TELECOM SudParis

IP PARIS

CSC 3101 – Cours et TP

Algorithmique et langage de programmation

Année 2023-2024

Coordinateurs : Julien Romero & Pierre Sutra

IP PARIS

Introduction

Algorithmique et langage de programmation
Julien Romero, Pierre Sutra
Gaël Thomas

Dans ce cours

Vous allez apprendre le langage Java

et acquérir des bases en algorithmique

Pourquoi Java ?

- Java est un langage de programmation
  - Première version en 1995 (J.Gosling et P. Naughton, SUN)
  - Actuellement en version 20 (Septembre 2023)
  - L'un des langages les plus utilisés depuis 30 ans
- Java permet de programmer n'importe quelle application
  - Des jeux, des serveurs, des applications mobiles...
- Java possède un ensemble de bibliothèques extraordinaires
  - Environ 2.000.000 lignes de code
  - Pour tous les domaines applicatifs

Pourquoi étudier l'algorithmique ?

- Pour pouvoir programmer : un programme n'est rien d'autre qu'un algorithme
- Pour comprendre comment fonctionnent les programmes
- Pour en créer de nouveaux
- Pour le plaisir : de nombreux algorithmes sont simplement beaux !

Pourquoi Java et l'algorithmique ?

- Car il faut un langage pour écrire une **application**
- Car une **application** met en œuvre des algorithmes
- Car des algorithmes s'expriment dans un langage

Que va-t-on apprendre dans ce module ?

- Les concepts de la programmation impérative
  - Les structures de contrôles, les types de données, les méthodes
- Les concepts de la programmation orientée objet
  - Les classes, l'héritage, les interfaces, les exceptions
- De nombreux algorithmes
  - Algorithmes de tris
  - Algorithmes de graphe
  - Les structures de données usuelles (arbre, table de hachage, liste chaînée)

Et à la fin, vous allez programmer

- Un petit interpréteur pour un langage maison
  (uniquement la partie interprétation)
- Un petit jeu vidéo
- Un serveur Web dynamique
  (que vous pourrez tester avec votre navigateur préféré)

La programmation est un artisanat

- Le but du programmeur est de construire un logiciel ou un système, comme un maçon construit une maison, un bottier des souliers, ou un ébéniste un meuble
  - Le processus prend du temps
  - Il faut acquérir de l'expérience pour avoir une bonne vision d'ensemble
  - La maîtrise des compétences fondamentales est cruciale

La programmation est un artisanat

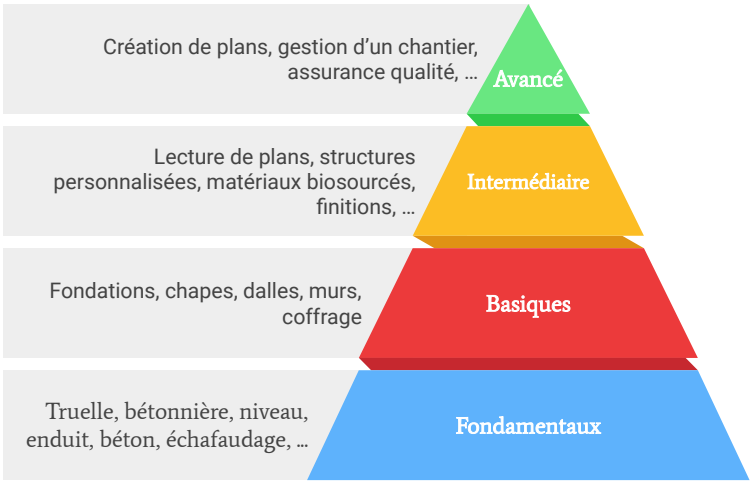
- Parfois une science, parfois un art

© Khürt Williams

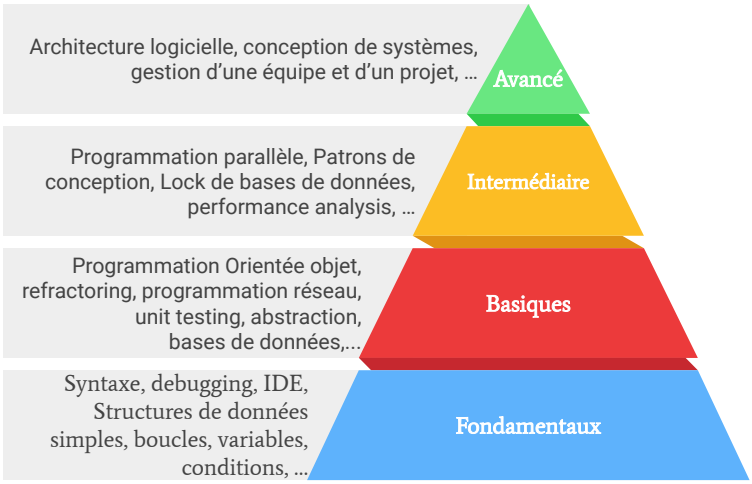
VS

La programmation est un artisanat

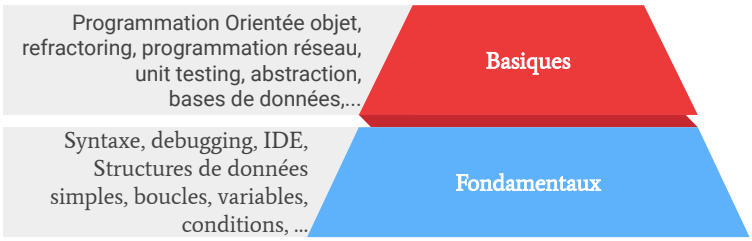
Jean-Pierre Dalbéra



La programmation est un artisanat



Dans ce cours, nous acquérons les bases



Comment pratiquer ?

- Faire les exercices donnés en cours
  - D'abord ceux obligatoires puis ceux donnés en entraînement pour aller plus loin.
- Résoudre des problèmes sur des plateformes spécialisées
  - https://leetcode.com/
  - https://www.codingame.com
  - https://projecteuler.net/
  - https://www.codewars.com/
  - ...
- Certains automatismes ne s'acquièrent qu'avec beaucoup de pratique

Organisation

- Le cours magistral d'aujourd'hui
- 9 séances de 3h en petits groupes d'une vingtaine d'étudiants
  - Une (petite) partie cours (environ 30 minutes)
  - Une (grosse) partie pratique sur machine (environ 2h30)
- Notation : 5% devoir maison + 95% contrôle final
  - Un devoir maison noté
  - Un contrôle
  - Un contrôle de rattrapage
- Attention! N'oubliez pas de ~~travailler~~ programmer ☺️
  car programmer s'apprend par la pratique

Assistants IA

- Il existe des assistants pour aider à coder: ChatGPT, GitHub Copilot, Bard, ...
- Il peuvent aider à augmenter sa productivité...
  - ... si l'on sait déjà programmer
- On déconseille fortement leur usage dans ce cours
  - Il faut savoir programmer pour reconnaître quand l'IA dit n'importe quoi
  - L'IA peut vous empêcher de progresser et de passer au niveau supérieur
  - L'IA sait faire les choses simples, mais a beaucoup de difficultés pour les concepts plus avancés et globaux
- Des cours sur cette thématique vont venir par la suite...

Premiers pas avec Java

Algorithmique et langage de programmation

Julien Romero, Gaël Thomas

Plan du cours

1. **Mon premier programme Java**
2. Exécution d'un programme Java
3. Variables et types de données
4. Les opérateurs du langage Java
5. Les conversions de type
6. Structures algorithmiques

Dans un programme Java, on trouve

- Des mots clés : des mots qui ont un sens réservé dans le langage : class, if, while, |, &&, do...
- Des symboles : des noms servant à identifier des éléments
  - Constitué de caractères, chiffres (sauf au début), _ ou $
  - La casse est significative (majuscule vs minuscule).
- Des types : spécifie la nature de certains symboles
  - Entier (int), chaîne de caractère (String), flottant (float)...
- Des valeurs littérales : des valeurs ayant un type donné

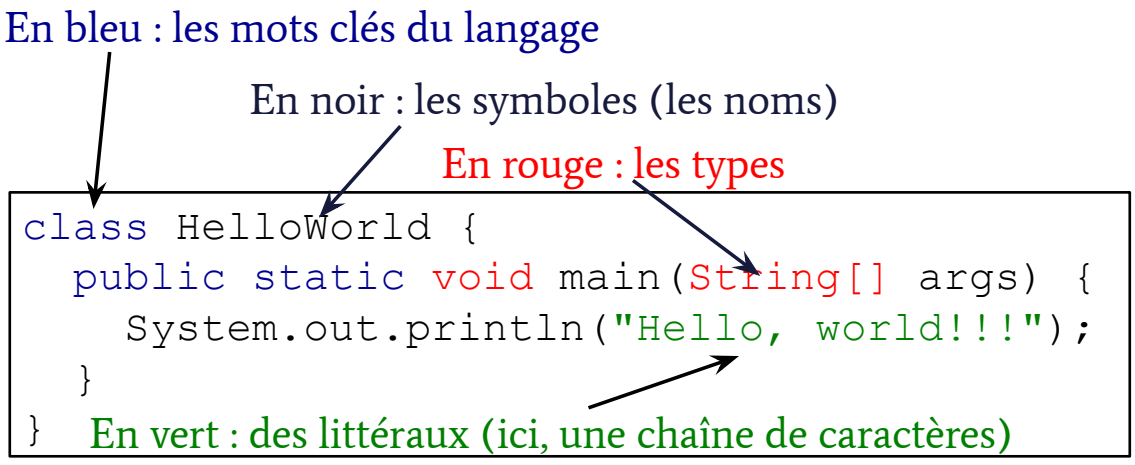
  42 (entier), "Bonjour, monde!" (chaîne de caractère), 3.14 (flottant)...

Dans un programme Java, on trouve

- Et des commentaires
  - Utilisés pour expliquer ce que fait le programme
  - Non exécutés, uniquement pour documenter le code

```
/*
 * ceci est un commentaire multi-lignes
 */

/* ceci est un commentaire multi-lignes sur une ligne */

// ceci est un commentaire mono-ligne
```

Mon premier programme Java



Mon premier programme Java

- Dans un premier temps, le mot clé `class` sert à définir le nom du programme
  - Ici, le nom du programme est `HelloWorld`
  - Le code du programme se trouve entre les accolades qui suivent

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

Mon premier programme Java

Interlude

L'équipe pédagogique tient à s'excuser pour un petit mensonge : le mot clé class est infiniment plus complexe que ce qui est présenté ici

Vous comprendrez le rôle exact du mot clé class dans les CI3 et CI4

Pour le moment, imaginez que class donne le nom du programme n'est pas totalement faux

Mon premier programme Java

- La ligne contenant `main` sera expliquée dans les CI3 à CI5
- Pour le moment
  - Cette ligne indique le début du programme…
  - …qui se trouve entre les accolades qui suivent

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

Mon premier programme Java

- Le code du programme est constitué de déclarations
  - `System.out.println` affiche son paramètre sur le terminal
    - ln ajoute une nouvelle ligne, `System.out.print` sinon
  - Le `"Hello, World!!!"` est le paramètre
  - Le point virgule (;) de fin de ligne indique la fin de la déclaration

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


Mon premier programme Java

- Les déclarations se terminent **toutes** par un point virgule !
  - Si on a une accolade fermante, alors pas de ;

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

Plan du cours

1. Mon premier programme Java
2. **Exécution d'un programme Java**
3. Variables et types de données
4. Les opérateurs du langage Java
5. Les conversions de type
6. Structures algorithmiques

Exécution d'un programme Java

- Étape 1 : traduction vers un fichier appelé bytecode
  - Le fichier source (contenant le programme Java) n'est pas exécutable en l'état, il doit être traduit
  - Le fichier bytecode contient une forme plus rapide à exécuter que le fichier source (validation syntaxique et sémantique effectuée)
  - La transformation source vers bytecode s'appelle la compilation
- Étape 2 : exécution dans une machine virtuelle Java
  - Un fichier de bytecode n'est pas directement exécutable par un processeur
  - Le programme `java` est capable d'interpréter un fichier bytecode

Exécution d'un programme Java

Fichier HelloWorld.java

```
class HelloWorld {… }
```

Compilation avec le programme javac

Fichier HelloWorld.class

```
0xcafebabe…
```

Interprétation avec le programme java

Mise en perspective

- En `Bash` (resp. `Python`)
  - Un programme source est un fichier `.sh` (resp. `.py`) (contient des déclarations `bash`, resp `python`)
  - Directement exécutable par l'interpréteur `bash` (resp. `python`)
- En `Java`
  - Un programme source est un fichier `.java` (contient des déclarations `Java`)
  - Doit être compilé en bytecode (fichier `.class`) avec `javac`
  - Le bytecode est exécutable par l'interpréteur `java`

Conception, compilation, exécution

```
$ emacs HelloWorld.java
```

```
class HelloWorld {
   …
}
```

Éditeur emacs
(fichier HelloWorld.java)

Première étape :
la conception

Conception, compilation, exécution

```
$ emacs HelloWorld.java
```

```
class HelloWorld {
   …
```

Attention : le nom du fichier
et le symbole qui suit class doivent coïncider

Convention majuscule en début
de symbole dans ce cas,
minuscule pour tous les autres symboles

Conception, compilation, exécution

```
$ emacs HelloWorld.java
$
```


Conception, compilation, exécution

```
$ emacs HelloWorld.java
$ ls
HelloWorld.java
$
```

Conception, compilation, exécution

```
$ emacs HelloWorld.java
$ ls
HelloWorld.java
$ javac HelloWorld.java
$
```

Deuxième étape : la compilation

Conception, compilation, exécution

```
$ emacs HelloWorld.java
$ ls
HelloWorld.java
$ javac HelloWorld.java
$ ls
HelloWorld.class
HelloWorld.java
$
```

Conception, compilation, exécution

```
$ emacs HelloWorld.java
$ ls
HelloWorld.java
$ javac HelloWorld.java
$ ls
HelloWorld.class
HelloWorld.java
$ java HelloWorld
Hello, world!!!
```

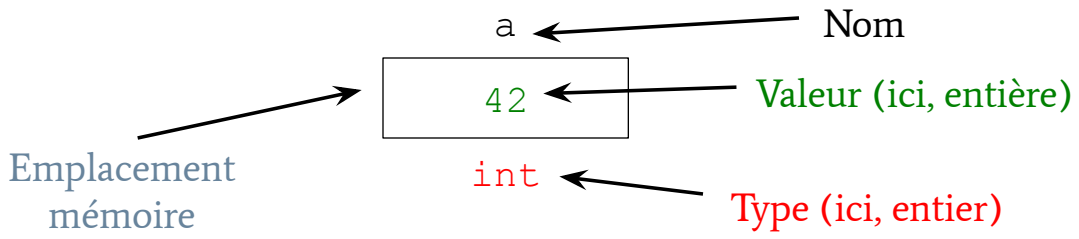
Troisième étape :
L'exécution

Plan du cours

1. Mon premier programme Java
2. Exécution d'un programme Java
3. **Variables et types de données**
4. Les opérateurs du langage Java
5. Les conversions de type
6. Structures algorithmiques

Les variables en Java (1/2)

- Une variable **est** un emplacement mémoire
  - Qui possède **un nom** (le nom de la variable)
  - **Un type** (la nature de ce que contient la variable)
  - Et **une valeur** (le contenu de la variable)



Les variables en Java (2/2)

- En Java, les types de base sont les :
  - Booléens : `boolean` (true ou false)
  - Entiers (signés) : `byte` (1 octet), `short` (2 octets), `int` (4 octets), `long` (8 octets)
  - Réels : `float` (4 octets), `double` (8 octets)
  - Caractères : `char` (un caractère **unicode**)
  - (Chaînes de caractères : `String` (plusieurs caractères), pas vraiment un type de base mais très utile)

Les variables en Java

- Définition avec :
  - `type symbole; /* ? valeur indéfinie ? */`
  - ou `type symbole = littéral;`

```
class HelloWorld {
  public static void main(String[] args) {
    String msg = "Coucou";/* Coucou */
    char c = '!';         /* ! */
    int x = 2;            /* 2 */
    int y = x + 5;        /* 7 (addition entière) */
    float z;              /* ? valeur indéfinie ? */
    boolean b = true;     /* true */
  }
}
```


Les littéraux en Java

- Un entier : une valeur entière sans symbole particulier
  - Si suivi de `l` (la lettre L minuscule), valeur de type `long` (ex: `21`)
  - Sinon de type `int` (ex: `2`)
- Un réel : une valeur avec un point sans symbole particulier (ou avec exposant 3.14e7 == $3.14 * 10^7$)
  - Si suivi de `f`, valeur de type `float` (ex: `3.14f`)
  - Sinon de type `double` (ex: `3.14`)
- Un caractère : un caractère entouré d'apostrophes (ex: `'a'`)
- Une chaîne de caractères : une suite de caractères entourée de guillemets (ex: "Ceci est une chaîne")

Plan du cours

1. Mon premier programme Java
2. Exécution d'un programme Java
3. Variables et types de données
4. **Les opérateurs du langage Java**
5. Les conversions de type
6. Structures algorithmiques

Les opérateurs du langage Java (1/2)

- Opérations arithmétiques sur les nombres (entiers ou flottants)
  - `+, -, *, /, %` (modulo), ++ (incrémente), -- (décrémente)
- Opérations bit à bit (sur les entiers)
  - `&` (et), `|` (ou), ^ (xor), ~ (complément), << (décalage à gauche), >> (décalage à droite)
- Opérations sur les booléens
  - `&&` (et), `||` (ou), `!` (non)
- Opérations sur les chaînes de caractères
  - + (concaténation)

Les opérateurs du langage Java (2/2)

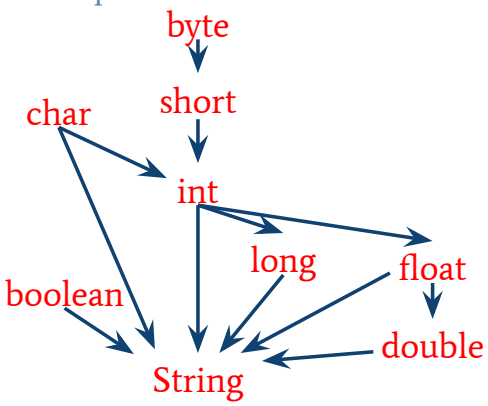
- Opérations de comparaison
  - `==, <=, <, >=, >, !=` (différent)
- Opération d'affectation (tous types)
  - `=`
- Combinaison d'affectation avec d'autres opérations possible `+=, /=, >>=` etc.
  - (Exemple : `a += 42` est équivalent à `a = a + 42`)

Plan du cours

1. Mon premier programme Java
2. Exécution d'un programme Java
3. Variables et types de données
4. Les opérateurs du langage Java
5. **Les conversions de type**
6. Structures algorithmiques

Conversion automatique de type

- Avant d'effectuer une opération, les valeurs peuvent être converties vers un type plus fort

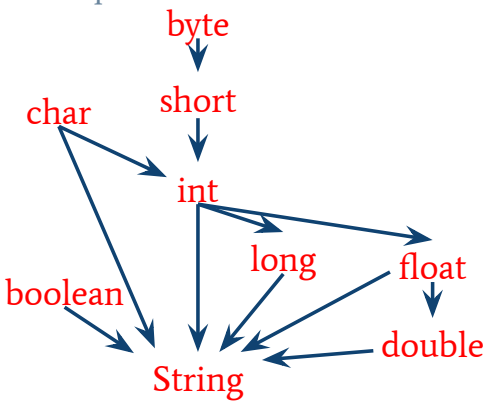


Type le plus faible

Type le plus fort

Conversion automatique de type

- Avant d'effectuer une opération, les valeurs peuvent être converties vers un type plus fort



Type le plus faible

Remarque : lors de la conversion d'un char vers un int c'est le numéro de caractère qui est renvoyé (par exemple 'a' a la valeur 97)

Type le plus fort

Conversion automatique de type - Unicode

Un ordinateur stocke des chiffres !

On doit donc utiliser une convention pour convertir les chiffres en lettres

Exemples:

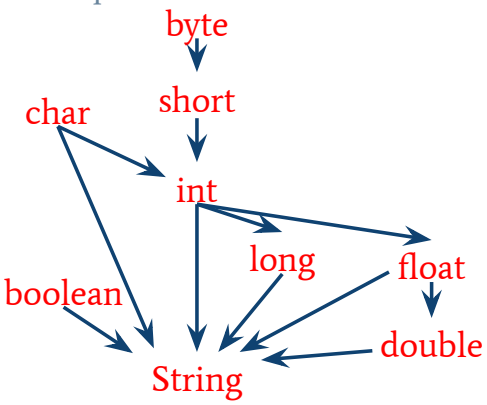
- ASCII: 7 bits, 128 caractères
- UTF-8: 8 bits, 256 caractères (+ extension unicode sur 4 bytes)
- UTF-16 (Java) : 16 bits, 65.536 caractères (+ extension unicode sur 4 bytes)
- Unicode : 4 bytes, 4 milliards de caractères

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Conversion automatique de type

- Avant d'effectuer une opération, les valeurs peuvent être converties vers un type plus fort




```
int x = 3;
double y = x + 2.2; /* 5.2 */
String s1 = "Coucou" + x;
    /* Coucou3 */
char c = 'a';
String s2 = "Coucou" + c;
    /* Coucoua */
int d = c + 1;
    /* 98 car 'a' = 97 */
```

Conversion explicite de type

- Quand il n'existe pas de conversion implicite, on utilise la conversion explicite avec `(type)`
  - Dans ce cas, Java tronque le nombre

```
double x = 97.7;
int y = (int)x;           /* 97 */
char c = (char)y;         /* 'a' car c vaut 97 */
byte b = (byte)(y * 3); /* 291 modulo 256 = 35 */
```


Plan du cours

1. Mon premier programme Java
2. Exécution d'un programme Java
3. Variables et types de données
4. Les opérateurs du langage Java
5. Les conversions de type
6. **Structures algorithmiques**

Structures algorithmiques et blocs

- La structure d'un programme Java est donnée par des blocs
  - Commence par un { et termine par un }
  - Regroupe un ensemble de déclarations

```
class Test {
  public static void main(String[] args) {
    if(0 == 1) {
      System.out.println("Cette machine est bizarre");
      System.out.println("Elle pense que  0 == 1 !");
    }
  }
}
```

Bloc exécuté si 0 == 1

  - Accolades optionnelles si une unique déclaration dans un bloc
  - Pour être plus précis, un bloc est simplement une déclaration…
  - Même si ce n'est pas obligatoire, on indente les blocs pour la lisibilité (comme en Python) !

