de la vraie vie (ou pas !)
 - 1 objet = une instance de classe

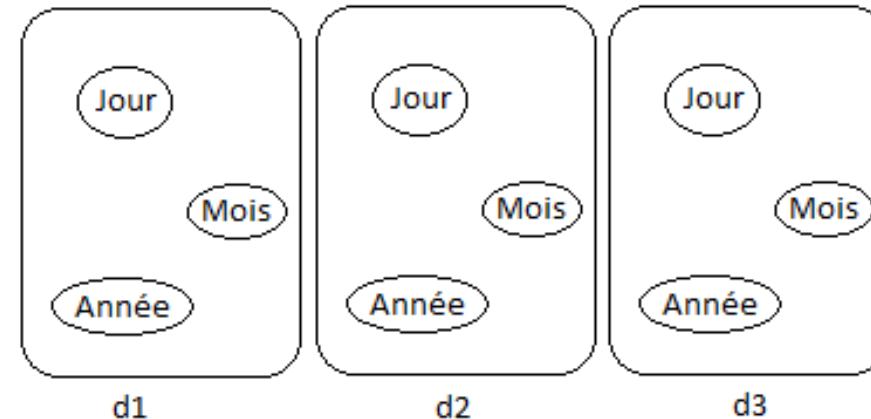
Date.java

```
01 public class Date {  
02     int jour;  
03     int mois;  
04     int annee;  
05 }
```

2 // [LA NOTION DE CLASSE]

■ Une classe

- correspond à un type personnalisé
 - constitué d'un assemblage de données : les propriétés
- Une variable de type Date est appelée instance de la classe Date
- Les variables dont le type est une classe sont appelées des objets
- Bien sûr, plusieurs objets peuvent être des instances de la même classe



2 // [LA NOTION DE CLASSE]

| Une classe | Un objet |
|---|--|
| Est un type (personnalisé) | Est une variable (instance de classe) |
| Décrit une famille d'objets du monde réel | Donne vie à UN objet décrit par une classe |
| N'occupe aucune place mémoire | Occupe de la place en mémoire |
| Ne peut donc pas être manipulée | Peut être manipulé comme une variable |

2 // [LA NOTION DE CLASSE]

■ Définir une classe

- 1) Grâce au mot-clé « class » (sans e)
- 2) Suivi du nom de la classe (choisi par le programmeur)
- 3) Puis d'un bloc qui décrit son contenu

Mot-clé

Nom
↓
class

Bloc
descriptif

A notre niveau : 1 classe = 1 fichier qui porte son nom !

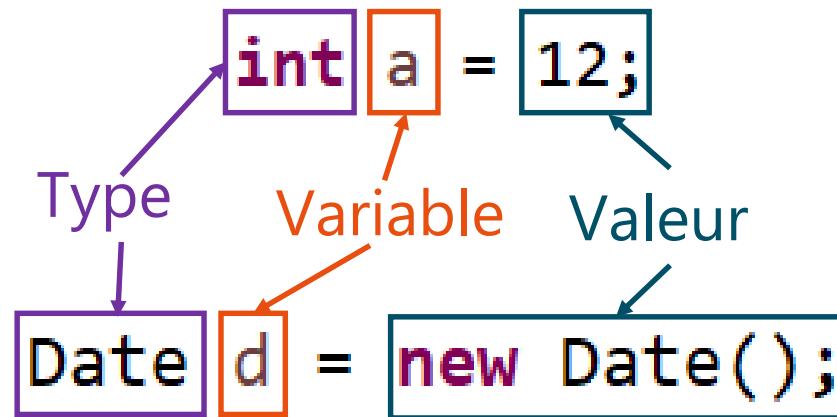


2 // [LA NOTION DE CLASSE]

- Créer un objet (instance de classe)

- 1) On déclare une variable de manière classique (`<type> <variable>;`)
 - 2) On construit un objet qu'on associe à cette variable (opérateur `new`)

- Analogie avec les types primitifs :



- Le mot-clé `new` a deux intérêts :

- manipuler de très gros objets
 - gérer le concept du **polymorphisme** (à découvrir prochainement)

2 // [LA NOTION DE CLASSE]

■ Manipuler une instance de classe

- Chaque objet possède son propre jeu de données (jour, mois, année pour une Date)
- Les propriétés sont accessibles grâce à l'opérateur . (d.jour par exemple)
- Les propriétés s'utilisent à partir de leurs types respectifs (d.jour est un int)

```
public class MainDate {  
    public static void main(String[] args) {  
        Date date1 = new Date();  
        Date date2 = new Date();  
  
        // Changement de la date 1  
        date1.jour=8;  
        date1.mois=10;  
        date1.annee=2018;  
  
        // Changement de la date 2  
        date2.jour=1;  
        date2.mois=1;  
        date2.annee=2000;  
  
        // Affichages  
        System.out.println("Date 1 : "+date1.jour+  
                           "/" +date1.mois+  
                           "/" +date1.annee);  
        System.out.println("Date 2 : "+date2.jour+  
                           "/" +date2.mois+  
                           "/" +date2.annee);  
    }  
}
```

2 // [LA NOTION DE CLASSE]

```
Date d = new Date();
```

- ATTENTION : d n'est pas l'objet lui-même, mais une référence

- Consequence 1 : on ne peut pas manipuler d, avant « new »

```
Date d;  
d.jour = 12;
```

The local variable d may not have been initialized

- Consequence 2 : 2 variables peuvent faire référence à la même date

```
Date d = new Date();  
d.jour=1;  
d.mois=1;  
d.annee=2018;  
System.out.println("Date = "+d.jour+"/"+d.mois+"/"+d.annee);  
Date d2 = d;  
d2.jour=2;  
d2.mois=2;  
d2.annee=2019;  
System.out.println("Date = "+d.jour+"/"+d.mois+"/"+d.annee);
```

Date = 1/1/2018
Date = 2/2/2019

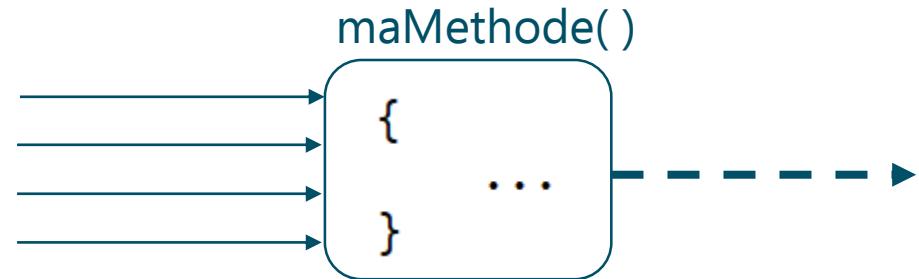
2 // [LA NOTION DE CLASSE]

- Questions dont vous avez maintenant la réponse :

- Pourquoi utiliser les classes ?
- Où et comment créer une classe ?
- Qu'est-ce qu'un objet ?
- Quel est le nom des variables contenues dans un objet ?
- Qu'est-ce qu'une référence ?
- A quoi sert le mot-clé « new » ?
- Comment accéder aux propriétés d'un objet ?

3 // [MÉTHODES DE CLASSE]

- Rappel : les **classes** ont des propriétés
- On peut aussi leur attribuer des **méthodes**
 - Outils permettant d'effectuer une **action** et/ou de **produire un résultat**
 - **Blocs** avec un nom, des entrées, et une sortie
 - Définis **dans** la classe
 - Repérés grâce à l'**opérateur ()**



- Ont accès aux **propriétés de la classe**
- S'utilisent comme les **propriétés**,
Mais avec l'**opérateur ()**

```
Date date1 = new Date();
date1.estBissextile(); }
```

```
class Date {
    int jour;
    int mois;
    int annee;
}
```

```
class Date {

    // Méthode indiquant si l'année
    // de la date est bissextile !
    boolean estBissextile() {

        ...

    }

    // Propriétés de la classe
    int jour;
    int mois;
    int annee;
}
```

3 // [MÉTHODES DE CLASSE]

- Crédit d'une méthode :

- 1) On choisit le nom qu'on souhaite donner à la méthode
 - Mêmes règles et bonnes pratiques que pour les propriétés !
 - 2) On détermine le type du résultat généré par la méthode
 - Un seul résultat généré
 - Type primitif ou non
 - 3) On définit les entrées nécessaires à la méthode
 - Autant qu'on veut,
 - Avec les types qu'on veut
 - 4) On établit le code source du bloc associé
 - Exactement de la même façon que ce que vous connaissez

```
class Date{  
  
    int jour;  
    int mois;  
    int annee;  
}
```

3 // MÉTHODES DE CLASSE

- Le mot clé **return** :
 - Permet de générer le résultat d'une méthode
 - Précède une variable ou une valeur **du bon type**
 - Interrompt l'exécution de la méthode
- Le mot clé **void** :
 - Peut remplacer le type du résultat généré
 - Indique qu'une méthode ne génère **aucun résultat**
 - Utile lorsqu'une méthode ne fait qu'agir sur les propriétés

```
class Date{  
    boolean estBissextile()  
{  
        if(annee%4==0)  
            return true;  
        else  
            return false;  
    }  
  
    int jour;  
    int mois;  
    int annee;  
}
```

3 // [MÉTHODES DE CLASSE]

- La gestion des **paramètres** :

- Définissent les **entrées** de la méthode
- On en met autant qu'on veut !
- Séparés par une virgule
- Déclarés dans les ()
- Type et nom pour chacun d'entre eux

- Le mot-clé **this** :

- Désigne l'**objet** propriétaire de la méthode
- Permet de lever l'ambiguïté avec les paramètres

```
void changer(int nouveauJour,  
            int nouveauMois,  
            int nouvelleAnnee) {  
    jour = nouveauJour;  
    mois = nouveauMois;  
    annee = nouvelleAnnee;  
}
```

```
void changerThis(int jour, int mois, int annee) {  
    jour = jour;  
    mois = mois;  
    annee = annee;  
}
```

```
void changerThis(int jour, int mois, int annee) {  
    this.jour = jour;  
    this.mois = mois;  
    this.annee = annee;  
}
```

3 // [MÉTHODES DE CLASSE]

- Appeler une méthode
 - Exécute le **bloc associé**
 - Pour **UN** jeu de paramètres donné
 - Pour **UN** objet donné
- Peut se faire
 - hors de la classe
 - depuis la classe (**this** facultatif)

```
// Méthode permettant de changer la valeur d'une date
// avec des noms de paramètres identiques aux propriétés
void changerThis(int jour, int mois, int annee) {
    // Appel de méthode depuis la classe
    // Les paramètres utilisés sont des variables !
    this.changer(jour,mois,annee);
}
```

```
void changer(int nouveauJour,
             int nouveauMois,
             int nouvelleAnnee) {
    jour = nouveauJour;
    mois = nouveauMois;
    annee = nouvelleAnnee;
}
```

```
public static void main(String[] args) {

    // On crée les objets d et d2
    Date d = new Date();
    Date d2 = new Date();

    // La Date d devient 1 Février 2010
    d.changer(1,2,2010);

    // La Date d2 devient 2 Mars 2015
    d2.changer(2,3,2015);
}
```

