



Global Knowledge®

# Algorithmique

## Les fondamentaux de la programmation

# PRESENTATION

➤ Qui suis-je ?



# But de la formation Algorithmique

*Etre capable de formuler de façon **structurée et indépendante** de toute contrainte matérielle ou logicielle, les différentes étapes conduisant à la **résolution d'un problème**.*

- *savoir analyser par étapes successives,*
- *définir des modules,*
- *être capable de construire des jeux d'essais*
- *Comprendre la gestion des données.*

# PRESENTATION

➤ **Qui êtes-vous?      Qu'attendez-vous de la formation ?**



# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS – PERSISTANCE**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**



# INTRODUCTION



# ALGORITHME

- Description d'une action complexe, décomposée en une suite d'actions qui le sont moins.
- Terme non purement informatique - Peut s'appliquer à n'importe quelle classe de « problème », dans tous les domaines d'activité.
- Exemple : Mode d'emploi d'un répondeur téléphonique
  - Algorithme pour interroger à distance,
  - Algorithme pour enregistrer l'annonce...
- Pour **un** problème déterminé, il existe **plusieurs** algorithmes possibles.
- Un algorithme peut être exprimé oralement, par un texte écrit, ou à l'aide d'un schéma.

# ALGORITHMIQUE

- L'algorithmique poursuit trois objectifs principaux :
  - Représentation compréhensible par le plus grand nombre.
  - Représentation aisément traduisible dans n'importe quel langage de programmation.
  - Représentation structurée.
- Ce dernier point est le plus important.



# La programmation structurée

- **La programmation structurée** n'est pas un phénomène de mode :
  - les actions de maintenance de plus en plus nombreuses (les applications évoluent) ont prouvé qu'un programme coûte plus cher en évolutions qu'en développement initial.
- Or bien souvent (c'est le cas de toutes les anciennes applications), les maintenances s'avèrent très difficiles - voire impossibles - à effectuer car les programmes ont été écrits sans aucune méthodologie.

# METHODOLOGIE

- **La méthodologie** est l'ensemble des actions à entreprendre pour parvenir à un but. Elle répond aux questions :
  - **QUE FAUT-IL FAIRE,**
  - **QUI LE FAIT**
  - **QUAND ?**
- **La méthode** est un ensemble de démarches raisonnées, suivies pour parvenir à un but. Elle répond à la question :
  - **COMMENT FAUT-IL LE FAIRE ?**

# La programmation structurée

- Le but de la ***programmation structurée*** est de :  
clarifier le texte d'un programme (le « source »),  
ceci pour trois raisons principales :
  - **Gagner du temps à l'écriture du programme**
  - **Améliorer la testabilité des programmes**
  - **Faciliter les maintenances futures**

# La programmation structurée :

## Gagner du temps

- Une erreur commune est de considérer l'écriture d'un algorithme sur papier comme une perte de temps.
- En réalité, on s'aperçoit vite que cette remarque n'est justifiée que pour des programmes très simples, ne demandant pas d'effort particulier de réflexion.
- Dans tous les autres cas, un **code structuré** permet au programmeur de toujours **maîtriser sa production**, même lorsque celle-ci devient complexe.

# La programmation structurée : Améliorer la testabilité

- Tester un programme équivaut à l'exécuter plusieurs fois en changeant les paramètres en entrée pour qu'au moins chaque combinaison d'instructions soit exécutée une fois.
- Si le programme utilise un grand nombre d'instructions alternatives par exemple, les tests peuvent s'avérer longs et fastidieux.
- Afin de faciliter cette étape primordiale, il est impératif que le source ait été écrit de manière logique pour pouvoir dresser une liste exhaustive des groupes d'instructions à tester.