Schéma alternatif (1/2)

- Schéma alternatif simple
  - si alors … sinon (si alors … sinon …)
  - Parties else if et else optionnelles

```
if(cond1) {
  body1
} else if(cond2) {
  body2
} else {
  body3
}
```

Exemple.java

```
class Exemple {
  static void main(String[] args) {
    int a = 21 * 2;
    if(a == 42) {
      System.out.println("Super !");
      System.out.println("Java, c'est facile !");
    }
  }
}
```

Schéma alternatif (2/2)

- Schéma alternatif complexe
  - Si val vaut v1, exécute body1
  - Sinon, si val vaut v2, exécute body2
  - Sinon, si ...
  - Sinon, exécute bodyn

```
switch(val) {
  case v1:
    body1;
    break;
  case v2:
    body2;
    break;
  …
  default:
    bodyd;
    break;
}
```

```
switch(c) {
  case 'a': System.out.println("Ceci est un a"); break;
  case 'b': System.out.println("Ceci est un b"); break;
  …
  default: System.out.println("Bizarre"); break;
}
```


Schéma alternatif (2/2)

- Schéma alternatif complexe
  - Si pas de `break`, continue l'exécution avec le `body` suivant
  - Attention au code spaghetti !

```
switch(val) {
  case v1:
    body1;
    break;
  case v2:
    body2;
    break;
  …
  default:
    bodyd;
    break;
}
```

```
switch(c) {
  case 'a': System.out.println("uniquement a");
  case 'b': System.out.println("a ou b"); break;
  case 'c': System.out.println("uniquement c");
  default:  System.out.println("ni a, ni b ☺");
}
```

Schémas itératifs

- Boucle `while`
  Tant que `cond` faire `body`
- Boucle `do … while`
  Faire `body` tant que `cond` (body exécuté au moins une fois)
  Attention au `;`
- Boucle `for`
  Exécute `init`
  Tant que `cond` faire
  `body` puis `iter`

```
while(cond) {
  body
}
```

```
do {
  body
} while(cond);
```

```
for(init; cond; iter) {
  body
}
```


Schémas itératifs – exemples

```
int x = 0;
while(x < 10) {
  System.out.println(x);
  x++;
}
```

```
for(int x=0; x<10; x++) {
  System.out.println(x);
}
```

```
int x;
do {
  x = Math.random() % 10;
} while(x == 3);
```

Java versus Python versus Bash

- En Java
  - Le type d'une variable est explicite
    (`int x=100` ⇒ entier, `String name="Arya"` ⇒ chaîne de caractères)
  - Les blocs sont explicitement délimités par des accolades `{ … }`
- En Python
  - Le type d'une variable est implicite
    (`x=100` ⇒ entier, `name="Arya"` ⇒ chaîne de caractères)
  - Les blocs sont implicitement donnés par l'indentation
- En Bash
  - Toutes les variables sont de type chaîne de caractères
    (`PATH=/usr/bin`)
  - Les blocs d'instructions sont délimités par des mots clés
    (`do … done`, `if …; then … fi` etc…)

Notions clés

- Conception, compilation et exécution d'un programme Java
- Déclaration et typage des variables : `type var;`
  `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `String`
- Opérateurs de base
- Structures algorithmiques
  - Schéma itératif (`if/else`)
  - Schéma alternatif (`while`, `for`, `do … while`)

If you want to know more

À propos des conversions de type en Java

Étrangeté

- Avant d'effectuer une opération arithmétique, Java convertit un `byte`, un `short` ou un `char` en `int`

```
byte x = 1;
byte y = 2;
byte z = x + y;
```

Interdit car :

Java convertit x et y en int avant d'effectuer l'opération

⇒ le résultat est un int et il est impossible d'affecter un int dans un byte

⇒ en additionnant deux bytes, on peut facilement dépasser la valeur maximale ou minimale

Encore plus d'étrangeté (pour votre culture)

- Si les deux membres de l'opération sont des littéraux entiers (`byte`, `short`, `int`) ou caractères (`char`), alors Java effectue **normalement** le calcul en `int`
- **Mais après le calcul**, s'il faut affecter le résultat à un des types entiers ou au type `char`, Java convertit le résultat si le nombre n'est pas trop grand
- **Raison**: Java peut deviner ou non si le résultat va dépasser le maximum/minimum au moment de la compilation et nous prévenir

Encore plus d'étrangeté (pour votre culture)

```
char c = 'a';
char d = c + 1;
```

Interdit car c n'est pas un littéral

(il est impossible de convertir de int vers char lors de l'affectation)

```
char c = 'a' + 1;
```

Autorisé car 'a' est un littéral

('a' devient le nombre 97, la somme vaut 98, qui représente le caractère 'b')

Les tableaux en Java

Algorithmique et langage de programmation
Julien Romero, Gaël Thomas

Qu'est ce qu'un tableau en Java

- Un tableau est une structure de données qui contient plusieurs éléments **du même type**

Un tableau de 6 entiers

1	17	4	7	2	13

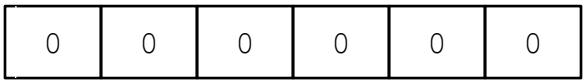
Allocation d'un tableau

- Un tableau doit être alloué dans la mémoire avec `new type[n]`

Allocation d'un tableau de `n` éléments ayant pour type `type`

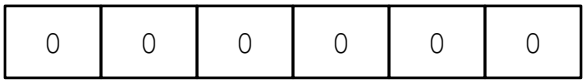
- Par exemple : `new int[6]`

Alloue un tableau de 6 entiers



Allocation d'un tableau

- L'opérateur `new` renvoie une référence vers un tableau
  (une référence est un identifiant unique d'une structure de données)
- Par exemple, `new int[6]` renvoie une référence vers ce tableau :



Note : Java met à 0 chaque élément lors d'une allocation

Déclaration

- Il n'existe pas de variable de type tableau en Java !
- En revanche, on peut déclarer une variable de type référence vers un tableau :

```
type[] var;
```

var est une variable
de type référence vers un tableau
contenant des éléments de type type

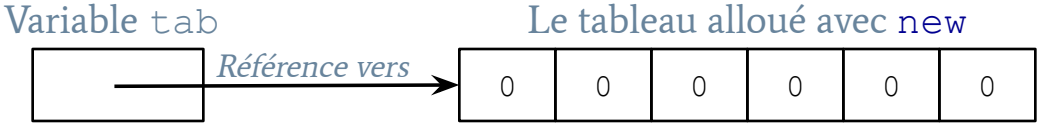
Exemple de déclaration

tab référence un tableau de int

Affectation de la référence

Allocation d'un tableau de 6 int

```
int[] tab = new int[6];
```



Allocation et initialisation

- On peut aussi allouer un tableau et l'initialiser avec

```
type[] tab = { x1, … xn };
```

- Par exemple :

```
double[] tab = { 2.3, 17.0, 3.14, 8.83, 7.26 };
```

- En détail, le programme va
  - Allouer le tableau (comme avec `new`) puis initialiser les éléments
  - Renvoyer une référence vers le tableau

Accès à un tableau

- Accès à la taille du tableau avec `tab.length`

- Accès à un élément avec `tab[indice]`
  - Exemple : `tab[i] = tab[j] * 2;`
  - Attention : les éléments sont indexés à partir de `0`
    ⇒ un tableau possède les éléments allant de `0` à `tab.length-1`

- Un accès en dehors des bornes du tableau provoque une erreur à l'exécution (`ArrayOutOfBoundsException`)

Parenthèse: La mémoire simplifiée

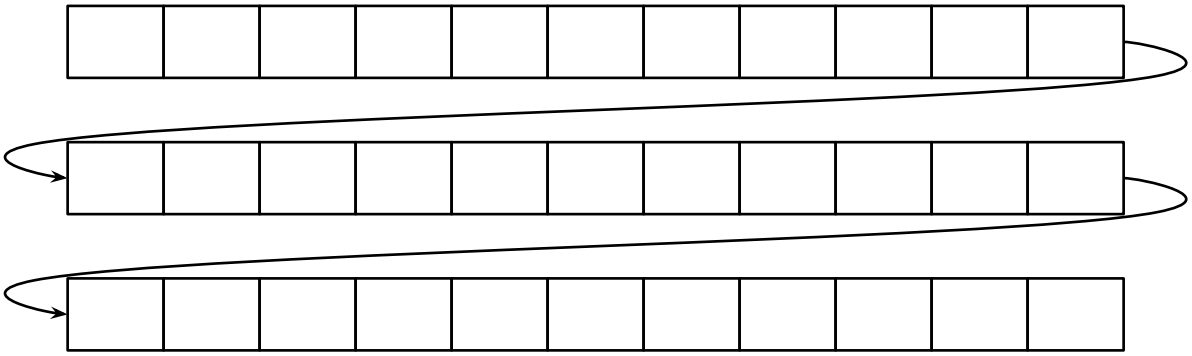
- Quelle différence entre un *octet* et un *byte* ?

Parenthèse: La mémoire simplifiée

- Quelle différence entre un *octet* et un *byte* ?
  - *Octet* = 8 bits (0/1)
  - *Byte* = plus petite unité de mémoire allouable, ce qui dépend des ordinateurs (8 bits en général, mais ça peut varier)
- En général, un *byte* est trop petit. On utilise donc un int comme unité de base.
  - Souvent, les calculs sont plus rapides avec un int
  - La taille d'un int dépend de l'ordinateur
  - En Java, on a une machine virtuelle qui fixe la taille des types pour tous les ordinateurs
    - byte = 1 octet
    - char = short = 2 octets
    - int = 4 octets
    - long = 8 octets
    - ...

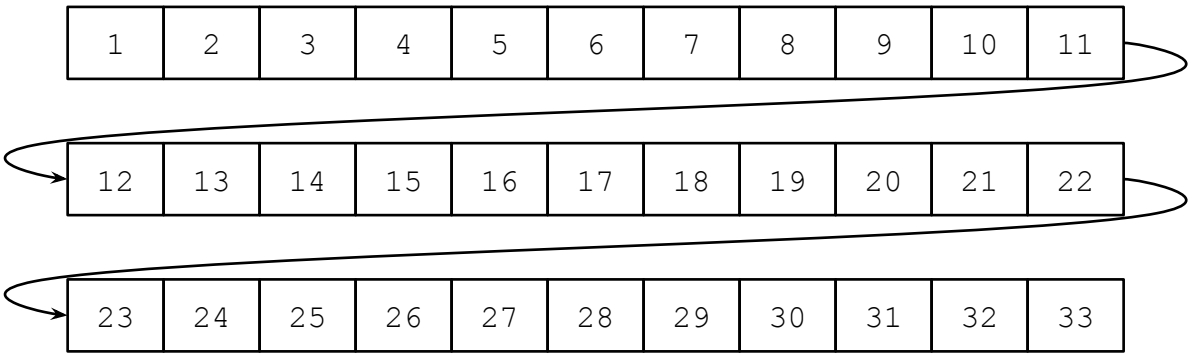
Parenthèse: La mémoire simplifiée

La mémoire est une séquence de “cases” allouables.



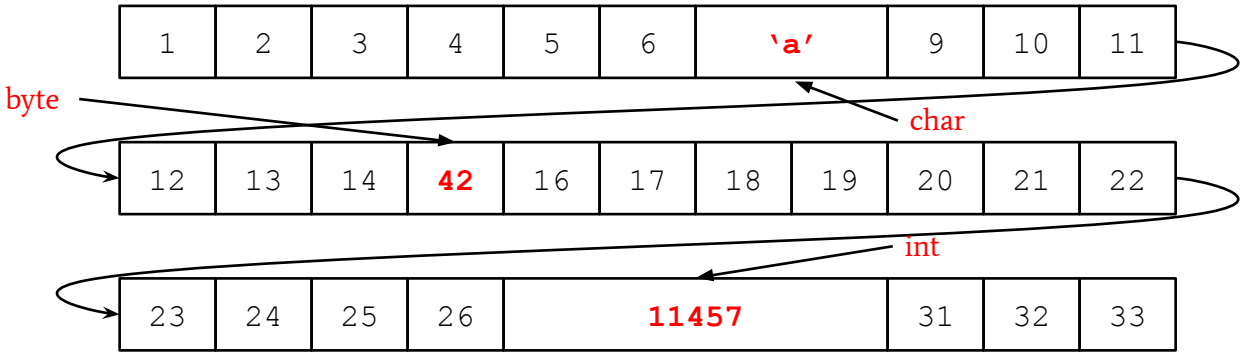
Parenthèse: La mémoire simplifiée

Pour se repérer dans la mémoire, chaque case a une **adresse**



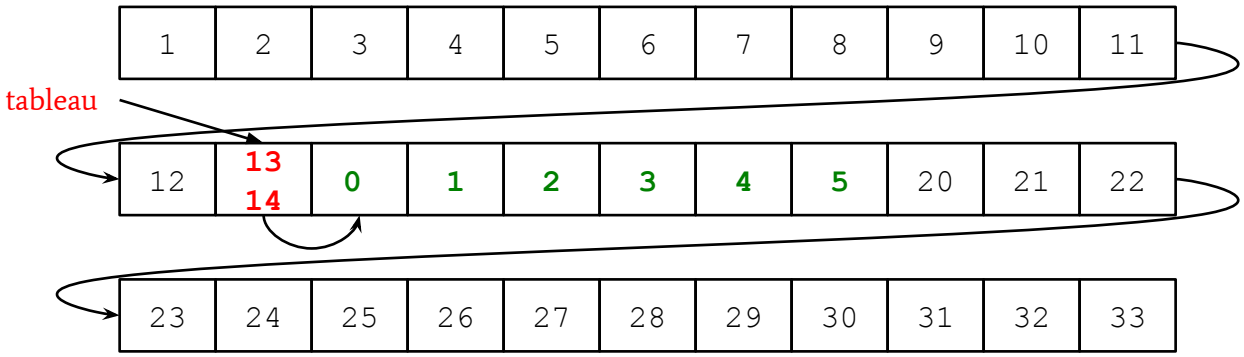
Parenthèse: La mémoire simplifiée

Les **types primitifs** (`boolean, char, byte, short, int, long, float, double`) ont une taille fixe et représentent **une seule valeur**, les adresses sont cachées



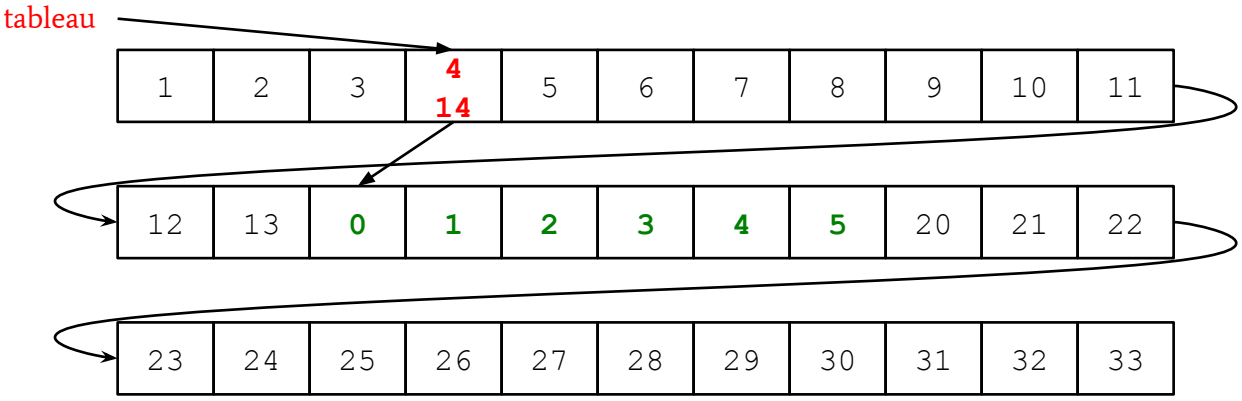
Parenthèse: La mémoire simplifiée

Pour les **types par référence** (ex: les tableaux), la taille des données peut varier. On ne donne donc que l'adresse de début des valeurs, Java se débrouille pour trouver le reste.



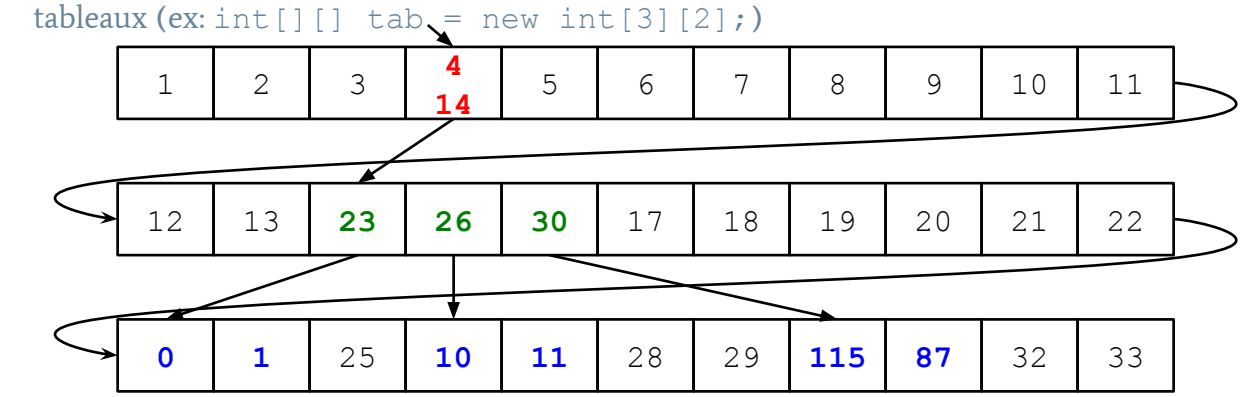
Parenthèse: La mémoire simplifiée

Attention! Les références (= les adresses) sont elles même des valeurs !



Parenthèse: La mémoire simplifiée

On peut avoir des tableaux de tableaux, i.e. des tableaux de références vers d'autres tableaux (ex: `int[][] tab = new int[3][2];`)

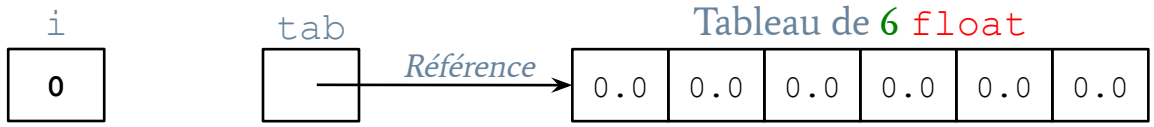


Exemple d'accès à un tableau

```
public static void main(String[] args) {
  float[] tab = new float[6];
  for(int i=0; i<tab.length; i++) {
    tab[i] = i + 2;
  }
}
```

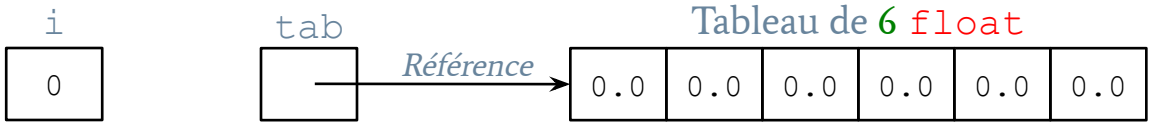
Exemple d'accès à un tableau

```
public static void main(String[] args) {
  float[] tab = new float[6];
  for(int i=0; i<tab.length; i++) {
    tab[i] = i + 2;
  }
}
```



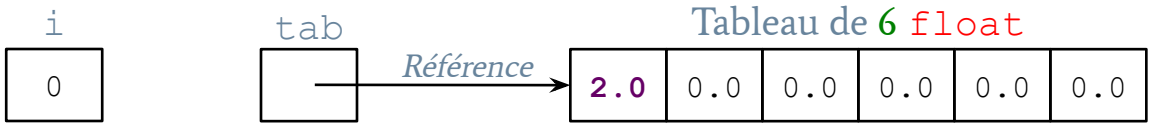
Exemple d'accès à un tableau

```
public static void main(String[] args) {
   float[] tab = new float[6];
   for(int i=0; i<tab.length; i++) {
      tab[i] = i + 2;
   }
}
```



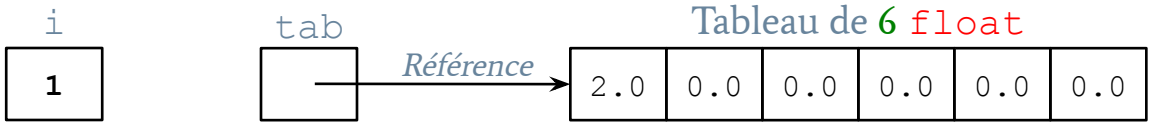
Exemple d'accès à un tableau

```
public static void main(String[] args) {
   float[] tab = new float[6];
   for(int i=0; i<tab.length; i++) {
      tab[i] = i + 2;
   }
}
```



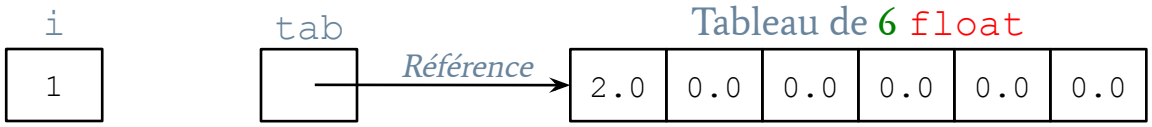
Exemple d'accès à un tableau

```
public static void main(String[] args) {
   float[] tab = new float[6];
   for(int i=0; i<tab.length; i++) {
      tab[i] = i + 2;
   }
}
```



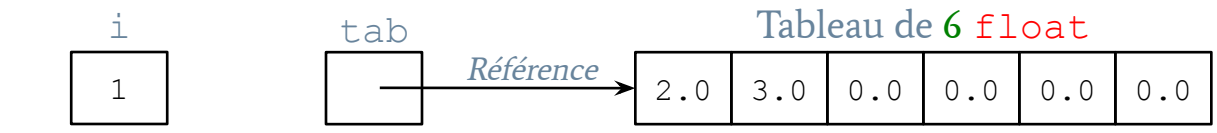
Exemple d'accès à un tableau

```
public static void main(String[] args) {
   float[] tab = new float[6];
   for(int i=0; i<tab.length; i++) {
      tab[i] = i + 2;
   }
}
```



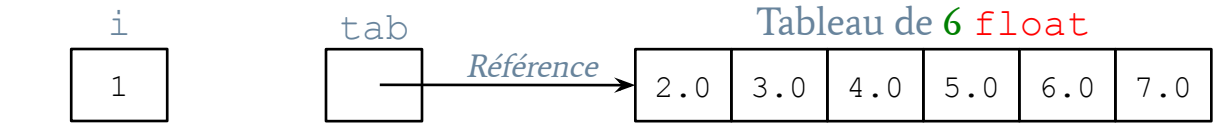
Exemple d'accès à un tableau

```
public static void main(String[] args) {
  float[] tab = new float[6];
  for(int i=0; i<tab.length; i++) {
    tab[i] = i + 2;
  }
}
```



Exemple d'accès à un tableau

```
public static void main(String[] args) {
  float[] tab = new float[6];
  for(int i=0; i<tab.length; i++) {
    tab[i] = i + 2;
  }
}
```

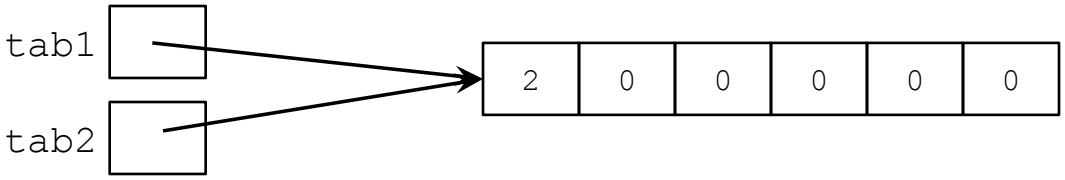


État à la fin de la boucle

Tableaux et aliasing

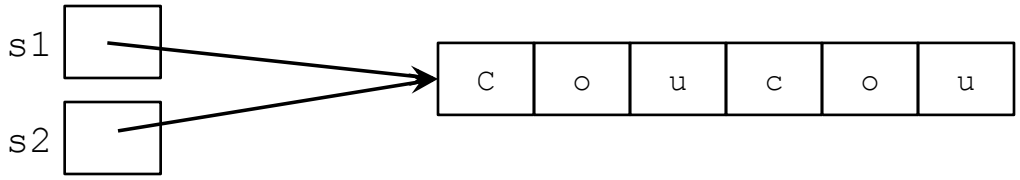
```
public static void main(String[] args) {
  byte[] tab1 = new byte[6];
  byte[] tab2 = tab1;
  tab2[0] = 2;
  System.out.println("tab1: " + tab1[0]);
} /* affiche tab1: 2 */
```

- Dans l'exemple précédent, `tab1` et `tab2` sont deux variables différentes, mais elles référencent le même tableau



String et aliasing

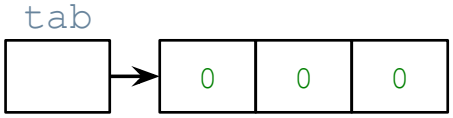
- Comme pour les tableaux, il n'existe en fait pas de variable de type String en Java !
- En revanche, `String` déclare une variable **référençant** une zone mémoire typée avec le type `String`
- Exemple : `String s1 = "Coucou"; String s2 = s1;`



Type référence versus type primitif

- Variable de type référence : contient une référence vers une structure de données
  - Tableaux et String
  - Très commun en Java !

```
int[] tab = new int[3];
```



- Variable de type primitif : contient une valeur
  - boolean, byte, short, char, int, long, float et double
  - On n'a pas de référence pour les types primitifs !

```
int x = 42;
```

Retour sur la méthode `main`

- Dans `public static void main(String[] args)`
  - `args` est une référence vers un tableau de chaînes de caractères correspondant aux arguments du programme
  - Si aucun argument : `args.length` vaut 0
  - Sinon, `args[0]` est le premier argument, `args[1]` le second, ...