3 // [MÉTHODES DE CLASSE]

- Paramètres formels :
 - Permettent d'écrire le code associé à une méthode
- Paramètres effectifs
 - Servent lors de l'appel d'une méthode
 - Transmettent leur valeur aux paramètres formels

```
// Méthode permettant de changer la valeur d'une date
// avec des noms de paramètres identiques aux propriétés
void changerThis(int jour, int mois, int annee) {
    // Appel de méthode depuis la classe
    // Les paramètres utilisés sont des variables !
    this.changer(jour,mois,annee);
}
```

```
void changer(int nouveauJour,
            int nouveauMois,
            int nouvelleAnnee) {
    jour = nouveauJour;
    mois = nouveauMois;
    annee = nouvelleAnnee;
}
```

```
public static void main(String[] args) {
    // On crée les objets d et d2
    Date d = new Date();
    Date d2 = new Date();

    // La Date d devient 1 Février 2010
    d.changer(1,2,2010);

    // La Date d2 devient 2 Mars 2015
    d2.changer(2,3,2015);
}
```

3 // [MÉTHODES DE CLASSE]

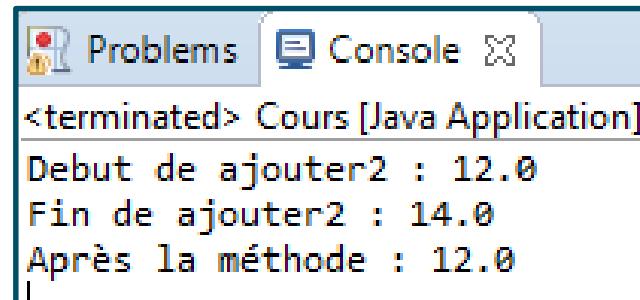
- Zoom sur le passage de paramètres

- Se fait par valeur !

- Types primitifs : on copie la valeur

- Question : à quoi correspond a ?

```
class Maths {  
    void ajouter2(double a) {  
        System.out.println("Début de ajouter2 : "+a);  
        a = a+2;  
        System.out.println("Fin de ajouter2 : "+a);  
    }  
}  
  
class Cours {  
    public static void main(String[] args) {  
        Maths maths = new Maths();  
        Point2D p = new Point2D();  
        maths.centre(p);  
        double a = 12;  
        maths.ajouter2(a);  
        System.out.println("Après la méthode : "+a);  
    }  
}
```



3 //// [MÉTHODES DE CLASSE]

■ Zoom sur le passage de paramètres

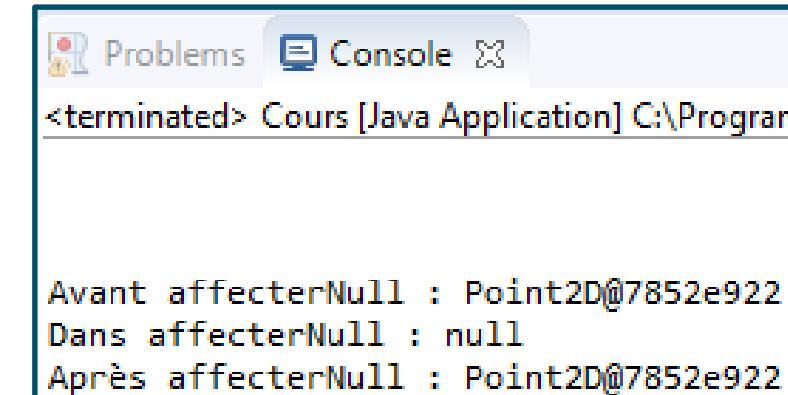
- Se fait par valeur !

- Types primitifs : on copie la valeur
- Types complexes : on copie la référence

```
class Point2D{  
    int x,y;  
}  
  
class Maths {  
    void affecterNull(Point2D p) {  
        p = null;  
        System.out.println("Dans affecterNull : "+p);  
    }  
}
```

```
class Cours {  
    public static void main(String[] args) {  
  
        Maths maths = new Maths();  
        Point2D p = new Point2D();  
        p.x=-5; p.y=8;
```

```
        System.out.println("Avant affecterNull : "+p);  
        maths.affecterNull(p);  
        System.out.println("Après affecterNull : "+p);  
    }  
}
```



3 //// [MÉTHODES DE CLASSE]

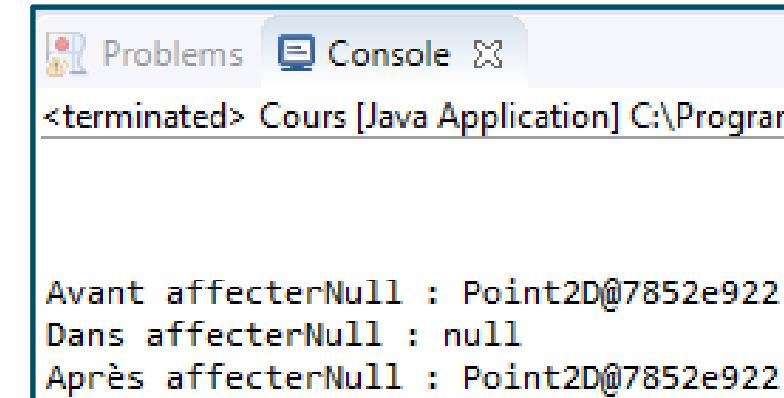
■ Zoom sur le passage de paramètres

- Se fait par valeur !

- Types primitifs : on copie la valeur
- Types complexes : on copie la référence

```
class Point2D{  
    int x,y;  
}  
  
class Maths {  
    void affecterNull(Point2D p) {  
        p = null;  
        System.out.println("Dans affecterNull : "+p);  
    }  
    void centrer(Point2D p) {  
        p.x = 0;  
        p.y = 0;  
        System.out.println("Dans centrer : "+p.x+","+p.y);  
    }  
}
```

```
class Cours {  
    public static void main(String[] args) {  
  
        Maths maths = new Maths();  
        Point2D p = new Point2D();  
        p.x=-5; p.y=8;  
  
        System.out.println("Avant centrer : "+p.x+","+p.y);  
        maths.centre(p);  
        System.out.println("Après centrer : "+p.x+","+p.y);  
  
        System.out.println("Avant affecterNull : "+p);  
        maths.affecterNull(p);  
        System.out.println("Après affecterNull : "+p);  
    }  
}
```



3 //// [/MÉTHODES DE CLASSE]

■ Surcharge de méthodes

- Créer plusieurs méthodes avec le même nom
- Mais avec une liste différente de paramètres formels

■ Retour sur la classe Date :

```
void changer(int nouveauJour,  
            int nouveauMois,  
            int nouvelleAnnee) {  
    jour = nouveauJour;  
    mois = nouveauMois;  
    annee = nouvelleAnnee;  
}
```

```
void changer(int nouveauJour) {  
    jour = nouveauJour;  
}
```

```
void changer(int nouveauMois) {  
    mois = nouveauMois;  
}
```

```
void changer(int nouveauJour,int nouveauMois) {  
    jour = nouveauJour;  
    mois = nouveauMois;  
}
```

```
public class Cours {  
  
    public static void main(String[] args) {  
        Date d = new Date();  
        // La Date d devient 1 Février 2010  
        d.changer(1,2,2010);  
  
        // La Date d devient 20 Février 2010  
        d.changer(20);  
  
        // La Date d devient 8 Avril 2010  
        d.changer(8,4);  
    }  
}
```

3 // [LES MÉTHODES DE CLASSE]

- Questions dont vous avez maintenant la réponse :

- Qu'est-ce qu'une méthode ?
- Comment préciser le type du résultat généré ?
- A quoi sert le mot-clé « void » ?
- Comment quitter une méthode en générant un résultat ?
- Comment exécuter le bloc associé à une méthode ?
- Quelle est la différence entre paramètres formels et effectifs ?
- Qu'implique un passage par valeur ?
- A quoi sert le mot-clé « this » ?
- Qu'est-ce que la surcharge ?

4 // [CONSTRUCTEUR(S)]

■ Question :

```
public static void main(String[] args) {  
  
    // On crée les objets d et d2  
    Date d = new Date();  
    Date d2 = new Date();  
  
    // La Date d devient 1 Février 2010  
    d.changer(1,2,2010);  
  
    // La Date d2 devient 2 Mars 2015  
    d2.changer(2,3,2015);  
  
}
```

Que valent d et d2 ici ?

■ Réponse : ça dépend du constructeur !

4 // [CONSTRUCTEUR(S)]

■ Retour sur le mot-clé new

- Enclenche l'appel d'un **constructeur** qui :
 - alloue la mémoire et l'associe à la variable
 - attribue une valeur à chaque propriété de l'instance
 - peut effectuer d'autres actions prédéterminées

■ Un **constructeur** est une méthode particulière

- qui porte le **même nom que la classe**
- qui ne renvoie rien (**même pas void**)
- qui **PEUT** avoir des arguments
- qui existe sans qu'on ait à le définir
- qui ne peut être appelée explicitement

```
public static void main(String[] args) {  
  
    // On crée les objets d et d2  
    Date d = new Date();  
    Date d2 = new Date();  
  
    // La Date d devient 1 Février 2010  
    d.changer(1,2,2010);  
  
    // La Date d2 devient 2 Mars 2015  
    d2.changer(2,3,2015);  
}
```

```
class Date {  
  
    // Propriétés de la classe  
    int jour;  
    int mois;  
    int annee;  
  
    ...  
}
```

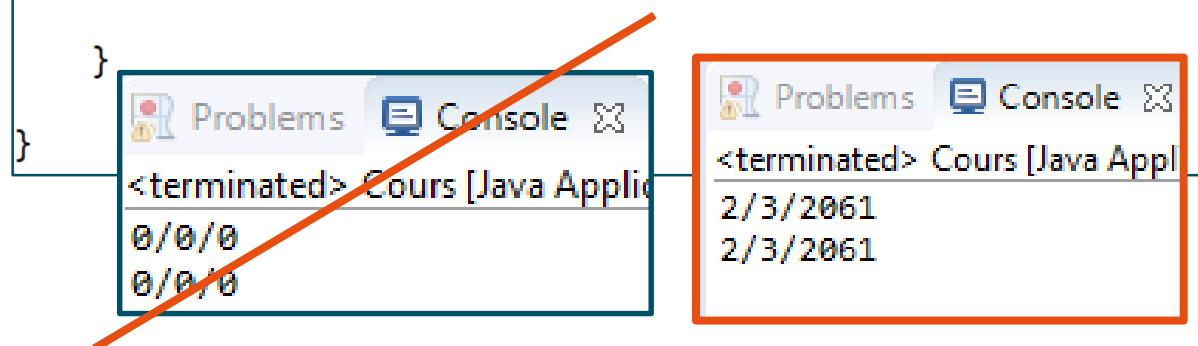
```
class Date {  
  
    // Propriétés de la classe  
    int jour;  
    int mois;  
    int annee;  
  
    Date(){  
    }  
  
    ...  
}
```