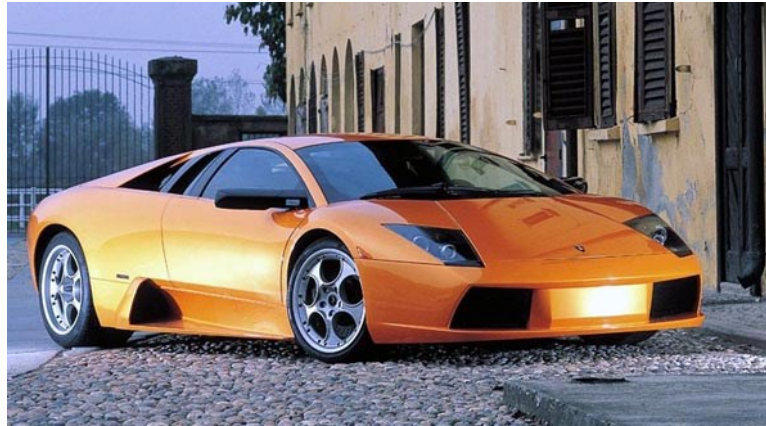


# La programmation structurée : Faciliter les maintenances

- Les besoins du service utilisant l'application dans laquelle vient s'insérer ce programme peuvent évoluer.
- Par ailleurs, un programme n'est jamais parfait et présente toujours un pourcentage plus ou moins important d'erreurs en fonction de la qualité des tests qui ont été menés.
- C'est pourquoi, un source fera toujours l'objet de maintenances (correctives ou évolutives).
  - Le plus souvent, celles-ci sont effectuées par un développeur ne connaissant pas le programme à modifier.
- Afin de faciliter le travail de l'équipe de maintenance, **le code doit avoir été écrit de façon structurée** pour être facilement compréhensible par tout autre développeur.

# DÉMARCHES DE CONCEPTION

- Objectif d'une démarche de conception
  - A pour but de réaliser un produit qui répond à un besoin exprimé par un utilisateur

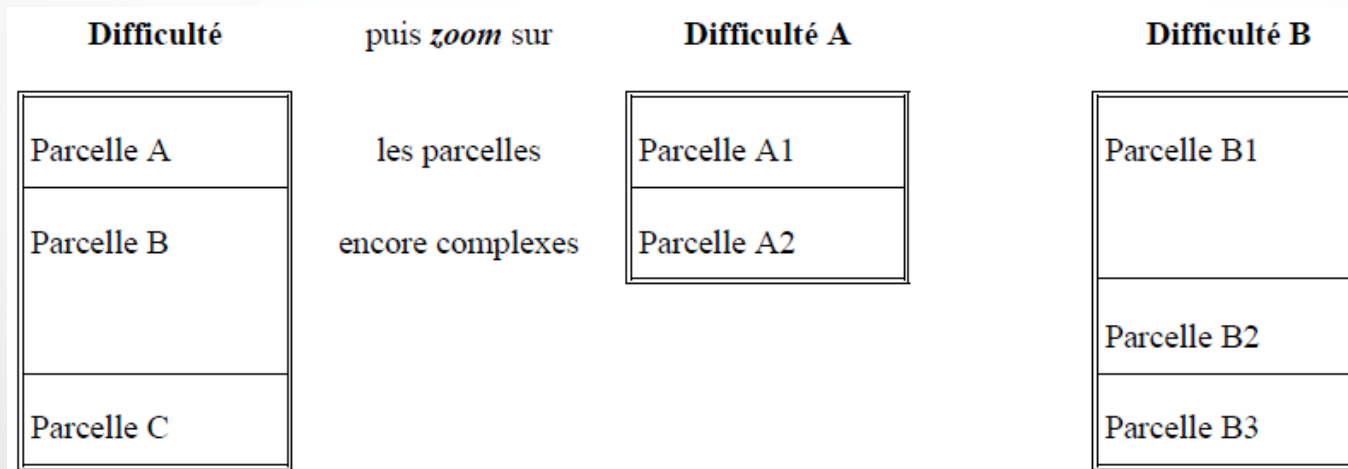


- Une voiture permet de déplacer des personnes et leurs biens d'un point à un autre en toute sécurité et protégés des intempéries

# DÉMARCHES DE CONCEPTION :

## Analyse descendante – Top-down

- L'analyse « descendante » ou « Top-down » est une application de la deuxième règle du Discours de la Méthode de Descartes :
  - « ***Diviser chacune des difficultés examinées en autant de parcelles qu'il est requis pour les mieux résoudre*** »

