Algorithmique et Programmation (1/3)

Objectifs :

► Approfondir l'algorithmique abordée au premier semestre : nouveaux types de données (énumérations, types composés), algorithmes de recherche, algorithmes de tris, récursivité

► Aborder de nouveaux aspects de l'algorithmique : complexité des algorithmes

Références :

Algorithmes en Java, R. Sedgewick, Pearson Education

Programmation – Cours et Exercices, G. Chaty & J. Vicard, Ellipses

Algorithmes et structures de données avec ADA, C++ et Java, A. Guerid, P. Breguet & H. Rothlisberger, PPUR

Page du module : *www.u-picardie.fr/~furst/algo_prog.php*

Page du module de S1 : *http://home.mis.u-picardie.fr/~groult/Enseignement/IntroInfo/*

Algorithmique et Programmation (2/3)

Informatique : sciences et techniques du *traitement automatisé de l'information*, c'est à dire des données (information structurée).

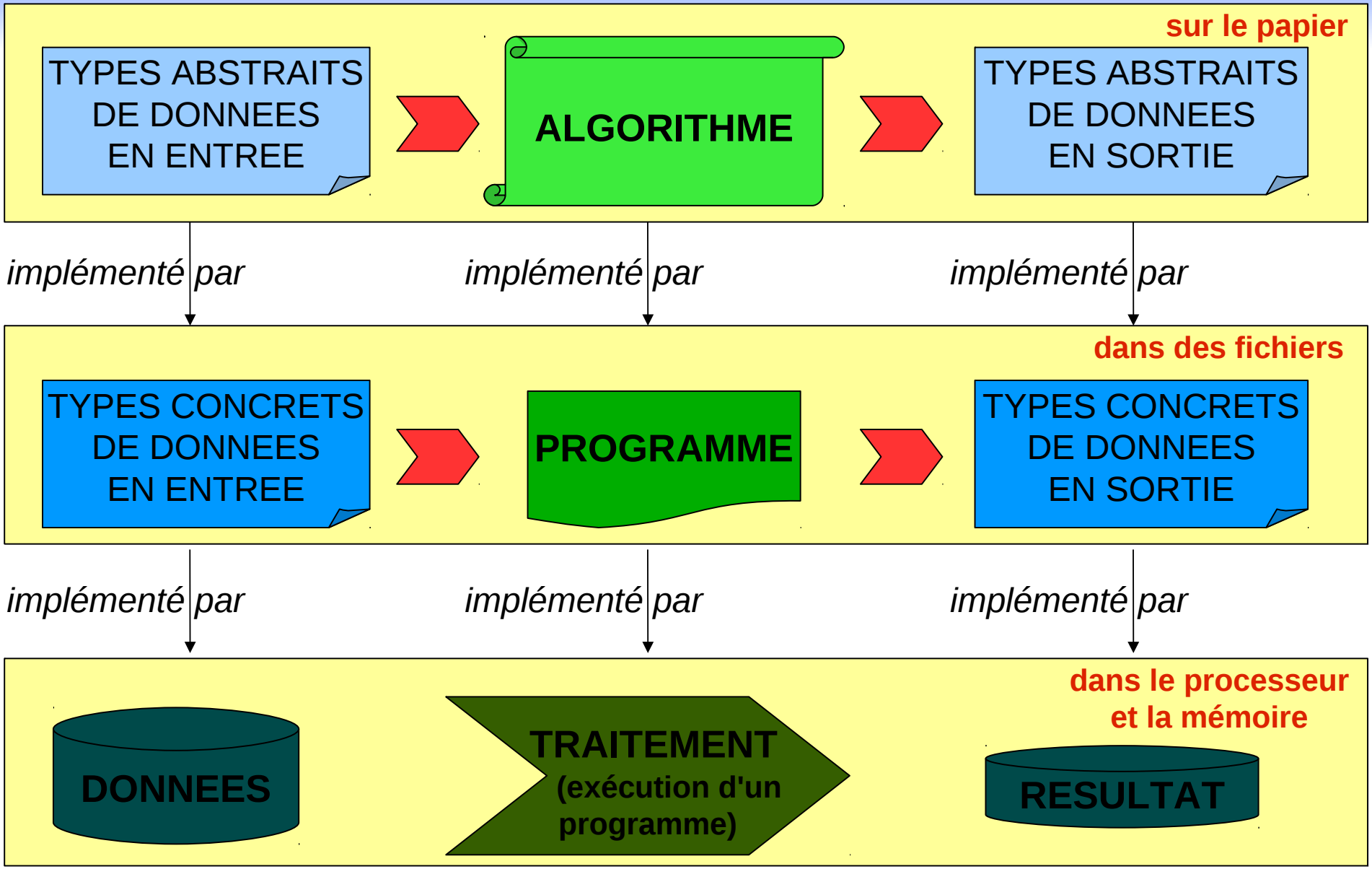
► On doit *définir quelle information traiter* : représentation et encodage des données

► On doit *définir comment traiter l'information* : algorithme

► On doit *faire exécuter cet algorithme par une machine* : programmation

« *Computer Science is no more about computers than astronomy is about telescopes.* » Edsger Dijkstra (prix Turing 1972)