4 // [CONSTRUCTEUR(S)]

- Le constructeur par défaut
 - Rappel : existe implicitement
 - Initialise toutes les propriétés à :
 - 0 pour les types primitifs
 - null pour les références
 - Peut être réécrit :

```
class Date {  
  
    Date(){  
        jour=2;  
        mois=3;  
        annee=2061;  
    }  
    ...  
}
```

```
public class Cours {  
  
    public static void main(String[] args) {  
  
        Date d = new Date();  
        Date d2 = new Date();  
        System.out.println(d.jour+"/"+d.mois+"/"+d.annee);  
        System.out.println(d2.jour+"/"+d2.mois+"/"+d2.annee);  
  
        // La Date d devient 1 Février 2010  
        d.changer(1,2,2010);  
  
        // La Date d2 devient 2 Mars 2015  
        d2.changer(2,3,2015);  
    }  
}
```



4 // [CONSTRUCTEUR(S)]

■ Surcharge de constructeur

- Bien sûr possible (c'est une méthode)
 - En respectant la même syntaxe
 - En ajoutant des arguments

■ Le choix du constructeur

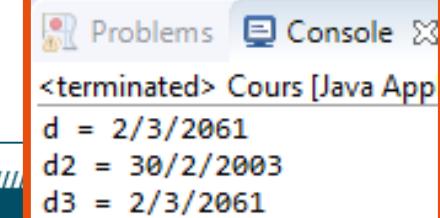
- Se fait à l'appel de **new**
 - via les paramètres effectifs
- Détermine l'initialisation de l'instance

```
Date(Date d){  
    jour=d.jour;  
    mois=d.mois;  
    annee=d.annee;  
}
```

```
Date(){  
    jour=2;  
    mois=3;  
    annee=2061;  
}
```

```
Date(int jour,int mois,int annee){  
    this.jour=jour;  
    this.mois=mois;  
    this.annee=annee;  
}
```

```
public class Cours {  
  
    public static void main(String[] args) {  
        Date d = new Date();  
        Date d2 = new Date(30,02,2003);  
        Date d3| = new Date(d);  
  
        System.out.println("d = "+d.jour+"/"+  
                           d.mois+"/"+d.annee);  
        System.out.println("d2 = "+d2.jour+"/"+  
                           d2.mois+"/"+d2.annee);  
        System.out.println("d3 = "+d3.jour+"/"+  
                           d3.mois+"/"+d3.annee);  
    }  
}
```



The screenshot shows the Java application window with the output of the console. The code creates three Date objects: d (without parameters), d2 (with parameters for day 30, month 2, year 2003), and d3 (with a reference to d). The console output shows the formatted dates: "d = 2/3/2061", "d2 = 30/2/2003", and "d3 = 2/3/2061".

```
Problems Console <terminated> Cours [Java App]  
d = 2/3/2061  
d2 = 30/2/2003  
d3 = 2/3/2061
```

4 // [CONSTRUCTEUR(S)]

■ Attention :

**La classe ne définit pas
de constructeur**



Java fabrique automatiquement un constructeur par défaut sans paramètre

**La classe définit
au moins un constructeur**



Le constructeur par défaut n'est plus disponible.
Java impose l'usage d'un des constructeur décrit dans la classe.

4 // [CONSTRUCTEUR(S)]

- Questions dont vous avez maintenant la réponse :

- A quoi sert un constructeur ?
- Comment différencier un constructeur d'une méthode classique ?
- Doit-on forcément ajouter un constructeur à une classe ?
- Qu'est-ce qu'un constructeur de recopie ?
- A quoi sert le mot-clé **null** ?

5 // [DROITS D'ACCÈS]

■ Question :

```
public class Cours {  
  
    public static void main(String[] args) {  
        Date d = new Date(23,10,2018);  
  
        d.jour = 53; Quel est le problème  
        d.mois = 21;  
    }  
}
```

■ Réponse :

- Le programmeur du main n'est pas le concepteur de Date !
- Il n'est pas forcément au courant de ce qu'il a le droit de faire

■ Solution : Ne pas lui laisser le choix

- Empêcher un accès direct aux propriétés sensibles (lecture et écriture)
- Imposer des vérifications quand une propriété change de valeur

5 // [DROITS D'ACCÈS]

- Idée : ajouter un mot-clé avant la définition de la propriété
- Le mot-clé **public** : `public int jour;`
 - Implicite quand rien n'est précisé `int jour;`
 - N'impose aucune restriction en lecture / écriture
- Le mot-clé **private** : `private int jour;`
 - Interdit l'accès en lecture / écriture depuis l'extérieur de la classe
 - Autorise l'accès depuis l'intérieur de la classe (méthodes)
 - Autorise l'accès depuis une autre instance de la même classe !
- Le mot-clé **protected** :
 - A découvrir plus tard !
- Utilisables sur les méthodes !
- Si c'est possible on met **private** !

```
class Date {  
    // Propriétés de la classe  
    int jour;  
    int mois;  
    int annee;  
}
```

| Droit d'accès | Accessibilité de l'extérieur de la classe | Accessibilité de l'intérieur de la classe |
|---------------|---|---|
| public | Oui | Oui |
| private | Non | Oui |

5 // [DROITS D'ACCÈS]

```
class Date {  
  
    // Propriétés de la classe  
    private int jour;  
    private int mois;  
    private int annee;  
  
    Date(){  
        jour = 23;  
        mois = 10;  
        annee = 2018;  
    }  
  
    void changer(int jour,int mois,int annee) {  
        this.jour=jour;  
        this.mois=mois;  
        this.annee=annee;  
    }  
}
```

```
class Date {  
  
    // Propriétés de la classe  
    private int jour;  
    private int mois;  
    private int annee;  
}
```

```
public class Cours {  
  
    public static void main(String[] args) {  
        Date d = new Date();  
  
        d.jour = 53;  
        d.mois = 21;  
  
        d.changer(53,21,2018);  
    }  
}
```

5 // [DROITS D'ACCÈS]

- Et si je veux utiliser le jour d'une date dans mon main ?

```
public class Cours {  
    public static void main(String[] args) {  
        Date d = new Date();  
  
        if(d.getJour() == 1)  
            System.out.println("c'est le début du mois...");  
    }  
}
```

- Si une propriété peut être divulguée, on doit définir un accesseur (getter)
 - Convention de nom : **getPropriete()**
 - Type de retour = type de la propriété
 - Aucun paramètre
 - Une seule ligne de code !
- Attention aux types non primitifs ... (copie défensive)

```
class Date {  
  
    // Propriétés de la classe  
    private int jour;  
    private int mois;  
    private int annee;  
  
    int getJour() {  
        return jour;  
    }  
}
```

5 // [DROITS D'ACCÈS]

- Et si je veux modifier le mois d'une date dans mon main ?

```
public class Cours {  
    public static void main(String[] args) {  
        Date d = new Date();  
  
        d.setMois(30);  
    }  
}
```

```
class Date {  
  
    // Propriétés de la classe  
    private int jour;  
    private int mois;  
    private int annee;  
  
    void setMois(int mois) {  
        if(mois>0&&mois<=12)  
            this.mois = mois;  
    }  
}
```

- Si une propriété peut être modifiée, on doit définir un mutateur (setter)
 - Convention de nom : **setPropriete()**
 - Type de retour = **void**
 - Paramètre = nouvelle valeur de la propriété
 - Potentiellement des milliers de lignes de code (tests de validité)
 - Attention aux types non primitifs ... (copie défensive)

5 // [DROITS D'ACCÈS]

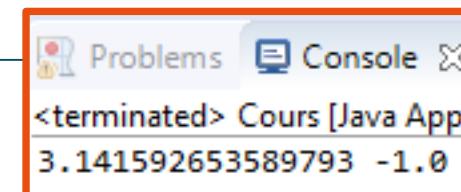
- Questions dont vous avez maintenant la réponse :

- A quoi servent les mots-clés public / private ?
- Qu'est-ce qu'un accesseur ? Un mutateur ?
- Comment correctement définir un getter / un setter ?
- Qu'est-ce qu'une copie défensive ?
- Dans quel cas une méthode doit-elle être private ?

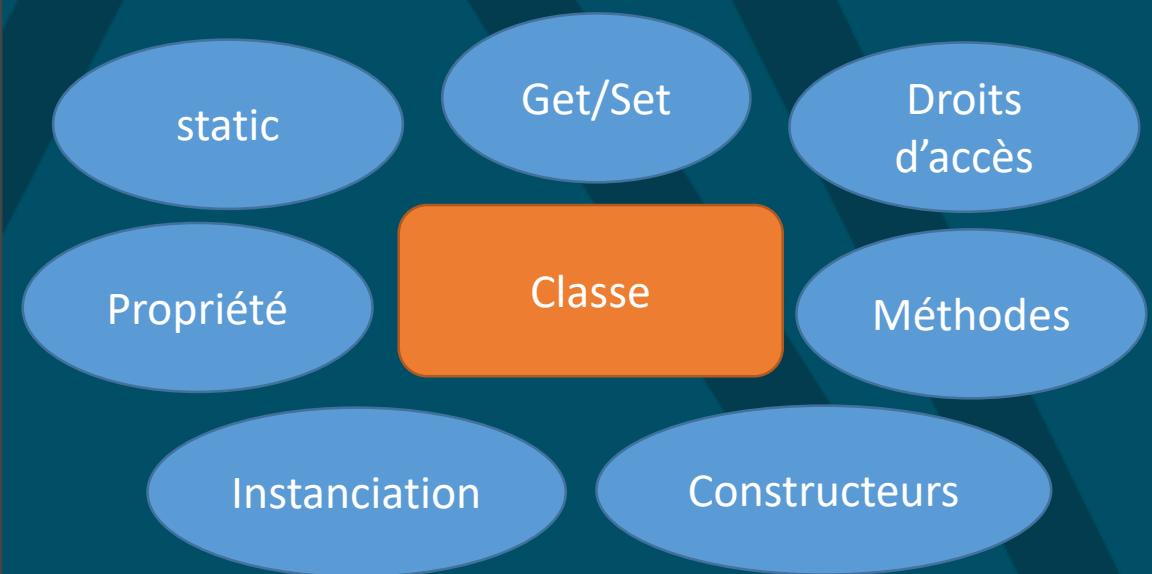
6 // [MÉTHODES ET PROPRIÉTÉS STATIQUES]

- Objectif : Associer une propriété / une méthode à une classe entière (pas à un objet)
 - Les méthodes n'ont donc plus accès aux propriétés d'une instance
 - Les propriétés statiques sont donc communes à toutes les instances de la classe
 - Exemple type : compteur d'instance
 - S'utilise sous la forme **Classe.propriété**, ou bien **Classe.méthode()**
 - Ajouter le mot clé **final** indique en plus que la propriété est **constante**
- Une classe où tout est statique ne peut pas produire d'objet !
- Exemple concret : la classe Math

```
import java.lang.Math;
public class Cours {
    public static void main(String[] args) {
        System.out.println(Math.PI+" "+Math.cos(Math.PI));
    }
}
```



PROGRAMMATION ORIENTÉE OBJET



QUESTIONS ?