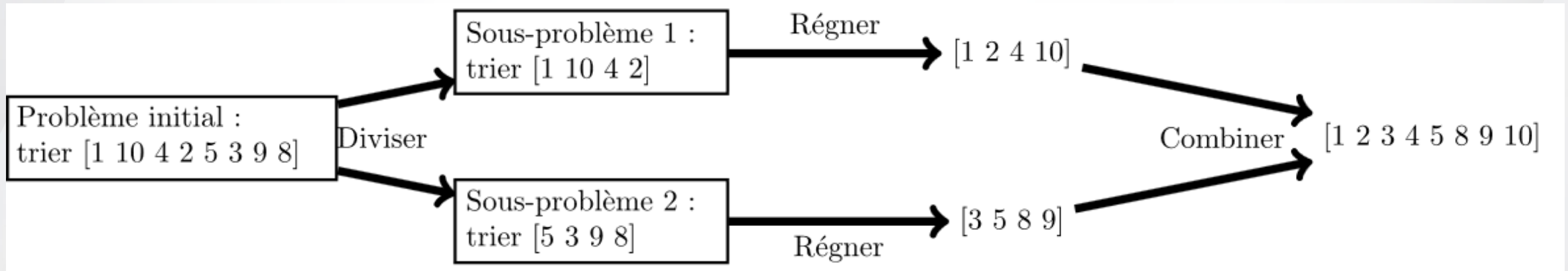


parcelles = modules

# DÉMARCHES DE CONCEPTION :

## Analyse descendante – Top-down

- L'analyse d'une tâche complexe consiste à la scinder en sous-problèmes dont la résolution est reportée à plus tard. ➔ Avoir 1 **Vue d'ensemble** du problème, pour «descendre» peu à peu **vers le détail** des éléments



- A chaque étape, il s'agit de **délimiter précisément le rôle qu'on attribue à chaque module**, sans se préoccuper de la façon dont on réalisera ces différentes fonctions

# DÉMARCHES DE CONCEPTION :

## Analyse ascendante - Bottom-up

Principe différent du top-down - Principe de construction, de lego

- on part du niveau de détail pour aboutir par regroupements successifs jusqu'à obtenir des fonctions suffisamment évoluées pour remplir les objectifs fixés au départ. »
- Inconvénient : ne permet pas d'avoir 1 vue d'ensemble - niveau du dév,
  - Au fur et à mesure de la progression on comprend la complexité du problème,
  - Ne permet pas de maîtriser les coûts et les délais de développement du système. Tant que l'on n'a pas pratiquement terminé
  - On ne sait pas vraiment quand on aura fini.