Retour sur la méthode `main`

```
class CmdLine {
  public static void main(String[] args) {
    for(int i=0; i<args.length; i++) {
      System.out.println(
        "args[" + i + "]: " + args[i]);
    }
  }
}
```

```
$ java CmdLine Bonjour monde
args[0]: Bonjour
args[1]: monde
$
```


Notions clés

- Allocation d'un tableau avec
  `new type[n]`
- Déclaration d'une variable référençant un tableau avec
  `type[] var`
- Accès à un élément avec
  `var[indice]`
- L'argument du `main` est le tableau des arguments du programme

Introduction à la complexité

Julien Romero

Décrire les performances des algorithmes

- Quand on a des algorithmes, on veut souvent analyser leurs performances :
  - D'un point de vue empirique: temps de calculs, usage de la mémoire moyenne, ...
  - D'un point de vue théorique: la complexité en instructions et en mémoire
- La complexité peut se décliner sur plusieurs cas
  - Dans le pire de cas (le plus fréquent, celui de ce cours): on calcule le nombre maximum d'instructions ou de mémoire utilisées
  - En moyenne: on calcule le nombre moyen d'instructions ou de mémoire utilisées. Souvent plus compliquée à calculer.
  - Dans le meilleur des cas (très rare): on calcule le nombre minimum d'instructions ou de mémoire utilisées

Mesurer les complexités

- Les complexités dans les cas asymptotiques sont exprimées en ordre de grandeur en fonction des entrées, souvent en utilisant des gros O: $\mathcal{O}(\dots)$
- Les complexités sont alors catégorisées en grands ensembles:
  - Constante: $\mathcal{O}(1)$
  - Logarithmique: $\mathcal{O}(log(n))$
  - Linéaire: $\mathcal{O}(n)$
  - Quadratique: $\mathcal{O}(n^2)$
  - Polynomiale: $\mathcal{O}(n^k)$
  - Exponentielle: $\mathcal{O}(k^n)$
- On simplifie au maximum les complexités !

$$\mathcal{O}(2n) \longrightarrow \mathcal{O}(n)$$

La notation grand O

- Soit n la taille des entrées on dit que $f(n)$ est en $\mathcal{O}(g(n))$ si:

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}, \forall n \geq n_0 : \ |f(n)| \leq c \cdot |g(n)|$$

- En d'autres mots, à partir d'un certain moment, la fonction g domine la fonction f.
  - Le temps d'exécution de la fonction sera bornée.
- En pratique, on va surtout avoir besoin de simplifier les $\mathcal{O}(\dots)$
  - On enlève les constantes multiplicatives $\mathcal{O}(2n) \longrightarrow \mathcal{O}(n)$
  - En ne gardant que le pire exposant/la pire fonction **pour chaque variable d'entrée** (ici n et m)

$$\mathcal{O}(1 + n + n^2) \longrightarrow \mathcal{O}(n^2)$$

$$\mathcal{O}(n^2 + 2^n) \longrightarrow \mathcal{O}(2^n)$$

$$\mathcal{O}(n^2 + 2^n + m + m^2) \longrightarrow \mathcal{O}(2^n + m^2)$$

$$\mathcal{O}(3) \longrightarrow \mathcal{O}(1)$$

$$\mathcal{O}(x) + \mathcal{O}(y) \longrightarrow max(\mathcal{O}(x), \mathcal{O}(y))$$

$$\mathcal{O}(x) \cdot \mathcal{O}(y) \longrightarrow \mathcal{O}(x \cdot y)$$

Comment calculer la complexité (dans le pire des cas)

- On parcourt notre algorithme et on ajoute la complexité de chaque ligne
  - Si on a une déclaration simple
    - La complexité est par défaut 1
    - Si on appelle une ou plusieurs fonctions, la complexité est la somme des complexités des appels de fonctions
  - Si on a une boucle:
    - La complexité est la complexité du contenu multipliée par le nombre de répétitions maximal dans le pire des cas.
  - Si on a une condition, on calcule la complexité de chaque branche et on prend la pire.
    - Cas spécial: La condition permet d'arrêter une récursion (voir plus loin)
  - Pour les boucles et les conditions, il faut aussi compter la vérification des conditions.
- À la fin, on combine toutes les complexités et on les simplifie pour obtenir un $\mathcal{O}(\dots)$
- On peut montrer que notre complexité est optimale en donnant un exemple d'entrée ayant la pire complexité.

Exemple 1 - Factorielle (pseudo-code)

```
factorielle(n):
  fact = 1;                  # 1
  for(i=1; i <= n; i++):     # 1 à chaque iter et Max iter: n
    fact = fact * i;         # 1
  return fact;
```

Complexité algorithmique: $\mathcal{O}(1 + n \cdot 1) = \mathcal{O}(n)$

Complexité mémoire: $\mathcal{O}(1)$ (2 variables créés)

Exemple 2 - Factorielle récursive (pseudo-code)

```
factorielle(n):
  if (n == 0):    # 1
      return 1;   # 1
  else:           # Pire des cas
      return n * factorielle(n - 1);   # 1 + C(n-1)
```

Complexité algorithmique:

- On l'appelle $C(n)$
- On a $C(n) = \mathcal{O}(1) + C(n-1)$ et $C(0) = \mathcal{O}(1)$
- On retrouve bien n fois $\mathcal{O}(1)$ (attention, on a bien un multiplication ici):

$$\mathcal{O}(1) + \ldots + \mathcal{O}(1) = \mathcal{O}(n)$$

Exemple 3 - Fibonacci (pseudo-code)

```
fibo(n):
  if (n == 0){ # 1
    return 0;  # 1
  else if (n == 1){ # 1
    return 1;       # 1
  else:             # Pire des cas
      return fibo(n - 1) + fibo(n - 2);  # C(n-1) + C(n-2)
```

Complexité algorithmique:

- On a $C(n) = C(n-1) + C(n-2) + \mathcal{O}(1)$ et $C(0) = \mathcal{O}(1)$ $C(1) = \mathcal{O}(1)$

Exemple 3 - Fibonacci - Calcul abbrégé

$$
\begin{aligned}
C(n) &= C(n-1) + C(n-2) + \mathcal{O}(1) \\
&= 2 \cdot C(n-2) + C(n-3) + 2 \cdot \mathcal{O}(1) \\
&= 3 \cdot C(n-3) + 2 \cdot C(n-4) + 4 \cdot O(1) \\
&= 5 \cdot C(n-4) + 3 \cdot C(n-5) + 7 \cdot O(1) \\
&= \dots \\
&= Fibo(k+1) \cdot C(n-k) + Fibo(k) \cdot C(n-k-1) + \left(\sum_{i=1}^{k} Fibo(i) \right) \mathcal{O}(1) \\
&= \dots = Fibo(n+1)C(1) + Fibo(n)C(0) + \left(\sum_{i=1}^{n} Fibo(k) \right) \mathcal{O}(1) \\
&= Fibo(n+1) \cdot \mathcal{O}(1) + Fibo(n) \cdot \mathcal{O}(1) + Fibo(n+2) \cdot \mathcal{O}(1) \\
&= \mathcal{O}(\phi^n)
\end{aligned}
$$

c.f la prépa

Exemple 4 - Fibonacci 2 (pseudo-code)

```
fibo2(n):
  if (n == 0){ # 1
    return 0;  # 1
  else if (n == 1){ # 1
    return 1;       # 1
  else:             # Pire des cas
    fibo_tmp = new int[n+1]; # n
    fibo_tmp[0] = 1; fibo_tmp[1] = 1; # 1
    for(i=2; i<=n; i++): # 1, Max iter: n
      fibo_tmp[i] = fibo_tmp[i-1] + fibo_tmp[i-2]; # 1
    return fibo_tmp[n] # 1
```

Complexité: $\mathcal{O}(n)$

Example 5 - Écart Type

```
ecartType(nombres):
  moyenne = mean(nombres);   # complexité mean=len(nombres)=n
  variance = 0;              # 1
  for (i=0; i < len(nombres); i++):          # 1, Max iter:n
    variance += (nombres[i] - moyenne) ** 2; # 1
  variance /= n;          # 1
  return sqrt(variance)  # 1
```

$$C(n) = \mathcal{O}(n + n \cdot 1 + 1) = \mathcal{O}(n)$$

Exemple 6 - Dichotomie

```
dichotomie(element, tableau_trie):
  if (tableau_trie.length <= 1):  # 1
    return 0;                     # 1
  m = tableau_trie.length // 2    # 1
  if tableau_trie[m] == element:  # 1
    return m                      # 1
  else if tableau_trie[m] > element :          # 1
    return dichotomie(element,tableau_trie[:m])# C(n//2)
  else:
    return m + dichotomie(element,tableau_trie[m:])# C(n//2)
```

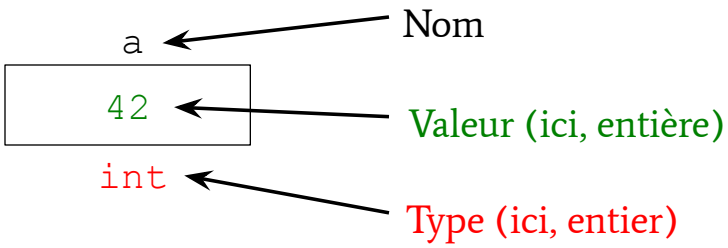
$$\begin{aligned} C(n) &= \mathcal{O}(1) + \mathcal{C}(n/2) \\ &= \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{C}(n/4) \\ &= \ldots = \log_2(n) \cdot \mathcal{O}(1) \\ &= O(\log_2(n)) \end{aligned}$$

Les méthodes de classe

Algorithmique et langage de programmation
Gaël Thomas

Rappel : la variable

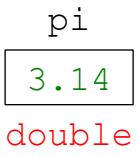
- Une variable **est** un emplacement mémoire
  - Qui possède **un nom**, **un type** et **une valeur**



Rappel : primitif versus référence

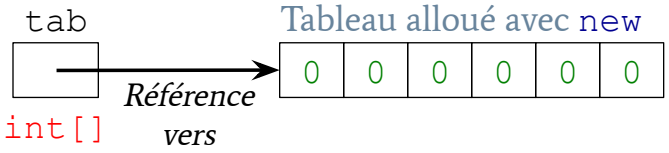
- Une variable contient
  - Soit une **valeur de type dit primitif**
    (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`)

```
double pi = 3.14
```



  - Soit une **valeur de type dit référence**
    (identifiant unique de tableau ou de `String`)

```
int[] tab = new int[6];
```



Qu'est-ce qu'une méthode ?

- Une méthode est un regroupement d'instructions
  - Création d'une macro-instruction
    - Permet de réutiliser du code
  - Peut prendre des arguments et renvoyer un résultat

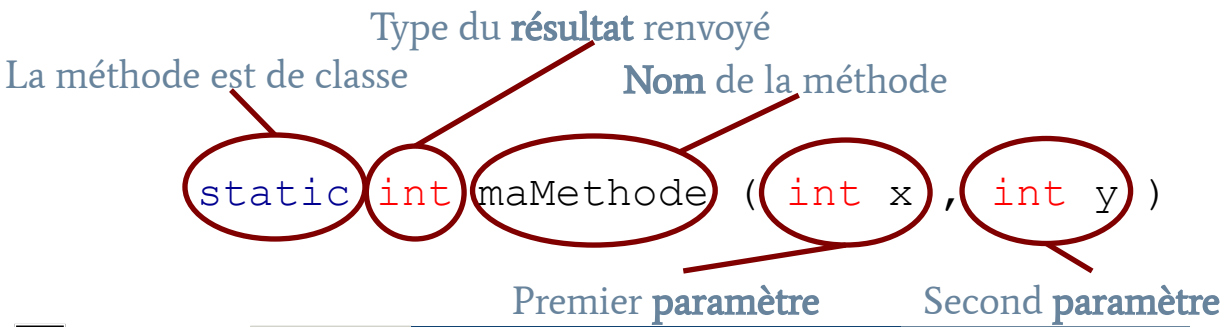
Qu'est-ce qu'une méthode ?

- Une méthode est un regroupement d'instructions
  - Création d'une macro-instruction
    - Permet de réutiliser du code
  - Peut prendre des arguments et renvoyer un résultat

- Dans ce cours, on étudie les méthodes **de classe**
  - Il existe aussi les méthodes d'instance, la différence sera expliquée en CI4

Définition d'une méthode de classe (1/2)

- Une méthode de classe possède
  - Un **nom** (⇔ nom de la macro-instruction)
  - Une liste de **paramètres** d'entrées sous la forme `type symbol`
  - Le type de **résultat** renvoyé (`void` si pas de résultat)



Définition d'une méthode de classe (2/2)

- Une méthode de classe possède un corps délimité par `{` et `}`
  - Le corps contient une suite d'instructions
  - Termine avec `return resultat;` (`return;` si pas de résultat)

```
static int maMethode(int x, int y) {
  if(y == 0) {
    System.out.println("div par 0");
    return -1;
  }
  return x / y;
}
```

Corps de la méthode

Termine la méthode de classe

La méthode main

Nom spécial qui indique que le programme commence ici

Ne renvoie pas de résultat

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

Déclaration magique (expliquée en cours 4)

Prend un unique paramètre : les arguments passés au programme

Invocation d'une méthode de classe

- Utilisation
  - Invocation avec `nomMethode(arg1, arg2…)`
  - Après l'invocation, l'expression est remplacée par le résultat

```
class MaClasse {
  static int add(int x , int y ) {
    return x + y;
  }
  public static void main(String[] args) {
    int a = 2;
    int res = add( 1 , a );
    System.out.println("1 + 2 = " + res);
  }}
```

Invoque `add` avec x valant 1 et y valant 2

Invocation d'une méthode de classe

- Utilisation
  - Invocation avec `nomMethode(arg1, arg2…)`
  - Après l'invocation, l'expression est remplacée par le résultat

```
class MaClasse {
  static int add(int  x  , int  y  ) {
    return x + y;
  }
  public static void main(String[] args) {
    int a = 2;
    int res = add( 1  ,  a  );
    System.out.println("1 + 2 = " + res);
  }}
```

Exécute `add` puis remplace par le résultat (ici 3)

Invocation d'une méthode de classe

- Utilisation
  - Invocation avec `nomMethode(arg1, arg2…)`
  - Après l'invocation, l'expression est remplacée par [illegible]

```
c
    int res = add( 1   ,   a   );
    System.out.println("1 + 2 = " + res);
  }}
```

Attention

Les méthodes sont déclarées dans une classe, pas dans d'autres méthodes

La surcharge de méthode

- Java permet de **surcharger** des méthodes
  - Deux méthodes peuvent avoir le **même nom** pourvu qu'elles n'aient **pas les mêmes types de paramètres d'entrée**
  - La méthode appelée est sélectionnée en fonction du type des arguments

```
class Test {
  static void f(double x) { … }
  static void f(int x) { … }
  static void g() { f(42); }
}
```

Appelle `f(int x)` car `42` est un entier

Mais où est le `return` du main ?

- Si une méthode ne retourne rien, un return est implicitement ajouté à la fin du corps de la méthode

```
public static void main(String[] args) {
  int res = add(1, 2);
  System.out.println("1 + 2 = " + res);
}
```

```
public static void main(String[] args) {
  int res = add(1, 2);
  System.out.println("1 + 2 = " + res);
  return;
}
```


Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode
- Lors d'une invocation de méthode, l'environnement d'exécution:
  - Crée un **cadre d'appel** pour accueillir les variables locales et les paramètres
  - Crée les variables locales/paramètres dans le cadre
  - Affecte les paramètres
- À la fin de l'invocation, l'environnement d'exécution
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

Variables locales (1/2)

- Variable locale = variable définie dans une méthode
  - **N'existe que le temps de l'invocation** de la méthode
  - Il en va de même des paramètres de la méthode
- Lors d'u on:
  - Cré es
  - Cré
  - Affe
- À la fin de l'invocation, l'environnement d'exécution
  - Détruit le cadre, ce qui détruit les variables locales/paramètres

cadre d'appel
=
call frame en anglais

Variables locales (2/2)

- Avant l'entrée dans `main`
  - Le tableau des arguments est initialisé

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

Tableau des arguments

Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de main

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

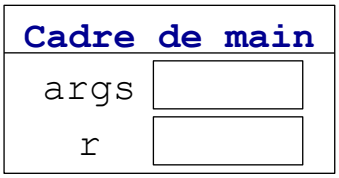


Tableau des arguments

Variables locales (2/2)

- À l'entrée dans `main`
  - Création cadre de main
  - Affectation paramètre `args` (référence vers tab. arguments)

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

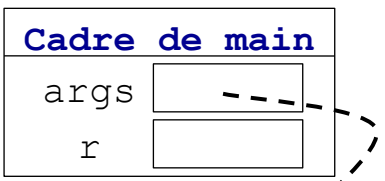
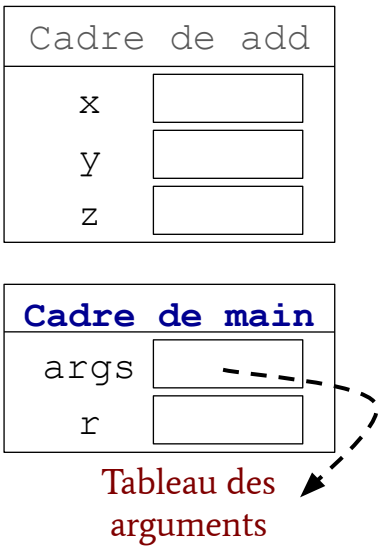


Tableau des arguments

Variables locales (2/2)

- À l'entrée dans `add`
  - Création cadre `add`

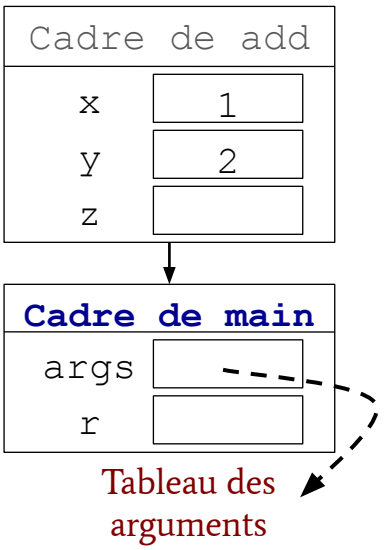
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À l'entrée dans `add`
  - Création cadre `add`
  - Affectation paramètres

```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```

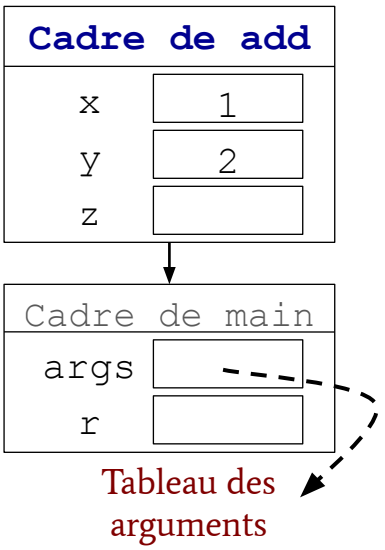


Variables locales (2/2)

- À l'entrée dans `add`
  - Activation du cadre de `add`

Cadre actif en bleu/gras

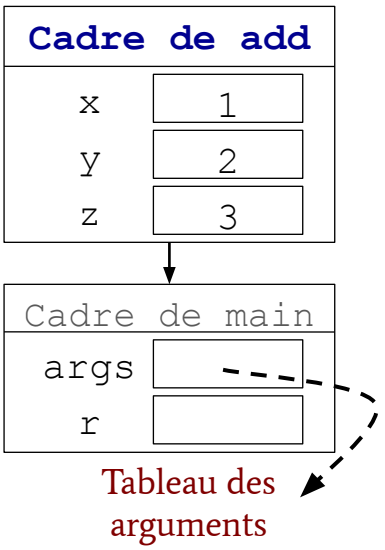
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- Pendant `add`
  - Utilisation des variables locales du **cadre actif**

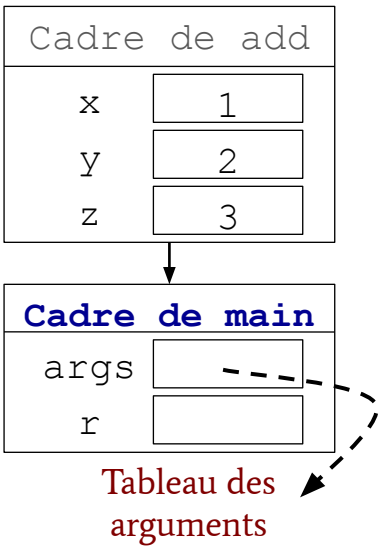
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À la sortie de `add`
  - Activation cadre précédent

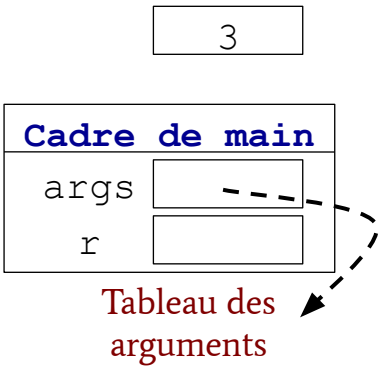
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À la sortie de `add`
  - Activation cadre précédent
  - Suppression cadre précédent + récupère valeur de retour

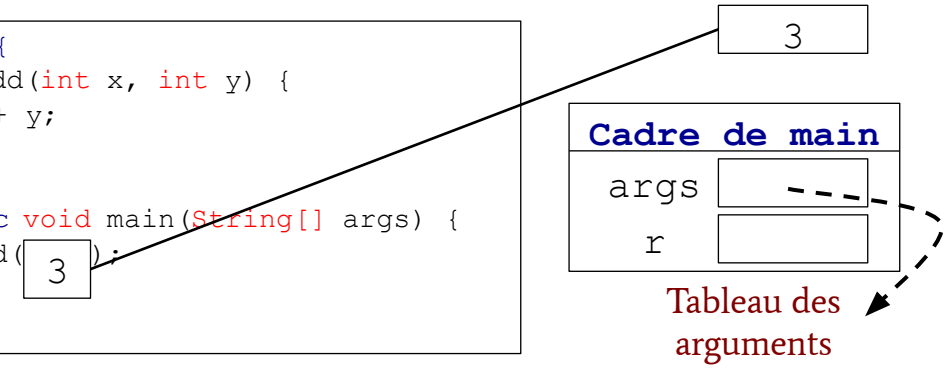
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat

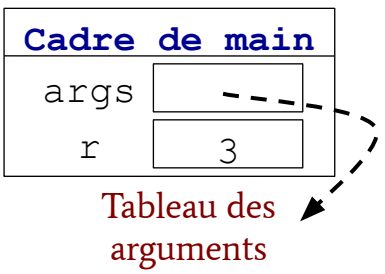
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales (2/2)

- À la sortie de `add`
  - Reprend l'exécution dans l'appelant avec le résultat

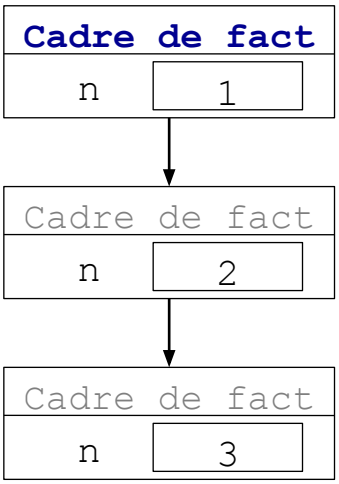
```
class MaClass {
  static int add(int x, int y) {
    int z = x + y;
    return z;
  }
  public static void main(String[] args) {
    int r = add(1, 2);
  }
}
```



Variables locales et appels récursifs

- Si une méthode s'appelle elle-même
  - Un nouveau cadre à chaque appel
    ⇒ de nouvelles variables locales à chaque appel

```
static int fact(int n) {
  if(n < 1)
    return 1;
  else
    return n * fact(n - 1);
}
/* fact(3) appelle fact(2)
   qui appelle fact(1) */
```



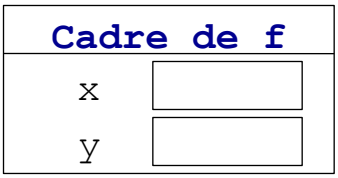
Passage par valeur et par référence

- Le passage des arguments peut se faire par
  - Valeur : l'appelé reçoit une copie d'une valeur
    - la copie et l'originale **sont différentes**
  - Référence : l'appelée reçoit une référence vers une valeur
    - la valeur est **partagée** entre l'appelant et l'appelé

- En Java :
  - Passage par valeur pour les 8 types primitifs
    - (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` et `double`)
  - Passage par référence pour les autres types
    - (`String` et tableaux pour l'instant)

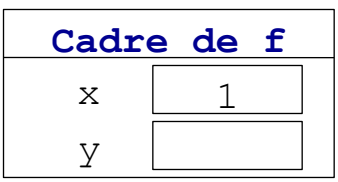
Passage par valeur – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur – exemple

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```

Cadre de f	
x	1
y	2

Passage par valeur – exemple

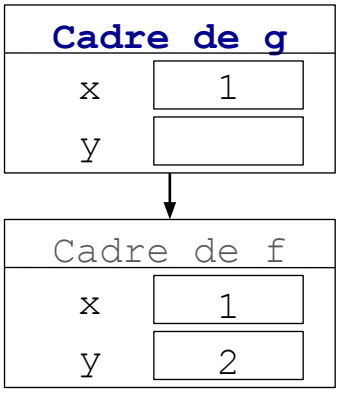
```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```

Cadre de f	
x	1
y	2

Passage par valeur – exemple

- À l'entrée dans g
  - Le x du cadre de f **est copié** dans le x du cadre de g

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



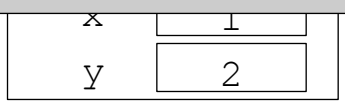
Passage par valeur – exemple

Attention !