Algorithmique et Programmation (3/3)



Variable

Dans un programme, les données sont manipulées via des variables :

- une variable est une *case mémoire*
- une variable est désignée par un nom (identifiant)
- une variable a un type de donnée (implicite dans certains langages)
- une variable contient une valeur du type et cette valeur peut varier

Cycle de vie d'une variable :

- déclaration de la variable (nom et type)
- affectations de valeurs à la variable
- suppression de la variable (souvent automatique)

Instructions

Déclaration de variable :

```
entier i;
```

Affectation de valeur à une variable :

```
i <- 23;
```

Expressions numériques et logiques :

```
((i+2) * (j-t/23)) mod 4
```

```
(a ou (b et non  c)) xor d
```

Lecture au clavier / Ecriture à l 'écran :

```
écrire "donnez votre nom";
chaine c;
lire c;
```

Structures de contrôle

Instructions conditionnelles :

```
si (température > 12) alors
  écrire "je vais me baigner";
finsi
```

```
si (a >= 16) alors
  écrire "mention bien";
sinon
  écrire "peut mieux faire";
finsi
```

Boucles :

```
chaine c <- "";
tantque (c ≠ "q") faire
  écrire "taper une chaine (q pour quitter)";
  lire c;
fintantque
```

```
pour (i allant de 1 à 30 pas 1) faire
  écrire "ceci est le " + i + "ème tour de boucle";
finpour
```

Algorithme

Un algorithme est une suite d'instructions séquentielles, éventuellement structurées par des conditionnelles et des boucles.

```
algorithme TestPrimalité // nom de l'algorithme (optionnel)
// déclarations des variables
     entier x,d;
     booléen b;
début // début des instructions
     écrire "donnez un entier positif";
     lire x;
     d <- 2;
     b <- false;
     tantque (non b et d*d <= x) faire
          si ((x mod d) == 0) alors
               écrire x+" n'est pas premier car il est divisible par "+d;
               b <- vrai;
          finsi
          d <- d+1;
     fintantque
     si (non b) a=alors
          écrire x+" est premier";
     finsi
fin
```

Comment écrire un algorithme (1/3)

Problème : écrire un algorithme qui calcule le PGCD de deux nombres

1- Bien comprendre le problème et, si le principe de l'algorithme n'est pas donné, trouver un principe de résolution

Méthode d'Euclide (~300 av. JC) : soient deux nombres entiers positifs A et B tels que A >= B. Si le reste R de la division de A par B est 0, le PGCD de A et B est B. Sinon, le PGCD de A et B est le PGCD de B et de R.

2- Si on ne comprend pas bien le principe, l'utiliser sur un exemple

Soit à calculer le PGCD de 123 et 27.

