

M1202 - Algorithmique

Page web du cours : <http://tinyurl.com/M1202-2016S1>

• Contact

- Courriel : philippe.gambette@u-pem.fr
(M1202 dans le sujet du courriel)
- De vive voix avant ou après le cours/TD/TP

• Matériel

- Ordinateur portable : pas pendant les cours, à discuter pour les TD/TP.
- Pas de téléphone portable pendant cours/TD/TP
- Salles informatiques : n’y point manger, n’y point boire, ne pas y débrancher les câbles

• Déroulement des enseignements

- Séparation cours/TP/TD :
 - nouvelles méthodes de travail
 - distinguer ce qui est important, à retenir
 - savoir où retrouver l'information
- En général, distribution de notes de cours à compléter
- En général, distribution de corrigés :
 - refaire les exercices !

• Notes et devoirs

- Interrogations QCM en début de cours ou TD (signalement des absences pour rattrapage)
- Du travail à la maison

• Note finale

- Prévision :
 - environ 2/3 “compétences”
 - environ 1/3 “motivation”
- Compétences : 2/3 devoir final (12 janvier 2017)
1/3 tests
- Motivation : cours à compléter
maison note générale de TP
bonus pour erreurs détectées

• Exercices supplémentaires d'entraînement

- Sur demande, par courriel
- Sur demande, possibilité d'organiser une séance d'exercices ou de préparation au devoir final.

• Sources

Le livre de Java premier langage, A. Tasso
<http://www.pise.info/algo/introduction.htm>
Cours INF120 de J.-G. Luque
<http://xkcd.com>, <http://xkcd.free.fr>

<http://serecom.univ-tln.fr/cours/index.php/Algorithmie>
Cours de J. Henriet :
<http://julienhenriet.olymp-network.com/Algo.html>

Sommaire

Introduction

- p. 3 A quoi sert un algorithme ?
- p. 4 Organigramme de la recette des crêpes
- p. 5 De l'algorithme au programme
- p. 6 Enjeux de l'algorithmique : complexité, correction

Premiers algorithmes

- p. 7 Composants d'un algorithme
- p. 8 Premier programme Java
- p. 9 Variables et affectation
- p. 10 Dictionnaire pseudo-code / Java
- p. 11 De l'organigramme au code Java

Les types en Java

- p. 12 Codage binaire
- p. 14 Codage des entiers 32 bits
- p. 15 Autres codages
- p. 16 Les booléens
- p. 17 Les opérations de base en Java

Les tableaux

- p. 18 Les tableaux en pseudo-code
- p. 19 Les tableaux en Java
- p. 20 Manipulation de tableaux
- p. 21 Affichage du contenu d'un tableau d'entiers
- p. 22 Graphique du nombre d'apparitions des mots dans un texte
- p. 24 La boucle for/pour tout

Périphériques d'entrée/sortie

- p. 25 Souris et autres périphériques d'entrée
- p. 26 Instructions d'entrée/sortie en Java et pseudo-code

Les fonctions

- p. 27 Introduction aux fonctions
- p. 28 Les spécificités de Java sur les fonctions
- p. 29 Exemples de fonctions


Concepts et compétences du cours M1202

- p. 30 Concepts
- p. 31 Compétences

A quoi sert un algorithme ?

- À décrire les étapes de résolution d'un problème :
 - de façon structurée et compacte (méthode **facile à comprendre, facile à transmettre**)
 - à partir d'opérations de base (méthode **adaptée aux moyens à disposition, adaptée aux connaissances de celui qui l'utilise**)
 - indépendamment d'un langage de programmation (méthode **adaptée pour des problèmes qui se traitent sans ordinateur, compréhensible sans apprendre un langage de programmation**)

(“étapes” aussi appelées “pas de l'algorithme”)

- Problème : données en entrée  résultat en sortie

- L' « *algorithme des crêpes* »

Ingrédients : beurre, oeufs, sachets de sucre vanillé, farine, lait, sel

Récipients : saladier, verre mesureur, poêle, assiette,

Opérations de base : **mettre dans un récipient**, **mélanger**, **attendre pendant ... minutes**, **retourner**, **laisser cuire pendant ... minutes**

Algorithme des crêpes :

Mettre 4 oeufs **dans** le saladier

Mettre 1 sachet de sucre vanillé **dans** le saladier

Mettre 250 g de farine **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** le saladier

Mettre 0,5 litre de lait **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** le saladier

Mettre 50 grammes de beurre **dans** la poêle

Laisser cuire la poêle **pendant 1 minute**

Mettre le contenu de la poêle **dans** le saladier

Mélanger le contenu du saladier

Attendre pendant 60 minutes

Mettre 5 grammes de beurre **dans** la poêle

Algorithmes sans ordinateurs :

- Euclide (vers -300) : calcul du PGCD de 2 nombres

- Al-Khwarizmi (825) : résolution d'équations

- Ada Lovelace (1842) : calcul des nombres de Bernoulli sur la *machine analytique* de Charles Babbage

Laisser cuire la poêle **pendant 0.5 minute**

Tant que le saladier n'est pas vide :

Mettre 5 cL du contenu du saladier **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** la poêle

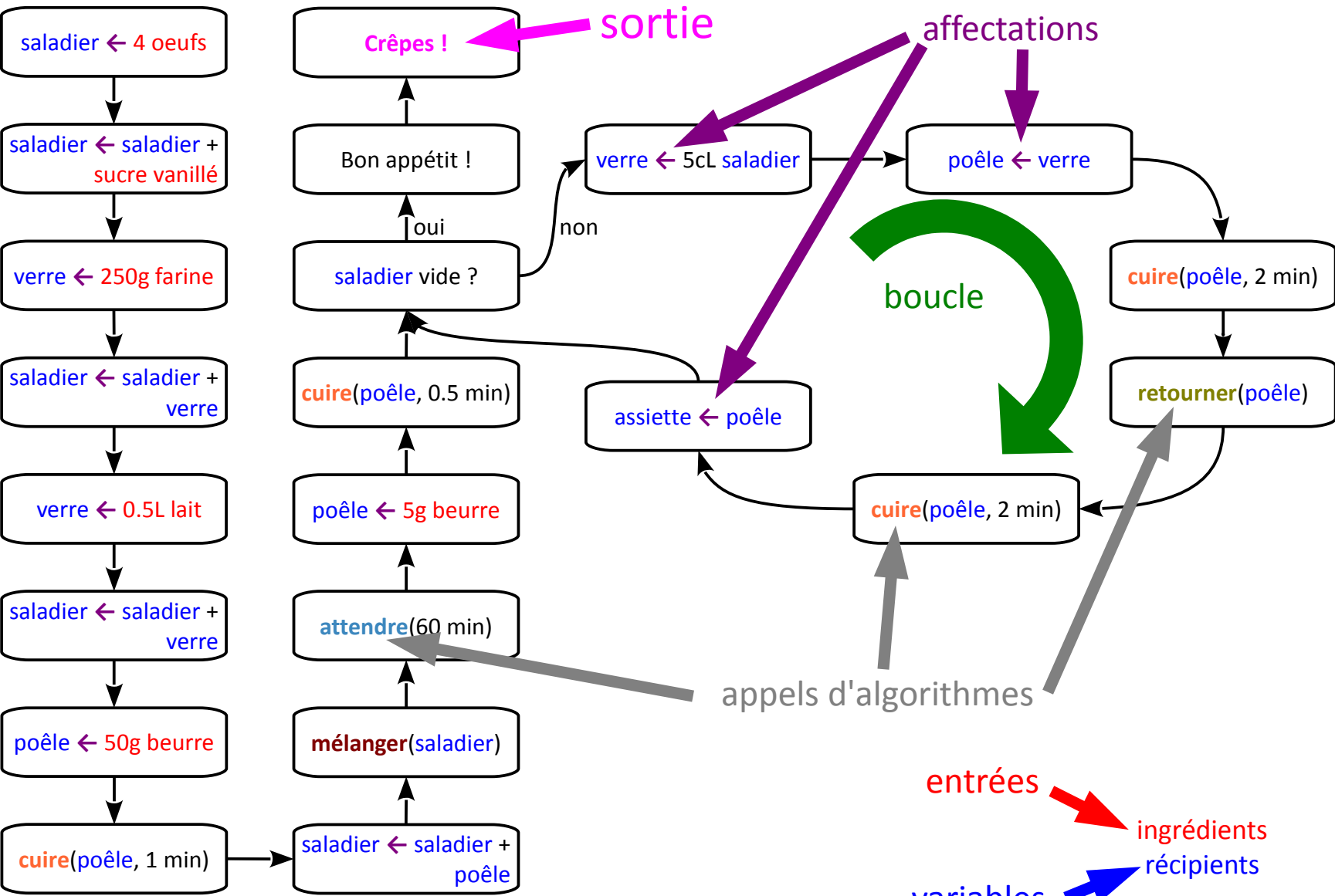
Laisser cuire la poêle **pendant 2 minutes**