Le fait que les variables de f et g aient
le même nom n'a aucune influence sur l'exécution

Si les variables de g se nommaient a et b,
le programme fonctionnerait exactement de la même façon !

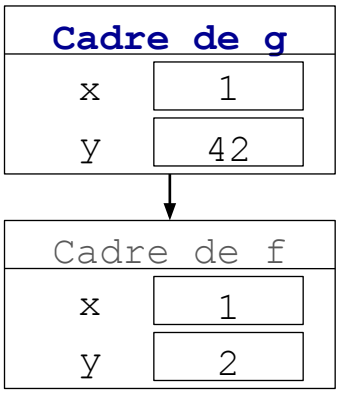
```
  g(x);
}
```



Passage par valeur – exemple

- L'affectation du `y` de `g` ne change pas la valeur du `y` de `f`

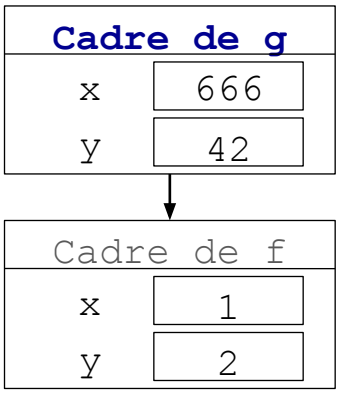
```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par valeur – exemple

- `g` modifie la copie de la variable `x` de `f`, pas celle de `f`

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



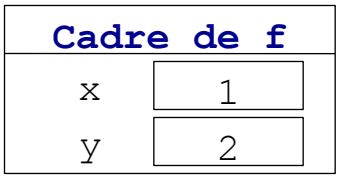
Passage par valeur – exemple

Pour résumer :

Les variables de l'appelant

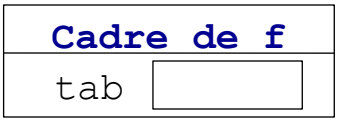
ne sont **jamais** modifiées par l'appelé

```
static void g(int x) {
  int y = 42;
  x = 666;
}

static void f() {
  int x = 1;
  int y = 2;
  g(x);
}
```



Passage par référence – exemple

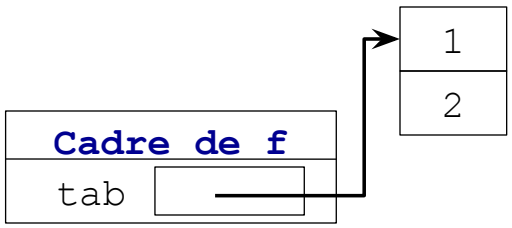
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par référence – exemple

- Rappel :
  - Un tableau est alloué dans la mémoire
  - La variable `tab` contient une **référence** vers ce tableau

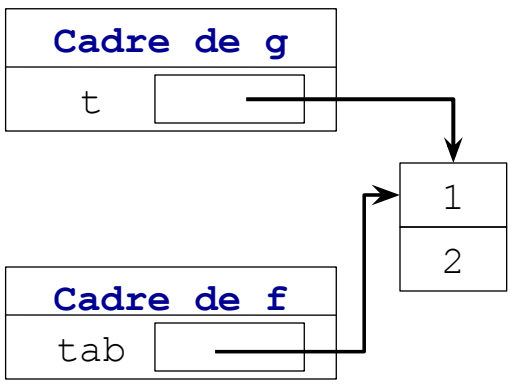
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par référence – exemple

- À l'entrée de g
  - `t` reçoit une copie de `tab`
  - `tab` étant une référence, **`t` et `tab` référence le même tableau**
    - le tableau est passé par référence

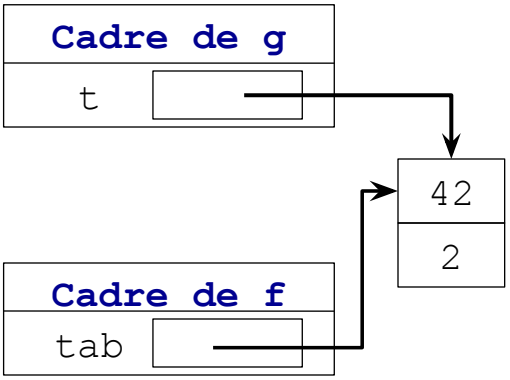
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par référence – exemple

- La modification dans `g` modifie le tableau référencé par `tab`

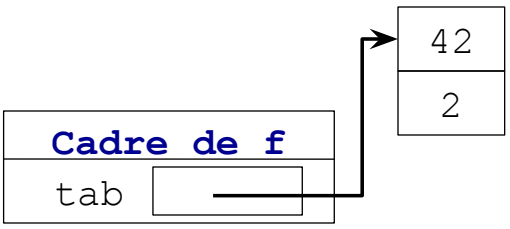
```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par référence – exemple

- L'appelant peut donc modifier une valeur passée par référence

```
static void g(int[] t) {
  t[0] = 42;
}

static void f() {
  int[] tab = { 1, 2 };
  g(tab);
}
```



Passage par valeur et par référence

Pour résumer :

Les variables de l'appelant

ne sont **jamais** modifiées par l'appelé

En revanche, les valeurs **référencées**

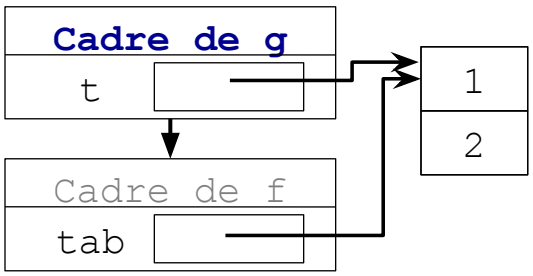
à la fois par l'appelant et l'appelé

peuvent être modifiées par l'appelé

Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

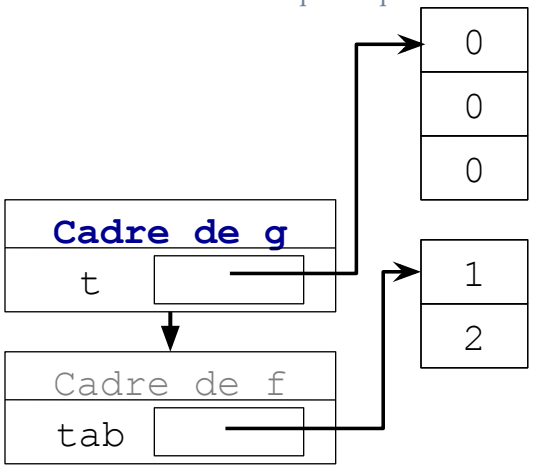
```
void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```



Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

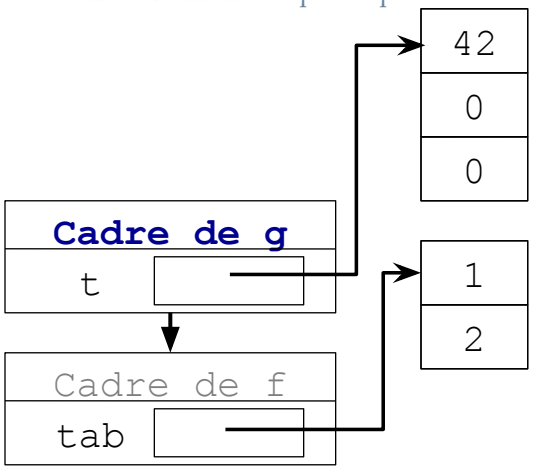
```
void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```



Piège !

- Attention, un tableau est passé par référence, mais la référence elle-même est passée par valeur...

```
void g(int[] t) {
  t = new int[3];
  t[0] = 42;
}

void f() {
  int[] tab = { 1, 2 };
  g(tab);
  System.out.println(tab[0]);
}
```



Notions clés

- Déclaration d'une méthode de classe
  ```
  static type nom(type param1, …) { corps }
  ```
- Appel d'une méthode de classe
  ```
  nom(arg1, …)
  ```
- Notions de cadre d'appel et de variables locales
  - Le cadre d'appel est détruit à la fin de l'invocation
- Passage par valeur et passage par référence
  - Par valeur pour les 8 types primitifs
  - Par référence pour les autres types

Les structures de données

Algorithmique et langage de programmation

Gaël Thomas, Julien Romero

1. **Les structures de données : objets et classes**
2. Manipulation de tuples en Java
3. Tuples versus tableaux en Java
4. Objets et références
5. Objets et invocation de méthodes

Les structures de données

- Structure de données = regroupement de données
  - Permet de lier entre elles des données
  - Simplifie le traitement de ces données

- Exemples
  - Une structure de données « Personnage » regroupant
    - une image (l'apparence du personnage),
    - une position,
    - un nombre de points de vie...
  - Une structure de données « ListePersonnages » regroupant
    - un ensemble de personnages

Deux familles de structures de données

- Le tableau (vu au CI2)
  - Regroupe un nombre fini d'éléments **homogènes**
  - Les éléments sont indexés par un **numéro**

- Le tuple (vu dans ce CI)
  - (aussi parfois appelé enregistrement ou structure)
  - Regroupe un nombre fini d'éléments **hétérogènes**
  - Les éléments sont indexés par un **symbole**
  - Les éléments s'appellent des **champs** ou **attributs**

En Java

- Une structure de données (tuple ou tableau) s'appelle un **objet**
- Un **objet** possède un **type**
- Le **type d'un objet** s'appelle une **classe**
- Si la classe d'un objet o est C, alors on dit que **o est une instance de C**

Théorie des types et théorie des ensembles

- Une grande partie de l'informatique est basée sur la **théorie des types**
  - Un programmeur ou une programmeuse crée des types de plus en plus complexes à partir des types primitifs pour modéliser des problèmes
- La théorie des types ressemble à la théorie des ensembles utilisée en maths
  - `int[n]` représente $\mathbb{Z}^n$
  - `int[]` représente $\mathbb{Z}^n$ pour tous les n
  - Un tuple peut s'apparenter à un produit cartésien
    - Une image représentée par un tableau en deux dimensions de triples : $(\mathbb{R} \times \mathbb{R} \times \mathbb{R})^{n \cdot n}$
    - Une position : $\mathbb{N} \times \mathbb{N}$
    - Un nombre de points de vie : $\mathbb{N}$
    - Un personnage : $\text{Image} \times \text{Position} \times \text{Point de vie}$

1. Les structures de données : objets et classes
2. **Manipulation de tuples en Java**
3. Tuples versus tableaux en Java
4. Objets et références
5. Objets et invocation de méthodes

Deux étapes pour créer un tuple

- Étape 1 : définition de la **classe** d'un tuple (i.e., de son type)
  - Donne une énumération des champs du tuple
  - Utilisation du mot clé `class` suivi d'un identifiant de type

```
class Perso {
  int pointsVie;
  int x;
  int y;
}
```

- Étape 2 : création d'une **instance** de la classe avec `new`

```
Perso bilbon = new Perso();
```

⇒ `bilbon` référence une instance de la classe `Perso`

Accès aux champs d'un tuple avec « . »

```
class Perso {
  int pointsVie;
  int x;
  int y;
}
```

```
class MonProg {
  public static void main(String[] a) {
    Perso bilbon = new Perso();

    bilbon.pointsVie = 10;
    bilbon.x = 0;
    bilbon.y = 0;

    Perso sauron = new Perso();

    sauron.pointsVie = 1000;
    sauron.x = 666;
    sauron.y = 666;
  }
}
```


Ne confondez pas classe et instance !

Une **classe** est une sorte de **moule**

Perso
pointsVie: int x: int y: int

Qui permet de créer des **instances** de même type

Perso
pointsVie:int = 10 x:int = 0 y:int = 0

Perso
pointsVie:int = 1000 x:int = 666 y:int = 666

Conventions de codage (1/2)

- Quand on code en Java, on utilise les conventions suivantes :
  - Les noms de classes des tuples commencent par une majuscule
  - Les variables et champs commencent par une minuscule
  - Les méthodes commencent par une minuscule

⇒ Visuellement, si on voit un symbole commençant par une majuscule, on sait qu'on parle d'une classe

Conventions de codage (2/2)

- On ne définit qu'une et une seule classe par fichier source
  (sauf pour les classes internes et anonymes, voir CI8)
- Le fichier source définissant la classe X s'appelle X.java

1. Les structures de données : objets et classes
2. Manipulation de tuples en Java
3. **Tuples versus tableaux en Java**
4. Objets et références
5. Objets et invocation de méthodes

Tuples versus tableaux : nommage

- La **classe d'un tuple** possède un nom librement défini
  - Mot clé `class` suivi du nom et de l'énumération des champs

```
class Perso { int pointsVie; int x; int y; }
```

Nom de la classe

- La **classe d'un tableau** possède un nom imposé
  - Type des éléments suivi du symbole `[]`

```
int[]
```

Nom de la classe

Tuples versus tableaux : allocation

- Un objet est alloué avec le mot clé `new` suivi de la classe
  - Suivi de parenthèses dans le cas des tuples, mais pas des tableaux
  - `new` renvoie une référence vers l'objet alloué

- Par exemple
  - `new Perso()` ⇒ alloue une instance de le classe `Perso`
  - `new int[5]` ⇒ alloue un tableau de 5 `int`

Tuples versus tableaux : accès

- Accès à un champ d'un tuple :
  - variable suivie d'un point et du nom du champ
  - `sauron.pointsVie = 1000;`

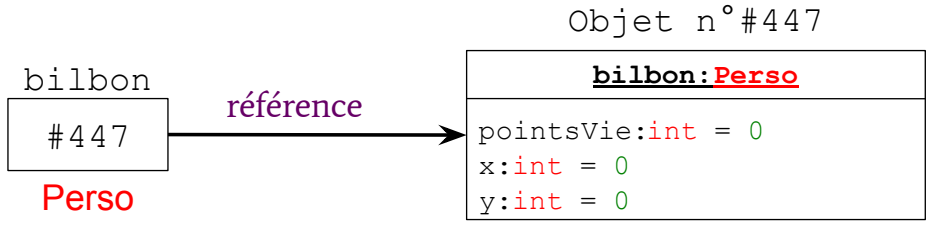
- Accès à élément d'un tableau :
  - variable suivie d'un indice entre crochets
  - `tab[3] = 42;`

1. Les structures de données : objets et classes
2. Manipulation de tuples en Java
3. Tuples versus tableaux en Java
4. **Objets et références**
5. Objets et invocation de méthodes

Objets et références (1/3)

- Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- `Perso p` déclare une **référence** vers un objet de type `Perso`

```
Perso bilbon = new Perso();
```



Objets et références (2/3)

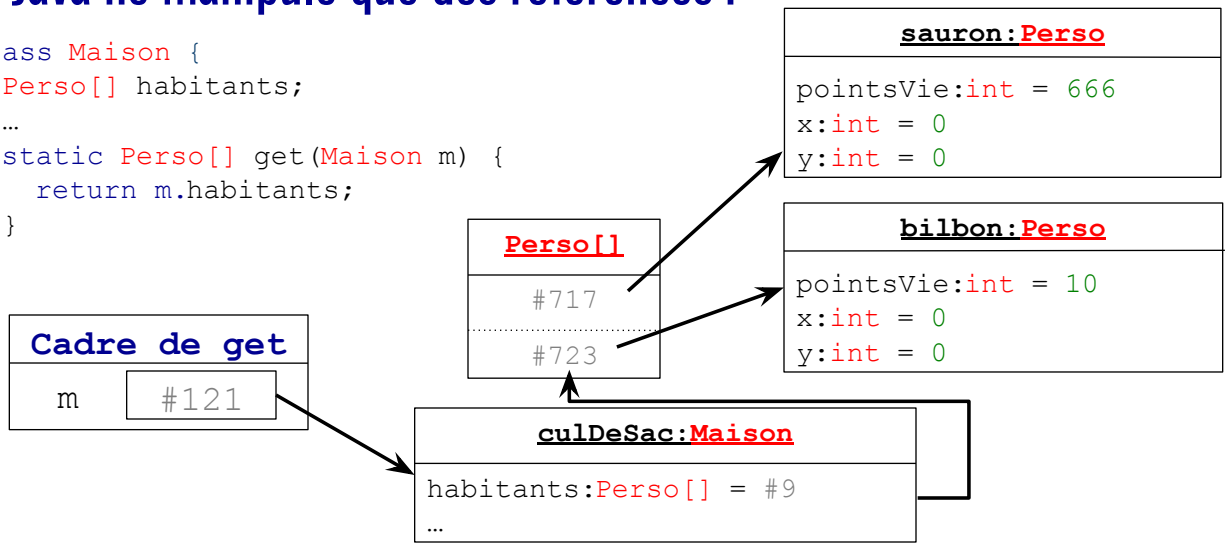
- Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- `Perso p` déclare une **référence** vers un objet de type `Perso`
- De la même façon, `int[] tab` déclare une **référence** vers un objet de type `int[]`

Objets et références (2/3)

- Java définit deux entités distinctes
  - Un objet est une structure de données en mémoire
  - Une référence est **un identifiant unique** d'un objet
- `Perso p` déclare une **référence** vers un objet de type `Perso`
- De la même façon, `int[] tab` déclare une **référence** vers un objet de type `int[]`
- Et `Perso[] tab` déclare donc une **référence** vers un tableau dans lequel chaque élément est une **référence** vers un `Perso`

Java ne manipule que des références !

```
class Maison {
  Perso[] habitants;
  …
  static Perso[] get(Maison m) {
    return m.habitants;
  }
}
```



La référence littérale `null`

- **`null`** : valeur littérale indiquant qu'aucun objet n'est référencé

```
Maison m = new Maison();
Perso bilbon = new Perso();

m.proprio = null; /* pas encore de propriétaire */
…
if(m.proprio == null)
  m.proprio = bilbon;
```

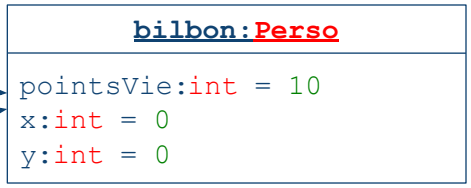
- Par défaut les champs (resp. éléments) de type références d'un tuple (resp. tableau) sont initialisés à `null`

1. Les structures de données : objets et classes
2. Manipulation de tuples en Java
3. Tuples versus tableaux en Java
4. Objets et références
5. **Objets et invocation de méthodes**

Objets et invocation de méthodes

- On dit que les objets sont passés par **référence** en Java (puisque le code ne manipule que des références)

```
static void init(Perso p) {
  p.pointsVie = 10;
}

static void f() {
  Perso bilbon = new Perso();
  init(bilbon)
  System.out.println(" ⇒ " + bilbon.pointsVie);
  // affiche 10 car init reçoit une référence vers bilbon
}
```



Invocation inter-classe de méthode (1/2)

- Le mot clé `class` a deux rôles différents en Java
  - Comme espace pour définir des classes définissant des tuples
  - Comme espace pour définir des méthodes de classe

```
class Perso {
  int pointsVie;
  int x;
  int y;
}
```

```
class MonProg {
  static void maFonction(int x) {
    …
  }
}
```

- On peut bien sûr combiner les deux rôles

```
class Perso { int pv; static void maFonc() { … } }
```


Invocation inter-classe de méthode (2/2)

- Par défaut, Java résout un appel de méthode dans la classe
  - Pour appeler une méthode d'une autre classe :
    préfixer le nom de la méthode avec la classe suivi d'un point

```
class MonProg {
  static void maFonction(int x) {
    Perso bilbon = new Perso();
    Perso.display(bilbon);
  }
}
```

```
class Perso {
  int pointsVie;

  static void display(Perso p) {
    …
  }
}
```

Notions clés

- Déclaration d'une classe définissant un tuple avec
  `class Nom { type1 champs1; type2 champs2; … }`

- Allocation d'un objet avec l'opérateur `new`
  `new Nom()` si tuple ou `new type[n]` si tableau

- En Java, il n'existe que des types références, pas de type objet

- Lors d'un appel de méthode, un objet est passé par référence

Les méthodes d'instance

Algorithmique et langage de programmation
Gaël Thomas

Petits rappels : l'objet

- Une structure de données (tuple ou tableau) s'appelle un **objet**
- Un **objet** possède un **type**
- Le **type d'un objet** s'appelle une **classe**
- Si la classe d'un objet o est C, alors on dit que **o est une instance de C**
- En Java, on ne manipule que des **références** vers des objets

Limite de la programmation impérative

- Prog. impérative = celle que vous utilisez jusqu'à maintenant
- Un programme est constitué de
  - Structures de données
  - Et de méthodes qui manipulent ces structures
- Pour réutiliser une structure de données dans un autre projet
  - Il faut trouver la structure de données et la copier
  - Mais il faut aussi **trouver les méthodes qui manipulent la structure de données et les copier**

    ⇒ **nécessite une structure claire du code**

Solution partielle utilisée en CI3

- Faire preuve de discipline quand on programme
  - Regrouper les méthodes qui manipulent une structure de données dans la classe qui définit la structure de données
  - Éviter de manipuler directement une structure de données à partir de l'extérieur de la classe
- Solution partiellement satisfaisante car aucune aide fournie par le langage
- Méthode d'instance : aide à faire preuve de discipline

La méthode d'instance

- But : simplifier la définition des méthodes qui manipulent une structure de données

- Principe : associer des méthodes aux instances
  - Méthode d'instance = méthode **sans** mot clé `static`
  - Méthode qui manipule une structure de données
  - Associée à la classe dans laquelle la méthode est définie
  - Reçoit un **paramètre caché nommé `this`** du type de l'instance
    ⇒ pas besoin de spécifier explicitement ce paramètre
    ⇒ simplifie le code

Méthode d'instance vs de classe

Avec méthode d'instance

```
class Monster {
  int health;

  void kill() {
    this.health = 0;
  }
}
```

Avec méthode de classe

```
class Monster {
  int health;

  static void kill(Monster m) {
    m.health = 0;
  }
}
```

Pas de static
⇒ paramètre Monster this implicitement ajouté

Méthode d'instance vs de classe

Avec méth asse

```
class Monster
  int health;

  void kill()
    this.heal
  }
}
```

m) {

On dit que **`this`** est le **receveur** de l'appel

Pas de static
⇒ paramètre Monster this implicitement ajouté

Invocation d'une méthode d'instance

- Le receveur d'un appel de méthode d'instance se met à gauche

```
Monster aMonster = Monster.create(aPicture);

aMonster.kill();
```

receveur

Méthode d'instance appelée

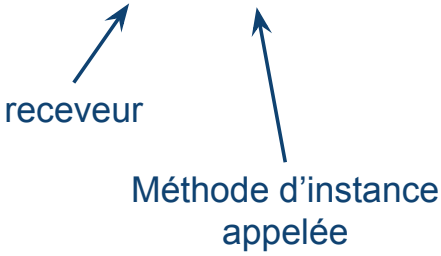
Un peu comme si on invoquait
`Monster.kill(aMonster);`

(i.e., `this` reçoit la valeur `aMonster`)

Invocation d'une méthode d'instance

- Le receveur d'un appel de méthode d'instance se met à gauche

```
Monster aMonster = Monster.create(aPicture);

aMonster.kill();
```



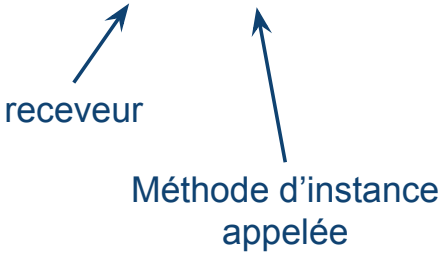
Remarque :

On appelle le `kill` de `Monster` car la classe du receveur (`aMonster`) est `Monster`

Invocation d'une méthode d'instance

- Le receveur d'un appel de méthode d'instance se met à gauche

```
Monster aMonster = Monster.create(aPicture);

aMonster.kill();
```



Remarque :

On appelle le `kill` de `Monster` car la classe du receveur (`aMonster`) est `Monster`

Remarque : si `aMonster` vaut `null` ⇒ erreur de type `NullPointerException`

Omission de `this`

```
class Monster {
  int health;

  void kill() {
    this.health = 0;
  }
}
```

⇔

```
class Monster {
  int health;

  void kill() {
    health = 0;
  }
}
```

En l'absence d'ambiguïté, `this` peut être omis

(`health` champ de `Monster`
⇒ remplacé par `this.health` à la compilation)

Omission de `this`

```
class Monster {



}
```

Attention !