123	27
15	4

27	15
12	1

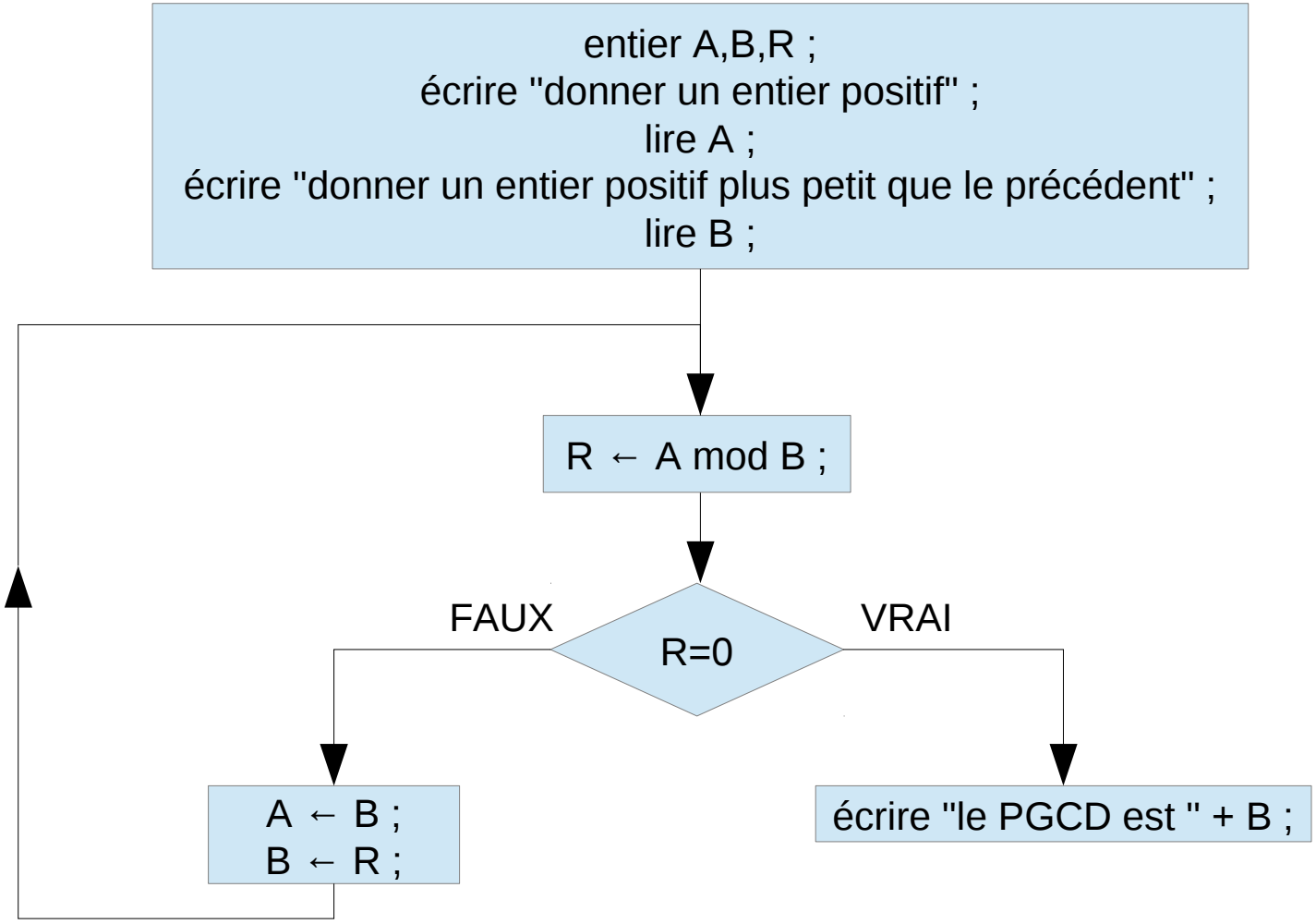
15	12
3	1

12	3
0	4

Le PGCD de 123 et 27 est donc 3.

Comment écrire un algorithme (2/3)

4- Si besoin, réaliser un organigramme pour mieux comprendre



Comment écrire un algorithme (3/3)

5- Ecrire l'algorithme

```
entier A,B,R;
début
      écrire "donner un entier positif";
      lire A;
      écrire "donner un entier positif plus petit que le précédent";
      lire B;
      R <- A mod B;
      tantque (R ≠ 0) faire
          A <- B;
          B <- R;
          R <- A mod B;
      fintantque
      écrire "le PGCD de A et B est " + B;
fin
```

6- Il peut être utile, voire nécessaire, de prouver l'algorithme

- l'algorithme va t'il toujours s'arrêter ?
- quand l'algorithme s'arrêtera, donnera t-il le bon résultat ?

7- Il peut être utile d'évaluer l'efficacité de l'algorithme

- n'existe t-il pas un algorithme plus rapide pour calculer le PGCD ?

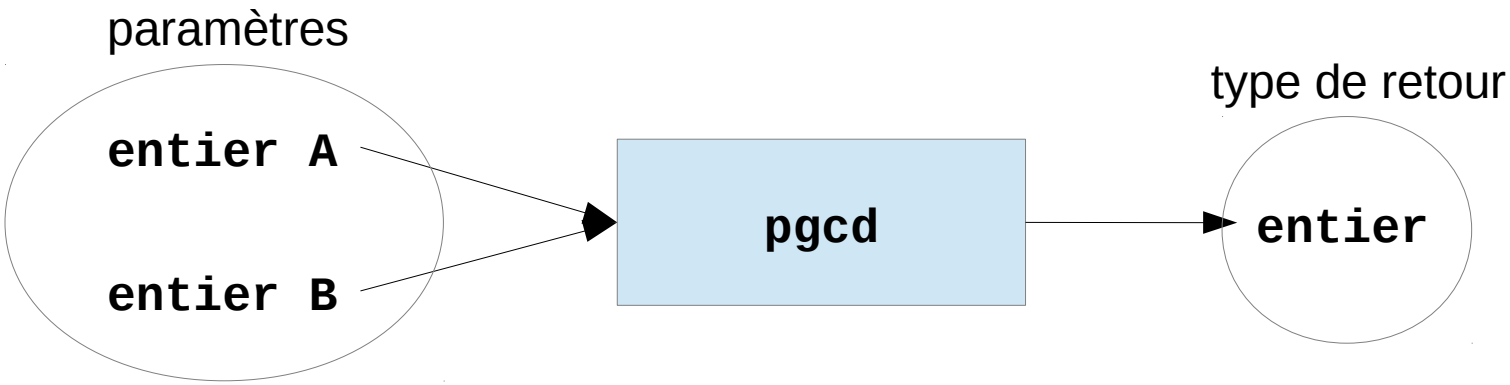
Fonction

Une fonction est un bloc d'instructions qui peut être appelé dans un autre bloc.

- la fonction a un nom (pour pouvoir être appelé)

- la fonction a des paramètres, contenant des valeurs ou des références sur des variables

- une fonction peut renvoyer une valeur au code qui l'a appelée. Le type de donnée de cette valeur est le type de retour de la fonction. Une fonction peut ne rien renvoyer (on parle alors parfois de procédure).