Retourner le contenu de la poêle

Laisser cuire la poêle **pendant 2 minutes**

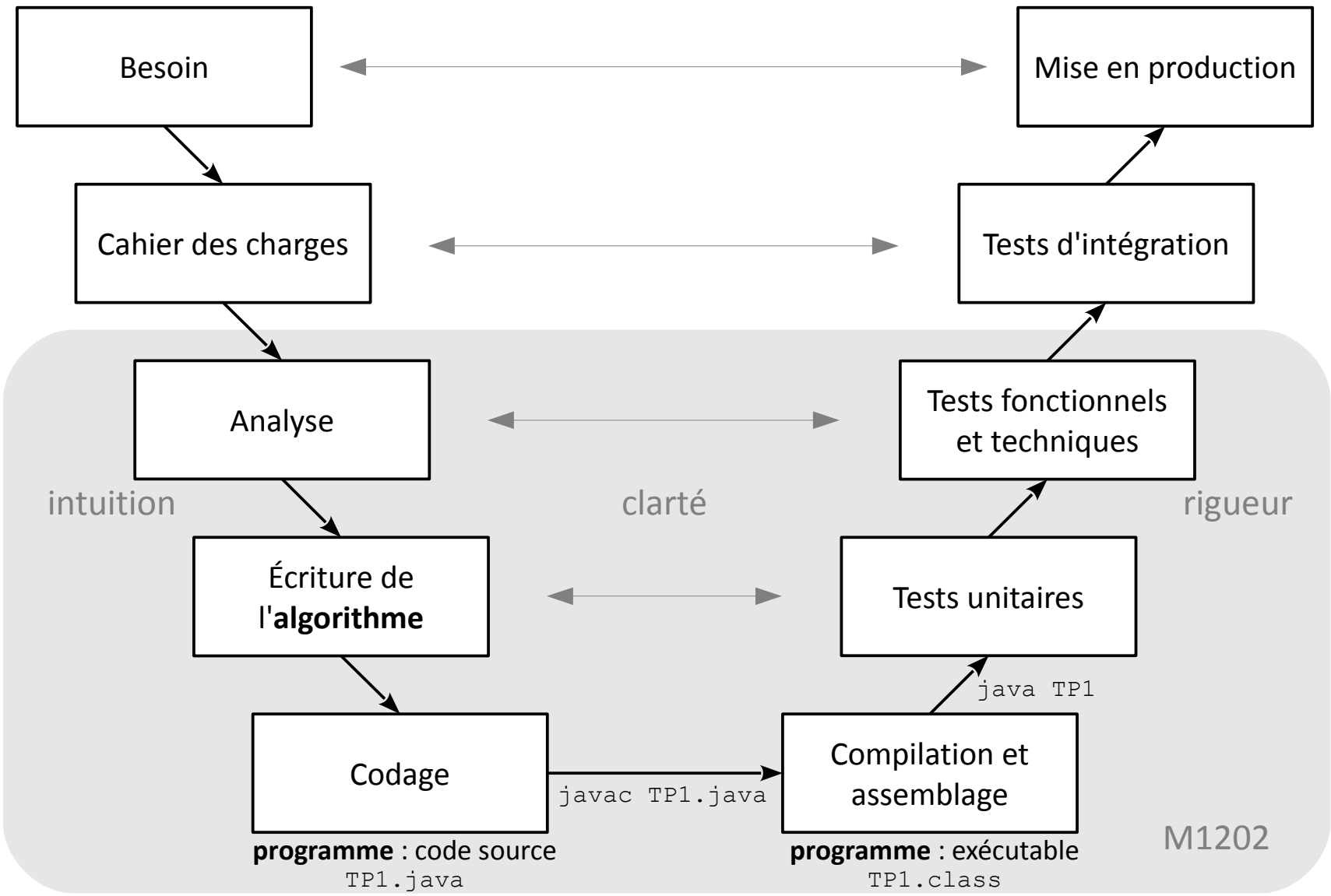
Mettre le contenu de la poêle **dans** l'assiette

Organigramme de la recette des crêpes



entrées → ingrédients
variables → récipients

De l'algorithmme au programme



Enjeux de l'algorithmique : correction, complexité

Algorithme correct ?

- donne le résultat attendu ? → **preuve de correction**
- quel que soit le type d'entrées ? → **débuggage, tests unitaires**

Preuve de correction :

- « invariant » : propriété vraie tout au long de l'algorithme
 - vraie à la première étape
 - si vraie à une étape, vraie à l'étape suivante
- ⇒ vrai à la fin

En pratique, pour débiter :

- vérifier sur les “cas de base”
- vérifier sur des exemples aléatoires (attention aux exemples trop « simples »)

Algorithme rapide ?

- se termine ? → **preuve de terminaison**
- en combien de temps ? → **complexité**

Preuve de terminaison :

L'algorithme des crêpes se termine-t-il ?

→ le saladier sera forcément vide à un moment donné (prouvable mathématiquement), donc **oui**.

Complexité : Combien de temps l'algorithme prend-il pour se terminer ?