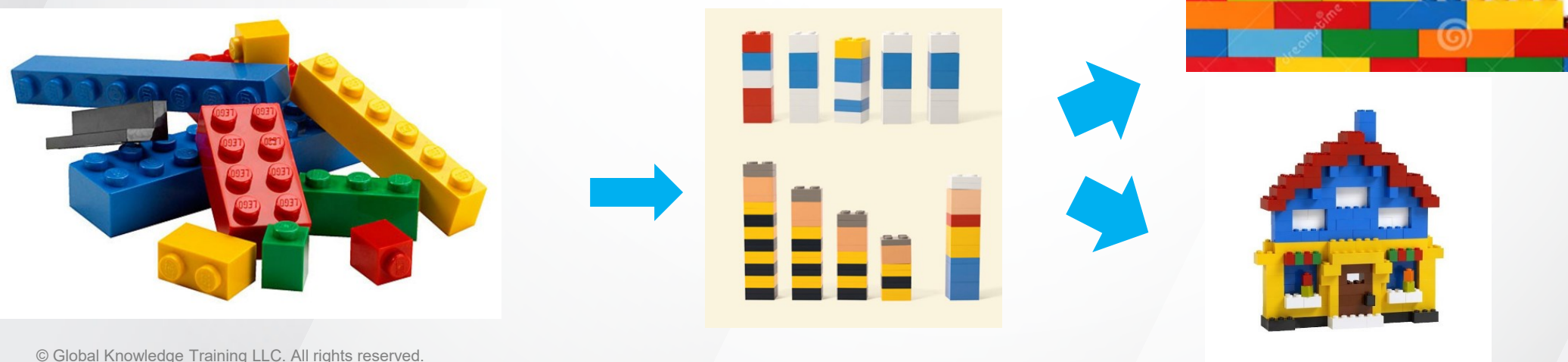




# DÉMARCHES DE CONCEPTION :

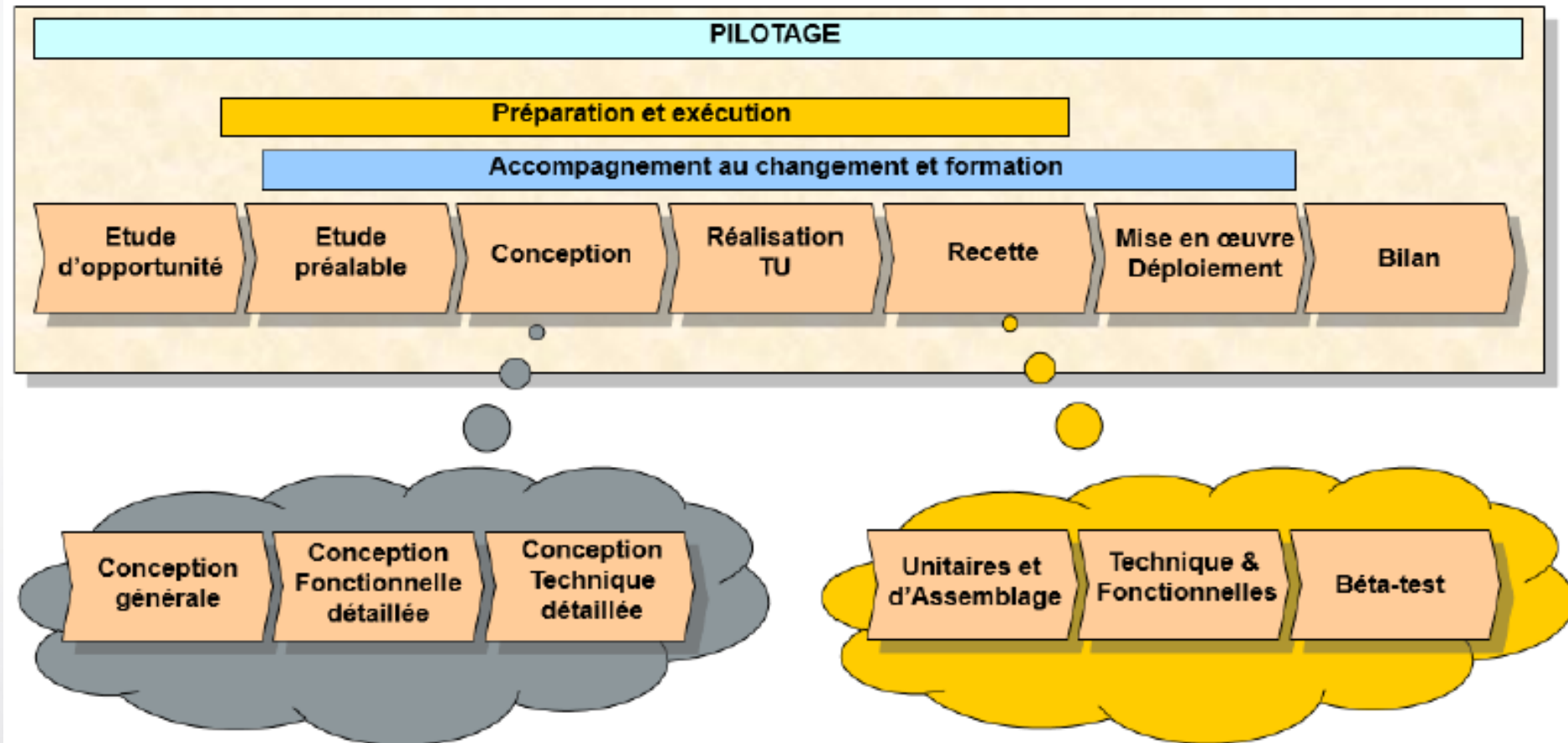
## Analyse ascendante - Bottom-up

- Cette méthode dite « Bottom-up » est utilisée en Conception, avant la phase de réalisation où se situe l'algorithme.
- C'est ainsi que l'on définit des modules réutilisables
- Cette méthode permet de guetter les éléments susceptibles d'être généralisés, tout en préservant l'autonomie des différents modules (en limitant le nombre de relations entre eux).