Déclaration de fonction

La déclaration d'une fonction comporte une signature qui indique comment utiliser la fonction en précisant son nom et ses entrées/sorties, et un corps qui est le code de la fonction.

signature

corps

```
fonction avec retour entier pgcd(entier A, entier B)
     entier R;
début
     R <- A mod B;
     tantque (R ≠ 0) faire
          A <- B;
          B <- R;
          R <- A mod B;
     fintantque
     retourne B;
fin
```

Une fonction peut ne pas avoir de retour (et donc de type de retour).

```
fonction sans retour ecritBonjour()
début
     écrire "Bonjour";
fin
```

Appel de fonction

On appelle une fonction par son nom, en lui fournissant les valeurs correspondant à ses paramètres (s'il y en a).

Si la fonction renvoie une valeur, on peut la récupérer dans une variable ou l'utiliser directement.

```
caractère c;
entier A, B;
début
     c ='O';
     tantque (c ≠ 'N') faire
          écrire "voulez vous continuer (O/N)";
          lire c;
          si (c ≠ 'N') alors
               écrire "donner un entier positif";
               lire A;
               écrire "donner un entier positif plus petit que le précédent";
               lire B;
               écrire "le PGCD de A et B est " + pgcd(A,B);
          finsi
     fintantque
fin
```

Paramètres et entrées/sorties

Attention : les paramètres d'une fonction servent à lui transmettre des valeurs. Il est donc redondant de lire au clavier des valeurs transmises par paramètres.

Il est également absurde de redéfinir les paramètres dans le corps de la fonction.

```
fonction avec retour entier pgcd(entier A, entier B)
     entier A,B,R;
début
     écrire "donner A";
     lire A;
     écrire "donner B";
     lire B;
     R <- A mod B;
     tantque (R ≠ 0) faire
          A <- B;
          B <- R;
          R <- A mod B;
     fintantque
     retourne B;
fin
```

Appel de fonction dans une fonction

```
fonction avec retour entier plusGrand2(entier a, entier b)
début
  si (a<b) alors
      retourne b;
  sinon
      retourne a;
  finsi;
fin
```

```
fonction avec retour entier plusGrand3(entier a, entier b, entier c)
début
  retourne plusGrand2(plusGrand2(a,b),c);
fin
```

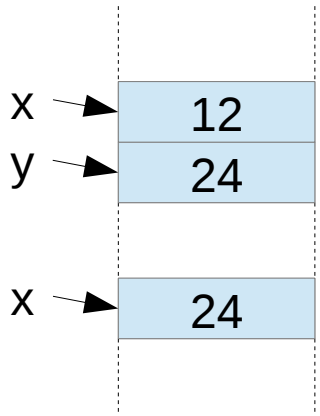
```
algorithme MonAlgo
  entier toto, titi, tutu;
début
  lire toto;
  lire titi;
  lire tutu;
  écrire "le plus grand est " + plusGrand3(toto,titi,tutu);
fin
```

Passage de paramètres (1/3)

Un paramètre peut être transmis à une fonction par valeur : la valeur est clonée et la fonction travaille sur une variable interne.

```
fonction avec retour entier multiplie2(entier x)
début
     x <- 2*x;
     retourne x;
fin
```

```
entier x,y;
x <- 12;
y <- multiplie2(x);
écrire "x vaut " + x;
```



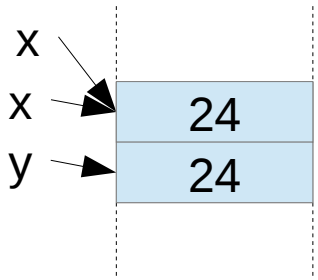
En algorithmique, on supposera que les paramètres de type primitif (entier, réel, booléen, caractère) sont toujours passés par valeur. C'est le cas en Java.

Passage de paramètres (2/3)

Un paramètre peut être transmis à une fonction par référence : la fonction travaille sur la case mémoire correspondant à la variable qui stocke la valeur.

```
fonction avec retour entier multiplie2(entier x)
début
     x <- 2*x;
     retourne x;
fin
```

```
entier x,y;
x <- 12;
y <- multiplie2(x);
écrire "x vaut " + x;
```



En algorithmique, on supposera que les paramètres de type composés (tableaux, enregistrements, etc) sont toujours passés par référence. C'est le cas en Java.

Passage de paramètres (3/3)

Les langages de programmation permettent souvent de choisir la façon dont les paramètres sont passés aux fonctions.

Exemple de passage d'un entier par référence en langage C :

```
void multiplie2(int* i){
     *i = (*i)*2;
}
```

```
int i;
i = 3;
multiplie2(&i);
```

Exemple de passage d'un entier par référence en langage ADA :

```
procedure multiplie2(i : in out Integer) is
begin
     i := i*2;
end multiplie2
```

```
i : Integer;
i = 3;
multiplie2(i);
```

Types de données

Un type de données est défini par un ensemble de valeurs et un ensemble d'opérations.

On distingue les types de données abtraits (au niveau algorithmique) et les types de données concrets (au niveau des langages de programmation).