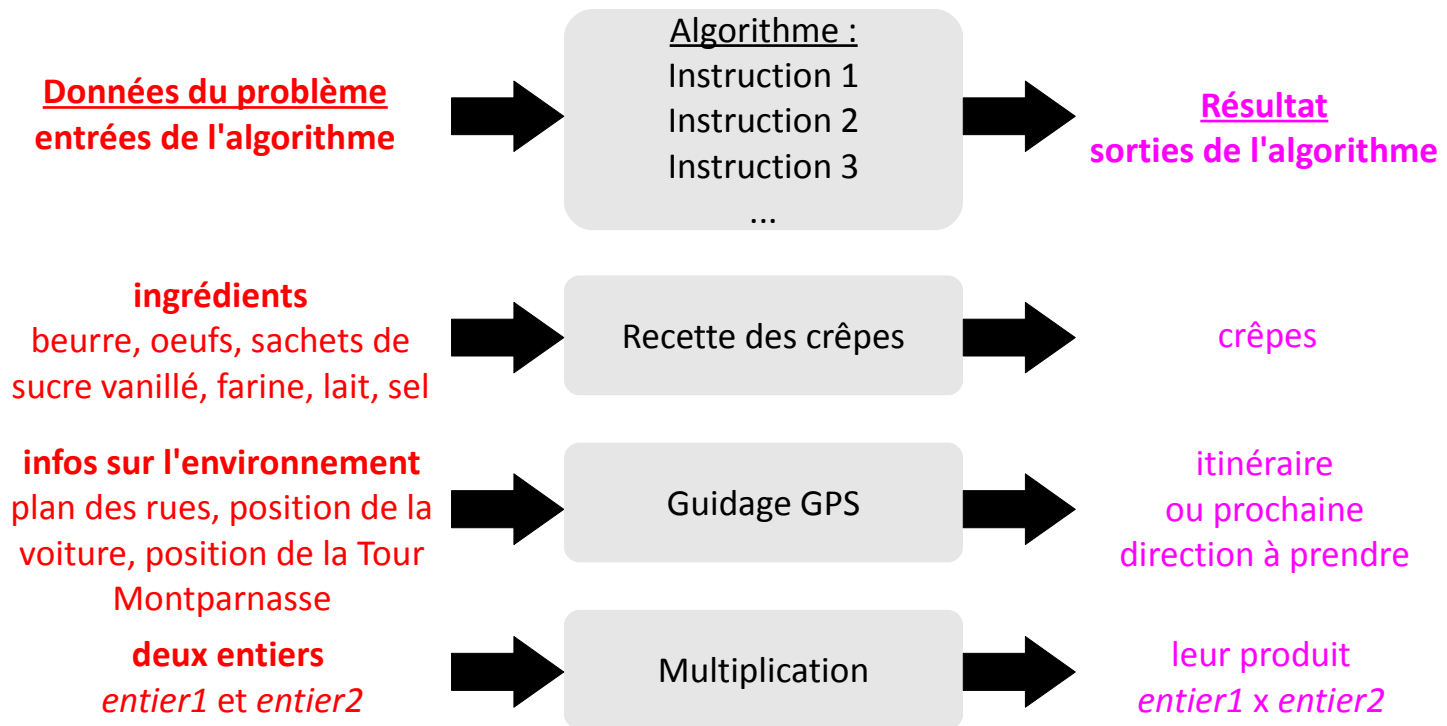
Théorie de la complexité :

- nombre d'opérations en fonction de la taille du problème, dans le pire cas
- prouver qu'on ne peut pas utiliser moins d'opérations pour résoudre le problème, dans le pire cas

En pratique, pour débiter :

- vérifier sur des exemples aléatoires
- connaître les cas difficiles

Composants d'un algorithme



Divers **types d'instructions** d'un algorithme :

- déclaration d'un algorithme
- appel d'un algorithme
- déclaration d'une **variable**
- **affectation** d'une **variable**
- **entrées** / **sorties**
- **boucle**
- test

Premier programme Java

La sortie de l'algorithme `reponseALaQuestion` est "gambette" si l'utilisateur a entré ce nom

```
public static void main(String[] arg){
    String nom, reponse;
    int nombreATrouver, nombreUtilisateur;

    //Identification de l'utilisateur
    nom = reponseALaQuestion("Comment vous appelez-vous ?");

    affiche(" ");
    affiche("Bonjour "+nom+" !");

    //Choix du nombre aléatoire
    nombreATrouver = nombreAleatoire(1,10);
    affiche("L'ordinateur a choisi un nombre entre 1 et 10.");
    affiche("Essayez de le deviner.");

    //Premier essai de l'utilisateur
    reponse = reponseALaQuestion("Premier essai :");
    nombreUtilisateur = Integer.parseInt(reponse);

    if (nombreUtilisateur == nombreATrouver){
        affiche("Bravo "+nom+", vous avez trouve !");
    }
}
```

entrées

La sortie de l'algorithme `nombreAleatoire` est 3 si l'ordinateur a choisi ce numéro au hasard

variables

affectations

appel d'algorithme

test

Variables et affectation

Dans un algorithme, une **variable** possède :

- un **nom**,
- une **valeur**,
- un **type** (ensemble des valeurs que peut prendre la variable).

La **valeur** d'une variable :

- est **fixe à un moment donné**,
- peut **changer au cours du temps**.

L'**affectation** change la valeur d'une variable :

- $a \leftarrow 5$ (**pseudo-code**) / $a=5$ (**Java**) :
 - la variable a prend la valeur 5
 - la valeur précédente est perdue (“écrasée”)
- $a \leftarrow b$ (**pseudo-code**) / $a=b$ (**Java**) :
 - la variable a prend la valeur de la variable b
 - la valeur précédente de a est perdue (“écrasée”)
 - la valeur de b n'est pas modifiée
 - a et b doivent être de même type
(ou de type compatible)

En revanche, le **nom** et le **type** d'une variable **ne changent pas**.

→ Dans un programme, **déclarer une variable** = fixer son type et son nom.

Pseudo-code : Variables : entiers a et b

Java : `int a,b;`

Dans un **algorithme**, choisir pour les variables :

- un nom composé de **lettres** et éventuellement de **chiffres**
- un nom **expressif**, par exemple :
 - *chaîne, requête1...* pour une chaîne de caractères
 - *n, a, compteur, nbOperations, longueur...* pour un entier
 - *x, y, température* pour un réel
 - *estEntier, testEntier, trouvé...* pour un booléen
- un nom **assez court** (il faut l'écrire !)
- éviter les **noms réservés** : *pour, tant que, si...*

Dans un **programme** :

- **éviter** les lettres accentuées et la ponctuation
- préférer l'**anglais** si votre code source est diffusé largement
- être **expressif** et **lisible** :
 - *est_entier* ou *estEntier* plutôt que *estentier*

Votre code sera relu, par vous ou par d'autres...

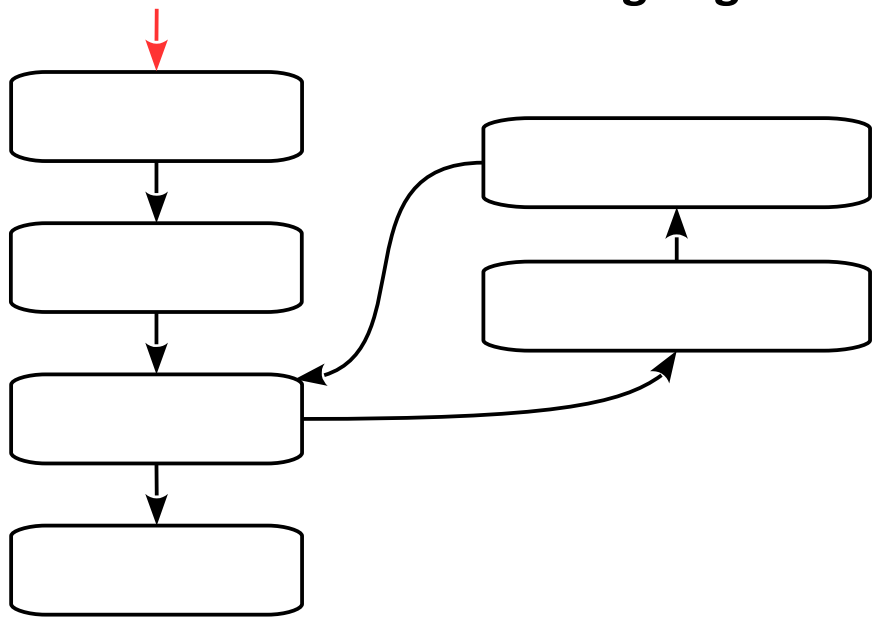
En **Java** le nom de variable doit commencer par une lettre, de préférence minuscule. Il peut contenir des lettres, des chiffres, et les symboles \$ et _. Il est sensible à la casse (majuscules/minuscules).