# PLACE DE L'ALGORITHMIQUE DANS LE PROJET

## ➤ Les différentes phases d'un projet informatique



# PLACE DE L'ALGORITHMIQUE DANS LE PROJET : Réalisation du système

- Cette partie recouvre la construction du produit final respectant les fonctionnalités désirées par l'utilisateur et l'architecture technique mise en place.
- La phase de réalisation recouvre les activités de :
  - Codage des programmes applicatifs, à partir **des algorithmes**,
  - Production de la documentation de programme, dans le but de faciliter les maintenances ultérieures,
  - Production des directives d'exploitation du système (politique de sauvegarde et de restauration, par exemple),
  - Production de guides utilisateurs et d'aides en ligne le cas échéant, voire des dispositifs de formation.

# PLACE DE L'ALGORITHMIQUE DANS LE PROJET : Les tests

- Plusieurs activités de test sont nécessaires pour garantir l'adéquation du système aux besoins exprimés :
  - Les tests unitaires
  - Les tests d'intégration
  - Les tests systèmes
  - Les tests opérationnels
- Le terme de **recette** désigne l'ensemble des tests que pratiquent les futurs utilisateurs d'un système lors de la livraison de l'application par les informaticiens. Ce terme implique un cadre contractuel

# PLACE DE L'ALGORITHMIQUE DANS LE PROJET : Les tests

- **Tests unitaires**, au cours desquels les différents programmes sont testés indépendamment les uns des autres.
  - Tests menés à partir de jeux d'essais, c'est-à-dire de cas d'utilisation conçus spécifiquement dans un but de validation,
- **Tests d'intégration**, tests de fonctionnalités qui enchaînent les différents programmes d'une même unité fonctionnelle, puis les unités fonctionnelles entre elles, menées, elles-aussi, sur la base de jeux d'essai,
- **Tests systèmes**, dont l'objectif est de vérifier la capacité du système à supporter la future application,
- **Tests opérationnels**, qui sont menés à partir de données et de contextes de production.



# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**

# CONSTITUANTS D'UN ALGORITHME



# PRESENTATION

- Un algorithme est la décomposition d'une action complexe qui, partant de données toutes définies, permet d'obtenir un (des) résultat(s) déterminé(s). Tout algorithme sera donc composé :
  - de structures d'enchaînements,
  - d'actions
  - des données

# L'ENCHAINEMENT

➤ Il existe trois familles d'enchaînement d'actions :

- Séquentiel
- Sélectif
- Répétitif

➤ Exemple : résolution d'une équation du 2<sup>nd</sup> degré

$$ax^2 + bx + c = 0$$

1. Connaître les valeurs de  $a$ ,  $b$  et  $c$  sachant  $a$  non nul
2. Calculer le discriminant  $D = b^2 - 4ac$
3. Si  $D < 0$  alors pas de solution
4. Si  $D = 0$  alors 1 solution  $[x = -b/2a]$
5. Si  $D > 0$  alors deux solutions  $[x_1 = (-b - \sqrt{\Delta})/2a \text{ et } x_2 = (-b + \sqrt{\Delta})/2a]$

# ENCHAINEMENT : Séquentiel

➤ Les actions sont effectuées l'une après l'autre, du début à la fin du **bloc logique** qu'elles forment (l'action plus complexe) :

- Début
  - Exécuter telle action
  - Exécuter telle autre action
- Fin du bloc

Par exemple : Appeler quelqu'un au téléphone :

- Début
    1. Décrocher le combine
    2. Attendre la tonalité
    3. Composer le numéro .../...
  - Fin
- (1 et 2) Prendre son Smartphone/choisir Téléphone





# ENCHAINEMENT : Sélectif

- L'alternative permet de choisir d'exécuter un bloc d'actions plutôt qu'un autre et peut s'exprimer sous la forme

SI la condition est vérifiée

Exécuter tel bloc d'actions

SINON

Exécuter tel autre bloc

Par exemple, Multiplier 2 nombres :

SI l'opération paraît simple

La faire de tête

SINON

La poser par écrit

- Un bloc peut tout à fait ne comporter aucune action

Par exemple, en faisant des courses :