N'oubliez pas qu'il y a un receveur `this` caché pour les méthodes d'instance

En l'at

(healt
⇒

...a la compilation)

En cas d'ambiguïté, utilisez `this`

Java utilise la portée lexicale : le compilateur cherche le symbole le plus proche en suivant les blocs de code

```
class Monster {
  int      health;
  int      x;
  int      y;

  void move(int x, int y) { this.x = x; this.y = y; }
}
```

`this` est nécessaire pour lever l'ambiguïté entre le champ `x` et l'argument `x`

Notions clés

- Avec `static`, la méthode s'appelle une **méthode de classe**
  - Marquée par le mot clé `static`
  - Uniquement les paramètres explicites

```
static void kill(Monster monster)
```

- Sans `static`, la méthode s'appelle une **méthode d'instance**
  - Reçoit un paramètre caché nommé `this` du type de la classe

```
void kill()
⇔
static void kill(Monster this)
```

  - Invocation avec receveur à **gauche** : `monster.kill();`

Notions clés

Attention !

Ce cours est court, mais **essentiel**

Ne confondez pas les méthodes de classe et les méthodes d'instance !

Les packages

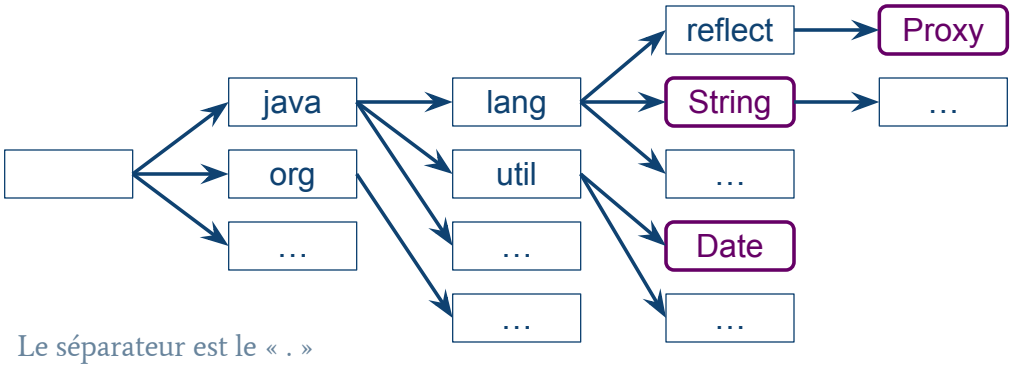
Algorithmique et langage de programmation
Gaël Thomas

À quoi servent les packages

- À ranger des classes, un peu comme des dossiers
  - Permet d'organiser l'ensemble des fichiers contenant le code
  - Évite les conflits de nom
    - (Que faire si deux développeurs nomment une de leurs classes `Test` ?)
- Parallèle avec CSC3102
  - Les packages sont les répertoires
  - Les classes sont les fichiers ordinaires

Organisation des packages

- Arbre dans lequel les feuilles sont des classes



- Le séparateur est le « . »

  Exemple : `java.util.Date` : classe `Date` dans `java.util`

Placer une classe dans un package

- Ajouter `package chemin-du-package;` au début du fichier

- Par exemple :

```
package org.tsp.csc3101;
class MaClass { … }
```

- Dans ce cas, la classe doit être définie dans `org/tsp/csc3101/MaClass.java`
  - Automatique avec Eclipse ou tout autre éditeur de code de qualité

Utiliser une classe d'un autre package

- Solution 1 : préfixer le nom de la classe avec le package

Par exemple :

```
class MaClass {
  static void test() {
    java.util.TimeZone.getDefault()
  }
}
```


Utiliser une classe d'un autre package

- Solution 2 : importer la classe en début de fichier
  - Par exemple :

```
import java.util.TimeZone;

class MaClass {
  static void test() {
    TimeZone.getDefault();
  }
}
```

  - On peut aussi importer en masse toutes les classes d'un package

```
import java.util.*;
```

Résolution du nom d'une classe

- Ordre de recherche des classes
  - Si nom qualifié (c'est-à-dire avec chemin explicite comme `a.C`)
    - Recherche à partir de la racine de l'arbre des packages
  - Sinon (c'est-à-dire sans chemin explicite comme `C`)
    - Recherche dans le package courant
    - Puis recherche dans les packages importés
      - Par défaut, `java.lang.*` est importé
      - Attention, les classes du package racine ne peuvent pas être importées

- Par exemple :
  - Cherche `Monster` dans `monsters` puis dans classes importées
  - Cherche `java.util.TimeZone` uniquement à partir de la racine

Notions clés

- Organisation des classes sous la forme d'un arbre de packages

- `package a.b;` au début du fichier ⇒ déf./rech. dans `a.b`

- `import a.b.X;` ⇒ importe `a.b.X` dans la classe courante

- Ordre de recherche d'une classe :
  - Dans le fichier
  - Puis parmi les classes importés
  - Puis à partir de la racine

Programmation orientée objet

Algorithmique et langage de programmation
Gaël Thomas

Petits rappels : l'objet

- Une structure de données (tuple ou tableau) s'appelle un **objet**
- Un **objet** possède un **type** appelé **sa classe**
  - Si la classe de l'objet o est C, on dit que **o est une instance de C**
- En Java, on ne manipule que des **références** vers des objets
- Une **méthode d'instance** est une méthode associée à l'objet
  - Possède un paramètre implicite du type de la classe nommé `this`

But de la programmation orientée objet

Améliorer la réutilisabilité du code

car une ligne de code coûte très cher !

(~1h de développement par ligne de code)

Que signifie réutiliser du code ?

- Quand on réutilise du code, on est en général intéressé par une **fonctionnalité**, pas par une mise en œuvre spécifique

- L'exemple de la classe `Army` dans l'armée de monstres
  - Objet sur lequel je peux appeler `addMonster`
  - Mais savoir que les monstres sont stockés dans un tableau extensible ou une liste chaînée n'est pas essentiel
    (sauf pour les performances)

Programmation orientée objet

- Concevoir une application en terme **d'objets** qui **interagissent**
  - Au lieu de la concevoir en terme de structures de données et de méthodes (programmation impérative)

⇒ On ne s'intéresse plus à la mise en œuvre d'un objet, mais d'abord aux fonctionnalités qu'il fournit

- **Objet** = entité du programme fournissant des fonctionnalités
  - Encapsule une structure de données et des méthodes qui manipulent cette structure de données
  - Expose des fonctionnalités

L'objet en Java

- Contient une mise en œuvre
  - Des champs
  - Des méthodes d'instances
  - Des **constructeurs** (méthodes d'initialisation vues dans ce cours)

- Expose des fonctionnalités
  - En empêchant l'accès à certains champs/méthodes/constructeurs à partir de l'extérieur de la classe

  ⇒ principe **d'encapsulation** vue dans ce cours

Plan du cours

1. **Le constructeur**
2. L'encapsulation
3. Les champs de classe

Création d'un objet

- Jusqu'à maintenant, pour créer un objet, on écrit une méthode de classe qui
  - Alloue l'objet
  - Initialise l'objet
  - Renvoie l'objet initialisé
- Par exemple :

```
static Compte create(String name) {
  Compte res = new Compte();
  res.name = name;
  res.solde = 0;
  return res;
}
```

Le constructeur

- Constructeur = méthode simplifiant la création d'un objet
- Méthode d'instance spéciale pour initialiser un objet
  - Méthode d'instance possédant le nom de la classe
  - Pas de type de retour
  - Peut posséder des paramètres
- Le constructeur est appelé automatiquement pendant un `new`
  - `new` commence par allouer un objet
  - Puis appelle le constructeur avec comme receveur le nouvel objet
  - Et, enfin, renvoie l'objet

Exemple avec la classe `Compte`

Avec constructeur

```
// alloc + constructeur
c = new Compte("Tyrion");
```

```
class Compte {
  String name;
  double solde;

  // déf. constructeur
  Compte(String name) {
    this.name = name;
    this.solde = 0;
  }
}
```

Sans constructeur

```
c = Compte.create("Tyrion");
```

```
class Compte {
  String name;
  double solde;

  static Compte create(
                          String name) {
    // alloc + constructeur vide
    Compte res = new Compte();
    res.name = name;
    res.solde = 0;
    return res;
  }
}
```

Exemple avec la classe `Compte`

Avec constructeur | Sans constructeur

Remarque :

Si pas de constructeur, Java génère un constructeur vide sans paramètre qui initialise les champs à 0/+0.0/faux/null

```
  }
}
```

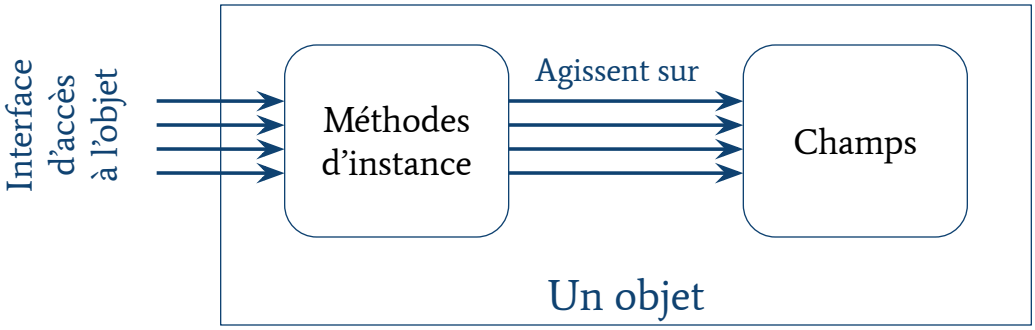
```
    res.name = name;
    res.solde = 0;
    return res;
  }
}
```


Plan du cours

1. Le constructeur
2. **L'encapsulation**
3. Les champs de classe

L'encapsulation

- Principe : cacher les détails de mise en œuvre d'un objet
  ⇒ pas d'accès direct aux champs de l'extérieur de l'objet



Mise en œuvre : l'encapsulation

- Chaque entité (classe, champ, méthode ou constructeur) possède un **niveau d'encapsulation**
  - Définit à partir d'où dans le programme une entité est visible
- Permet de masquer les détails de mise en œuvre d'un objet

L'encapsulation en Java

- Trois niveaux de visibilité pour une entité en Java
  - (techniquement quatre, mais le dernier est vu en CI6)
  - Invisible en dehors de la classe : mot clé private
  - Invisible en dehors du package : comportement par défaut
  - Visible de n'importe où : mot clé public

- En général
  - Les champs sont privés (inaccessibles en dehors de la classe)
  - Les méthodes sont publiques

L'encapsulation par l'exemple

```
package tsp.bank;

public class Bank {   /* visible partout */
  private Node node; /* invisible de l'extérieur
                                du fichier (de Bank) */

  public Bank() { … } /* visible partout */
  public void lookup() (String name) { … } /* visible
                                          partout */

  void display(Ba) { … } /* invisible en dehors
                            du package tsp.bank */
}
```

Plan du cours

1. Le constructeur
2. L'encapsulation
3. **Les champs de classe**

Les champs de classe

- Champ de classe
  - Champ précédé du mot clé `static`
  - Définit **une variable globale**, indépendante d'une instance
  - Exemple : `System.out` est un champ de classe de la classe `System`
- Attention !
  - Un champ de classe est une variable mais pas un champ d'instance : un champs d'instance est un symbole nommant un élément d'une structure de donnée
- Dans la suite du cours
  - On n'utilise pas de champs de classe
  - **On utilise static uniquement pour la méthode main**

Notions clés

- Programmation orientée objet
  - Conception d'un programme à partir d'objets qui interagissent
  - On s'intéresse à la fonctionnalité d'un objet avant de s'intéresser à sa mise en œuvre

- Le constructeur
  - Méthode appelée pendant `new` pour initialiser un objet

- Principe d'encapsulation pour cacher la mise en œuvre
  - `private` : invisible en dehors de la classe
  - Par défaut : invisible en dehors du package
  - `public` : visible de partout

L'héritage

Algorithmique et langage de programmation
Gaël Thomas

Petits rappels : l'objet

- Une structure de données (tuple ou tableau) s'appelle un **objet**
  - Un **objet** possède un **type** appelé **sa classe**
  - Si la classe de l'objet o est C, on dit que **o est une instance de C**
- La classe d'un objet définit des
  - **Champs**
  - **Méthodes d'instances**
    - Possède un paramètre implicite du type de la classe nommé `this`
  - **Constructeurs** : méthodes spéciales appelées après l'allocation
- Chacun de ces éléments a une visibilité
  - `public` (partout), `private` (local), pas de mot clé (package)

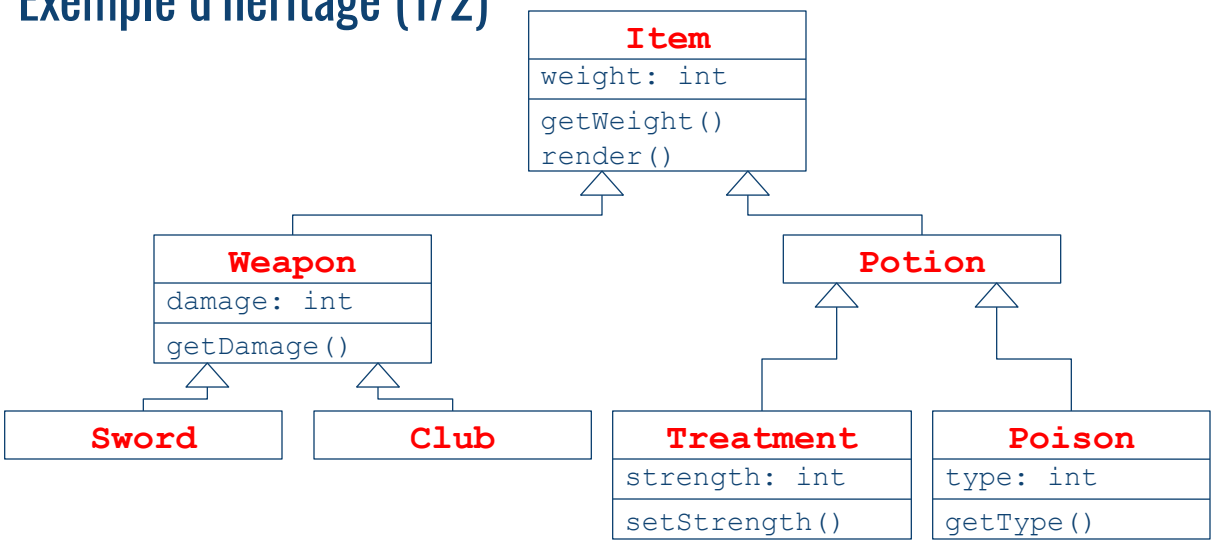
Principe de l'héritage

- Une classe dite fille peut hériter d'une autre classe dite mère
  - La classe fille possède les champs et méthodes de la mère
  - Et peut en ajouter de nouveaux pour spécialiser la classe mère
  - La classe fille définit un nouveau type
  - Ce type est compatible avec le type de la mère
- L'héritage est, par définition transitif
  - Si C hérite de B et B hérite de A, alors C hérite de A

But de l'héritage

- Améliorer la réutilisabilité du code
  - En écrivant du code générique et spécialisable
- Exemples
  - Un objet générique dans un jeu, spécialisable en épée ou tunique
  - Un nœud générique dans un arbre binaire de recherche
  - Un flux générique spécialisable en flux réseau, flux vers un fichier, flux reposant sur un tube
  - ...

Exemple d'héritage (1/2)



Exemple d'héritage (2/2)

- Le sac du joueur est constitué d'objets de types `Item`
  - Affichés à l'écran avec la méthode `render()`
  - Le poids du sac est calculé en appelant `getWeight()` sur chaque `Item`

  ⇒ le code de gestion du sac est générique

- Le système de combat manipule des objets de type `Weapon`
  - Les dégâts sont connus avec la méthode `getDamage()`
  - Le fait de se battre à l'épée ou la massue ne change rien

  ⇒le code de gestion des combats est générique

L'héritage en Java

- Une classe hérite **au plus d'une unique** classe
  - Mot clé `extends` suivi de la classe mère

```
class Item {
  private int weight;
  public void setWeight(int w) { weight = w; }
  public Item() {}
}

class Weapon extends Item {
  public Weapon(int w) { setWeight(w); }
}
```


L'héritage en Java

- Une classe hérite **au plus d'une unique** classe
  - Mot clé `extends` suivi de la classe mère

```
class Item {
  private int weight;
  public void setWeight(int w) { weight = w; }
  public Item() {}
}

class Weapon extends Item {
  public Weapon(int w) { setWeight(w); }
}
```

Weapon **hérite** de Item ⇒ possède les champs/méthodes de Item

Héritage et constructeur

- La première instruction du constructeur d'une classe fille **doit** être une invocation du constructeur de la classe mère
  - Invocation constructeur mère avec le nom clé `super`

```
class Item {
  private int weight;
  public void setWeight(int w) { weight = w; }
  public Item(int w) {weight = w;}
}

class Weapon extends Item {
  public Weapon(int w) { super(w); }
}
```

Appel du constructeur de Item avec le paramètre w

Héritage et constructeur

- La première instruction du constructeur d'une classe fille **doit** être une invocation du constructeur de la classe mère
  - Invocation constructeur mère avec le nom clé `super`
  - Invocation implicite si constructeur mère sans paramètre

```
class Item {
  private int weight;
  public void setWeight(int w) { weight = w; }
  public Item() {}
}

class Weapon extends Item {
  public Weapon(int w) { super(); setWeight(w);}
}
```

Si super omis
⇒ appel à super implicite (ajouté automatiquement à la compilation)

Héritage et transtypage

- Une fille possède aussi le type de sa mère ⇒ sur-typage valide

```
Item item = new Sword(); /* valide (upcast) */
```

- Un objet du type de la mère ne possède pas en général le type de la fille
  - Possibilité de sous-typer (*downcast*) un objet explicitement
  - Erreur de transtypage détectée **à l'exécution**

```
Item item = new Sword();
Sword asSw = (Sword)item;   /* valide à l'exécution */
Club asCl = (Club)item;     /* erreur à l'exécution */
```


Héritage et transtypage

- Un
  Ite
- Un
  -
  -

instanceof permet de tester si un objet est une instance d'une classe donnée

⇒

```
if(item instanceof Club) {
  Club asCl = (Club)item;
  System.out.println("This is a club");
}
```

```
Item it
Sword asSw = (Sword)item;   /* valide à l'exécution */
Club asCl = (Club)item;     /* erreur à l'exécution */
```

Héritage et appel de méthode d'instance

- Si une fille **surdéfinit** une méthode de la mère, l'appel va toujours vers celui de la **fille** (liaison tardive)

```
class Item {
  public void render() {
    /* affiche un ? */
  }
}

class Sword extends Weapon {
  public void render() {
    /* affiche une épée */
  }
}
```

```
class Engine {
  public void test() {
    Item i = new Sword();
    i.render();
  }
}
```

Appel Sword.render() car i est un objet de type effectif Sword
(même si son type statique dans le code source est Item)

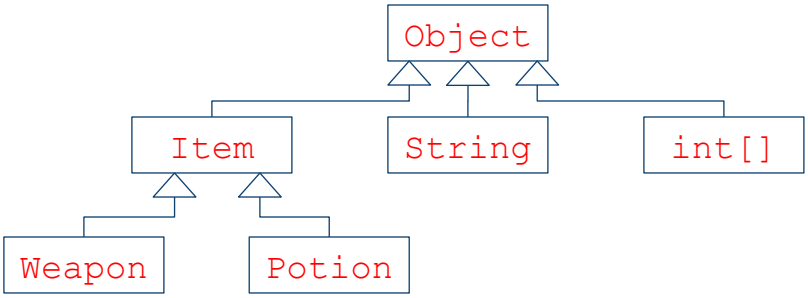
La visibilité `protected`

- `protected` permet de spécifier qu'une entité (champ ou méthode) est visible des classes filles et du package

La classe Object

- Toutes les classes héritent d'une classe nommée `Object`
  - Héritage implicite (`extends Object`) si pas de `extends`
  - Héritage par transitivité sinon
  - Vrai aussi pour les classes de tableaux

⇒ La relation d'héritage construit un arbre dont `Object` est la racine



La méthode `String toString()`

- `Object` fournit quelques méthodes génériques
  - La plus importante à ce stade est `String toString()`
  - Méthode appelée automatiquement pour convertir un `Objet` en `String` quand conversion implicite requise (voir CM1)

```
class City {
  private String name;
  public String toString() {
    return "City: " + name;
  }
}
```

```
City city = new City("Paris");

System.out.println(city);
/* ⇒ affiche City: Paris */
```

La méthode `String toString()`

- Object fournit ...

La méthode `String toString()` sert principalement à déverminer les programmes

Utilisez là !

Règle générale : toute classe devrait surdéfinir la méthode `String toString()` (par défaut, affiche la référence de l'objet)

```
c
}
```


Les classes et méthodes abstraites (1/2)

- Parfois, donner le code d'une méthode dans une classe mère n'a pas de sens
  - La méthode n'est définie que pour être invoquée, c'est aux filles de la mettre en œuvre
  - Exemple typique : la méthode `getDamage()` de `Weapon`

```
class Weapon extends Item {
  public int getDamage() {
    /* quelle valeur doit-on renvoyer ici ? */
  }
}

class Sword extends Weapon {
  public int getDamage() { return 17; }
}
```

Les classes et méthodes abstraites (2/2)

- Parfois, donner le code d'une méthode dans une classe mère n'a pas de sens
  - La méthode n'est définie que pour être invoquée, c'est aux filles de la mettre en œuvre
  - Exemple typique : la méthode `getWeight()` de `Item`

- Dans ce cas, on peut omettre le code d'une méthode
  - Marquer la méthode d'instance comme `abstract`
  - Marquer la classe comme `abstract` (⇒ impossible à instancier)

=> oblige les descendantes concrètes à mettre en œuvre les méthodes marquées `abstract` chez la mère

Utilisation des classes abstraites (1/3)

```
// classes abstraites versus classes concrètes

abstract class Item {
  public void printWeight() {
    System.out.println("Weight: " + getWeight());
  }
  public abstract int getWeight();
}
```

Club et Sword mettent en œuvre `getWeight`
(on suppose que Weapon hérite de Item)

```
class Club extends Weapon {
  public int getWeight() {
    return 42;
  } }
```

```
class Sword extends Weapon {
  public int getWeight() {
    return 17;
  } }
```

Utili

Remarque : comme `Weapon` ne met pas elle-même en œuvre `getWeight()`, `Weapon` est une classe abstraite :

```
abstract class Weapon extends Item { }
```

```
// classe

abstract class Item {
  public void printWeight() {
    System.out.println("Weight: " + getWeight());
  }
  public abstract int getWeight();
}
```

Club et Sword mettent en œuvre `getWeight`
(on suppose que Weapon hérite de Item)

```
class Club extends Weapon {
  public int getWeight() {
    return 42;
  } }
```

```
class Sword extends Weapon {
  public int getWeight() {
    return 17;
  } }
```


Utilisation des classes abstraites (2/3)

```
class Bag {
  Item[] items;

  // exemple d'initialisation de items
  Bag() {
    items = new Item[2];     // tableau de réfs vers Item
    items[0] = new Club();   // ok car Club est un Item
    items[1] = new Sword(); // ok car Sword est un Item
  }
  …
```

```
class Club extends Weapon {
  public int getWeight() {
    return 42;
  } }
```

```
class Sword extends Weapon {
  public int getWeight() {
    return 17;
  } }
```

Utilisation des classes abstraites (2/3)

```
class Bag {
  … // items vaut toujours { new Club(), new Sword() }

  // exemple d'utilisation de getWeight
  int bagWeight() {
    int tot = 0;
    for(int i=0; i<items.length; i++)
      tot += items[i].getWeight();
    return tot;
  }
```

⇒ `tot` vaut 59
à la fin de la boucle

`getWeight()` pour `items[0]`

`getWeight()` pour `items[1]`

```
class Club extends Weapon {
  public int getWeight() {
    return 42;
  } }
```

```
class Sword extends Weapon {
  public int getWeight() {
    return 17;
  } }
```


L'interface

- Limitation de l'héritage Java
  - Une classe hérite **au plus** d'une unique classe
  - Que faire, par exemple, pour une classe `Poison` qui serait à la fois une arme (`Weapon`) et une potion (`Potion`) ?