Dictionnaire pseudo-code / Java

	Pseudo-code	Java
Déclaration d'un algorithme	addition Entrées : entiers <i>i</i> et <i>j</i> Type de sortie : entier Début ... renvoyer ... Fin	<pre>public static int addition(int i, int j){ ... return ... }</pre>
Déclaration d'une variable	Variables : entier <i>i</i>	<code>int i;</code>
Affectation	$i \leftarrow 1$	<code>i = 1;</code>
Test	Si <i>i</i> =1 alors ... Sinon } <i>Le bloc du « Sinon »</i> ... } <i>est facultatif</i> Fin Si	<pre>if (i==1){ ... } else { ... } <i>Le bloc du « else »</i> est facultatif }</pre>
Boucle	Tant que <i>i</i> <3 faire ... Fin Tant que	<pre>while (i<3) { ... }</pre>

De l'organigramme au code Java

organigramme



pseudo-code

deuxPuissance :

Entrées :

Type de sortie :

Variables :

Début

Fin

code Java

```
public static
```

Trace (valeur de chaque variable après chaque instruction de l'algorithme) :

Valeurs des variables `compteur` et `resultat` après le `i`-ième passage dans la boucle `while` :

<code>i</code>	<code>compteur</code>	<code>resultat</code>
0		
1		
2		
3		
4		

Codage binaire

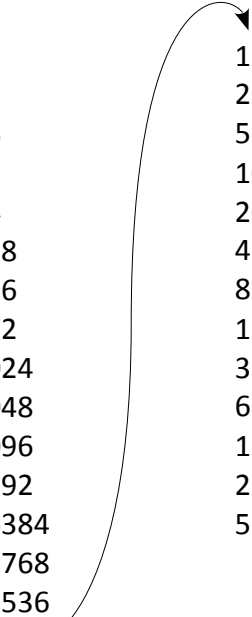
La "minute votes SMS"

Programme Java :

```
import java.io.*;
public class Boucle{
    public static void main(String [] arg){
        int i=1;
        while(i>0){
            i=i*2;
        }
        System.out.print("J'ai fini !");
    }
}
```

Résultat du programme :

2	
4	131072
8	262144
16	524288
32	1048576
64	2097152
128	4194304
256	8388608
512	16777216
1024	33554432
2048	67108864
4096	134217728
8192	268435456
16384	536870912
32768	
65536	



La "minute mathématique"

Pour le stockage comme pour le traitement d'instructions, il est nécessaire que toutes les données traitées par un ordinateur soient codées en **binaire**, par des **0** et des **1**.

Exemple de nombre entier en binaire : 1101100001101

0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	1
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

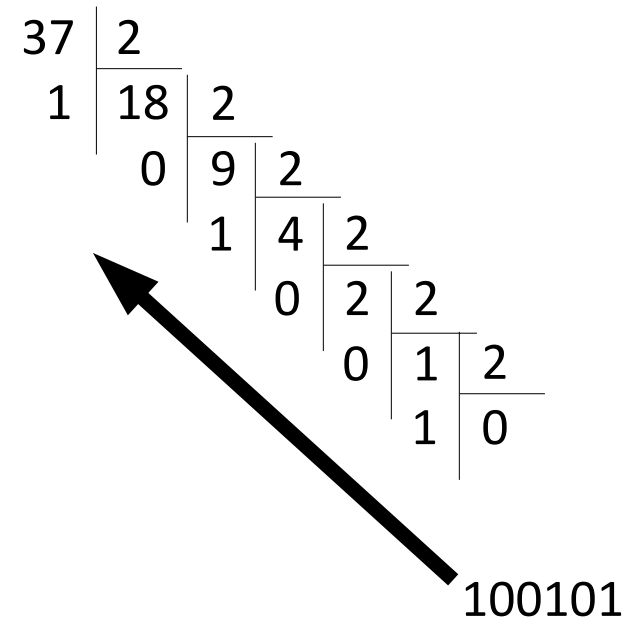
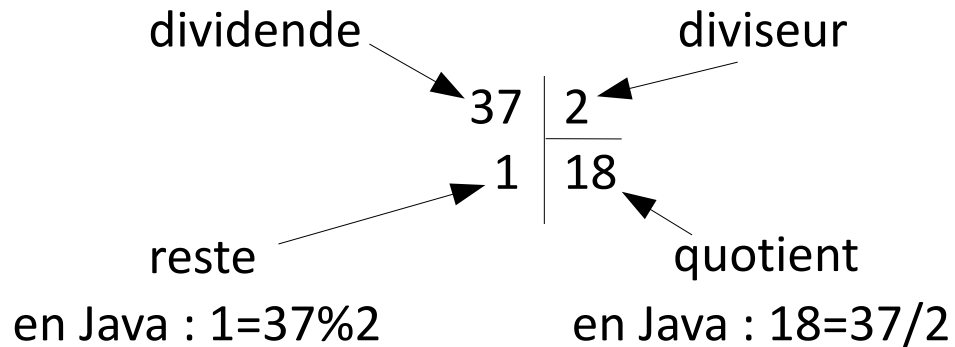
$$2^{12} + 2^{11} + 2^9 + 2^8 + 2^3 + 2^2 + 2^0 = 4096 + 2048 + 512 + 256 + 8 + 4 + 1 = 6925$$

Codage binaire

Pour le stockage comme pour le traitement d'instructions, il est nécessaire que toutes les données traitées par un ordinateur soient codées en **binaire**, par des **0** et des **1**.

Exemple : écrire 37 en binaire ?

Division euclidienne :



Pour faire son geek :

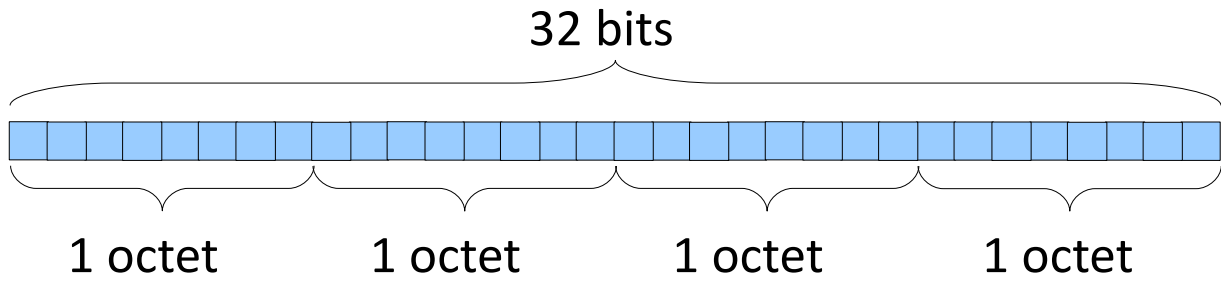
- compter sur ses doigts en binaire, jusqu'à 2^{10}
- faire des estimations de nombres données en binaire : $2^{10} = 1024$ donc $2^{10} \approx 1000$