SI le produit convient

Le prendre du rayon

SINON

RIEN

# Un exercice, à vous de jouer !

action N

SI truc > 4

    action P

        SI chose < 3

            action C

            action D

        SINON action V

        action W

    SINON action I

action T

Soit l'algorithme (à gauche)

Indiquez les actions effectuées quand

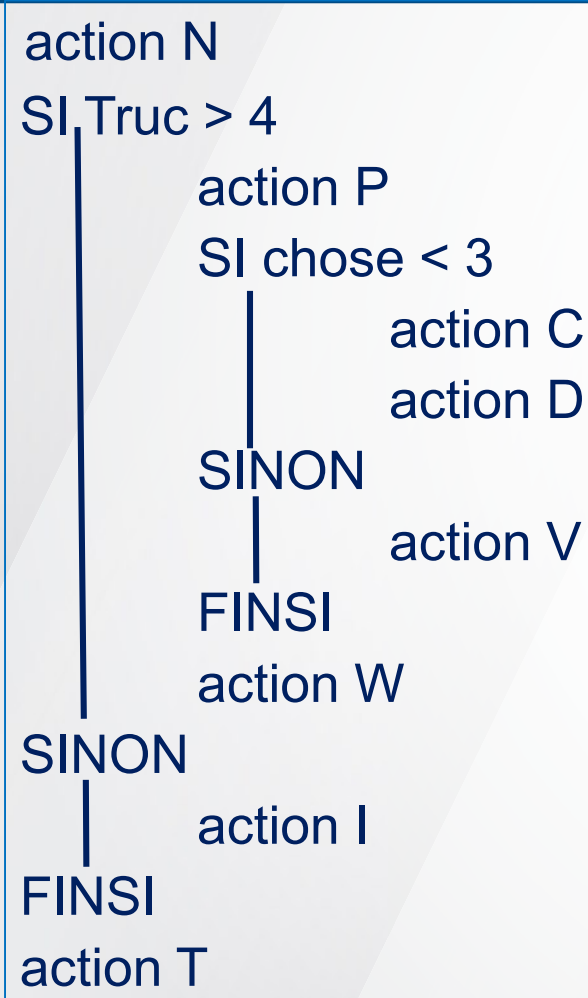
        truc est > 4    et quand truc <= 4