TABLEAUX

1 // [INTRODUCTION]

- Un tableau est
 - Une collection : ensemble d'éléments (type primitif ou objet)
 - Homogène : tous les éléments sont du même type
 - d'une quantité fixe d'éléments : le nombre d'éléments est fixé à la création
 - avec un accès indexé : Les éléments sont numérotés
- En java, un tableau est un unique objet. Il faut :
 - Lui donner un nom
 - Définir le type des éléments qui le composent
 - Préciser la quantité de valeurs à mémoriser
 - Réserver l'espace mémoire (mot-clé new)
- Pour manipuler un tableau :
 - On utilise l'opérateur []
 - On retient que le premier élément à l'indice 0

| | Façon 1 | Façon 2 |
|-------------------------------------|--|---|
| Déclaration puis allocation | <code>int[] notes; notes = new int[8];</code> | <code>int notes[]; notes = new int[8];</code> |
| Déclaration et allocation combinées | <code>int[] notes = new int[8];</code> | <code>int notes[] = new int[8]</code> |
| Effet | Crée 8 valeurs de types int : <code>notes[0], notes[1], notes[2], notes[3], notes[4], notes[5], notes[6], notes[7]</code> | |

1 // INTRODUCTION

Programme Java

```
01 import java.util.Scanner;
02
03 public class Cours {
04     public static void main(String[] args) {
05         double [] notes = new double[8];
06         double somme = 0, moyenne = 0;
07         int i;
08         Scanner entree = new Scanner(System.in);
09
10         for(i=0;i<8;i++){
11             do {
12                 System.out.print("Entrez la note "+(i+1)+" entre 0 et 20 : ");
13                 notes[i] = entree.nextDouble();
14             } while(notes[i]<0 || notes[i]>20);
15             somme += notes[i];
16         }
17         moyenne = somme / 8.0;
18         System.out.println("La moyenne de ces notes est : "+moyenne);
19     }
20 }
```

- Remarque : il serait possible de demander le nombre de notes à l'utilisateur
 - Donc d'avoir un tableau dont la taille varie à chaque exécution

1 // [INTRODUCTION]

- Initialiser un tableau à la déclaration
 - Possible via l'opérateur {}
 - Dans ce cas, pas de new []

Propriété length

Programme Java

```
01 public class Cours {  
02     public static void main(String[] args) {  
03         int tab[]={24,78,54,76,12}; ← Initialise le tableau avec 5 éléments  
04         int i;  
05  
06         for(i=0;i<tab.length;i++){  
07             System.out.println("tab["+i+"] = "+tab[i]); ← Affiche  
08         }  
09     }  
10 }
```

Affiche

tab[0] = 24
tab[1] = 78
tab[2] = 54
tab[3] = 76
tab[4] = 12

1 // [INTRODUCTION]

■ Tableau d'objets :

- Il faut appeler new pour le tableau lui-même
- Puis new pour chaque instance (chaque cellule du tableau)
- Chaque case du tableau s'utilise comme une instance à part entière
- Il est possible d'initialiser à la création

```
public class Cours {  
    public static void main(String[] args) {  
        Date[] tab = new Date[3];  
  
    }  
}
```

```
public class Cours {  
    public static void main(String[] args) {  
        Date[] tab = {new Date(),new Date(),new Date()};  
        tab[1].setMois(12);  
    }  
}
```

1 // [INTRODUCTION]

- Un tableau est une référence
- L'opérateur = ne fait donc que copier la référence, pas le contenu
- Pour réaliser une copie, on utilise la méthode **clone()** prédéfinie

```
public class Cours {  
    public static void main(String[] args) {  
        int tab1[]={24,78,54,76,12};  
        int tab2[];  
        int tab3[];  
  
        tab2 = tab1;  
        tab3 = tab1.clone();  
  
        System.out.println("tab1 = "+tab1);  
        System.out.println("tab2 = "+tab2);  
        System.out.println("tab3 = "+tab3);  
  
        for(int d:tab3){  
            System.out.print(d+" ");  
        }  
    }  
}
```

Affiche
tab1 = [I@584479b2
tab2 = [I@584479b2
tab3 = [I@7791c26
24 78 54 76 12

2 // [LA CLASSE ARRAYS]

- Classe de Java contenant des méthodes statiques pour manipuler les tableaux
 - pas d'instance de la classe
 - Méthodes définies pour tous les types primitifs
 - Traitements courants sur les tableaux
- Pour appeler une méthode de la classe Arrays :
 - `Arrays.laMethode()`
- Nécessite l'import de `java.util.Arrays`

2 // [LA CLASSE ARRAYS]

| Opération | Méthodes |
|---|---|
| Tri par ordre ascendant | <pre>void sort(int[] a)</pre> <p>Réalise un tri par ordre croissant du tableau passé en paramètre. int[] a peut être remplacé par long[], short[], char[], byte[], float[], double[]</p> <pre>void sort(int[] a, int fromIndex, int toIndex)</pre> <p>Réalise un tri par ordre croissant du tableau passé en paramètre restreint entre les indices fromIndex et toIndex. int[] a peut être remplacé par long[], short[], char[], byte[], float[], double[]</p> |
| Recherche binaire (sur un tableau trié) | <pre>int binarySearch(int[] a, int key)</pre> <p>Retourne l'indice de la clé si elle est trouvée, une valeur négative sinon (codant le point d'insertion) int[] a peut être remplacé par long[], short[], char[], byte[], float[], double[] et int key par long, short, char, byte, float, double</p> <pre>int binarySearch(int[] a, int fromIndex, int toIndex, int key)</pre> <p>Retourne l'indice de la clé si elle est trouvée, une valeur négative sinon (codant le point d'insertion) int[] a peut être remplacé par long[], short[], char[], byte[], float[], double[] et int key par long, short, char, byte, float, double</p> |

2 // [LA CLASSE ARRAYS]

| Opération | Méthodes |
|---------------------------------|--|
| Test de contenu identique | <code>boolean equals(int[] a1, int[] a2)</code> Retourne vrai si a1 et a2 contiennent les mêmes éléments dans le même ordre. <code>int[] a1</code> et <code>a2</code> peuvent être remplacés par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> (<code>a1</code> et <code>a2</code> doivent être du même type). |
| Remplissage avec une valeur | <code>void fill(int[] a, int val)</code> Remplit le tableau avec la valeur <code>val</code> . <code>int[] a</code> peut être remplacé par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> et <code>int key</code> par <code>long</code> , <code>short</code> , <code>char</code> , <code>byte</code> , <code>float</code> , <code>double</code> , <code>boolean</code> |
| | <code>void fill(int[] a, int fromIndex, int toIndex, int val)</code> Remplit le tableau avec la valeur <code>val</code> pour les éléments entre les indices <code>fromIndex</code> et <code>toIndex</code> . <code>int[] a</code> peut être remplacé par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> et <code>int val</code> par <code>long</code> , <code>short</code> , <code>char</code> , <code>byte</code> , <code>float</code> , <code>double</code> , <code>boolean</code> |
| Copie avec changement de taille | <code>int[] copyOf(int[] original, int newLength)</code> Copie le tableau <code>original</code> en ajustant la taille à <code>newLength</code> soit par troncature, soit par ajout d'éléments à 0. <code>int[] a</code> peut être remplacé par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> . |
| Copie partielle | <code>int[] copyOfRange(int[] original, int from, int to)</code> Copie le tableau <code>original</code> entre les indices <code>from</code> et <code>to</code> . <code>int[] a</code> peut être remplacé par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> . |

2 // [LA CLASSE ARRAYS]

| Opération | Méthodes |
|-----------------------------------|--|
| Conversion en chaîne de caractère | <code>String toString(int[] a)</code> Convertit un tableau en chaîne de caractères au format [, , ,]. <code>int[] a</code> peut être remplacé par <code>long[]</code> , <code>short[]</code> , <code>char[]</code> , <code>byte[]</code> , <code>float[]</code> , <code>double[]</code> , <code>boolean[]</code> . |

```
import java.util.Arrays; ← Obligatoire !  
  
public class Cours {  
    public static void main(String[] args) {  
        byte tab[]={24,78,54,76,12,112,3,78};  
  
        System.out.println(Arrays.toString(tab)); ← Affiche  
    }  
}
```

2 // [LA CLASSE ARRAYS]

Programme Java

```
01 import java.util.Arrays;  
02  
03 public class Cours {  
04     public static void main(String[] args) {  
05         byte tab[]={24,78,54,76,12,112,3,78};  
06  
07         System.out.println("Avant le tri");  
08         System.out.println(Arrays.toString(tab));  
09  
10         Arrays.sort(tab);  
11  
12         System.out.println("Après le tri");  
13         System.out.println(Arrays.toString(tab));  
14     }  
15 }
```

Affiche
Avant le tri
[24, 78, 54, 76, 12, 112, 3, 78]
Après le tri
[3, 12, 24, 54, 76, 78, 78, 112]

Le tri par ordre décroissant est possible, mais nécessite des éléments que nous n'avons pas encore vus.

3 // [LES COLLECTIONS]

- Classes Java permettant de manipuler des **ensembles d'objets**
- Alternative aux tableaux traditionnels
- Basé sur un concept que nous verrons plus tard : **les interfaces**
- Plusieurs variantes en fonction des besoins
 - Map : stockage d'objets sous un format clé / valeur
 - Set : Liste d'objets uniques (doublons interdits)
 - List : Doublons autorisés, et possibilité d'accéder aux éléments par leur indice
- Nombreuses variantes avec des propriétés spécifiques
 - **ArrayList** : Tableau redimensionnable
 - **LinkedList** : Liste chaînée
 - **SortedSet** : Liste triée d'objets
 - **Vector** : Comme **ArrayList**, mais synchronisé
 - ...