$$2^{32} \approx 4 \text{ milliards}$$

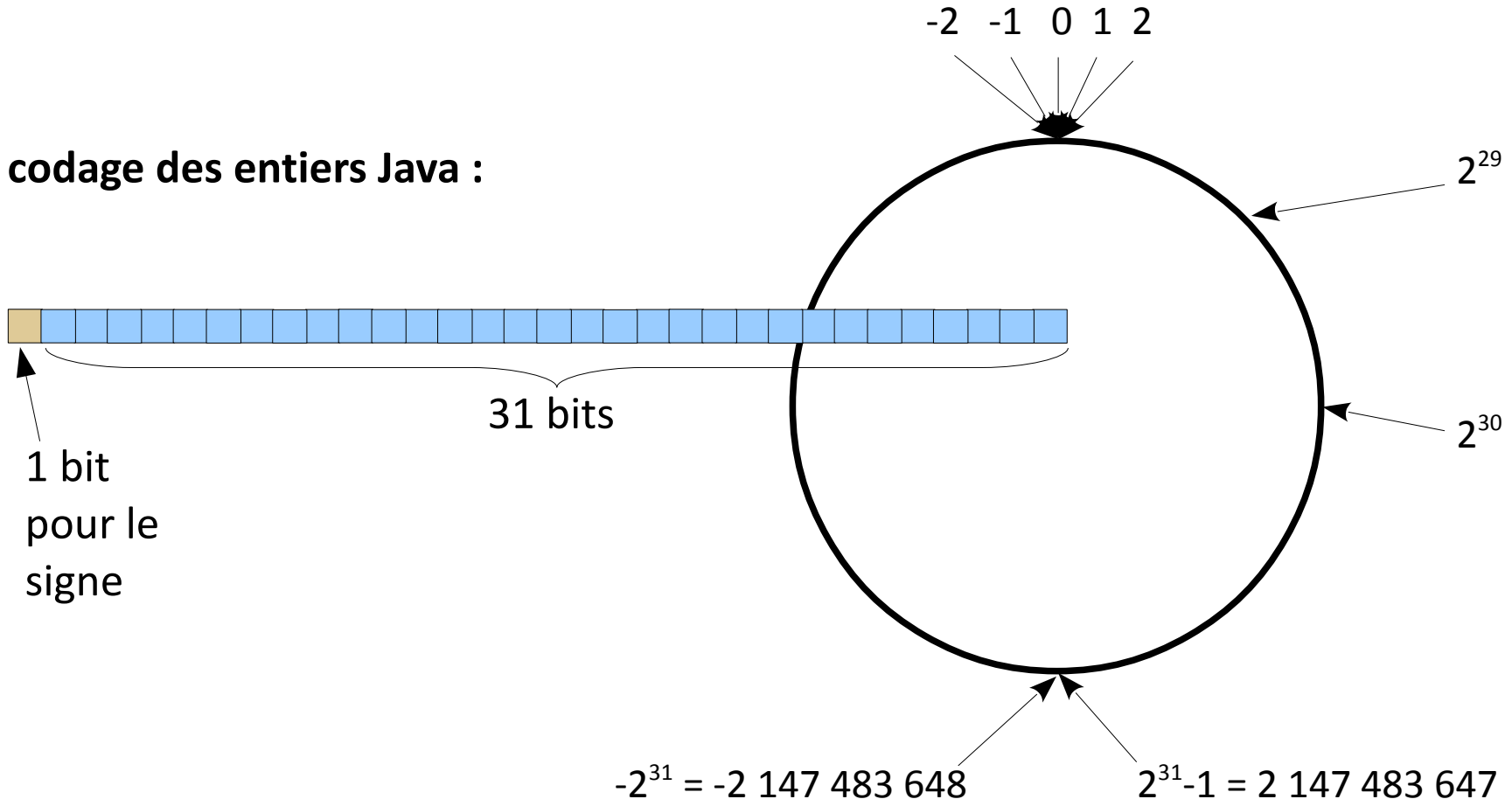
$$2^{32} = 4\,294\,967\,296$$

Codage des entiers 32 bits

- 1 bit = 0 ou 1
- 1 octet = 8 bits
- 1 Ko (kiloctet) = 1024 octets
- 1 Mo (mégaoctet) = 1024 Ko (disquette)
- 1 Go (gigaoctet) = 1024 Mo (carte mémoire, 2h de vidéo en DivX)
- 1 To (téraoctet) = 1024 Go (disque dur externe)



Le codage des entiers Java :



Autres codages

• Chaînes de caractères

- ASCII : 7 bits, caractères simples codés de 32 à 127
- ANSI : 8 bits, caractères simples codés de 32 à 127, caractères accentués de 128 à 255
- UTF-8 : de 1 à 4 octets

• Couleurs d'une image

- RGB : "red, green, blue", 1 octet pour chacun :
 - valeurs entre 0 et 255
 - codage hexadécimal avec 2 symboles

La "minute culturelle"

Hexadécimal : en base 16 (ἕξάς : six, decem : dix)

Codé par les chiffres de 0 à 9 et les lettres A B C D E F

	A	B	C	D	E	F
	↑	↑	↑	↑	↑	↑
	10	11	12	13	14	15

- Deux symboles pour un octet :
 16^2 valeurs possibles = 256

- Utilisé pour coder les couleurs en HTML :
couleur="#RRGGBB"
rouge="#FF0000", vert="#00FF00"
#800080 ?

Les booléens

- **Opérations sur les booléens :**
et, ou, non

ET	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

NON	VRAI	FAUX
	FAUX	VRAI

- Une **égalité** est un **booléen** : $i=4$ est soit VRAI, soit FAUX
 - Une **inégalité** est un **booléen** : $i>10$ est soit VRAI, soit FAUX
- on peut placer utiliser les opérations sur les booléens pour des inégalités, des égalités, etc.
- Exemples : Si $(i=4)$ OU $(i>10)$ alors ... / Si NON $(i=4)$ alors ...

Les opérations de base en Java

- **Type entier `int`**

+ (addition), - (soustraction), * (multiplication), / (division entière),
% (reste modulo), ^ (puissance), == (égalité), < et > (inégalité stricte),
<= et >= (inégalité large), != (non égalité)

- **Type flottant `float`, ou `double` (plus précis)**

+ (addition), - (soustraction), * (multiplication), / (division),
% (reste modulo), == (égalité), < et > (inégalité stricte),
<= et >= (inégalité large), != (non égalité)

- **Type booléen `boolean`**

`false` (faux), `true` (vrai), `&&` (et), `||` (ou), `!` (non)

- **Type chaîne de caractères `String`**

+ (concaténation : `"M"+"1202"` est équivalent à `"M1202"`)

Les tableaux en pseudo-code

Les tableaux sont des variables qui contiennent **plusieurs variables de même type**, stockées chacune dans une des cases du tableau.

en pseudo-code

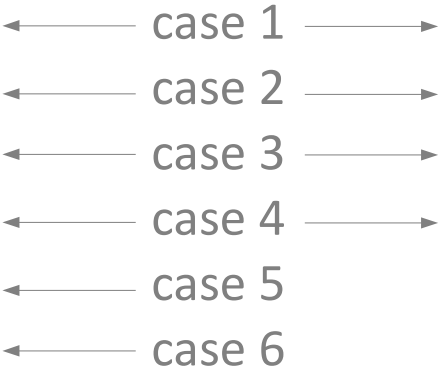
Variables : *tableau1*, un tableau d'entiers,
tableau2, un tableau de chaînes de caractères

Par exemple,

Un **tableau d'entiers** :

Un **tableau de chaînes de caractères** :

4
5
1
23
8
9



"chaine1"
"chaine2"
"blabla"
"toto"

longueur 4

longueur d'un tableau = nombre de cases
↘ **longueur(tableau2)**

longueur 6

```
tableau1 ← nouveauTableau(6)  
case(tableau1,1) ← 4  
case(tableau1,2) ← 5  
case(tableau1,3) ← 1  
case(tableau1,4) ← 23  
case(tableau1,5) ← 8  
case(tableau1,6) ← 9
```

```
tableau2 ← nouveauTableau(4)  
case(tableau2,1) ← "chaine1"  
case(tableau2,2) ← "chaine2"  
case(tableau2,3) ← "blabla"  
case(tableau2,4) ← "toto"
```



Les tableaux en Java

en Java

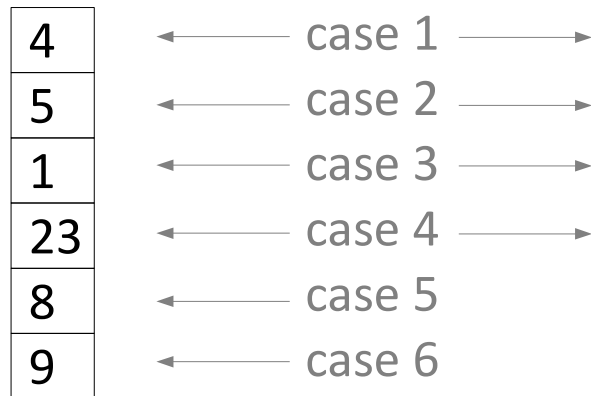
Les tableaux sont des variables qui contiennent **plusieurs variables de même type**, stockées chacune dans une des cases du tableau.