Exemples :

*- le type abstrait entier a pour valeurs tous les entiers positifs ou négatifs, et pour opérations +, -, *, / et mod (ainsi que l'opération externe =).*

*- le type concret int en Java a pour valeurs les entiers compris entre 2^{31}-1 et -2^{31} et pour opérations +, -, *, / et mod (ainsi que l'opération externe ==).*

Les types de données servent à faciliter la programmation et surtout à tester à la compilation la correction des programmes.

Création de types de données

Certains langages permettent de créer des sous types de types existants en restreignant l'ensemble des valeurs et/ou les opérations.

Exemple en ADA :

```
subtype Age is INTEGER range 0..100;
subtype Lettre is Character range 'a'..'Z';
```

La plupart des langages permettent de créer de nouveaux types.

- en créant des ensembles de valeurs définies par le programmeur : énumérations, ensembles, unions.

- en assemblant des variables au sein de types composés : tableaux, enregistrement, listes, arbres, graphes, etc.

Types énumérés (1/4)

Un type énuméré (ou énumération) est un type dont les valeurs sont données in extenso par le programmeur.

Un type énuméré permet de définir des valeurs n'existant pas dans les types fournis par le langage.

Un type énuméré s'utilise comme n'importe quel type pour typer des variables ou des paramètres.

Exemple : type énuméré en Java (ou C ou C++ ou C#)

```
enum ArcEnCiel {rouge, orange, jaune, vert, bleu, indigo, magenta, violet};
ArcEnciel aec =  ArcEnCiel.jaune;

enum Primaire {rouge, vert, bleu};
Primaire p = Primaire.rouge;
```

Remarque : en Java, une valeur d'un type énuméré doit être préfixée par le nom du type, pour éviter toute ambiguité. En algorithmique, ce n'est pas nécessaire.

Types énumérés (2/4)

Certains langages (dont Java et C) permettent de donner aux types énumérés des valeurs d'autres types stockées dans des variables.

Exemple en Java :

```
int vrai = 1;
int faux = 0;
String possible = "possible";
enum ValeurDeVerite {vrai, faux, possible};
```

Tous les langages permettant les types énumérés offrent des opérations d'égalité et d'inégalité sur ces types.

Exemple en Java :

```
ValeurDeVerite vv1, vv2;
...
if ((vv1 == vv2) && (vv1 != ValeurDeVerite.faux))
  System.out.println("Tout est possible");
```

Types énumérés (3/4)

Il existe souvent une relation d'ordre sur les valeurs énumérées.

Exemple : en Java v.ordinal() renvoie le numéro d'ordre de v dans le type.

```
enum ArcEnCiel {rouge, orange, jaune, vert, bleu, indigo, magenta, violet};
ArcEnciel aec1, aec2;
aec1 = ArcEnCiel.rouge; // aec1.ordinal() vaut 0
aec2 = ArcEnCiel.bleu; // aec2.ordinal() vaut 4
if (aec1.ordinal() > aec2.ordinal()) {...}
```

Il peut exister des opérations de parcours des valeurs d'un type énuméré.

Exemple en Java :

```
for (ArcEnCiel aec : ArcEnCiel.values()) {
     System.out.println(aec.name());
}
```

Types énumérés (4/4)

Le programmeur peut toujours ajouter des opérations sur un type énuméré qu'il a créé, en écrivant des fonctions.

```
fonction avec retour ValeurDeVerite et(ValeurDeVerite vv1, ValeurDeVerite vv2)
début
  si (vv1 = ValeurDeVerite.vrai) alors
      si (vv2 = ValeurDeVerite.vrai) alors retourne ValeurDeVerite.vrai;
      sinon
          si (vv2 = ValeurDeVerite.possible) alors
              retourne ValeurDeVerite.possible;
          sinon
              retourne ValeurDeVerite.faux;
          finsi
      finsi
  finsi
  si (vv1 = ValeurDeVerite.faux) alors retourne ValeurDeVerite.faux;
  finsi
  si (vv1 = ValeurDeVerite.possible) alors
      si ((vv2 = ValeurDeVerite.vrai) ou (vv1 = ValeurDeVerite.possible))
          retourne ValeurDeVerite.possible;
      sinon
          retourne ValeurDeVerite.faux;
      finsi
  finsi
fin
```

Types énumérés en algorithmique

En algorithmique, pour les types énumérés, on adoptera la syntaxe la plus courante qui est celle du langage C (syntaxe reprise en C++ et en Java, entre autres).

Les types énumérés seront déclarés dans la partie déclaration, avec les déclarations de variables.

```
algorithme MonAlgorithme
  entier vrai, faux;
  chaine possible;
  enum ValeurDeVerite {vrai, faux, possible};
  ValeurDeVerite vv1,vv2;
début
  vrai <- 1;
  faux <- 0;
  possible <- "possible";
  vv1 <- ValeurDeVerite.possible;
  vv2 <- ValeurDeVerite.faux;
fin
```

Comme en Java, on considère que les valeurs des types énumérés sont passés aux fonctions par référence.

Types composés

Les données sont souvent complexes et les types primitifs (entier, booléen, ...) ne suffisent pas à les représenter de façon efficace.