3 // [LES COLLECTIONS]

| Collection | Ordonné | Accès direct | Clé / valeur | Doublons | Null | Thread Safe |
|----------------------|---------|--------------|--------------|----------|------|-------------|
| ArrayList | Oui | Oui | Non | Oui | Oui | Non |
| LinkedList | Oui | Non | Non | Oui | Oui | Non |
| HashSet | Non | Non | Non | Non | Oui | Non |
| TreeSet | Oui | Non | Non | Non | Non | Non |
| HashMap | Non | Oui | Oui | Non | Oui | Non |
| TreeMap | Oui | Oui | Oui | Non | Non | Non |
| Vector | Oui | Oui | Non | Oui | Oui | Oui |
| Hashtable | Non | Oui | Oui | Non | Non | Oui |
| Properties | Non | Oui | Oui | Non | Non | Oui |
| Stack | Oui | Non | Non | Oui | Oui | Oui |
| CopyOnWriteArrayList | Oui | Oui | Non | Oui | Oui | Oui |
| ConcurrentHashMap | Non | Oui | Oui | Non | Non | Oui |
| CopyOnWriteArraySet | Non | Non | Non | Non | Oui | Oui |

3 // [LES COLLECTIONS]

- Une classe à connaître : **ArrayList**
 - Tableau d'objet dont la taille est dynamique
- Basé sur un tableau dont la taille s'adapte au nombre d'éléments à stocker
- Syntaxe particulière :
 - Il faut préciser le type d'objet contenu
 - Les types primitifs sont interdits

```
ArrayList<Integer> liste = new ArrayList<Integer>();
```

- L'accès au contenu se fait grâce à la méthode **add** et aux accesseurs mutateurs

```
for(int i=0;i<12;i++) {  
    liste.add(i);  
}  
liste.set(4,100);  
for(int i=0;i<12;i++) {  
    System.out.println(liste.get(i));  
}
```

| |
|-----|
| 0 |
| 1 |
| 2 |
| 3 |
| 100 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |

3 // [LES COLLECTIONS]

| Constructeur | Rôle |
|--------------------------------|---|
| ArrayList() | Créer une instance vide de la collection avec une capacité initiale de 10 |
| ArrayList(int initialCapacity) | Créer une instance vide de la collection avec la capacité initiale fournie en paramètre |

| Méthode | Rôle |
|----------------------------|--|
| boolean add(Object) | Ajouter un élément à la fin du tableau |
| void clear() | Supprimer tous les éléments du tableau |
| Object get(index) | Renvoyer l'élément du tableau dont la position est précisée |
| int indexOf(Object) | Renvoyer la position de la première occurrence de l'élément fourni en paramètre |
| boolean isEmpty() | Indiquer si le tableau est vide |
| int lastIndexOf(Object) | Renvoyer la position de la dernière occurrence de l'élément fourni en paramètre |
| Object remove(int) | Supprimer dans le tableau l'élément fourni en paramètre |
| void removeRange(int, int) | Supprimer tous les éléments du tableau de la première position fournie incluse jusqu'à la dernière position fournie exclue |
| Object set(int, Object) | Remplacer l'élément à la position indiquée par celui fourni en paramètre |
| int size() | Renvoyer le nombre d'éléments du tableau |

3 // [LES COLLECTIONS]

■ ArrayList -> Tableau

```
Object[] tab = liste.toArray();
for(int i=0;i<12;i++) {
    System.out.println((int)tab[i]);
}
```

■ Tableau -> ArrayList

```
String tab[] = {"A","B","C"};
ArrayList<String> liste = new ArrayList<String>();
Collections.addAll(liste,tab);
for(int i=0;i<liste.size();i++) {
    System.out.println(liste.get(i));
}
```

3 // [LES COLLECTIONS]

- Résumé des avantages de l'ArrayList :

- Dynamique : le nombre d'éléments n'a pas besoin d'être fixé à l'avance
- Insertion simplifiée : Possibilité d'insérer un indice au milieu du tableau sans gérer les décalages
- Recherche simplifiée :
 - `contains(Object o)` : indique si un élément est présent dans la liste
 - `indexOf(Object o)` : indique la position d'un élément dans la liste (-1 sinon)
 - `lastIndexOf(Object o)` : indique la dernière position d'un élément
- Facile à manipuler :
 - `subList(int fromIndex, int toIndex)` : construction d'une liste partielle
 - `clear()` : supprimer tous les éléments
 - `remove(int)` : supprimer la i-ème case (la mise à jour est automatique)
 - `removeRange(int,int)` : supprime une succession de cases (idem)

3 // [LES COLLECTIONS]

- Résumé des inconvénients de l'ArrayList :
 - Syntaxe moins intuitive
 - On ne peut plus utiliser les crochets []
 - Les types primitifs sont interdits (il faut utiliser les Wrappers)

RÉFÉRENCES & TABLEAUX

QUESTIONS ?

HÉRITAGE & POLYMORPHISME

1 // [DÉ ET DÉ TRUQUÉ]

- On souhaite programmer un jeu de société dans lequel des dés sont lancés
 - On va donc créer une classe Dé (qui peut avoir un nombre de faces quelconque)

| De |
|------------------|
| -valeur : int |
| -faces : int |
| +De() |
| +De(faces:int) |
| +lancer():void |
| +getValeur():int |

```
public class De {  
    private int valeur;  
    private int faces;  
  
    public De() {  
        faces = 6;  
        lancer();  
    }  
  
    public De(int faces) {  
        this.faces = faces;  
        lancer();  
    }  
  
    public int getValeur() {  
        return valeur;  
    }  
  
    public void lancer() {  
        valeur = (int)(Math.random()*faces)+1;  
    }  
}
```

1 // [DÉ ET DÉ TRUQUÉ]

- On souhaite également pouvoir disposer d'un dé truqué (dé normal mais qui sort une valeur max tous les 4 lancers)
 - On va donc aussi créer une classe DéTruqué

| DéTruque | |
|------------------|--|
| -valeur : int | |
| -faces : int | |
| -lancers:int | |
| +De() | |
| +De(faces:int) | |
| +lancer():void | |
| +getValeur():int | |

```
public class DeTruque {  
    private int valeur;  
    private int faces;  
    private int lancers = 0;  
  
    public DeTruque() {  
        faces = 6;  
        lancer();  
    }  
  
    public DeTruque(int faces) {  
        this.faces = faces;  
        lancer();  
    }  
  
    public int getValeur() {  
        return valeur;  
    }  
  
    public void lancer() {  
        lancers++;  
        if(lancers%4==0)  
            valeur = faces;  
        else  
            valeur = (int)(Math.random()*faces)+1;  
    }  
}
```

1 // [DÉ ET DÉ TRUQUÉ]

- Qu'en est-il du code source ?

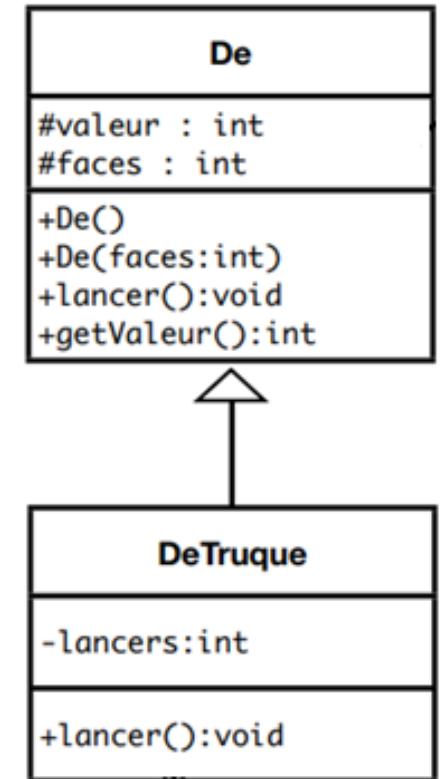
- Quasiment le même ! Copier-coller ? Mauvaise idée 😞

```
01 class De {  
02     private int valeur;  
03     private int faces;  
04  
05     public De(){  
06         faces = 6;  
07         lancer();  
08     }  
09  
10    public De(int faces){  
11        this.faces = faces;  
12        lancer();  
13    }  
14  
15    public void lancer(){  
16        valeur = (int)(Math.random()*faces)+1;  
17    }  
18  
19    public int getValeur(){  
20        return valeur;  
21    }  
22 }
```

```
01 class DeTruque {  
02     private int valeur;  
03     private int faces;  
04     private int lancers = 0;  
05  
06     public DeTruque(){  
07         faces = 6;  
08         lancer();  
09     }  
10    public DeTruque(int faces){  
11        this.faces = faces;  
12        lancer();  
13    }  
14    public void lancer(){  
15        lancers++;  
16        if(lancers%4==0) { valeur = faces; }  
17        else { valeur = (int)(Math.random()*faces)+1; }  
18    }  
19    public int getValeur(){  
20        return valeur;  
21    }  
22 }
```

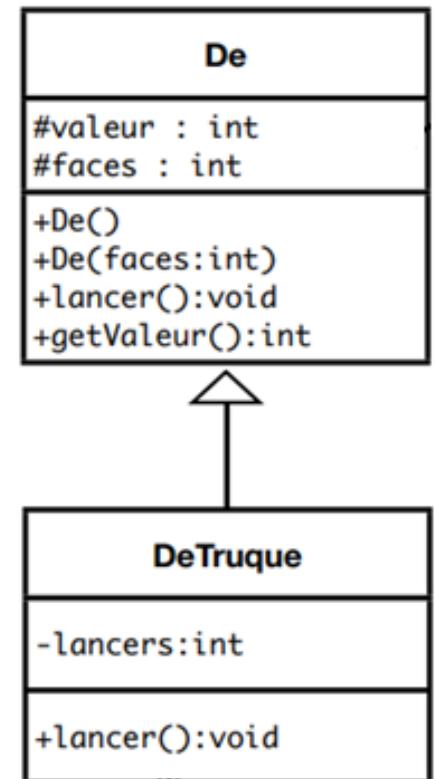
2 // [LA NOTION D'HÉRITAGE]

- Objectif : faire en sorte que
 - Un DéTruqué possède tout ce que possède un Dé (propriétés + méthodes)
 - Un DéTruqué puisse quand même avoir ses spécificités
 - Une propriété lancers qui ne sert qu'à elle
 - Une méthode lancer() différente de celle de la classe Dé
 - Une spécificité d'un DéTruqué soit prioritaire
- Solution : Héritage
 - Un DéTruqué « EST UN » Dé
 - Mais avec ses spécificités
 - La classe Dé sera alors la **classe mère (ou superclasse)**
 - La classe DéTruqué sera alors la **classe fille (ou sousclasse)**