Déclaration des variables de tableau (tableau d'entiers, et tableau de chaînes de caractères) :

```
int[] tableau1; String[] tableau2;
```

Par exemple,

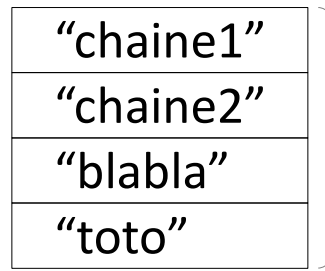
Un **tableau d'entiers** :



longueur 6

```
tableau1=new int[6];  
tableau1[0]=4;  
tableau1[1]=5;  
tableau1[2]=1;  
tableau1[3]=23;  
tableau1[4]=8;  
tableau1[5]=9;
```

Un **tableau de chaînes de caractères** :



longueur 4

longueur d'un tableau
= nombre de cases

→ `tableau2.length`

```
tableau2=new String[4];  
tableau2[0]="chaine1";  
tableau2[1]="chaine2";  
tableau2[2]="blabla";  
tableau2[3]="toto";
```

Attention, cases du tableau `t` numérotées de 0 à `t.length-1` en Java.

Manipulation de tableaux

Pour lire le contenu d'un tableau...

il faut une **boucle pour aller lire chaque case** !

Si le tableau a été prévu trop court au début, **impossible de changer sa longueur...** il faut une boucle pour le recopier dans un tableau plus grand !

Possibilité de créer des **tableaux de tableaux...**

L'algorithme de base, le **parcours du tableau**, pour visiter chaque case :

Variables : tableau d'entiers *tab*, entier *i*

$i \leftarrow 1$

Tant que $i < \text{longueur}(tab)+1$ faire :

 [des choses avec la *i*-ième case du tableau **case**(*tab*,*i*)...]

$i \leftarrow i+1$

Fin Tant que

Affichage du contenu d'un tableau d'entiers

Algorithme **afficheTableau**

Variable d'entrée :

Variable :

Début

Fin

```
public static      afficheTableau(      tableau1) {
```

```
}
```



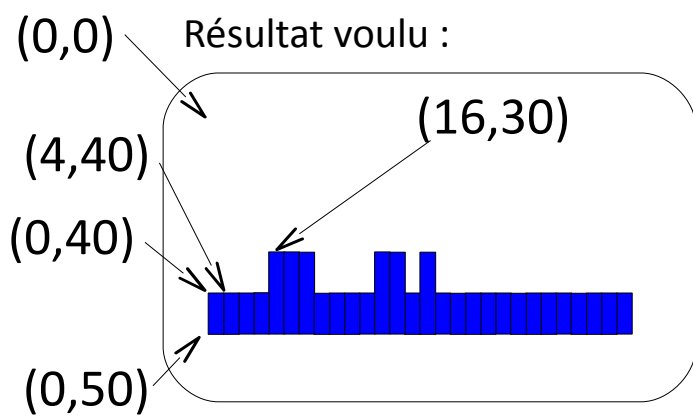
Graphique du nombre d'apparitions des mots dans un texte

J'ai cueilli ce
brin de
bruyère
L'automne
est morte
souviens-t'en
Nous ne nous
verrons plus
sur terre
Odeur du
temps brin de
bruyère
Et souviens-
toi que je
t'attends

j	1
ai	1
cueilli	1
ce	1
brin	2
de	2
bruyère	2
l	1
automne	1
est	1
morte	1
souviens	2
t	2
en	1
nous	2
ne	1
verrons	1
plus	1
sur	1
terre	1
odeur	1
du	1
temps	1
et	1
toi	1
que	1
je	1
attends	1

tableau
de chaînes
de caractères
mots

tableau d'entiers *nbApparitions*



Graphique du nombre d'apparitions des mots dans un texte



Algorithme **dessineHistogramme**

Entrée : tableau de chaînes de caractères *mots*
et tableau d'entiers *nbApparitions*.

Variable : entier *i*

Début

$i \leftarrow 1$

Tant que

faire :

$i \leftarrow 1 + i$

Fin Tant que

Fin

La boucle “for” / “Pour tout...”

La boucle “for” / “Pour tout”

Une boucle pour **parcourir tous les entiers entre deux valeurs entières.**

Algorithme **dessineHistogramme**

Entrée : tableau de chaînes de caractères *mots* et tableau d'entiers *nbApparitions*.

Variable : entier *compteur*

Début

compteur ← 1

Tant que *compteur* < **longueur**(*mots*)+1 **faire** :

```
    dessineRectanglePlein(compteur*4-4,  
                          50-10*case(nbApparitions,compteur),  
                          4,10*case(nbApparitions,compteur),  
                          couleurRGB(0,0,255))  
    compteur ← 1 + compteur
```

Fin Tant que

Fin

En Java :

```
int compteur;  
compteur=1;  
while (compteur<mots.length+1) {  
    ...  
}
```

Algorithme **dessineHistogramme**

Entrée : tableau de chaînes de caractères *mots* et tableau d'entiers *nbApparitions*.

Variable : entier *compteur*

Début

Pour tout *compteur* de 1 à **longueur**(*mots*) **faire** :

```
    dessineRectanglePlein(compteur*4-4,  
                          50-10*case(nbApparitions,compteur),  
                          4,10*case(nbApparitions,compteur),  
                          couleurRGB(0,0,255))
```

Fin Pour

Fin

```
    déclaration +  
    initialisation      condition d'arrêt      mise à jour  
for (int compteur=1; compteur<mots.length+1; compteur++) {  
    ...  
}
```


Souris et autres périphériques d'entrée

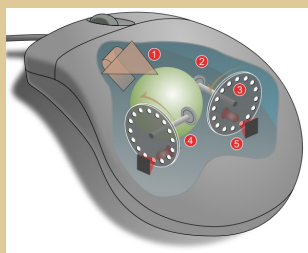
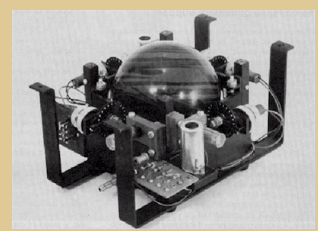
La "minute culturelle"

L'invention de la souris

1952 Trackball (boule de commande)
Tom Cranston et Fred Longstaff
(Marine Royale Canadienne)

1963 Souris mécanique
Douglas Engelbart et Bill English
(Stanford Research Institute)

1977 Souris optique
Jean-Daniel Nicoud et André Guignard
(Ecole polytechnique fédérale de Lausanne)



Entrées-sorties dans la communication ordinateur – utilisateur

Quel **type de données** utiliser en **algorithmique** pour coder les entrées-sorties ?