et quand

        chose est < 3 et chose >= 3



# Un exercice, à vous de jouer



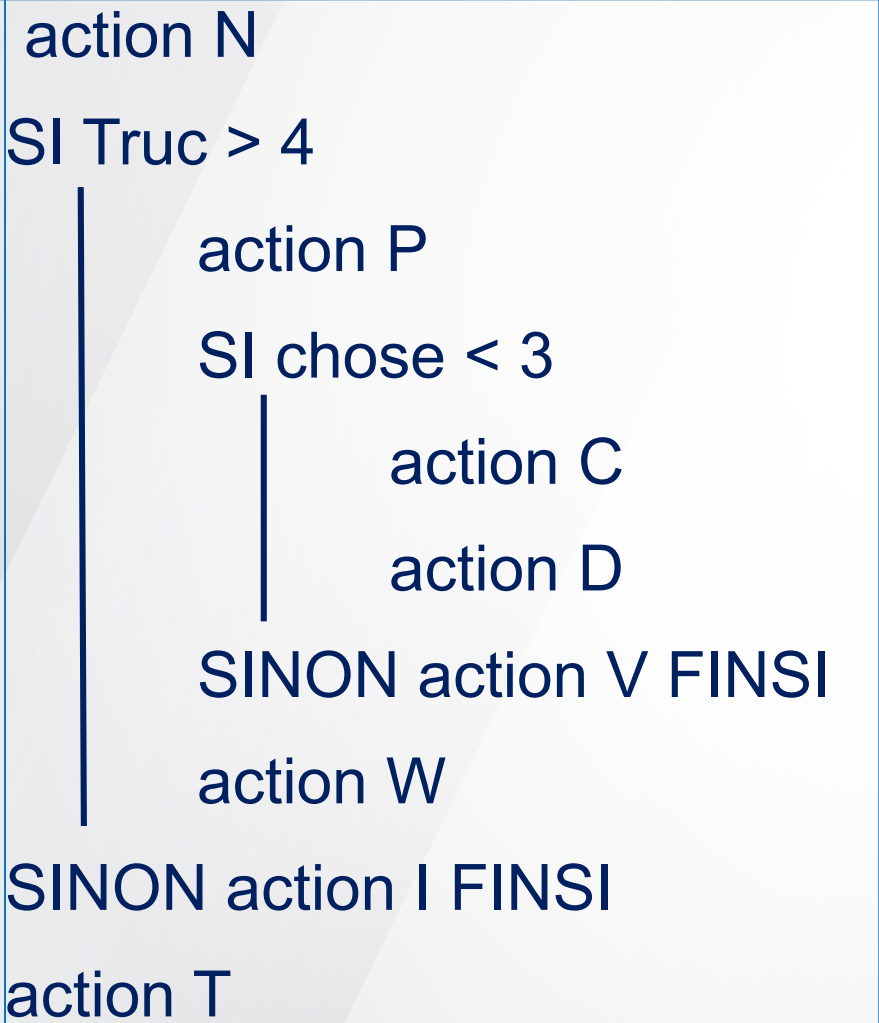
Algorithme plus lisible avec des FIN/SI

L'indentation est utile

Résultats à obtenir

	Chose < 3	Chose >= 3
Truc > 4		
Truc <= 4		

# Corrigé



Algorithme plus lisible avec des FIN/SI

L'indentation est utile

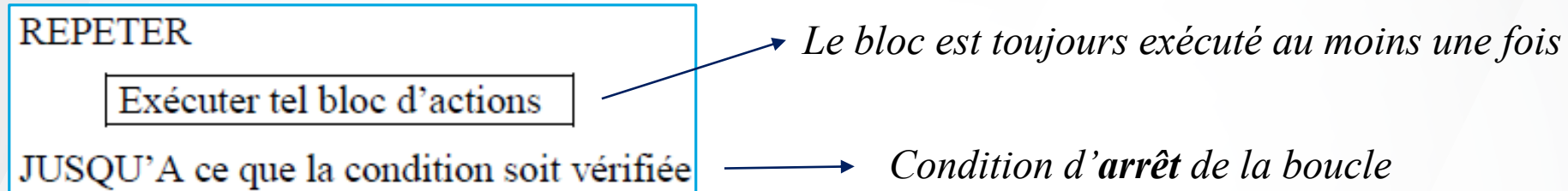
Résultats à obtenir

	Chose < 3	Chose >= 3
Truc > 4	N P C D W T	N P V W T
Truc <= 4	N I T	N I T

# ENCHAINEMENT : Répétitif

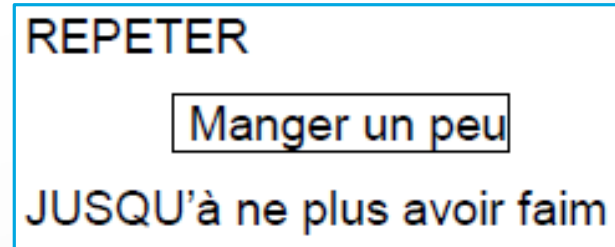
- Les « boucles » servent à attendre un état prédéterminé, en répétant un bloc d'actions 0, 1, ou plusieurs fois, selon l'expression choisie :
- Les deux boucles les plus courantes sont :
  - REPETER action JUSQU'A condition
  - TANT QUE condition ok Faire Action

# ENCHAINEMENT : Répétitif



- **Attention** : avec un REPETER, le test sur la condition d'arrêt se fait **après** l'exécution du bloc d'action.
- Cette boucle s'effectue donc de **1 à plusieurs fois**.

- Exemple, prendre un repas





# ENCHAINEMENT : Répétitif

TANT QUE la condition est vérifiée

→ Condition de *continuation* de la boucle

Exécuter tel bloc d'actions

→ Le bloc peut ne jamais être exécuté

- **Attention** : avec un TANT QUE, le test sur la condition d'arrêt se fait **avant** l'exécution du bloc d'action.
- Cette boucle s'effectue donc de **0 à plusieurs fois**

- Exemple, prendre un repas

TANT QUE j'ai faim

Manger un peu

# ENCHAINEMENT - L'imbrication de structures

- Attention : L'imbrication de structures d'enchaînement est à utiliser avec discernement et modération :
- Trop de niveaux (plus de 3) entraînent de très sérieux problèmes de compréhension.