2 // [LA NOTION D'HÉRITAGE]

- Les propriétés de Dé sont maintenant **protected**
 - Nécessaire pour être héritables
- La classe DéTruqué hérite
 - des propriétés **valeur** et **faces**
 - des méthodes **getValeur()** et **lancer()**
- La classe DéTruqué redéfinit
 - sa propre méthode **lancer()** (masquage)
- La classe DéTruqué ne redéfinit pas la méthode **getValeur()**
 - Ce sera donc celle de la classe Dé qui sera utilisée



2 // [LA NOTION D'HÉRITAGE]

```
public class De {  
    protected int valeur;  
    protected int faces;  
  
    public De() {  
        faces = 6;  
        lancer();  
    }  
  
    public De(int faces) {  
        this.faces = faces;  
        lancer();  
    }  
  
    public int getValeur() {  
        return valeur;  
    }  
  
    public void lancer() {  
        valeur = (int)(Math.random()*faces)+1;  
    }  
}
```

Le mot-clé **extends** exprime la relation d'héritage

```
public class DeTruque extends De {  
    private int lancers = 0;  
  
    public void lancer() {  
        lancers++;  
        if(lancers%4==0) {  
            valeur = faces;  
        }  
        else {  
            super.lancer();  
        }  
    }  
}
```

Le mot-clé **super** désigne la classe mère. Ici, **super.lancer()** implique l'exécution de la méthode **lancer()** de la classe Dé

2 // [LA NOTION D'HÉRITAGE]

- Exemple d'utilisation :
 - Comme s'il s'agissait de deux classes indépendantes !

```
public class Main {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        De de1 = new De();  
        DeTruque de2 = new DeTruque();  
  
        for (int i = 0; i < 24; i++) {  
            de1.lancer();  
            de2.lancer();  
  
            System.out.println("lancer " + i + " : " + de1.getValeur() + " // " + de2.getValeur());  
        }  
    }  
}
```

Appelle la méthode lancer() de DeTruque
(masquage)

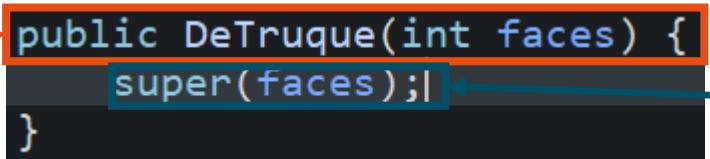
Appelle la méthode
lancer() de De

Lancers truqués

| |
|--------------------|
| lancer 0 : 5 // 5 |
| lancer 1 : 2 // 5 |
| lancer 2 : 3 // 6 |
| lancer 3 : 5 // 6 |
| lancer 4 : 5 // 1 |
| lancer 5 : 5 // 3 |
| lancer 6 : 4 // 1 |
| lancer 7 : 6 // 6 |
| lancer 8 : 4 // 4 |
| lancer 9 : 2 // 3 |
| lancer 10 : 4 // 5 |
| lancer 11 : 3 // 6 |
| lancer 12 : 2 // 1 |
| lancer 13 : 3 // 3 |
| lancer 14 : 1 // 1 |
| lancer 15 : 6 // 6 |
| lancer 16 : 1 // 3 |
| lancer 17 : 1 // 3 |
| lancer 18 : 3 // 4 |
| lancer 19 : 6 // 6 |
| lancer 20 : 3 // 6 |
| lancer 21 : 4 // 6 |
| lancer 22 : 1 // 2 |
| lancer 23 : 1 // 6 |

2 // [LA NOTION D'HÉRITAGE]

- Question : Comment construire un DéTruqué à 8 faces ?
 - Réponse : impossible !
 - Pas de constructeur avec paramètres pour un DéTruqué
- On le rajoute !
 - En faisant un appel explicite au constructeur de Dé
 - `super()` désigne un constructeur de la classe mère

Constructeur de
DéTruqué → 
Appel explicite au
constructeur de Dé

```
public DeTruque(int faces) {  
    super(faces);  
}
```

- Pour construire un objet d'une classe fille
 - Il faut appeler le constructeur de sa classe mère
 - Par défaut, c'est le constructeur par défaut qui est appelé

3 // [LE POLYMORPHISME]

- Les classes interagissent entre elles :
 - Par composition (« est formé de »)
 - Exemple : une Promo « est formée » d'Etudiants
 - Exemple 2 : Une Forme « est formée » de Carres (Tetris)
 - Par héritage (« est un »)
 - Exemple : Un DéTruqué « est un » Dé
 - Puisqu'un DéTruqué « est un » Dé :
 - On peut écrire : `De de3 = new DeTruque();`
 - C'est le polymorphisme

3 // [LE POLYMORPHISME]

```
public class Main {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        De de1 = new De();  
        De de3 = new DeTruque();  
  
        for (int i = 0; i < 24; i++) {  
            de1.lancer();  
            de3.lancer();  
  
            System.out.println("lancer " + i + " : " + de1.getValeur() + " // " + de3.getValeur());  
        }  
    }  
}
```

Appelle la méthode lancer() de De

Appelle la méthode lancer() de DeTruque (polymorphisme)

3 // [LE POLYMORPHISME]

- Et si ma classe DéTruqué
 - contient une méthode tricheur()
 - qui n'existe pas dans la classe Dé
- Erreur : pour qu'il y ait polymorphisme, il faut que la classe de la variable contienne la méthode !
 - Solution possible : cast explicite
 - il faut alors être sur que d3 soit un DeTruque()

```
public class Main {  
    public static void main(String[] args) {  
  
        De de3 = new DeTruque();  
  
        de3.tricheur();  
    }  
}
```

```
public void tricheur() {  
    System.out.println("Je suis un dé tricheur");  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        De de3 = new DeTruque();  
  
        ((DeTruque)de3).tricheur();   
    }  
}
```

HÉRITAGE & POLYMORPHISME

1 / HÉRITAGE
2 / POLYMORPHISME

QUESTIONS ?

ABSTRACTION & INTERFACES

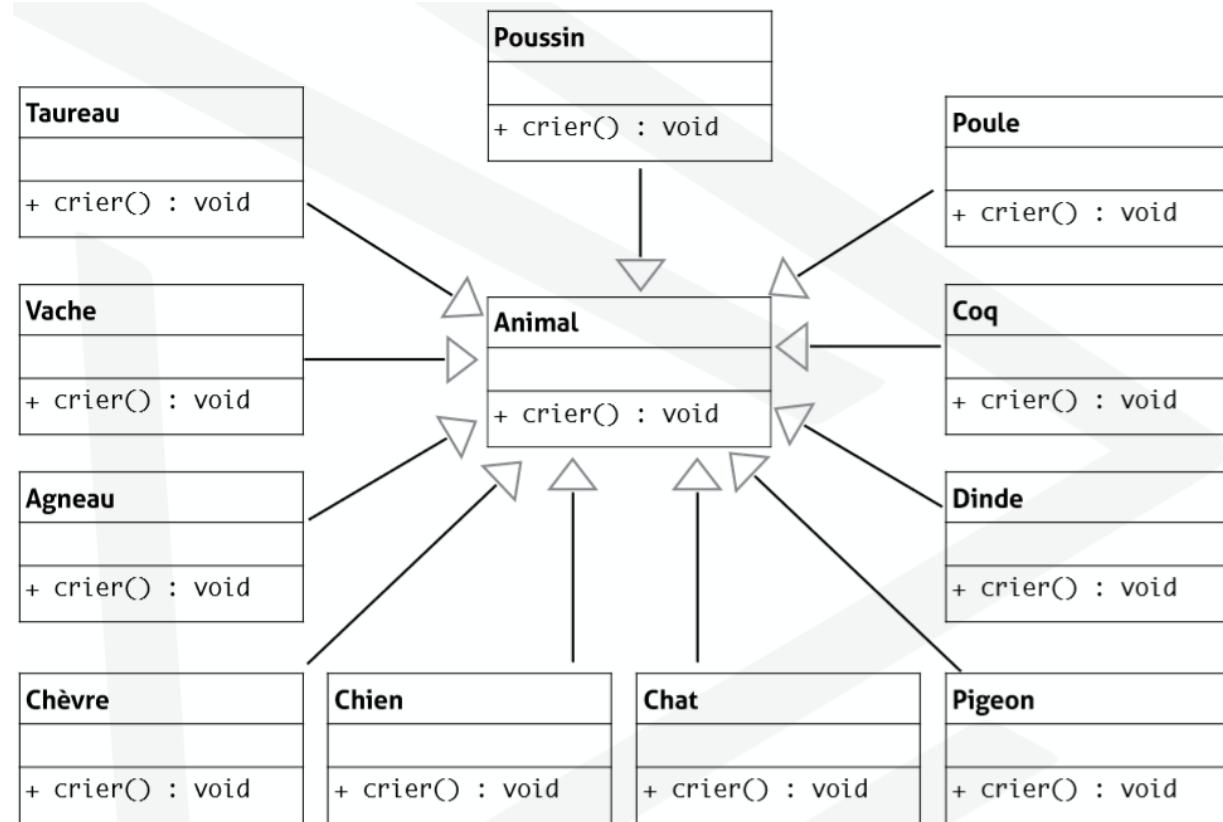
1 // [L'ABSTRACTION]

- Cherchons à modéliser les animaux de la ferme :
 - En particulier leurs cris

| Animal | Cri | Animal | Cri |
|---------|--------------|---------|-----------|
| Poussin | Piou | Chien | Ouaf Ouaf |
| Poule | Cot Cot | Chèvre | Bêê |
| Coq | Cocorico | Agneau | Mèè |
| Dinde | Glouglouglou | Vache | Meuh |
| Pigeon | Roucoule | Taureau | Muu |
| Chat | Miaou | | |