- Solution : l'interface = généralisation des classes abstraites
  - Ne définit **que** des méthodes abstraites, pas de champs
  - Mot clé `interface` au lieu de `class`, plus besoin de marquer les méthodes `abstract`
  - Possibilité pour un objet d'exposer **plusieurs** interfaces
  - On dit alors que la classe **met en œuvre** les interface

L'interface – exemple (1/2)

```
interface Weapon {
  public int getDamage();
}
```

A peu près équivalent à

```
abstract class Weapon {
  abstract public int getDamage();
}
```


L'interface – exemple (2/2)

```
interface Weapon {
  public int getDamage();
}
```

```
interface Potion {
  public void drink(Player p);
}
```

```
class Poison implements Weapon, Potion {
  public int getDamage() { … }
  public void drink(Player player) { … }
}
```

Utilisation :

```
Potion p = new Poison();
p.drink(sauron);
```

Notions clés

- Si `Y` est fille de `X`
  - `Y` possède les champs et méthodes de `X`, `Y` est aussi du type `X`
  - Déclaration avec `class Y extends X { … }`
  - Appel du constructeur parent avec le mot clé `super`
  - Le receveur d'un appel de méthode est donné par le type effectif
- Méthode `abstract`
  - Méthode d'instance sans corps
  - La classe doit être marquée `abstract`, elle est non instanciable
- Interface = classe avec uniquement des méthodes `abstract`
  - Mise en œuvre d'une interface avec `implements`

Les classes génériques

Algorithmique et langage de programmation
Gaël Thomas

Héritage et réutilisation de code (rappel)

- Une structure de données peut souvent être réutilisée
  - Exemple typique : le tableau extensible de `Monster` (voir CI3)

```
class Army {
  Monster[] content; /* tableau de monstres */
  int top;           /* qui est occupé jusqu'à top */

  void add(Monster m) {
    if(top == content.length) {
      Monster[] tmp = new Monster[content.length * 2];
      for(int i=0; i<content.length; i++)
        tmp[i] = content[i];
      content = tmp;
    }

    content[top++] = m; } }
```

Héritage et réutilisation de code (rappel)

- Pour rendre le code réutilisable dans d'autres contextes, il suffit de :
  - Remplacer `Army` par `ArrayList`
  - Remplacer `Monster` par `Object` puisque `Monster` hérite de `Object`

```
class ArrayList {
  Object[] content; /* tableau d'objets */
  int top;          /* qui est occupé jusqu'à top */

  void add(Object m) {
    if(top == content.length) {
      Object[] tmp = new Object[content.length * 2];
      for(int i=0; i<content.length; i++)
        tmp[i] = content[i];
      content = tmp;
    }

    content[top++] = m; } }
```


Pourquoi la programmation générique

- Problème : nécessite des sous-typage explicites à l'utilisation

```
class ArrayList {
  …
  Object get(int i) { return content[i]; }
}

class Game {
  ArrayList army;

  void display(int i) {
    Monster m = (Monster)army.get(i);
    m.display();
  }
}
```

renvoie `Object` car le type des objets contenus n'est pas connu à la conception

ce qui force un sous-typage explicite à l'utilisation

Pourquoi la programmation générique

Ces sous-typages explicites ne sont pas idéals pour le développeur !

- Les sous-typages explicites rendent le code confus et difficile à lire
- Et ne permettent pas au compilateur de s'assurer que `ArrayList` ne contient que des `Monster`
  => risque d'erreurs difficiles à détecter si le programme stocke par erreur autre chose qu'un `Monster` dans le `ArrayList`

La classe générique

- Classe générique = classe paramétrée par une autre classe

```
class Game {
  /* ArrayList stocke des Monster et non des Object */

  ArrayList<Monster> army;

  void display(int i) {
    /* plus besoin de sous-typer le résultat */

    Monster m = army.get(i);
    m.display();
  } }
```

- Avantages
  - Détection des incohérences de types à la compilation
    (impossible de stocker autre chose qu'un `Monster` dans `army`)
  - Code plus facile à lire car plus de sous-typages explicites

Utilisation simple (1/2)

```
// la classe Bag est paramétrée par E
// ⇔ E inconnu ici, il sera connu à la déclaration
class Bag<E> {
  private E[] elements;
  public Bag() { elements = new E[42]; }
  public E get(int i) { return element[i]; }
}

Bag<Potion> b;   // déclare un Bag avec E valant Potion
b = new Bag<>(); // alloue un Bag (le paramètre Potion est
   automatiquement déduit à partir du type de b)
```

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Utilisation simple (1/2)

```
// la classe Bag es
// ⇔ E inconnu ici
class Bag<E> {
  private E[] eleme
  public Bag() { el
  public E get(int
}

Bag<Potion> b;   // déclare un Bag avec E valant Potion
b = new Bag<>(); // alloue un Bag (le paramètre Potion est
   automatiquement déduit à partir du type de b)
```

Remarque 1 : on peut aussi compacter déclaration et allocation avec

`Bag<Potion> b = new Bag<>();`

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Remarque 2 : on peut omettre le "<Potion>", dans ce cas, un "<Object>" est ajouté de façon transparente par le compilateur

Pas recommandé

Remarque 3 : on peut omettre le "<>"

Pas recommandé

```
a déclaration

Bag<Potion> b;    // déclare un Bag avec E valant Potion
b = new Bag<>(); // alloue un Bag (le paramètre Potion est
   automatiquement déduit à partir du type de b)
```

b est un sac à potions, on peut faire : `Potion p = b.get(0);`

Utilisation simple (2/2)

```
class Tree<E, F> { /* paramétré par E, F */
  private Tree<E, F> left;
  private Tree<E, F> right;
  private E          key;
  private F          value;
  …
}
```

⇒ `Tree<String, Account>` déclare un `Tree` avec `E` valant `String` et `F` valant `Account`

Utilisation avancée

- Utilisation de `extends` si il faut que le type paramètre hérite d'une classe précise

```
class Bag<E extends Item> {
  private E[] elmts;
  public int getWeight() {…
    for(…) tot += elmts[i].getWeight(); …}
}

class Item {
  abstract public int getWeight();
}
```

`E` doit hériter de `Item` car `E` doit posséder la méthode `getWeigth`

U

-

Attention

Si un type paramètre doit mettre en œuvre une **interface** précise, on utilise **quand même** `extends`

```
  public int getWeight() {…
    for(…) tot += elmts[i].getWeight(); …}
}

class Item {
  abstract public int getWeight();
}
```

Tableau de classes génériques

- Attention : pas d'allocation d'un tableau de classes génériques avec "<>"

```
Truc<String>[] t = new Truc<>[10];
```

- Allocation du tableau "comme si" `Truc` n'était pas générique

```
Truc<String>[] t = new Truc[10]; /* autorisé */
```

- En revanche, allocation des éléments de façon normale

```
t[0] = new Truc<>("Hello");
```


Les classes enveloppes

- Les types primitifs ne font pas partis de la hiérarchie des classes (ce ne sont pas des classes) et donc ne peuvent pas être utilisés dans les classes génériques.
- Java fournit des classes enveloppes chargées représenter les types primitifs sous forme de classe:
  - `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` et `Double`
  - Ces classes héritent de `Number` ou directement de `Object`
  - Elles fournissent des méthodes très utiles (ex: `Interger.parseInt(String s)`)
- On doit utiliser les classes enveloppes avec les classes génériques

```
ArrayList<int> myList = new ArrayList<>();
```

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

Notions clés

- Classe générique = classe paramétrée par d'autres types

```
class Truc<X, Y, Z extends Bidule>
```

- Déclaration en spécifiant les paramètres

```
Truc<A, B, C> truc;
```

- Allocation en ajoutant "<>" après le mot clé `new`

```
truc = new Truc<>();
```

- Pas de "<>" à côté du `new` pour allouer un tableau de classes génériques

```
Truc<A, B, C>[] trucs = new Truc[10];
```


Notion avancée : déduction de type

- Le compilateur Java déduit automatiquement que le paramètre de `new Bag<>()` est `Potion` via le type de `b` dans

```
Bag<Potion> b = new Bag<>();
```

- Mais le compilateur n'est pas toujours capable de déduire le type

```
class Potion { void drink() { … } }

class Bag<E> { E e; E get() { return e; } }

(new Bag<>()).get().drink(); // déduction impossible
```

- Dans ce cas, il faut explicitement donner le paramètre lors de l'allocation

```
(new Bag<Potion>()).get().drink();
```

Les exceptions

Algorithmique et langage de programmation
Gaël Thomas

À quoi servent les exceptions ?

Les exceptions servent à gérer les cas d'exécution exceptionnels
(c'est-à-dire les cas d'exécution rares, par exemple les erreurs)

Évite de traiter les cas exceptionnels
dans le flot d'exécution normal
⇒ code plus clair et plus facilement réutilisable

Problème 1 : signaler un cas d'exécution exceptionnel

- Exemple : méthode calculant le logarithme népérien
  `double ln(double x)`

- Que faire si x est inférieur ou égal à 0 (cas exceptionnel) ?
  - Renvoyer une valeur absurde… Mais `ln` est surjective dans `R` !
  - Renvoyer un objet avec un champ indiquant l'erreur et un champ indiquant le résultat… Mais coûteux en calcul/mémoire !

⇒ pas de solution satisfaisante

Problème 2 : propager un cas d'exécution exceptionnel

- Bien souvent, on est obligé de propager d'appelé vers appelant les situations exceptionnelles

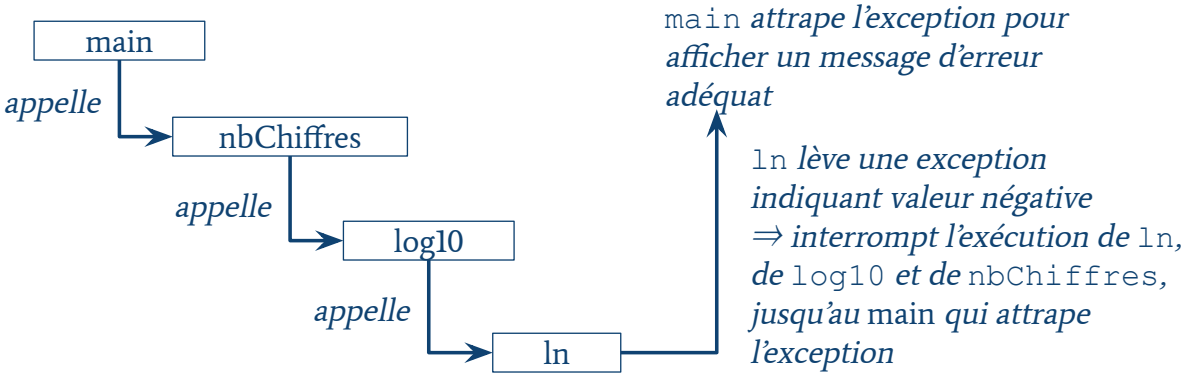
- Exemple : calcul du nombre de chiffres d'un nombre

```
… main() { System.out.println(nbChiffres( 3024)); }
int nbChiffres(double x) { return log10(x); }
double log10(double x) { return ln(x)/ln(10); }
double ln(double x) { /* calcul log. népérien */ }
```

`log10` et `nbChiffres` doivent propager l'erreur si erreur dans `ln`
⇒ cette propagation rendrait le code vite illisible !

La solution : les exceptions

- Exception : objet qu'on peut lever et attraper
  - **Lever** une exception : interrompt la chaîne d'appel jusqu'à un appelant capable d'**attraper** l'exception



L'objet de type exception

- Une exception est un objet qui hérite de la classe `Exception`

```
class NegativeValue extends Exception { }
```

  ⇒ toute instance de `NegativeValue` est une exception

- **Une exception est un objet comme les autres**, mais il peut aussi être levé et attrapé

- Deux principaux constructeurs offerts par `Exception`
  - Pas d'argument
  - Un message de type `String` accessible via `getMessage()`

Lever une exception

- `throw e` où `e` est un objet de type exception
  Exemple dans le code de la méthode `ln`:
  ```
  if(x <= 0) throw new NegativeValue();
  ```

- Les exceptions levées par une méthode (directement ou indirectement) doivent être indiquées avec `throws`

```
int nbChiffres(double x) throws NegativeValue {
  return log10(x); }
double log10(double x) throws NegativeValue {
  return ln(x)/ln(10); }
double ln(double x) throws NegativeValue { … }
```


Lever une exception

Attention : ne confondez pas

`throw` (sans s) pour lever une exception

avec

`throws` (avec s) pour indiquer quelles exceptions sont levées

Attraper une exception

- Trois éléments + 1 optionnel
  - Délimiteur de zone d'attrapage avec `try`
  - Suivi de filtres d'exceptions attrapées avec `catch`
  - Suivi du code à exécuter si exception attrapée
  - Optionnellement suivi de `finally` exécuté dans tous les cas

```
try { nbChiffres(); }
catch(NegativeValue|ZeroValue e) { … }
catch(AutreException e) { … }
catch(Exception e) { … }
finally { … }
```

Si exception levée dans ce bloc
⇒ applique les filtres

Attrape tout objet qui hérite de `Exception`
⇒ toutes les autres Exceptions

Un exemple complet

```
class NegativeValue extends Exception {}
class Test {
    static double log(double x) throws NegativeValue {
      if(x <= 0) { throw new NegativeValue(); }
      else { return Math.log(x); } }

    public static void main(String[] args) {
      try {
        double val = Double.parseDouble(args[0]);
        System.out.println(log(val));
      } catch(NegativeValue v) {
        System.out.println("Chiffre positif attendu");
      } } }
```

Bonnes pratiques

- N'utilisez les exceptions **que** pour les cas exceptionnels
  - Les exceptions rendent le code difficile à lire dans le cas normal (plus facile à lire **uniquement** dans les cas exceptionnels)
  - Allouer une exception avec `new` et lever une exception avec `throw` sont deux opérations particulièrement lentes

    (`try`/`catch` est en revanche gratuit)

- N'attrapez les exceptions **que** si vous avez quelque chose d'intéressant à faire
  - Exemple à ne jamais faire :

```
try { f() } catch(Exception e) { throw e; }
```


Bon à savoir

- Si `e` est de type `Exception`
  - `e.getMessage()` ⇒ renvoie le message associé à l'exception
  - `e.printStackTrace()` ⇒ affiche la pile d'exécution (la suite d'appels ayant mené jusqu'au `throw`)
  - La machine virtuelle Java affiche message/pile d'exécution si `main` lève une exception

4 exceptions prédéfinies à connaître

- `ClassCastException` : mauvais transtypage
- `IllegalArgumentException` : mauvais argument (vous pouvez réutiliser cette exception, elle est là pour ça)
- `IndexOutOfBoundsException` : dépassement de capacité d'une structure (tableau, chaîne de caractères, vecteurs)
- `NullPointerException` : accès à la référence `null`

Notions clés

- Le mécanisme d'exception sert à gérer **les cas exceptionnels**
- Exception = objet qui hérite de la classe `Exception`
  - Peut être levée avec `throw`
  - Peut être attrapée avec `try/catch`

Les collections et les listes d'associations

Algorithmique et langage de programmation

Gaël Thomas

Depuis le début du module

- Vous avez mis en œuvre 4 grandes structures de données
  - Des tableaux dynamiques avec l'armée de monstres (CI3)
  - Des listes chaînées avec l'armée de monstres (CI3)
  - Des arbres binaires de recherche avec la banque Lannister (CI4)
  - Des tables de hachage pour votre voyage à Bilbao (CI7)

Bravo !

Vous avez donc gagné le droit d'utiliser
la bibliothèque Java qui vous fournit
ces structures de données ☺

La bibliothèque `java.util`

- Fournit un ensemble d'interfaces abstrayant les structures de données les plus fréquemment utilisées
- Fournit un ensemble de classes mettant en œuvre ces structures de données
- Toutes ces classes et interfaces sont génériques

Avant de commencer (1/2)

- La bibliothèque de structures de données Java nécessite des méthodes utilitaires
  - Pour effectuer une comparaison par valeurs entre objets
    (comme avec `CityId.equals` dans nos tables de hachage)
  - Pour connaître le code hachage d'un objet
    (comme avec `CityId.hashCode` dans nos tables de hachage)
  - Pour savoir si un objet est plus petit qu'un autre
    (comme avec la banque Lannister avec les comptes classés suivant un ordre lexicographique dans l'arbre binaire de recherche)

Avant de commencer (1/2)

- La bibliothèque de structures de données Java nécessite des méthodes utilitaires
  - Pour effectuer une comparaison par valeurs entre objets
    Offert directement par `Object` via la méthode `equals`
  - Pour connaître le code hachage d'un objet
    Offert directement par `Object` via la méthode `hashCode`
  - Pour savoir si un objet est plus petit qu'un autre
    Offert par l'interface `Comparable` via la méthode `compareTo`

Deux familles de structures de données

- La **collection** stocke une collection d'éléments
  - Tableaux extensibles (stocke des monstres)
  - Listes chaînées (stocke des monstres)
- La **liste d'association** associe des clés et des valeurs
  - La table de hachage (associe des noms à des villes)
  - L'arbre binaire de recherche (associe des noms à des comptes)

Deux familles de structures de données

- La **collection** stocke une collection d'éléments
  - Tableaux extensibles — `java.util.ArrayList<E>`
  - Listes chaînées — `java.util.LinkedList<E>`

- La **liste d'association** associe des clés et des valeurs
  - La table de hachage — `java.util.HashMap<K, V>`
  - L'arbre binaire de recherche (rouge-noir) — `java.util.TreeMap<K, V>`

La collection d'éléments

- L'interface `Collection<E>` représente une collection
  - `boolean add(E e)` : ajoute l'élément `e`
  - `boolean remove(Object o)` : supprime l'élément `o`
  - `boolean contains(Object o)` : vrai si collection contient `o`

  - `contains` et `remove` prennent en argument un `Object` et non un `E` pour permettre à l'utilisateur de tester la présence ou de supprimer un élément en utilisant une instance d'une autre classe

Exemple d'utilisation des collections

```
/* ArrayList ⇒ tableau extensible (cas ici) */
/* LinkedList ⇒ liste chaînée */

Collection<Monster> col = new ArrayList<>();
Monster m = new Monster("Pikachu");

col.add(m);
if(col.contains(m)) {
  System.out.println("Les collections, c'est facile !");
}
```


La classe `Iterator`

- Un `Iterator` permet de parcourir une collection
  - `E next()` : renvoie l'élément suivant et avance dans la collection
  - `boolean hasNext()` : renvoie vrai si il existe un suivant
  - `void remove()` : supprime l'élément courant
- Exemple d'utilisation

```
Collection<Monster> col = new LinkedList<>();
… /* ajoute des monstres à la collection ici */
Iterator<Monster> it = col.iterator();
while(it.hasNext()) {
  Monster cur = it.next();
  System.out.println("Monstre: " + cur.name);
}
```

La boucle « `pour chaque` »

- Sucre syntaxique introduit dans Java 5

```
for(Monster cur: col) {
  /* fait quelque chose avec cur */
}

Iterator<Monster> it = col.iterator();
while(it.hasNext()) {
  Monster cur = it.next();
  System.out.println("Monstre: " + cur.name);
}
```

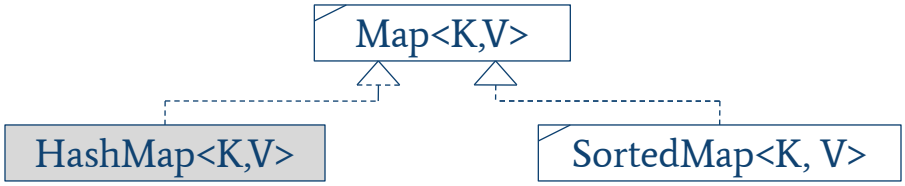

La boucle « pour cha

Bon à savoir

on peut aussi utiliser une boucle « pour chaque » pour parcourir un tableau

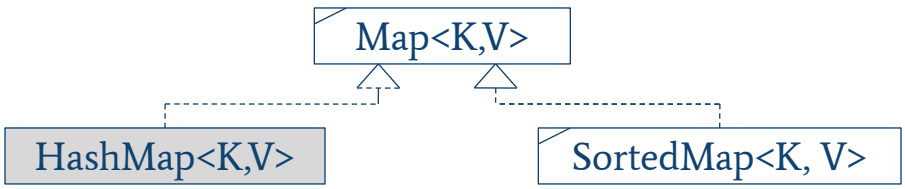
```
String[] tab = { "a", "b" };

for(String s: tab) {
    /* … */
}
```

Les tables d'association en Java (1/2)



- `Map` définit principalement quatre méthodes
  - `V put(K k, V v)` : associe `k` à `v` (renvoie ancienne valeur ou `null`)
  - `V get(K k)` : renvoie la valeur associée à `k` ou `null`
  - `boolean remove(K k)` : supprime l'association
  - `Collection<V> values()` : renvoie la collection des valeurs

Les tables d'association en Java (1/2)



- Deux principales mises en œuvre
  - La table de hachage (`HashMap`)
    - `K` doit mettre en œuvre `equals` et `hashCode`
    - Table d'association + éléments non triés
  - L'arbre binaire rouge-noir (`TreeMap` de type `SortedMap`)
    - `K` doit mettre en œuvre `Comparable`
    - Table d'association + éléments triés suivant l'ordre des clés

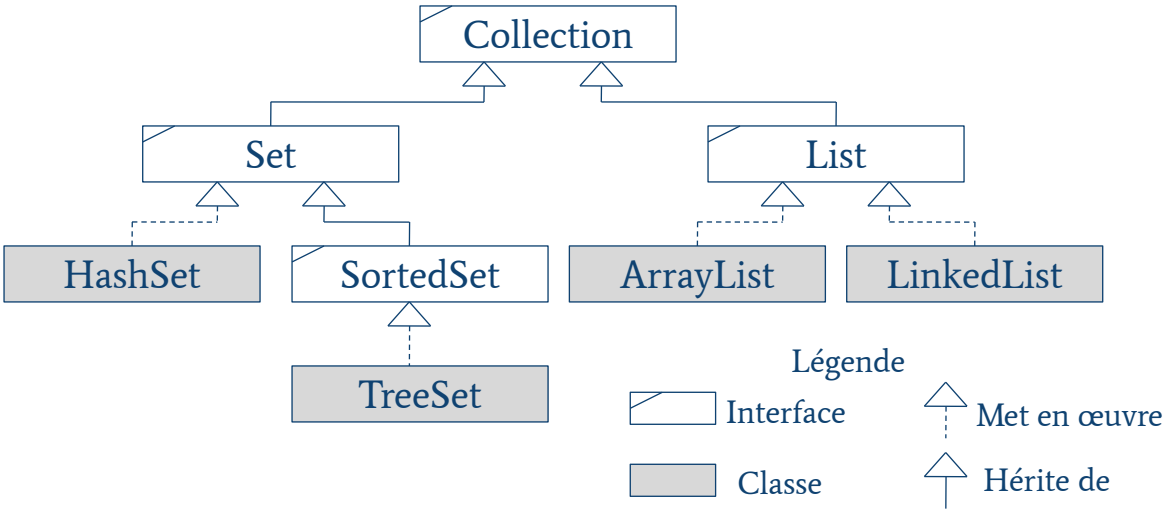
Notions clés

- `Collection` pour stocker des collections d'éléments
  - Deux principales mises en œuvre
    - `ArrayList` : tableau extensible, ajout lent (extension du tableau), accès aléatoire rapide
    - `LinkedList` : liste chaînée, ajout rapide, accès aléatoire lent
  - Parcours avec `Iterator` ou boucle « pour chaque »