Exemple : on veut représenter dans un programme les notes de 100 étudiants inscrits dans 5 modules : il faut 500 variables.

```
réel noteModule1Etudiant1, noteModule1Etudiant2, noteModule1Etudiant3, ...
noteModule2Etudiant1, ...
...
... , noteModule5Etudiant100;
```

Un type de données composé (ou structuré) est un type dont les valeurs sont composées de plusieurs valeurs.

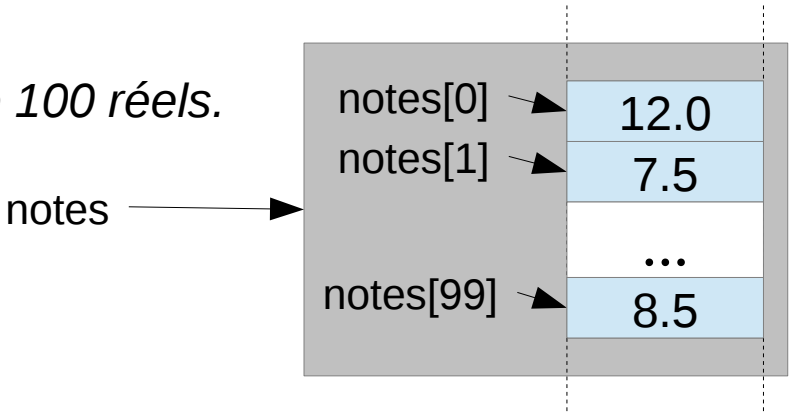
noteModule1Etudiant1	noteModule1Etudiant2	...	noteModule1Etudiant99	noteModule1Etudiant100
		...		
noteModule5Etudiant1	noteModule5Etudiant2	...	noteModule5Etudiant99	noteModule5Etudiant100

Tableaux

Un type tableau est un type composé dont chaque valeur est composée d'autres valeurs (les “cases” du tableau), toutes du même type, et indicées par un type discret ordonné.

Généralement, les indices sont des entiers, allant de 0 à la longueur du tableau-1.

Exemple : tableau de 100 réels.




```
réel[100] notes;
notes[0] <- 12.0;
notes[1] <- 7.5;
...
notes[99] <- 8.5;
```

Remarque : un type tableau est un type, un tableau est une valeur du type

Exemple : - “tableau de caractères” est un type

- le tableau de caractères ['t','o','t','o'] est une valeur de type tableau de caractères

Déclaration d'un tableau

Comme toute variable, une variable de type tableau possède un nom et un type (celui de ses éléments). Un tableau possède également un nombre d'éléments.

Exemple : déclaration de tableaux dans une syntaxe C ou C++

```
// déclaration d'un tableau de chaines de 12 cases
char*[12] monTableauDeChaines;

// déclaration d'un tableau d'entiers de 5 cases
int monTableauDEntiers[5];
```

En algorithmique, on utilisera la même syntaxe (deux formes possibles) :

```
<type des éléments>[<nombre d'éléments>] <nom de la variable>;
```

```
<type des éléments> <nom de la variable>[<nombre d'éléments>];
```

Remarque : déclarer le tableau entraine la déclaration de toutes les "variables" contenues dans le tableau mais pas leur initialisation!

Accès aux cases d'un tableau

On accède à une case d'un tableau en précisant son indice entre crochets.

```
chaine[3] monTableauDeChaines;

monTableauDeChaines[0] <- "toto";
monTableauDeChaines[1] <- "titi";
montableauDeChaines[2] <- "tutu";

écrire monTableauDeChaines[1];
monTableauDeChaines[0] <- monTableauDeChaines[3];
```

Remarques :

- *une tentative d'accès à une case qui n'existe pas fera planter le programme!*
- *accéder à une case non initialisée peut conduire à des erreurs d'exécution.*

Initialisation d'un tableau (1/2)

Exemple : initialisation à 1.0 des cases d'un tableau de 38 réels

```
réel[38] monTableauDeReels;
entier i;
début
     pour (i allant de 0 à 37 pas 1) faire
          monTableauDeReels[i] <- 1.0;
     finpour
fin
```

Exemple : initialisation d'un tableau de 45 caractères à 'a' pour les cases d'indice pair et à 'b' pour les cases d'indice impair.

```
caractère[45] t;
entier i;
début
     pour (i allant de 0 à 44 pas 1) faire
          si(i mod 2 = 0) alors
               t[i] <- 'a';
          sinon
               t[i] <- 'b';
          finsi
     finpour
fin
```

Initialisation d'un tableau (2/2)

Beaucoup de langages proposent une syntaxe pour initialiser les valeurs d'un tableau par énumération.

- initialisation et déclaration sont alors regroupées dans la même instruction
- on ne précise pas la taille du tableau dans la partie déclaration

Cette méthode n'est utile que pour de petits tableaux.