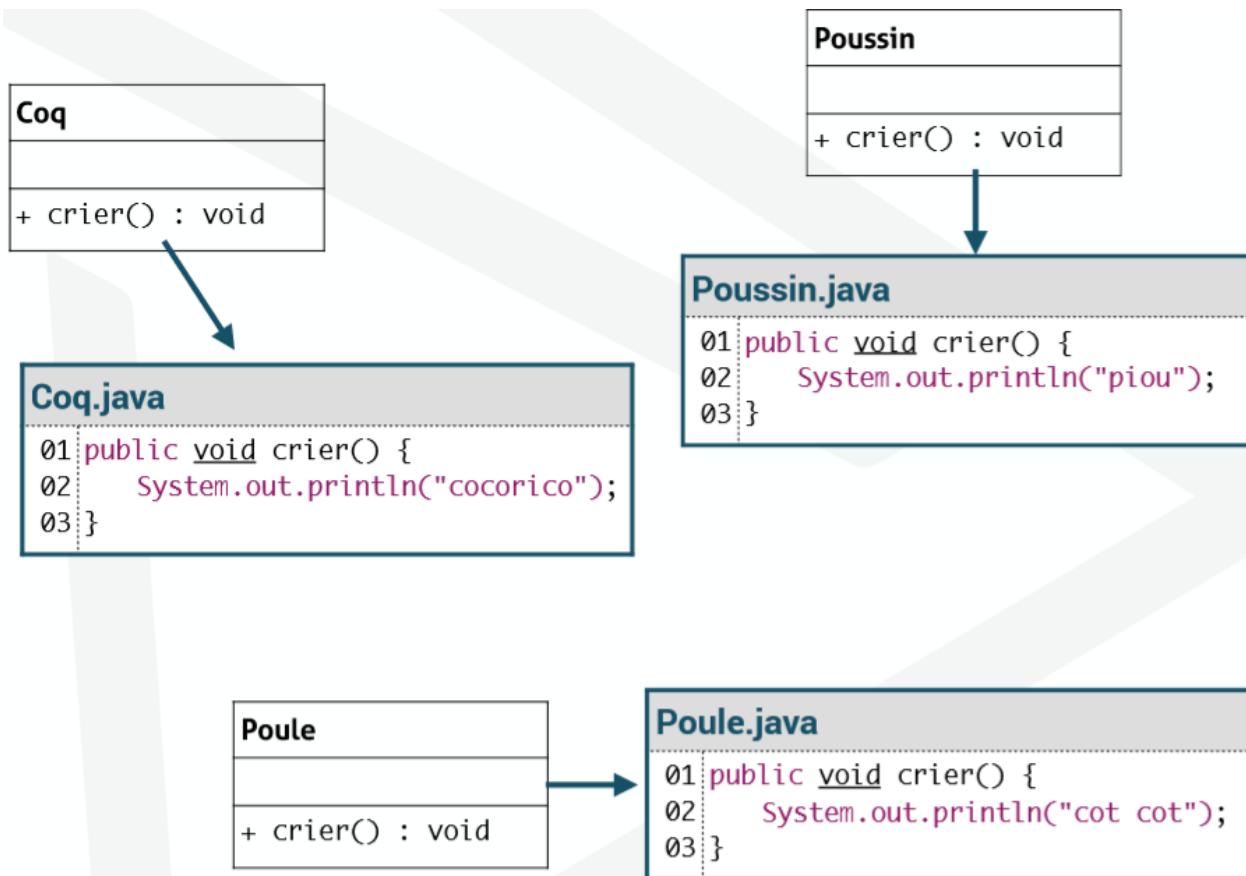
1 // [L'ABSTRACTION]

- Si chaque animal est représenté par une classe, on obtient logiquement le diagramme suivant :



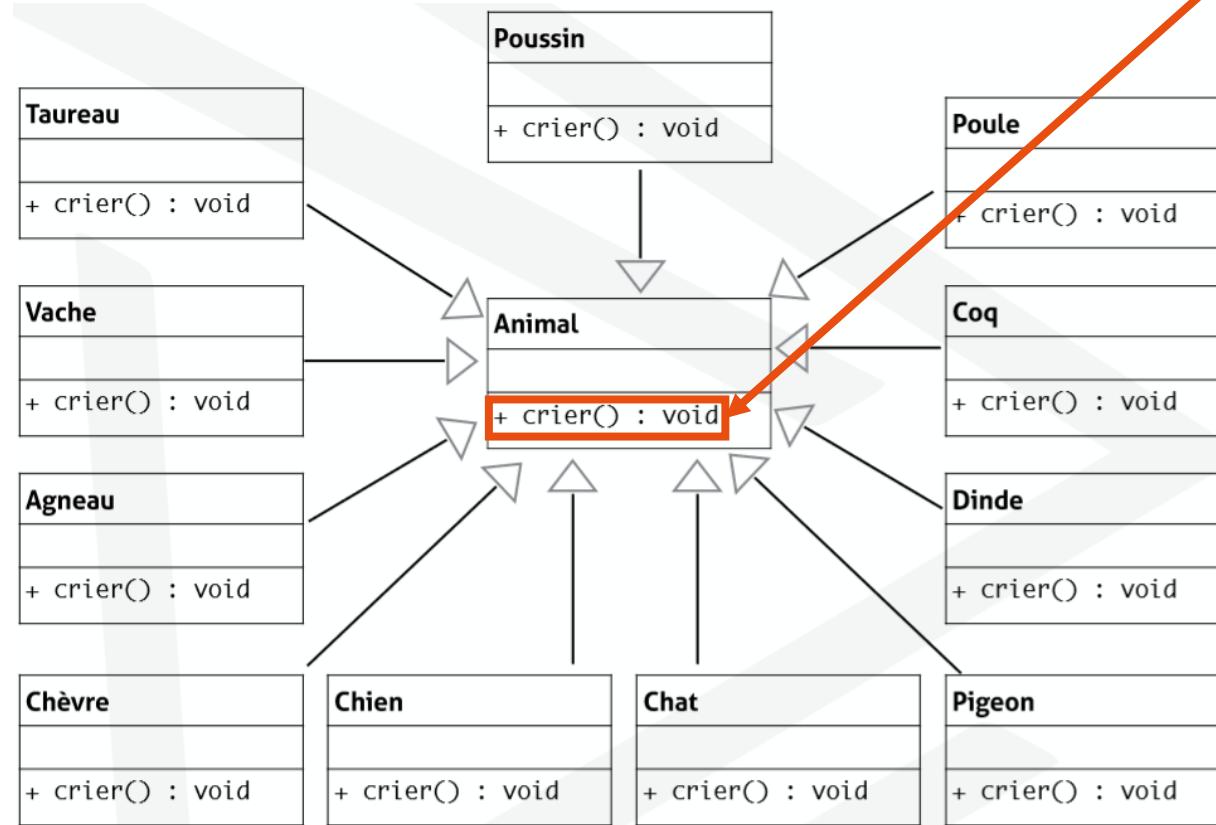
1 // [L'ABSTRACTION]

- Bien sûr, chaque animal ayant son propre cri, il suffit de redéfinir la méthode crier() :



1 // [L'ABSTRACTION]

- Problème : quel est le cri d'un animal ?



Animal.java

```
01 public void crier() {  
02 }
```

Le cri d'un animal c'est ... RIEN ???
Ce n'est pas satisfaisant !

```
public static void main(String[] args) {  
    Animal poussin = new Poussin();  
    poussin.crier();  
  
    Animal animal = new Animal();  
    animal.crier();  
}
```

Autre problème :
Ici on crée un animal, sans savoir lequel ...
Pas satisfaisant non plus

1 // [L'ABSTRACTION]

- Solution : préciser que la classe Animal est **abstraite** c'est-à-dire :
 - une classe normale, avec ses méthodes et propriétés, mais
 - Incomplète (certaines méthodes ne sont pas définies)
 - donc impossible à instancier
- On utilise le mot-clé **abstract** devant le nom de la classe
- Intérêt :
 - Maintenir le polymorphisme
 - Tout en empêchant la création d'un Animal

On peut par exemple créer un tableau d'Animal, mais pour remplir une case, il faut préciser de quel animal il s'agit !

Animal.java

```
01 public abstract class Animal {  
02     public void crier() {  
03     }  
04 }
```

Main.java

```
01 public static void main(String[] args) {  
02     Animal animal = new Animal();  
03     animal.crier();  
04 }
```

 Cannot instantiate the type Animal

1 // [L'ABSTRACTION]

- Une classe peut être abstraite
- Une méthode peut également être abstraite !
 - On ne précise alors que son prototype (visibilité, nom, paramètres, retour)
 - Mais pas son contenu
- Une méthode abstraite doit **FORCEMENT** se trouver dans une classe abstraite
- Intérêt :
 - Montrer qu'on a **volontairement** omis le contenu d'une méthode

Animal.java

```
01 public abstract class Animal {  
02     public abstract void crier();  
03 }
```

Poussin.java

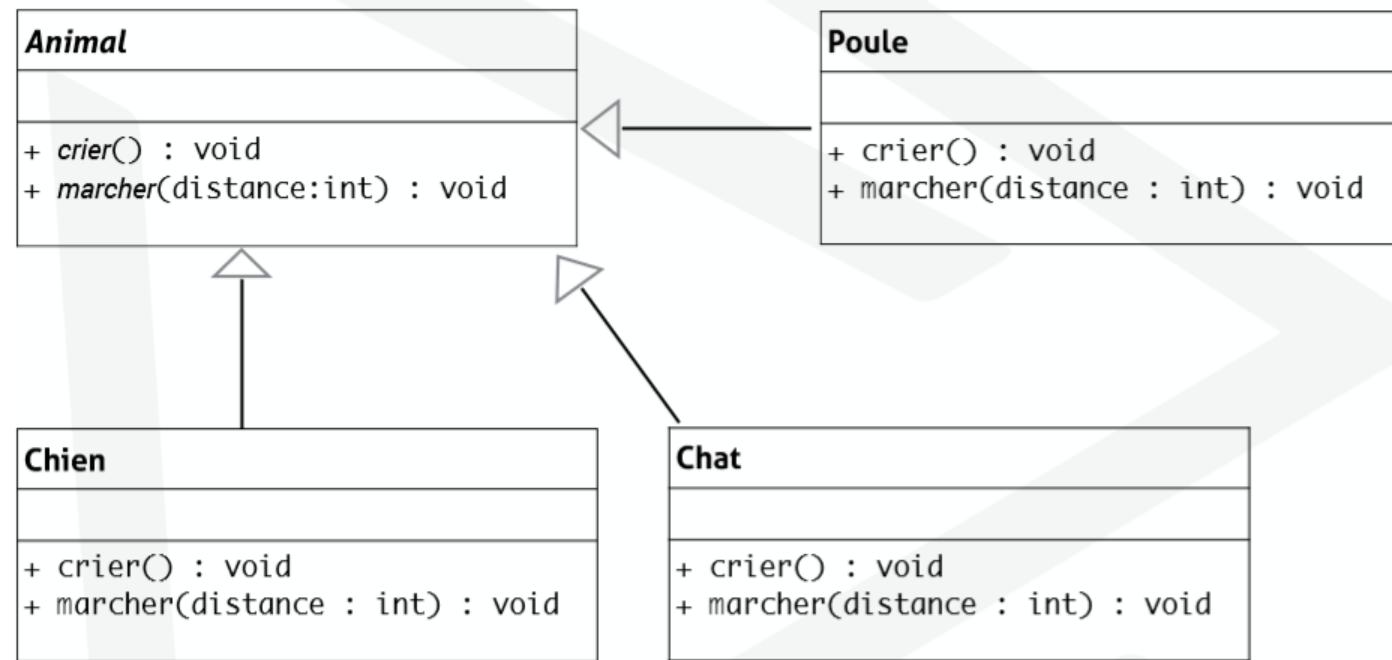
```
01 public class Poussin extends Animal {  
02     @Override  
03     public void crier() {  
04         System.out.println("Piou");  
05     }  
06 }
```