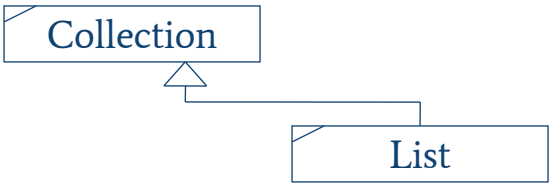
- `Map` pour associer clé et valeur
  - Deux principales mises en œuvre
    - `HashMap` : table de hachage, non triée
    - `TreeMap` : arbre binaire rouge-noir, trié suivant les clés

Pour approfondir

L'arbre d'héritage de `Collection`

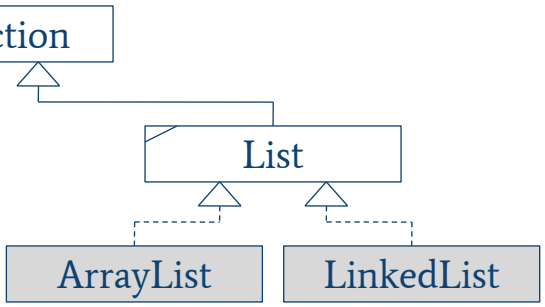


L'interface `List` (1/2)



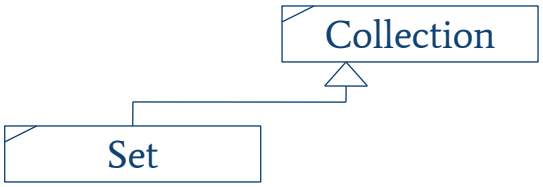
- Propriétés :
  - Les éléments possèdent un indice ⇒ généralisation des tableaux
  - Duplication d'éléments autorisée
- Ajoute principalement quatre nouvelles méthodes
  - `void add(int i, E e)`
  - `void set(int i, E e)`
  - `E get(int i)`
  - `E remove(int i)`

L'interface `List` (2/2)



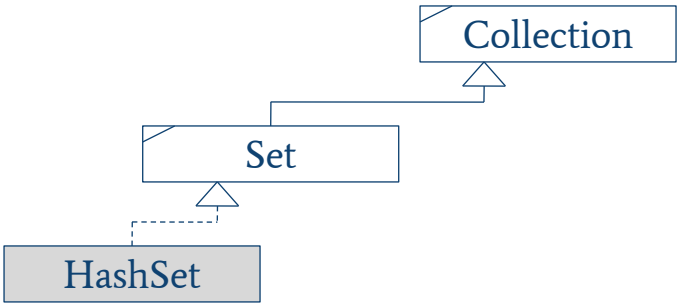
- Deux principales mises en œuvre
  - `ArrayList` : tableau extensible
  - `LinkedList` : liste chaînée

L'interface `Set` (1/2)



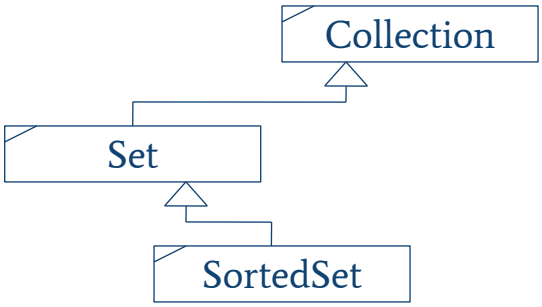
- Propriétés :
  - Les éléments ne possèdent pas d'indice
  - Duplication d'éléments interdite
- Pas de nouvelle méthode ajoutée

L'interface `Set` (2/2)



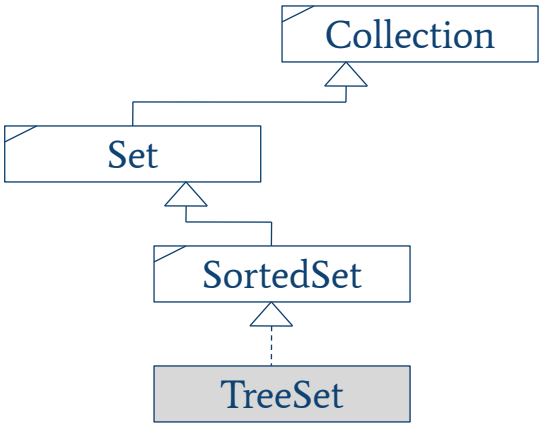
- `HashSet` = table de hachage dans laquelle clé == valeur
  - Les éléments doivent mettre en œuvre `equals` et `hashCode`
  - Les éléments sont non triés

L'interface `SortedSet` (1/2)



- Propriétés :
  - Les éléments sont triés (mettent en œuvre `Comparable`)
- Pas de nouvelle méthode ajoutée

L'interface `SortedSet` (2/2)



- `TreeSet` = arbre binaire rouge-noir dans lequel clé == valeur

Complexités des collections de Java

Structure	add(el)	add(idx, el) put(key, val)	get(idx/ key)	remove(el)	remove(0) it.remove()	contains(el/ key)	next
ArrayList	$\mathcal{O}(1)$ or $\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
LinkedList	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
HashMap	X	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	X	$\mathcal{O}(1)$	$\mathcal{O}(h/n)$
TreeMap	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$
HashSet	$\mathcal{O}(1)$	X	X	$\mathcal{O}(1)$	X	$\mathcal{O}(1)$	$\mathcal{O}(h/n)$
TreeSet	$\mathcal{O}(log(n))$	X	X	$\mathcal{O}(log(n))$	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

h = capacité = taille de la table interne (c.f. CI 7)

Choisir la bonne structure de donnée

- Les complexités nous donnent des indications générales sur les structures à choisir en fonction de nos besoins:
  - Je veux souvent regarder si un élément fait partie de ma structure : Set
  - Je veux pouvoir rapidement parcourir tous les éléments : List
  - Je veux associer une clef et une valeur : Map
- Cependant, les complexités restent approximatives et ne garantissent pas les performances temporelles en pratique dans tous les cas. Il faut donc:
  - Comprendre les implémentations. Par exemple, `HashSet` n'est pas thread-safe, alors que `ConcurrentSkipListSet` si. `LinkedHashSet` a de meilleures performances que `HashSet` si l'on doit parcourir le set... La Javadoc est très utile pour cela !
  - Mesurer les performances réelles de votre programme (profiling). On chronomètre des tests en fonction de la structure utilisée. L'encapsulation est primordial ici !

Exemple de profiling

```
static int maxElements = 10000;
static int nbExperiments = 10000;

static double testAdd(List<Integer> l){
        double start = System.nanoTime();
        for (int i = 0; i < maxElements; i++){
                l.add(i);
        }
        double time = System.nanoTime() - start;
        return time / maxElements;
}

static double testRemove(List<Integer> l){
        double start = System.nanoTime();
        for (int i = 0; i < maxElements; i++){
                l.remove(0);
        }
        double time = System.nanoTime() - start;
        return time / maxElements;
}
```

```
public static void main(String[ ] args) {
        double totalAddAL = 0;
        double totalAddLL = 0;
        double totalRemoveAL = 0;
        double totalRemoveLL = 0;
        for (int i = 0; i < nbExperiments; i++){
                ArrayList<Integer> arrayList = new ArrayList<>();
                LinkedList<Integer> linkedList = new LinkedList<>();
                totalAddAL += testAdd(arrayList);
                totalAddLL += testAdd(linkedList);
                totalRemoveAL += testRemove(arrayList);
                totalRemoveLL += testRemove(linkedList);
        }
        totalAddAL /= nbExperiments;
        totalAddLL /= nbExperiments;
        totalRemoveAL /= nbExperiments;
        totalRemoveLL /= nbExperiments;
        System.out.println("Average time to add element in ArrayList: "
                        + totalAddAL);
        System.out.println("Average time to add element in LinkedList: "
                        + totalAddLL);
        System.out.println("Average time to remove element in ArrayList: " +
totalRemoveAL);
        System.out.println("Average time to remove element in LinkedList: "
                        + totalRemoveLL);
}
```

Exemple de profiling

```
Average time to add element in ArrayList: 5.076008439999996
Average time to add element in LinkedList: 5.5253209499999745
```

Dans notre cas, les `ArrayList` sont 10% plus rapides, malgré une complexité similaire.

```
Average time to remove element in ArrayList: 146.90044398000063
Average time to remove element in LinkedList: 3.127785470000008
```

Pour des complexités différentes, les différences sont flagrantes :
`LinkedList` est 47 fois plus rapide !

```
public static void main(String[] args) {
        double totalAddAL = 0;
        double totalAddLL = 0;
        double totalRemoveAL = 0;
        double totalRemoveLL = 0;
        for (int i = 0; i < nbExperiments; i++){
static i
static i
static d
}
static d
        double time = System.nanoTime() - start;
        return time / maxElements;
}
                                                                    totalRemoveAL);
                                                            System.out.println("Average time to remove element in LinkedList: "
                                                                    + totalRemoveLL);
                                                    }
```


Le profiling est plus compliqué que ça...

- Beaucoup d'impacts extérieurs au programme :
  - Hardware/OS de l'ordinateur
  - Autres applications tournent en parallèle ?
  - Utilisation aléatoire du garbage collector
  - ...
- Mise en œuvre non triviale :
  - Écrire des tests pour tous les cas possibles d'utilisations
  - Dépendances de performances entre les fonctions
  - ...
- Des outils avancés existent (non vus dans ce cours)

Les classes anonymes

Algorithmique et langage de programmation
Gaël Thomas

Le problème

- La syntaxe utilisée pour l'héritage de classe/mise en œuvre d'interface est trop verbeuse pour des codes simples

```
interface Bird { void fly(); }

class MyBird implements Bird {
  void fly() {
    System.out.println("fly!");
  } }

class Test {
  void f() {
    Bird bird = new MyBird();
  } }
```

Nécessite une nouvelle classe, donc un fichier, le tout pour allouer une **unique instance** et peu de lignes de code

Les classes anonymes

- But : simplifier la syntaxe utilisée pour l'héritage de classe ou la mise en œuvre d'interface pour les codes simples
  - Allocation d'une **unique instance** de la classe dans le code
  - Peu de méthodes et de lignes de code dans la classe
- Principes :
  - Omettre le nom de la classe
  - Donner le code de la mise en œuvre au moment de l'allocation

Les classes anonymes par l'exemple

```
interface Bird { void fly(); }

class MyBird {
  void fly() {
    System.out...;
  } }

class Test {
  void f() {
    Bird bird = new MyBird();
  } }
```

Mise en œuvre au moment de l'allocation

Définit une nouvelle classe qui hérite de `Bird` et qui n'a pas de nom

```
class Test {
  void f() {
    Bird bird = new Bird() {
      void fly() {
        System.out...
      }
    }}
}
```

Les expressions lambda (1/2)

- Pour les cas les plus simple, le code reste verbeux, même avec les classes anonymes
- But des expressions lambda : simplifier la mise en œuvre d'une interface **fonctionnelle**
  - Interface **fonctionnelle** = interface avec une unique méthode
- Expression lambda = expression courte d'une méthode
  - `(arguments sans type) -> corps de la méthode;`
  - Exemple : `(x, y) -> x + y;`
  - Remarque : il existe des variations, non étudiée dans le cours, dans la façon de déclarer une expression lambda

Les expressions lambda par l'exemple

- Remplacement de la mise en œuvre de l'interface fonctionnelle par une expression lambda

```
interface Bird { void fly(); }
```

```
Bird bird = () -> { System.out.println("fly!"); };
```

⇔

```
Bird bird = new Bird() {
  void fly() {
    System.out.println("fly!");
  }
};
```

On ne garde que la partie « intéressante » de la déclaration

Portée de variables

- Accès aux champs de la classe anonyme, mais aussi de la classe englobante et aux variables de la méthode englobante
  - Accès en lecture seule aux variables de la méthode

```
class Nid {
  int x;      /* accès en lecture/écriture */

  void f() {
    int y;    /* en lecture seule */
    Bird bird = new Bird() {
      int z; /* accès en lecture/écriture */
      void fly() { z = x + y; }
    };
  } }
```


Notions clés

- Classe anonyme
  - Simplifie l'héritage de classe et la mise en œuvre d'interface dans les cas simples
  - `Bird bird = new Bird() { void fly() { … } };`

- Expressions lambda :
  - Simplifie encore le code pour les interfaces fonctionnelles
  - Interface fonctionnelles = interface avec une unique méthode
  - `Bird bird = () -> { System.out.println("fly!"); }`

Pour approfondir

Le but de ce sous-cours optionnel est de comprendre comment sont construites les classes anonymes et les lambda de façon à comprendre la portée des variables et des champs

Plan du sous-cours

- Classe anonyme = classe interne de méthode sans nom

Plan du cours

1. Les classes internes
2. Les classes internes de méthodes
3. Les classes anonymes

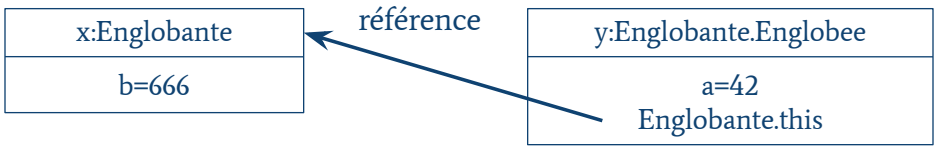
Classes internes

- Une classe interne est une classe définie dans une autre classe
  - Permet de **coupler** fortement deux classes
    - (la classe interne a accès aux champs de la classe externe)

```
class Englobante {
  class Englobee {
    int a;
    void f() { a = 42; b = 666; }
  }
  private int b;
  private Englobee englobee;
}
```


Mise en œuvre d'une classe interne

- Une classe interne possède un champ ajouté automatiquement permettant d'accéder à la classe externe




```
void f() { a = 42; b = 666; }
```

⇔

```
void f() { this.a = 42; Englobante.this.b = 666; }
```

Allocation d'une classe interne

- À partir de la classe externe, allocation de façon normale

```
class Englobante {
  class Englobee { … }
  Englobante() {
    englobee = new Englobee();
  }
}
```

- À partir de l'extérieur de la classe englobante :

```
Englobante x = new Englobante();
Englobante.Englobee y = x.new Englobee();
x.englobee = y; /* à faire à la main */
```


Classe interne et visibilité

- Les champs de la classe englobée sont toujours accessibles à partir de la classe englobante et vice-versa
- Composition des opérateurs de visibilités pour l'extérieur

```
public class Englobante { /* accessible partout */
  class Englobee {  /* accessible du package */
    private int x;  /* accessible de Englobante */
  }
  private int b;    /* accessible de Englobee */
  private Englobee englobee; /* idem */
}
```

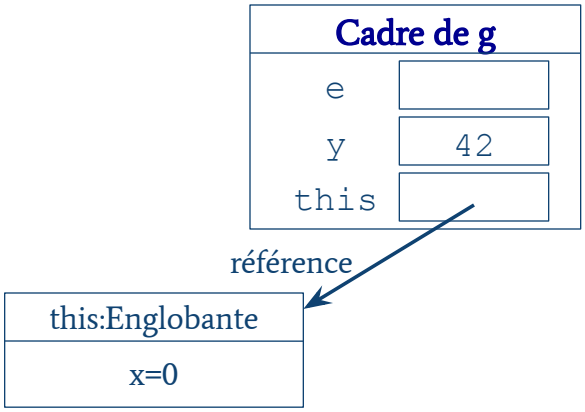
Classe interne de méthode

- Classe interne de méthode = classe définie dans une méthode
- La classe interne peut aussi accéder aux variables locales de la méthode (si var. que accédées en **lecture** après la déclaration)

```
class Englobante {
  int x;
  void f() {
    int y = 42; /* y en lecture seule dans Englobee */
    class Englobee { void g() { x = y; } };
    Englobee e = new Englobee();
    e.g();
  } }
```

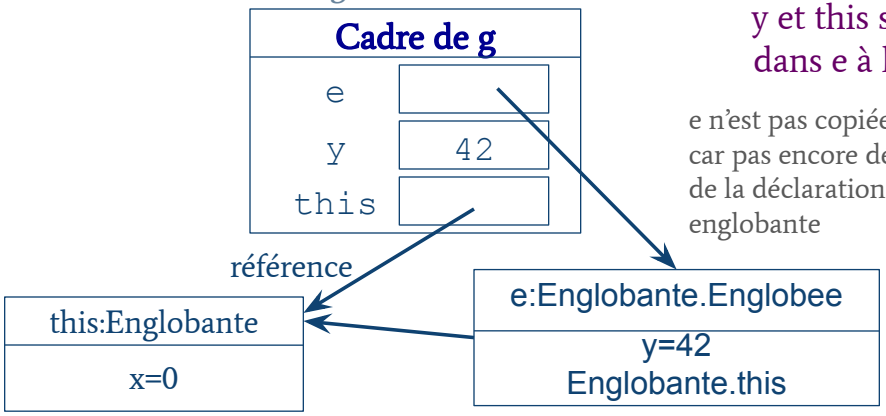

Mise en œuvre

- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



Mise en œuvre

- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



y et this sont copiés dans e à l'allocation

e n'est pas copiée car pas encore définie lors de la déclaration de la classe englobante

Classes anonymes (1/2)

- Classe anonyme = classe interne de méthode sans nom

```
void f() {                  /* en version nommée */
  class AvecNom extends Bird {
    void fly() { System.out.println("fly!"); }
  };
  Bird bird = new AvecNom();
}
```

⇔

```
void f() {                 /* en version anonyme */
  Bird bird = new Bird() {
    void fly() { System.out.println("fly!"); }
  };
}
```

Classes anonymes (2/2)

- Une classe anonyme est une classe interne de méthode
  - Si méthode d'instance, peut accéder aux champs de l'instance
  - Peut accéder aux variables locales de la méthode
    - *à condition que ces variables ne soient accédées qu'en lecture à partir de la déclaration de la classe anonyme*

```
class Test {
  int x;  /* lecture/écriture à partir de cl. anon. */
  void f() {
    int a = 42;     /* lecture à partir de cl. anon. */
    Bird bird = new Bird() { void fly() { x = a; } };
    bird.fly();
  } }
```


Et avec des expressions lambda

```
interface Bird { void fly(); }

class Test {
  int x;     /* lecture/écriture à partir de lambda */
  void f() {
    int a = 42; /* lecture seule à partir de lambda */
    Bird bird = () -> x = a;
    bird.fly();
  }
}
```

Notions clés (du sous-cours optionnel)

- Classe anonyme = classe interne de méthode sans nom
- Classe interne de méthode = classe interne définie dans une méthode
  Peut accéder aux variables de la méthode (si variables que lues)
- Classe interne = classe définit dans une autre classe
  Possède une référence vers une instance de la classe englobante

Pour votre culture : classe interne statique

- Si la classe interne est marquée `static`
  - pas liée à une instance de la classe englobante
  - pas d'accès aux champs d'instance d'une instance englobante
  - allocation sans instance d'une classe Englobante

```
class A {
  static class B {
    int a;
    void f() { a = 42; c = 666; } /* pas d'accès à b */
  }
  private int b;
  private static int c;
} /* allocation avec A.B x = new A.B(); */
```

Fiches algorithmes et structures de données

Julien Romero

Algorithmes de tri - CI2

- Entrée: Un tableau d'éléments comparables
- Sortie: Rien (tri sur place) ou nouveau tableau avec les éléments ordonnés

Algorithme	Pire des cas	Moyenne	Spatiale Moyenne
Tri bulle	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Tri par insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Tri fusion	$\mathcal{O}(n \cdot log(n))$	$\mathcal{O}(n \cdot log(n))$	$\mathcal{O}(n)$
Tri rapide	$\mathcal{O}(n^2)$	$\mathcal{O}(n \cdot log(n))$	$\mathcal{O}(log(n))$

- En Java: `Arrays.sort` (tri rapide)

Tableau extensible - CI3

- Un tableau qui grandit quand il est plein
- En Java: `ArrayList`

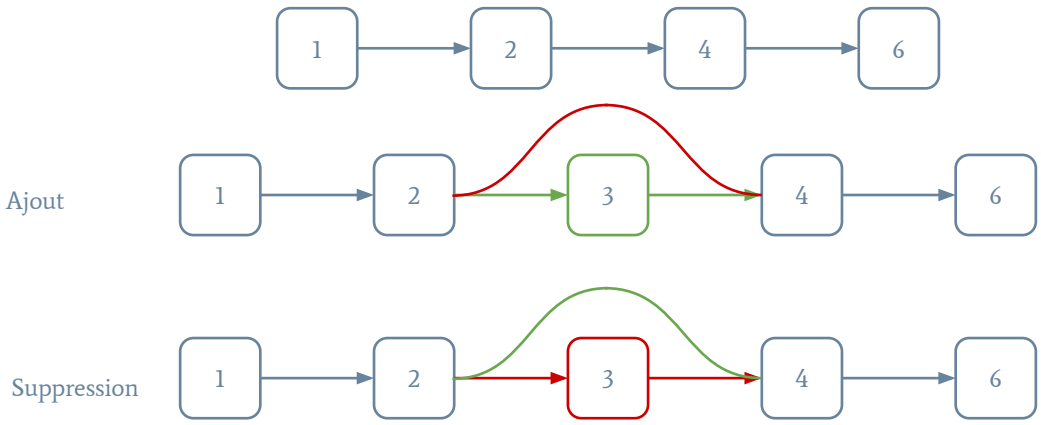
Opération	Pire des cas	Moyenne/Amortie
add(element)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
add(index, element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
get(index)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
remove(index or element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
contains(element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Liste chainée - CI3

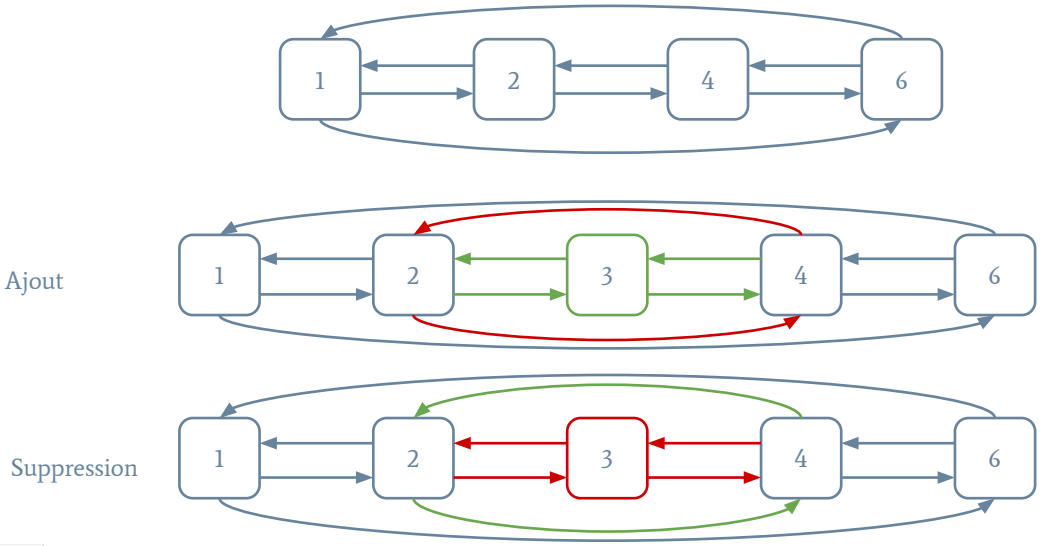
- Chaque élément de la liste a une valeur et ne connaît que ses voisins.
- En Java: `LinkedList`