Périphérique	Type de données transmises
Clavier	chaîne de caractères
Souris à 1 bouton	deux entiers (abscisse et ordonnée) + un booléen (clic ou pas)
Webcam	image, donc tableau de tableaux de couleurs RGB
Kinect	image + tableau de tableaux d'entiers (profondeur)
Ecran	Si ligne de commande : chaîne de caractères Si interface graphique : image, donc tableau de tableaux de couleurs RGB

La "minute xkcd" : 243

Crédits :
http://en.wikipedia.org/wiki/File:DATAR_trackball.jpg
http://cerncourier.com/cws/article/cern/28358/1/cernbooks2_12-00
 Jeremymkemp, Pbroks13,
http://fr.wikipedia.org/wiki/Fichier:Mouse_mechanism_diagram.svg
 Stéphane Magnenat (User:Nct)
<http://en.wikipedia.org/wiki/File:SmakyMouseAG.jpeg>

Instructions d'entrée-sortie en Java et pseudo-code

Entrées clavier

en pseudo-code

chaîne de caractères



reponseALaQuestion(*questionAAfficher*)

affiche la question *questionAAfficher* et renvoie une chaîne de caractères.

Exemple : **reponseALaQuestion**("Quel est votre nom") me laisse taper mon nom au clavier et renvoie "Gambette"

en Java

```
Scanner lectureClavier = new Scanner(System.in);  
String chaineLue = lectureClavier.next();
```

LectureClavier.next() renvoie une chaîne de caractères.

```
int entierLu = lectureClavier.nextInt();
```

LectureClavier.nextInt() renvoie un entier.

Sorties écran (ligne de commande)

en pseudo-code

chaîne de caractères



affiche(*chaineAAfficher*)

affiche la chaîne de caractères *chaineAAfficher* et ne renvoie rien.

en Java

```
String chaineAAfficher="blabla";  
System.out.println(chaineAAfficher);  
affiche la chaîne de caractères questionAAfficher,  
puis retourne à la ligne, mais ne renvoie rien.
```

```
System.out.print(chaineAAfficher);  
affiche la chaîne de caractères questionAAfficher et  
ne renvoie rien.
```

Les fonctions – Introduction

Algorithme pour **construire une maison** ?

→ faire appel à un **maître d'oeuvre**

Que fait le maître d'oeuvre ?

→ il réunit les informations sur la maison qui va être construite

→ il trouve des artisans

→ à chaque artisan, il donne des informations sur ce qu'il attend, et récupère le résultat

→ chaque artisan peut lui-même sous-traiter une partie de son travail à un autre artisan : il donne les informations sur ce qu'il attend, et récupère le résultat

→ la maison se construit

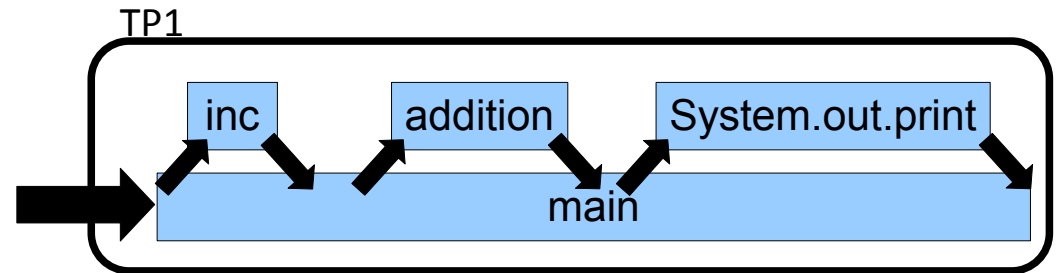
La construction de la maison : le programme TP1

Le maître d'oeuvre : la fonction main

L'artisan 1 : la fonction inc

L'artisan 2 : la fonction addition

L'artisan 3 : la fonction System.out.print



```
public class TP1{
    public static void main(String[] arg){
        int i,j;
        i=5;
        j=34;
        System.out.print("i+1="+inc(i)+" , i="+i+" ,
            j="+j+" , somme : "+addition(i,j));
    }
    public static int inc(int i){
        i=i+1;
        return i;
    }
    public static int addition(int i, int j){
        return i+j;
    }
}
```

Les fonctions – spécificités en Java

Les fonctions = les algorithmes

Zéro, un ou plusieurs paramètres en entrée...

Une sortie (ou aucune : void en Java)...

En Java :

Toujours une fonction **main** qui ne renvoie rien (type de sortie : void) qui prend en entrée les paramètres du programme (tableau arg) et qui est la seule fonction exécutée quand on exécute le programme.

Déclaration des autres fonctions après la fonction main : ne s'exécutent que si elles sont appelées pendant l'exécution de la fonction main

Les paramètres du programme :

Si on exécute dans le terminal la commande
java TP1 toto 1 10 10.5
alors la variable `arg` contient le tableau de chaînes de caractères :

"toto"	"1"	"10"	"10.5"
--------	-----	------	--------

Visibilité des variables : toute variable déclarée à l'intérieur d'une fonction n'est valable que dans cette fonction et ne peut pas être utilisée ailleurs.
→ Variables « locales »

```
public class TP1{
    public static void main(String[] arg){
        int i,j;
        i=5;
        j=34;
        System.out.print("i+1="+inc(i)+" , i="+i+" ,
            j="+j+" , somme : "+addition(i,j));
    }
    public static int inc(int i){
        i=i+1;
        return i;
    }
    public static int addition(int i, int j){
        return i+j;
    }
}
```

pas la même variable *i* même si elles ont la même valeur !