Main.java

```
01 public static void main(String[] args) {  
02     Animal animal = new Poussin();  
03     animal.crier();  
04 }
```

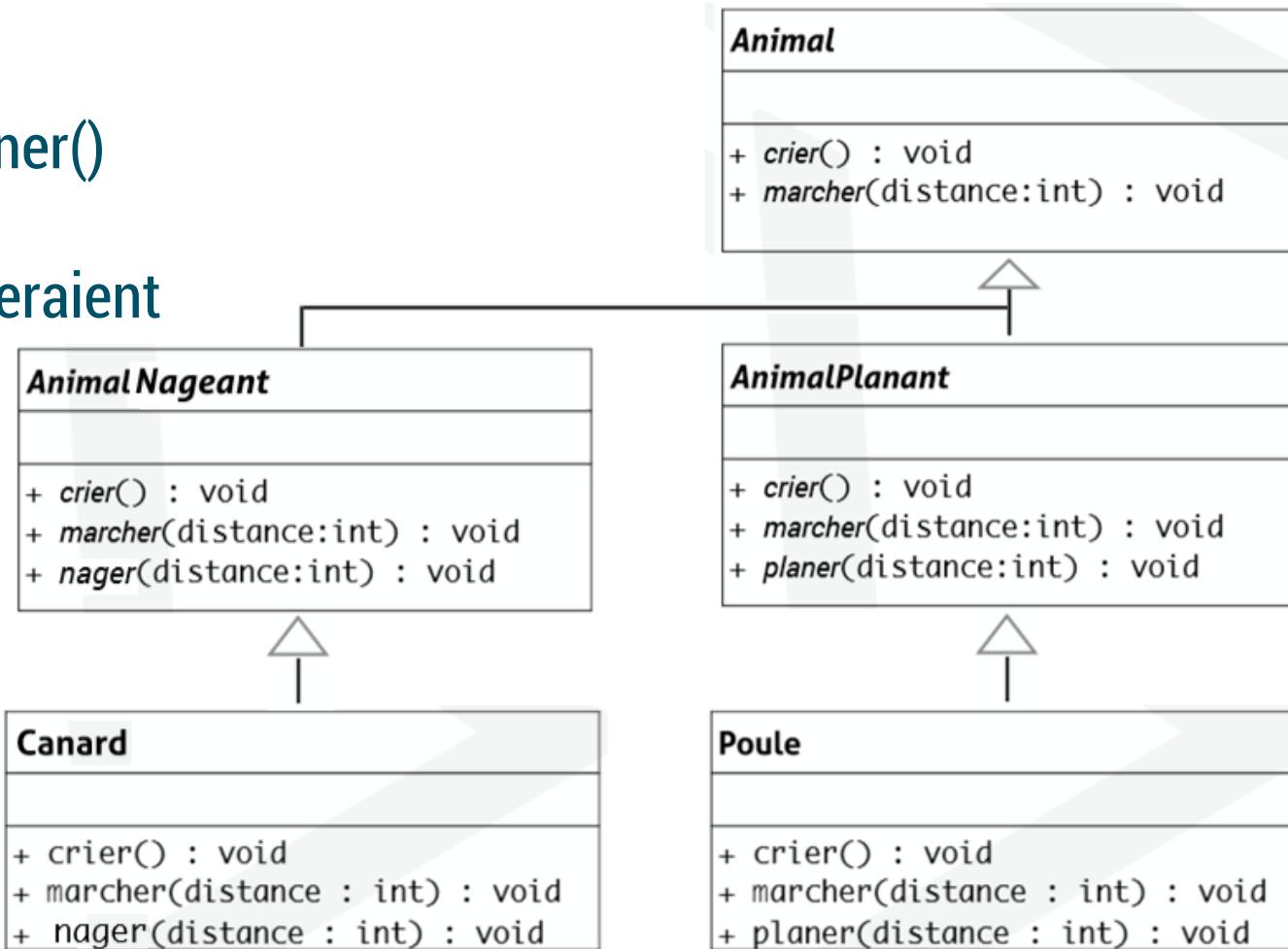
2 // [LES INTERFACES]

- Et si les animaux marchaient ?
 - Supposons que tous les animaux marchent
 - Facile :
 - on ajoute une méthode abstraite marcher() dans la classe Animal
 - et on écrit son contenu pour chaque classe Fille



2 // [LES INTERFACES]

- Et puis certains animaux planent
 - Idée : ajouter une méthode abstraite planer() dans la classe Animal
 - Problème : Tous les animaux possèderaient une méthode planer 😞
 - Idée 2 : ajouter une classe abstraite intermédiaire
 - Ca marche ?!
- Et puis certains animaux nagent !
 - On ajoute une nouvelle classe abstraite intermédiaire ?
 - Ca marche encore !

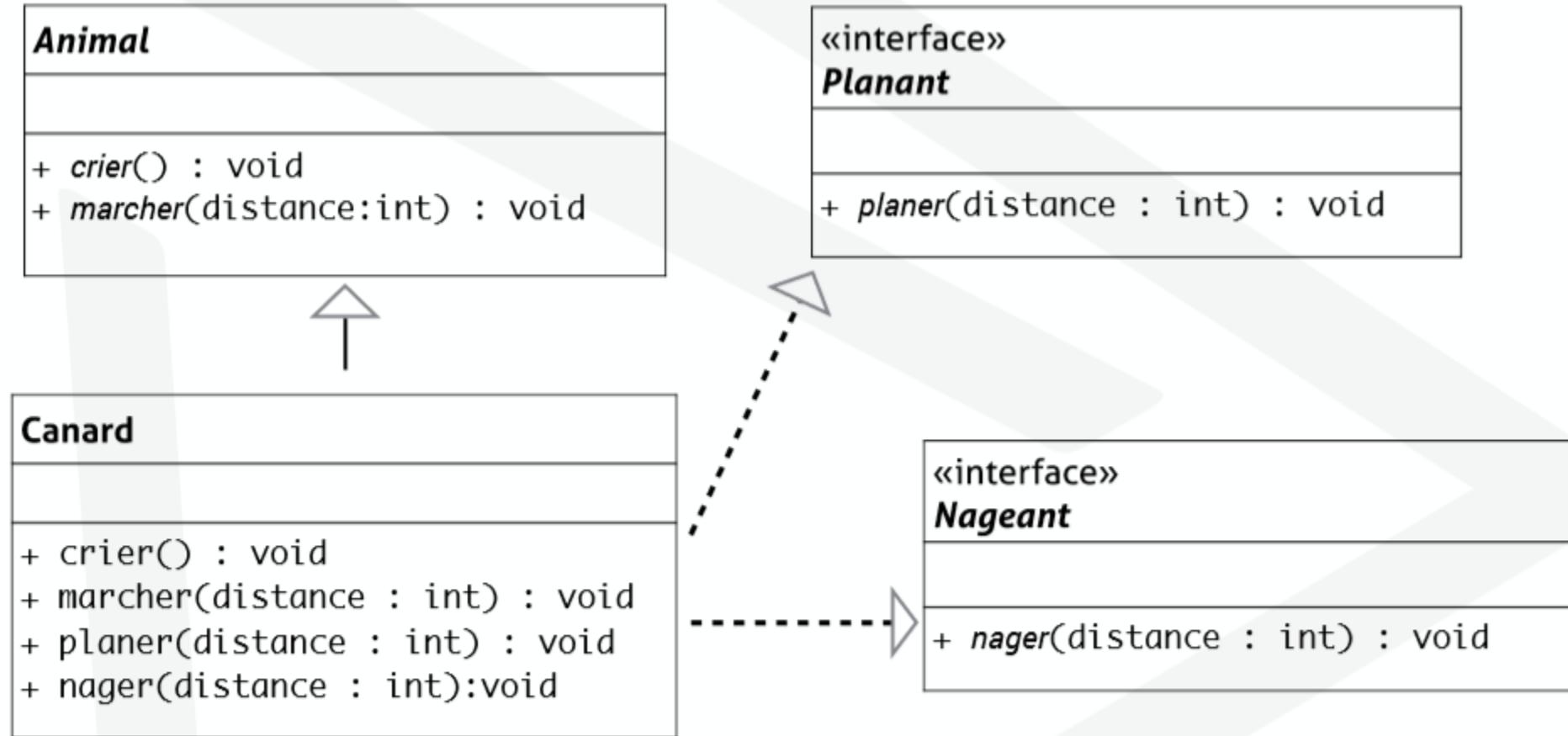


2 // [LES INTERFACES]

- Mais ... Un canard ca plane aussi !
 - AnimalNageant ou AnimalVolant ? Il faut choisir !
 - Pas d'héritage multiple en java !
- Solution : les interfaces
 - collection de méthodes abstraites
 - pas de propriété
 - pas de méthode avec du code
- Lorsqu'une classe implémente une interface, elle doit décrire le contenu de chacune des méthodes de la collection
- Une classe peut implémenter plusieurs méthodes

2 // [LES INTERFACES]

- Et pour notre canard ?



2 // [LES INTERFACES]

- Et pour notre canard ?

Canard.java

```
01 public class Canard extends Animal implements Nageant, Planant {  
02     @Override  
03     public void planer(int distance) {  
04         System.out.println("Je vole");  
05     }  
06  
07     @Override  
08     public void nager(int distance) {  
09         System.out.println("Je nage");  
10    }  
11  
12    @Override  
13    public void crier() {  
14        System.out.println("Coin coin");  
15    }  
16}
```

Nageant.java

```
01 public interface Nageant {  
02     public void nager(int distance);  
03 }
```

Planant.java

```
01 public interface Planant {  
02     public void planer(int distance);  
03 }
```

2 // [LES INTERFACES]

| Caractéristiques | Classe Abstraite | Interface |
|---|-----------------------------|--------------------------------------|
| propriétés | OUI | NON |
| méthodes concrètes | OUI | NON |
| méthodes abstraites | OUI | OUI |
| relation | "est un" | "se comporte comme" |
| mécanisme d'intégration dans une classe | héritage, simple uniquement | implémentation, possibilité multiple |

ABSTRACTIONS & INTERFACES

1 / MOT-CLÉ ABSTRACT
2 / INTERFACES

QUESTIONS ?

PACKAGES

PACKAGES

1 /
2 /
3 /

QUESTIONS ?