Exemples :

```
// déclaration et remplissage d'un tableau de réels
réel[] monTableauDeReels <- {3.2,4.0,3.14,2.78,10.6};

// déclaration et remplissage d'un tableau de chaines
chaine monTableauDeChaines[] <- {"toto","titi","tutu","tete"};
```

Longueur d'un tableau

L'opérateur longueur (**`length`** en Java) permet de connaitre la taille d'un tableau.

```
chaine[12] monTableauDeChaines;
booléen[monTableauDeChaines.longueur*2] tableauDeBooleens;
début
  ...
  écrire "le tableau a " + monTableauDeChaines.longueur + " éléments";
fin
```

Remarques :

- la taille d'un tableau ne peut pas varier

- il faut savoir, avant de créer un tableau, de combien de cases on a besoin

- il ne faut pas déclarer plus de cases que ce dont on a besoin, pour éviter d'occuper de la place en mémoire pour rien

- si on ne sait vraiment pas de combien de cases on aura besoin, il existe des structures linéaires dynamiques (listes) dont la taille peut varier (cf. cours de L2)

Déclaration de types tableau

Certains langages permettent la déclaration des types tableau (et non uniquement la déclaration des tableaux).

```
// déclaration d'un type tableau de 5 entiers en ADA
type MonTypeTableau is array(0 .. 5) of Integer;

// utilisation du type tableau pour déclarer un tableau
MonTypeTableau monTableau;
```

```
// déclaration d'un type tableau de 5 entiers en Java
class MonTypeTableau{
     int[] t = new int[5];
}

// utilisation du type tableau pour déclarer un tableau
MonTypeTableau monTableau = new MonTypeTableau();
MonTableau.t[0] <- 2;
...
```

En pratique, il est souvent sans intérêt de créer des types tableaux.

Tableau paramètre

Rappel : les tableaux passés en paramètres de fonctions le sont par référence.

```
fonction sans retour multiplie2Tab(entier[] t)
     entier i;
début
     pour (i allant de 0 à t.longueur-1 pas de 1) faire
          t[i] <- t[i]*2;
     finpour
fin
```

```
entier[] tab <- {1,2,3};
multiplie2Tab(tab);
écrire tab[0];
```

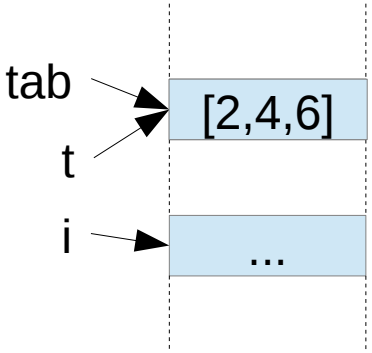


Tableau dynamique (1/2)

Problèmes posés lorsqu'on fixe le nombre d'éléments d'un tableau à la déclaration :

- on est obligé de savoir à l'avance combien il y aura de cases dans les tableaux retournés par les fonctions

```
// fonction retournant le tableau des noms des étudiants d'un groupe
chaine[12] fonction nomsEtudiants(int numeroGroupe) ...

chaine[35] etudiantsDuGroupe2 = nomsEtudiants(2);
```

- on est obligé de connaitre à l'avance le nombre de cases d'un tableau passé en paramètre d'une fonction

```
// fonction retournant la moyenne des réels du tableau paramètre
réel fonction calculMoyenne(réel[56] notes) ...

réel[35] notesGroupe3 = ...
réel m <- calculMoyenne(notesGroupe3);
```

Tableau dynamique (2/2)

Un tableau est dit statique si sa taille est fixée à la déclaration et dynamique si sa taille est fixée plus tard.

```
chaine[27] monTableauStatique
chaine[] monTableauDynamique
...
redim monTableauDynamique[27]; // redimensionnement
```

Les tableaux dynamiques permettent aux fonctions de travailler sur des tableaux de taille arbitraire.

```
chaine[] fonction nomsEtudiants(int numeroGroupe) ...

chaine[] etudiantsDuGroupe2 = nomsEtudiants(2);

reel fonction calculMoyenne(reel[] notes) ...
```

Déclaration de tableau en Java

En Java, les tableaux sont toujours dynamiques :

```
// déclaration d'un tableau de chaines en Java
String[] monTableauDeChaines;
// redimemsionnement du tableau
monTableauDeChaines = new String[12];
// déclaration et redimemsionnement sur la même ligne
String[] monAutreTableau = new String[12];
// déclaration et redimemsionnement d'un tableau d'entiers
int monTableauDEntiers[] = new int[23];

double tab[12]; // erreur à la compilation
```

Remarque : dans la partie déclaration, les crochets peuvent être placés avant ou après le nom du tableau, mais attention en cas de déclaration multiple.

```
int[] t1,t2; // t1 et t2 sont des tableaux d'entiers
int t1,t2[]; // t1 est un entier et t2 un tableau d'entiers
int t1[],t2; // t1 est un tableau d'entiers et t2 un entier
```

Tableaux dynamiques en algorithmique

Rappel : déclaration d'un tableau statique.

```
<type des éléments>[<nombre d'éléments>] <nom de la variable>;
```

```
<type des éléments> <nom de la variable>[<nombre d'éléments>];
```

Déclaration et redimensionnement d'un tableau dynamique : deux syntaxes possibles pour la déclaration, une seule pour le redimensionnement.

```
<type des éléments>[] <nom de la variable>;
```

```
<type des éléments> <nom de la variable>[];
```

```
redim <nom de la variable>[<nombre d'éléments>];
```

Tableau vide (1/2)

Pour des raisons pratiques, il peut être nécessaire de traiter des tableaux vides.