appel d'algorithmme

type de sortie

Ligne de déclaration d'un algorithmme / d'une fonction

variables d'entrée précédées de leur type

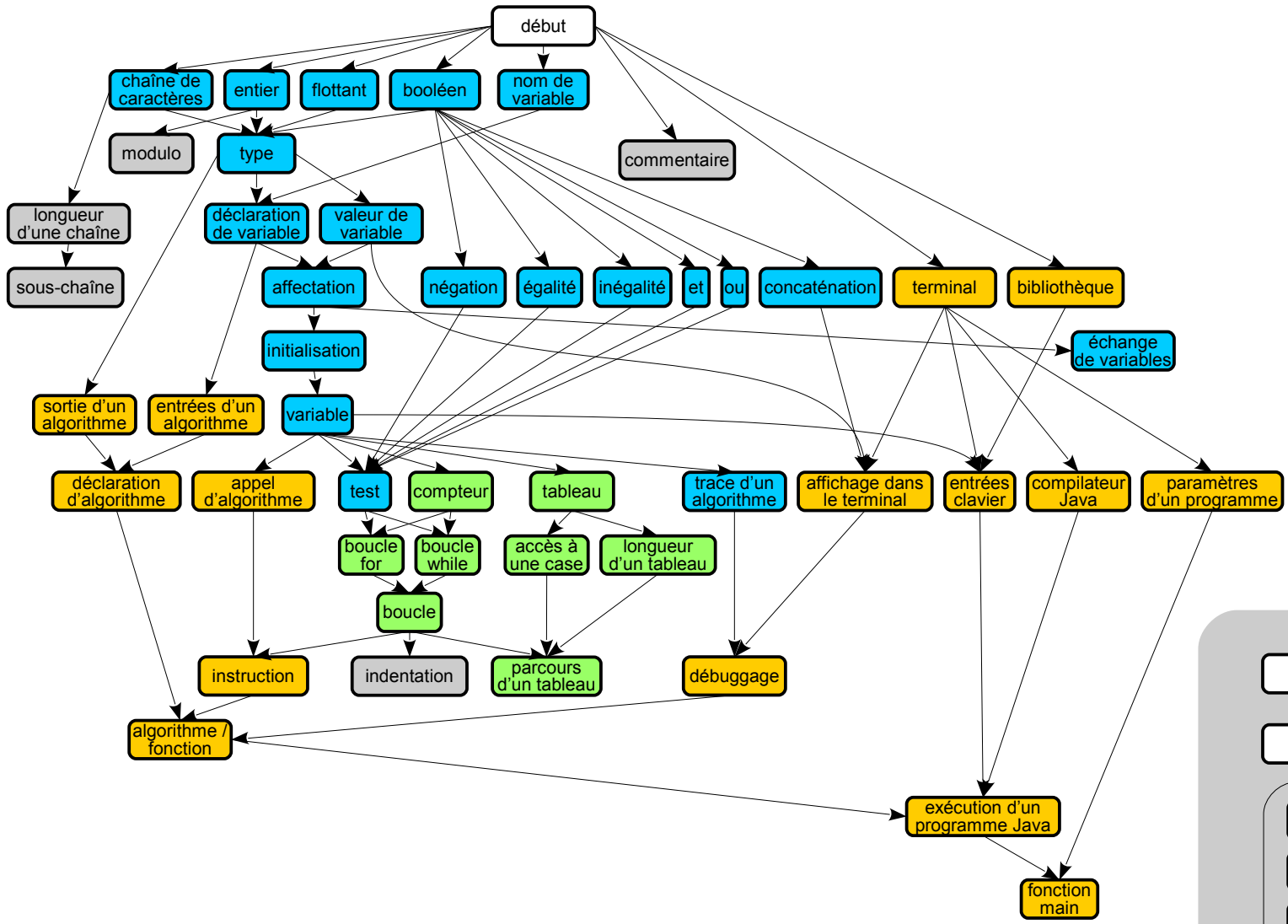
Les fonctions – Exemples

La “minute mathématique”

fonction	exemple	entrées possibles	sortie
cosinus	$\text{cosinus}(1.047)=0.5$	flottant	flottant
somme	$\text{somme}(2,3)=5$	2 entiers 2 flottants	entier flottant
opposé	$\text{opposé}(4)=-4$	entier flottant	entier flottant
inverse	$\text{inverse}(10)=0.1$	flottant	flottant
différence	$\text{différence}(2,3)=-1$	2 entiers 2 flottants	entier flottant
estPositif	$\text{estPositif}(-5)=\text{FAUX}$	entier	booléen
partieEntière	$\text{partieEntière}(5.6)=5$	flottant	entier
moyenne	$\text{moyenne}(2,4,6)=4$	3 flottants	flottant
min	$\text{min}(\{6,2,4,3\})=2$	tableau d'entiers tableau de flottants	entier flottant

Attention : point au lieu de virgule dans les flottants (pour différencier de la virgule qui sépare les entrées d'une fonction)

Concepts du cours M1202



A

B

Il est utile d'avoir bien compris A avant de bien comprendre B

fin du cours 2

fin du cours 3

fin du cours 5

Semestre 1 M1202

Compétences du cours M1202

Être capable de :

C1) comprendre le fonctionnement d'un algorithme :

C1a) en identifiant les différents éléments de base de l'algorithme

C1b) en simulant son comportement à l'aide d'une trace

C1c) en interprétant une courte description en français de ses spécifications

C2) concevoir un algorithme pour résoudre un problème :

C2a) en analysant le problème :

- pour comprendre les besoins

- pour le découper éventuellement en sous-problèmes

C2b) en écrivant un algorithme correctement structuré :

- avec des entrées et des sorties correctement définies

- en utilisant des structures conditionnelles

- en utilisant des répétitions d'opérations (à l'aide de boucles)

- en faisant appel à d'autres algorithmes dont les spécifications sont connues, notamment pour interagir avec l'utilisateur

- en découpant le code de manière modulaire en divers composants indépendants et réutilisables

C3) comprendre le fonctionnement d'un code Java :

C3a) en identifiant les divers éléments de structure du code Java

C3b) en identifiant les éventuelles erreurs qu'il contient par un processus de débogage

C4) écrire un programme Java :

C4a) en respectant la syntaxe Java

C4b) en choisissant ou en respectant des conventions de nommage appropriées

C4c) en testant le code obtenu (par compilation puis exécution)

Méthodo : Lire et comprendre un algorithme

Premiers éléments à identifier :

- qu'est-ce que l'algorithme **prend en entrée** ? **Combien** de variables, de quel **type** ?
- qu'est-ce que l'algorithme **renvoie en sortie** ? **Rien** ? Ou bien **une** variable ? De quel **type** ?

Ensuite :

- quels sont les autres **algorithmes appelés** par l'algorithme ?

Enfin :

- faire la **trace** de l'algorithme, c'est-à-dire l'essayer sur un **exemple** (... ou plusieurs pour passer au moins une fois par toutes les instructions de l'algorithme) et voir ce que valent **toutes les variables à chaque étape** (et noter ces valeurs dans un tableau contenant une ligne par variable et une colonne par étape),
- noter en particulier le **résultat obtenu en sortie** pour une **entrée testée**.

Méthodo : Concevoir un algorithme

Premiers éléments à identifier :

- quels sont les **outils à disposition** ? (pour ces outils : données en entrée, type de données en entrée, résultat en sortie, type de résultat en sortie, résultat attendu sur un exemple...)
- quel est le **comportement attendu** pour mon algorithme ? (données en entrée, type de données en entrée, résultat en sortie, type de résultat en sortie, résultat attendu sur un exemple...)

Ensuite, résoudre le problème en utilisant ces outils :

- comment résoudre le problème **étape par étape** ? (essayer sur l'exemple testé)
- est-ce que les **outils à disposition** sont **utilisables** pour réaliser chaque étape ?

Enfin :

- comment **structurer** l'utilisation des outils à disposition ? (**combinaison** des différents outils à l'intérieur de structure de **boucles**, de **tests**, utilisation d'un **organigramme**...)
- comment **décomposer** le problème ? (et **reformuler** chaque sous-problème pour le résoudre avec les outils à disposition, écrire un algorithme par sous-problème)

Méthodo : Connaître les éléments de base d'un algorithme

en pseudo-code

Un algorithme résout un problème, en fournissant un **résultat en sortie** à partir de **données en entrée**.

Entrées : entiers i et j
Type de sortie : entier

une instruction par ligne !

Pour cela, il utilise plusieurs types d'instructions :

- des **affectations** dans des **variables** (mémoires) $\text{produit} \leftarrow \text{entier1} + \text{produit}$
variable valeur
- des **boucles** **Tant que ... faire : ... Fin TantQue**
- des **appels** à d'autres algorithmes **Addition(3,5)** { entrées de l'algorithme entre parenthèses, respectant l'ordre de déclaration des entrées
- des **tests** **Si ... alors : ... sinon : ... Fin Si**
- des **"lectures"** d'entrées et **"renvois"** de sorties
renvoyer $i+j$

Chaque **variable** a un **nom**. On doit :

- la **déclarer** en définissant son **type** (ensemble de valeurs possibles)
 - puis l'**initialiser** (lui donner une **valeur** par une affectation) avant de l'utiliser.
- Variables : entiers a et b
type
- $a \leftarrow 2$
 $b \leftarrow a+2$

Méthodo : Connaître les éléments de base d'un algorithme

en Java

Un algorithme résout un problème, en fournissant un **résultat en sortie** à partir de **données en entrée**.

```
public static int Addition(int i, int j){...}
type de sortie entrées précédées de leur type
```

Pour cela, il utilise plusieurs types d'instructions : *instructions finies par ";"*

- des **affectations** dans des **variables** (mémoires)
- des **boucles** `while(...){...}`
- des **appels** à d'autres algorithmes `Addition(3,5);`
- des **tests** `if(...){...}else{...}`
- des **"lectures"** d'entrées et **"renvois"** de sorties

```
produit=entier1+produit;
variable valeur
```

entrées de l'algorithme entre parenthèses, respectant l'ordre de déclaration des entrées

```
return i+j;
```

Chaque **variable** a un **nom**. On doit :

- la **déclarer** en définissant son **type** (ensemble de valeurs possibles)
- puis l'**initialiser** (lui donner une **valeur** par une affectation) avant de l'utiliser.

```
int a, b;
type
```

```
a=2;
b=a+2;
```