Opération	Pire des cas	Moyenne/Amortie
add(element)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
add(index, element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
get(index)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
remove(index or element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
contains(element)	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Liste chainée - CI3

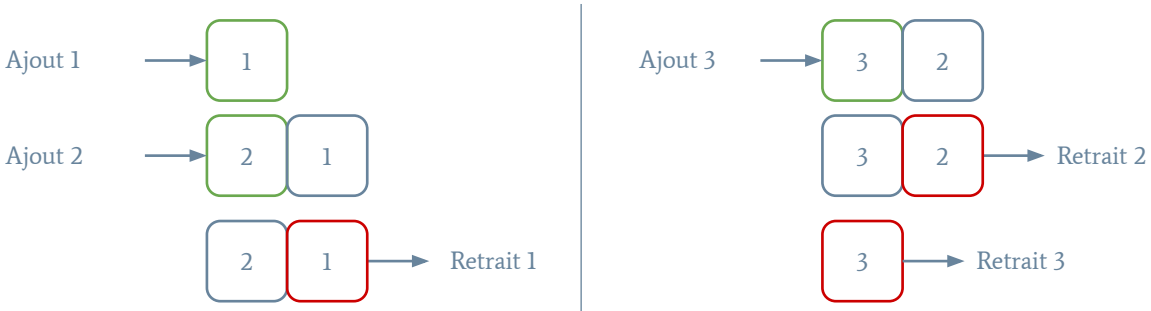


Liste doublement chainée circulaire - CI7



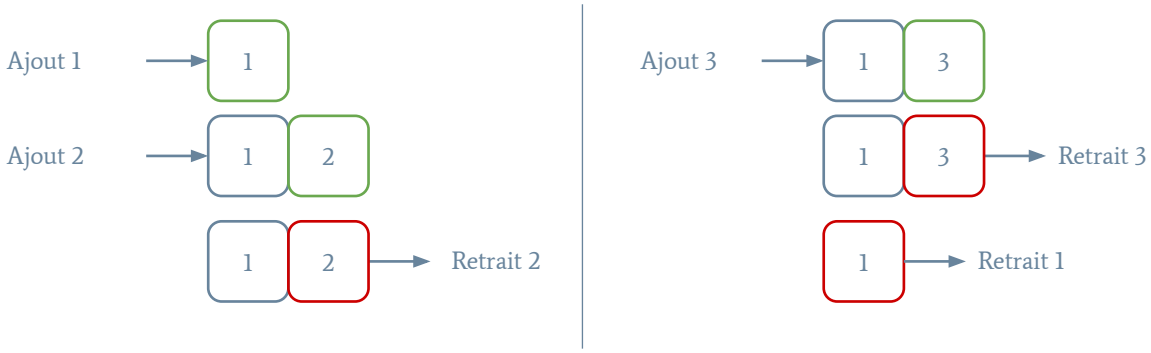
Pile (Stack) - CI7 (optionnel)

- Structure permettant de récupérer les éléments dans l'ordre d'insertion, le tout en temps constant.
- En Java: `Stack`



File (Queue) - CI7 (optionnel)

- Structure permettant de récupérer les éléments dans l'ordre **inverse** d'insertion, le tout en temps constant.
- En Java: `LinkedList` (interface `Queue`)



File à priorité (Priority Queue) - CI5 (optionnel)

- Variante de la file avec un ordre de sortie dépend d'une priorité.
- En Java: `PriorityQueue`

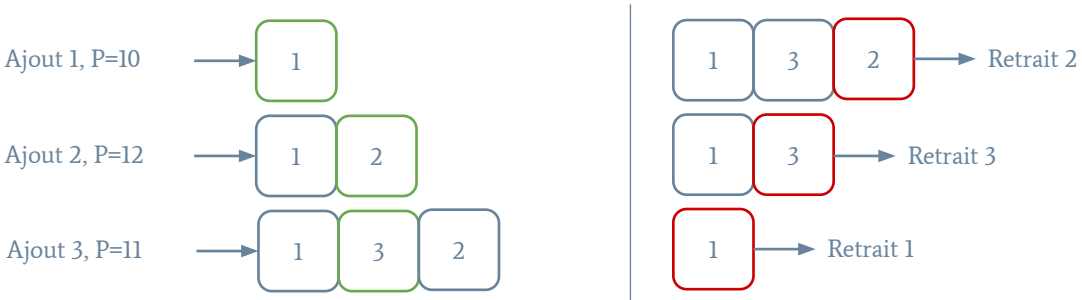


Table de hachage - CI7

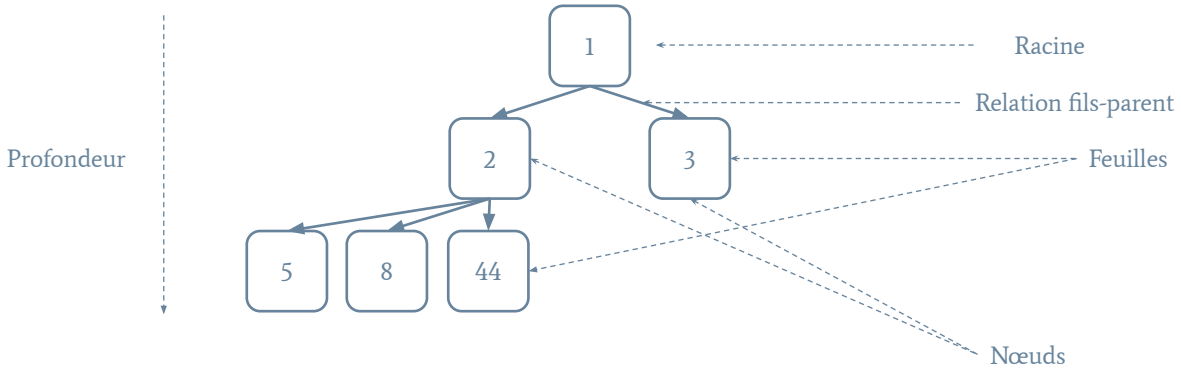
- Structure algorithmique associant des clefs à des valeurs
- L'ajout de paire de clef/valeur ou l'accès à la valeur associée à une clef se fait en *temps constant*.
- En Java: `HashMap`

Ensemble - CI7 (optionnel)

- Structure algorithmique permettant de rapidement ajouter un élément et de rapidement vérifier que l'ensemble contient un élément.
- Ajouter un élément et vérifier sa présence se fait en temps constant. Les autres opérations peuvent être plus coûteuses par rapport à une liste ou un tableau.
- En Java: Interface `Set`
  - En pratique : `HashSet`

Arbre - CI4

- Structure hiérarchique utilisée dans de nombreux algorithmes.
- En Java: À réécrire suivant les cas d'usage



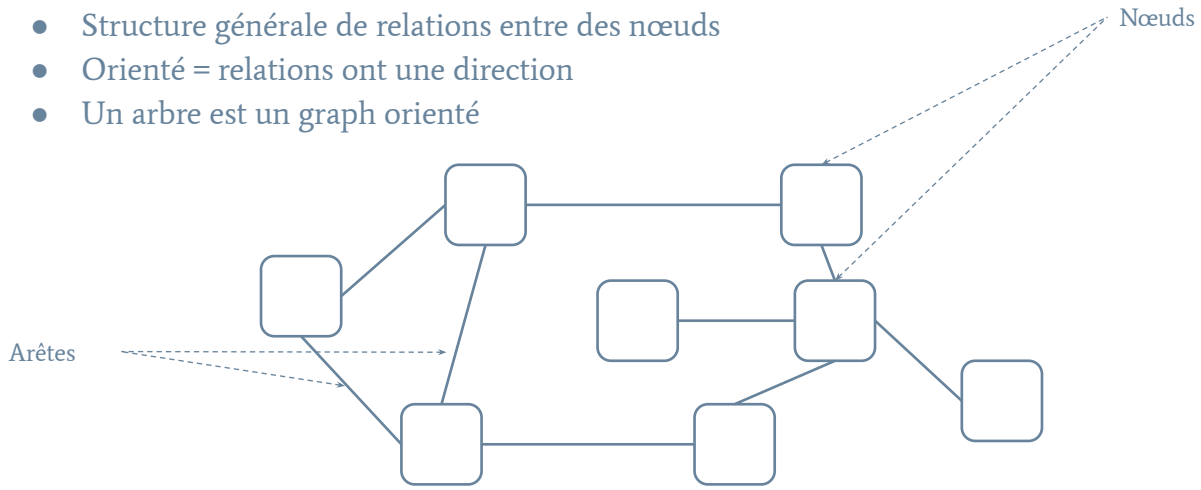
Arbre binaire de recherche - CI4

- Une structure d'arbre permettant de rapidement ajouter, rechercher et supprimer des éléments.
- Arbre binaire = nombre de fils est 0 (feuille) ou 2.

Opération	Pire des cas	Moyenne/Amortie
add	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$
delete	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$
search	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$

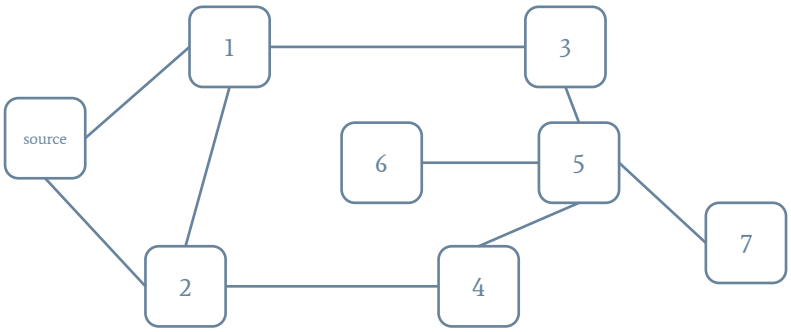
Graphe - CI5

- Structure générale de relations entre des nœuds
- Orienté = relations ont une direction
- Un arbre est un graph orienté



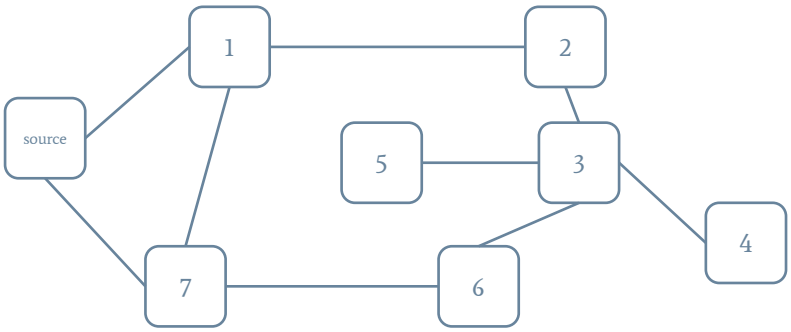
Parcours en largeur/profondeur - CI5 + CI9

- Algorithme visant à viser tous les nœuds accessible depuis un nœud source.
- Parcours en largeur: On explore d'abord les nœuds à un distance 1, puis 2, ..., puis k, puis k+1, ...



Parcours en largeur/profondeur - CI5 + CI9

- Algorithme visant à viser tous les nœuds accessible depuis un nœud source.
- Parcours en profondeur: On suit un chemin jusqu'au bout puis on le remonte pour explorer les bifurcations.



Algorithme de Dijkstra - CI5

- Permet de trouver le plus court chemin entre deux nœuds d'un graphe pondéré.
- Graph pondéré = graphe avec des poids sur chaque arête.
- Complexité (A = nombre d'arêtes, N = nombre de nœuds):
  - Avec un tas binaire: $\mathcal{O}((\mathcal{A} + \mathcal{N}) \cdot log(\mathcal{N}))$
  - Avec un tas de Fibonacci: $\mathcal{O}(\mathcal{A}+\mathcal{N} \cdot log(\mathcal{N}))$

Parcours A* - CI9

- Variante du parcours en profondeur et en largeur, mais l'ordre d'exploration dépend d'une fonction d'exploration appelée heuristique.
- Exemple d'heuristique : la distance de Manhattan entre deux points.
- Souvent utilisé dans des intelligences artificielles de jeux vidéos.

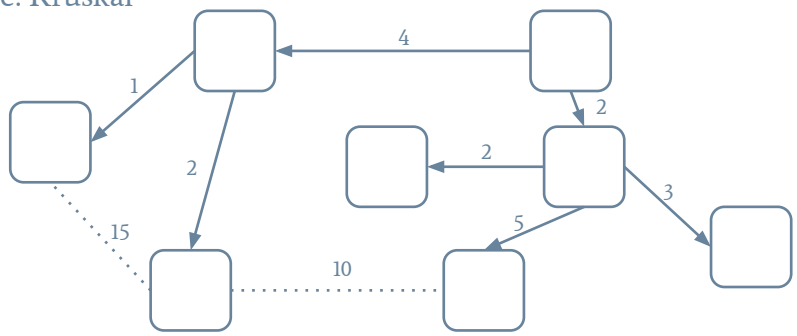
Tas binaire - CI5 (optionnel)

- Structure de donnée combinant un arbre binaire complet et un tas (un nœud est plus grand/petit que ses fils) utilisée pour rapidement trouver le minimum/maximum d'un ensemble.

Opération	Pire des cas	Moyenne/Amortie
add	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$
removeMin/removeMax	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$
update	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

Arbre couvrant minimal - CI6 (optionnel)

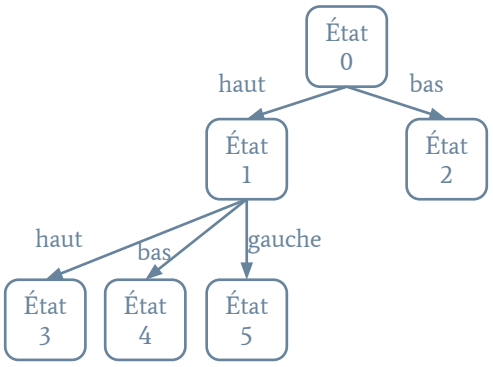
- Problème consistant à trouver un arbre "couvrant" tous les nœuds d'un graphe de façon continue de telle sorte que le poids des arêtes soit minimal.
- Utile pour de nombreux algorithmes
- Algorithme: Kruskal



Intelligence artificielle pour des jeux - CI9

- Représentation du monde avec un état
  - Exemple : pour Pacman, l'aire de jeu avec la position de la nourriture et de Pacman.
- Dans chaque état, un agent (i.e., une entité du jeu) choisit une action parmi un choix d'actions légales
  - Exemple d'agents : Pacman et les fantômes, les noirs et les blancs aux échecs
  - Exemple d'actions : Pacman : Haut, bas, gauche, droite. Échecs : déplacement de pièce.
- Il est souvent nécessaire de simuler l'état suivant à partir d'une action
- On peut former un graphe (ou plus simplement un arbre) dont les nœuds sont états et les arêtes les actions possibles.

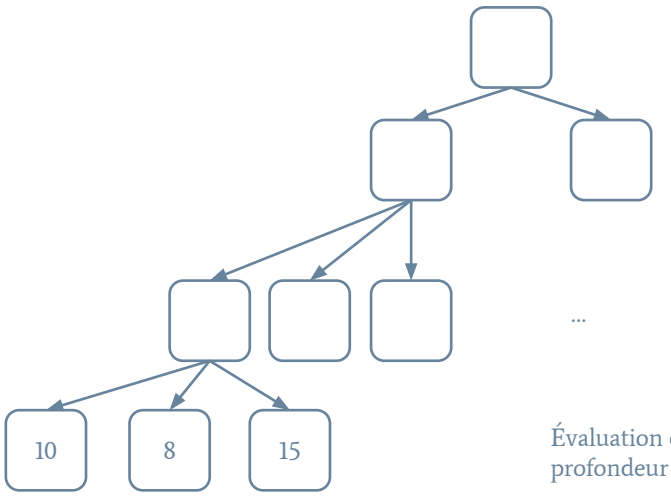
Intelligence artificielle pour des jeux - CI9



Minimax - CI9 (optionnel)

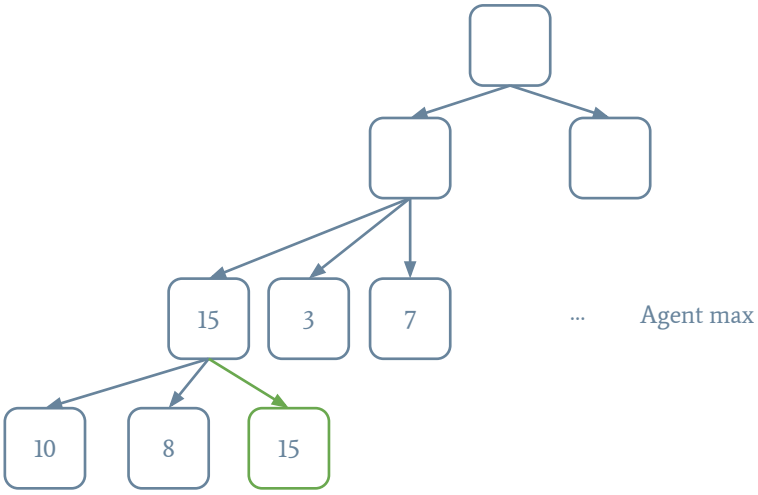
- Algorithme d'exploration souvent utilisé dans les intelligences artificielles qui consiste à simuler toutes les actions possibles de tous les agents jusqu'à une certaine profondeur.
- Les états sont évalués avec une fonction d'évaluation à la profondeur maximale.
- Deux types d'agents:
  - Agent max : choisit toujours l'évaluation maximale (le "gentil")
  - Agent min : choisit toujours l'évaluation minimale (le "méchant")
- Complexité exponentielle en la profondeur.

Minimax - CI9 (optionnel)

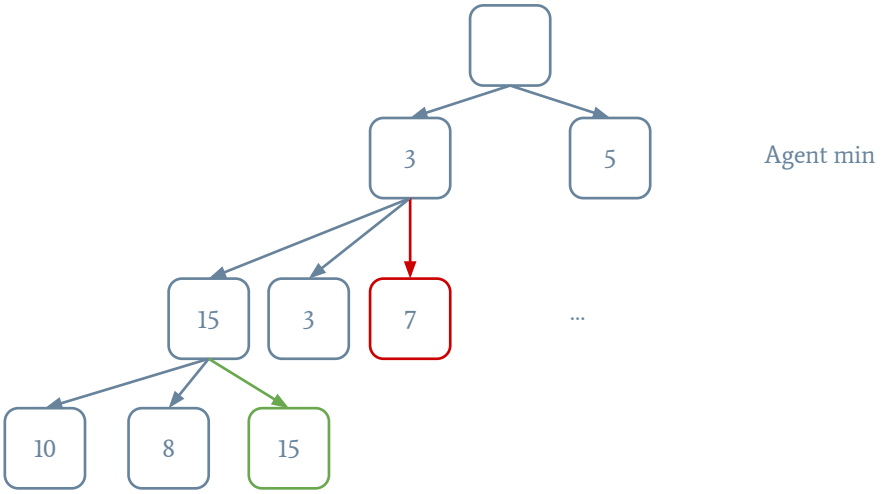


Évaluation des états à la profondeur maximale

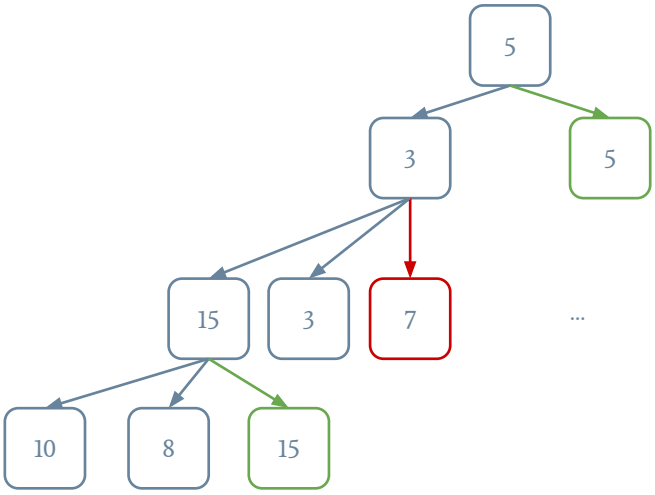
Minimax - CI9 (optionnel)



Minimax - CI9 (optionnel)



Minimax - CI9 (optionnel)



Agent max

Élagage AlphaBeta - CI9 (optionnel)

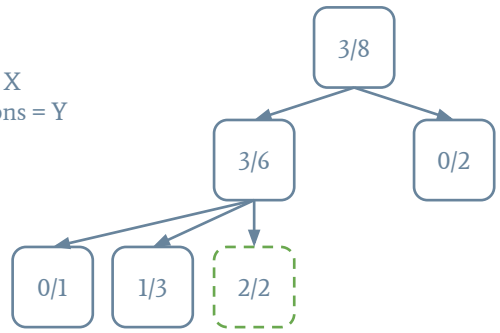
- Variante de Minimax qui supprime de manière intelligente des branches de l'arbre
- Accélération des calculs mais pas de changement de complexité

Monte Carlo Tree Search - CI9 (optionnel)

- Algorithme d'exploration qui approxime la valeur d'un état à l'aide de simulations aléatoires.
- Construit un arbre d'états au fur et à mesure
- Quatre actions principales :
  - La sélection : On choisit la feuille de l'arbre d'où partira la simulation aléatoire. La sélection se fait avec une métrique, UCB (Upper Confidence Bound) qui mélange désir de choisir la meilleure solution et d'explorer de nouveaux états
  - L'expansion : On ajoute les fils de la feuille précédemment sélectionnée à l'arbre
  - La simulation : On simule le jeu de manière aléatoire jusqu'à atteindre un état final ou une profondeur max.
  - La backpropagation : On évalue l'état à la fin de la simulation et on fait remonter son score dans l'arbre d'exploration.

Monte Carlo Tree Search - CI9 (optionnel) - Sélection

X/Y =>
Somme des scores = X
Nombre d'explorations = Y

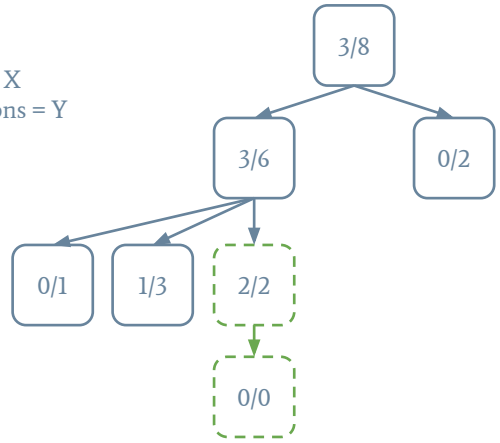


La sélection dépend de la métrique (e.x. UCB)

Pour simplifier, nous considérons qu'il n'y a qu'un seul agent.

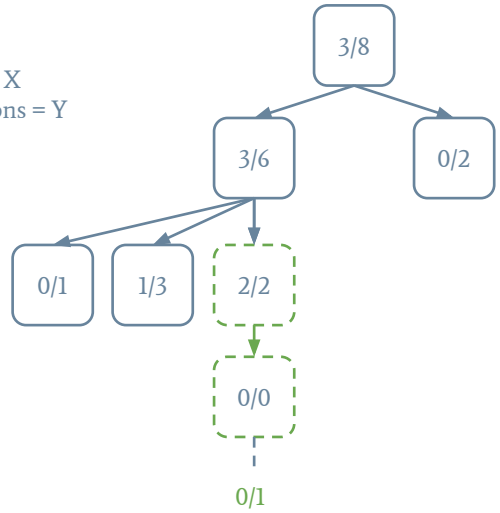
Monte Carlo Tree Search - CI9 (optionnel) - Expansion

X/Y =>
Somme des scores = X
Nombre d'explorations = Y



Monte Carlo Tree Search - CI9 (optionnel) - Simulation

X/Y =>
Somme des scores = X
Nombre d'explorations = Y



Monte Carlo Tree Search - CI9 (optionnel) - Backpropagation

X/Y =>
Somme des scores = X
Nombre d'explorations = Y