Exemple : fonction qui renvoie le tableau des entiers pairs contenus dans un tableau d'entiers passé en paramètre.

```
fonction avec retour entier[] pairs(entier[] t)
     entier[] resultat;
     entier i,nb;
début
     nb <- 0;
     pour (i allant de 0 à t.longueur-1 pas 1) faire
          si (t[i] mod 2 = 0) alors
               nb <- nb+1;
          finsi
     finpour
     redim resultat[nb];
     nb <- 0;
     pour (i allant de 0 à t.longueur-1 pas 1) faire
          si (t[i] mod 2 = 0) alors
               resultat[nb] <- t[i];
               nb <- nb+1;
          finsi
     finpour
     retourne resultat;
fin
```

Tableau vide (2/2)

En algorithmique, on s'autorise à créer des tableaux vides de toutes les manières imaginables. On peut tester si un tableau est vide en testant sa longueur.

```
entier[] titi; // titi est vide
redim titi[0]; // titi est toujours vide
entier[0] tutu; // tutu est vide
entier[] tata <- {}; // tata est vide

entier[] toto <- {3,5,75,231};
entier[] t <- pairs(toto);
si(t.longueur = 0) alors
     ...
```

En Java, un tableau qui n'a pas été dimensionné sera égal à **`null`**.

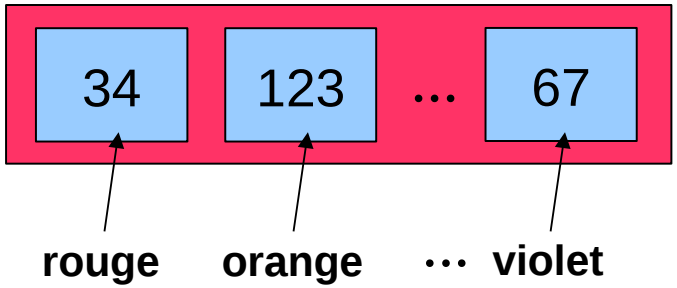
```
int[] titi; // le compilateur indiquera une erreur d'initialisation
int[] tete = new int[0]; // tete est vide
int[0] tutu; // interdit en Java, les tableaux sont forcément dynamiques
int[] tata <- {}; // tata est vide

int[] toto <- {3,5,75,231};
int[] t <- pairs(toto);
if(t != null && t.length != 0){
     ...
```

Tableaux associatifs

Il est possible d'indicer les tableaux par n'importe quel type énuméré ordonné, pas seulement pas des entiers de 0 à la longueur du tableau – 1.

Exemples : tableau d'entiers indicé par les valeurs du type ArcEnCiel en Java




```
enum ArcEnCiel {rouge, orange, jaune, vert, bleu, indigo, magenta, violet};
HashTable<ArcEnCiel,Integer> t = new HashTable<ArcEnCiel,Integer>();
t.put(ArcEnCiel.rouge,34);
t.put(ArcEnCiel.orange,123);
...
t.put(ArcEnciel.violet,67);
...
t.get(ArcEnCiel.orange); // renvoie 123
```

Enregistrement (1/2)

Exemple : on veut représenter dans un programme les données décrivant 50 personnes avec pour chacune un nom, un prénom, un âge et une taille.

```
chaîne nom1, prénom1;
entier age1;
réel taille1;
...
chaîne nom50, prénom50;
entier âge50;
réel taille50;
...
nom1 <- "Duchmol";
prenom1 <- "Robert";
age1 <- 24;
taille1 <- 1.80;
```

Un enregistrement est un type composé qui permet de regrouper des valeurs de types différents.

Enregistrement (2/2)

Un enregistrement a un nom et un certains nombre de champs, qui sont des variables typées.

L'accès à un champ se fait en préfixant l'identifiant du champ par le nom de la variable qui contient l'enregistrement.

```
algorithme monAlgorithme
     // déclaration d'un enregistrement
     enregistrement Personne
          chaine nom;
          chaine prenom;
          entier age;
          réel taille;
     finenregistrement
     ...
     Personne p1, p2;
début
     // Initialisation d'un enregistrement
     p1.nom <- "Duchmol";
     p1.prenom <- "Robert";
     p1.age <- 24;
     p1.taille <- 1.80;
     ...
fin
```

Remarque : un enregistrement est un type, mais les valeurs du type sont aussi appelées enregistrement (comme pour les types primitifs).

Enregistrement en Java

```
public class MonProgramme{

     static class Personne{
          String nom;
          String prenom;
          int age;
          float taille;
     }

     public static void main(String arg[]){
          Personne p1 = new Personne();
          Personne p2 = new Personne();
          p1.nom = "Duchmol";
          p1.prenom = "Robert";
          p1.age = 24;
          p1.taille = 1.80;
          ...
     }

}
```

Rappel : les enregistrements passés en paramètre de fonction le sont par référence (en Java et en algorithmique).