# LES DONNEES

- Une donnée est une **information nécessaire** au processus décrit. Toute donnée se définit par **trois caractéristiques** :
  - Type
  - Valeur
  - Identificateur

# LES DONNEES : Type

➤ Le type d'une donnée désigne :

- L'ensemble des valeurs qu'elle peut prendre.
  - Exemples de types : entier, âge, heure, carte de tarot, booléen...
- Les **opérations** possibles avec elle.
  - Exemples :
    - nom + prénom (mettre à la suite)
    - prix + 20.6 % (ajouter)
    - habiter Paris ET avoir moins de 25 ans → Condition

1 condition est 1 expression booléenne

# LES DONNEES : Valeur

- Celle-ci peut être **constante** (3.1416, 20.6...)
- ou **varier** pendant l'action – à la manière et dans les limites de son type. Grace aux instructions d'affectation

Syntaxe:

1. `nom_variable ← valeur`

Ex: `Note2 ← 15`

2. `nom_variable1 ← nom_variable2`

Ex: `Note1 ← Note2`

3. `nom_variable ← expression`

Ex: `Moyenne ← (Note1*2 + Note2)/3`



# LES DONNEES : Identificateur

- C'est le **mot** qui symbolise la donnée. Son nom que l'on utilise pour la désigner dans l'algorithme.
- Plus les identificateurs sont imagés, plus l'algorithme est compréhensible :
  - Exemples :
    - prénom, lampe\_allumée, pi, taux\_TVA... valent mieux que : X, CDVLI, truc, compteur...
  - *Remarque* : En nommant les constantes, vous généraliserez à peu de frais vos algorithmes

# LES DONNEES : Déclaration

- Tout algorithme comporte une partie « **déclarative** » qui recense et définit l'ensemble des données qu'il utilise, c'est son **dictionnaire des données**.
- Pour chacune, on y indique :
  - son identificateur,
  - le type et la longueur de la donnée,
  - sa valeur - s'il s'agit d'une constante,
  - éventuellement son nombre d'éléments

➤ Par exemple

Prénom	caractères	20	<i>20 caractères maximum pour un prénom</i>
Age	Numérique	3	<i>3 chiffres maximum pour l'âge</i>
Lampe_allumée	booléen		2 valeurs possible VRAI ou FAUX

# LES ACTIONS

- Quand doit-on arrêter de décomposer les actions en actions plus élémentaires ?
  - ➔ Quand chacune d'entre-elles est connue de celui à qui l'on s'adresse, c'est à dire quand ces actions sont traduisibles dans le langage de programmation concerné.
- Vous trouverez dans tout langage de programmation au moins **trois instructions de base** :
  - Lire la valeur d'une variable (au clavier, sur le disque...)
  - Mémoriser ou modifier la valeur d'une donnée (faire des calculs...)
  - Ecrire la valeur d'une donnée (à l'écran, sur le disque, à l'imprimante...)

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**




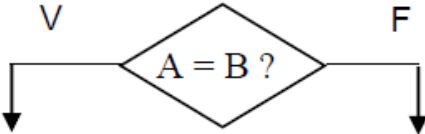
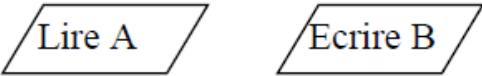
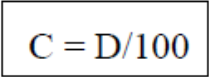
# FORMALISME





# ORDINOGRAMME

- Un « **ordinogramme** », encore appelé « **organigramme** », représente chaque élément de l'algorithme par un schéma.
- Il existe des schémas normalisés pour chaque type d'élément

Début / Fin du processus	
Enchaînement	
Branchement éloigné	
Expression d'une condition	
Opération d'Entrée / Sortie	
Affectation de valeur	

# Un exercice : Faire l'ordinogramme

## ALGO communication Téléphonique

NumTel, tempAtt numérique

### DEBUT

Lire TempAtt

Décrocher combiné

Attendre tonalité

Composer NumTel

**SI** il décroche

### REPETER

Parler

**JUSQUA** pas de sujet de discussion

### SINON

**TANTQUE** (attente < TempAtt) et (Bip non entendu )

Attendre 1s de plus

**FIN TANTQUE**

**SI** Bip entendu

laisser un message

**FIN SI**

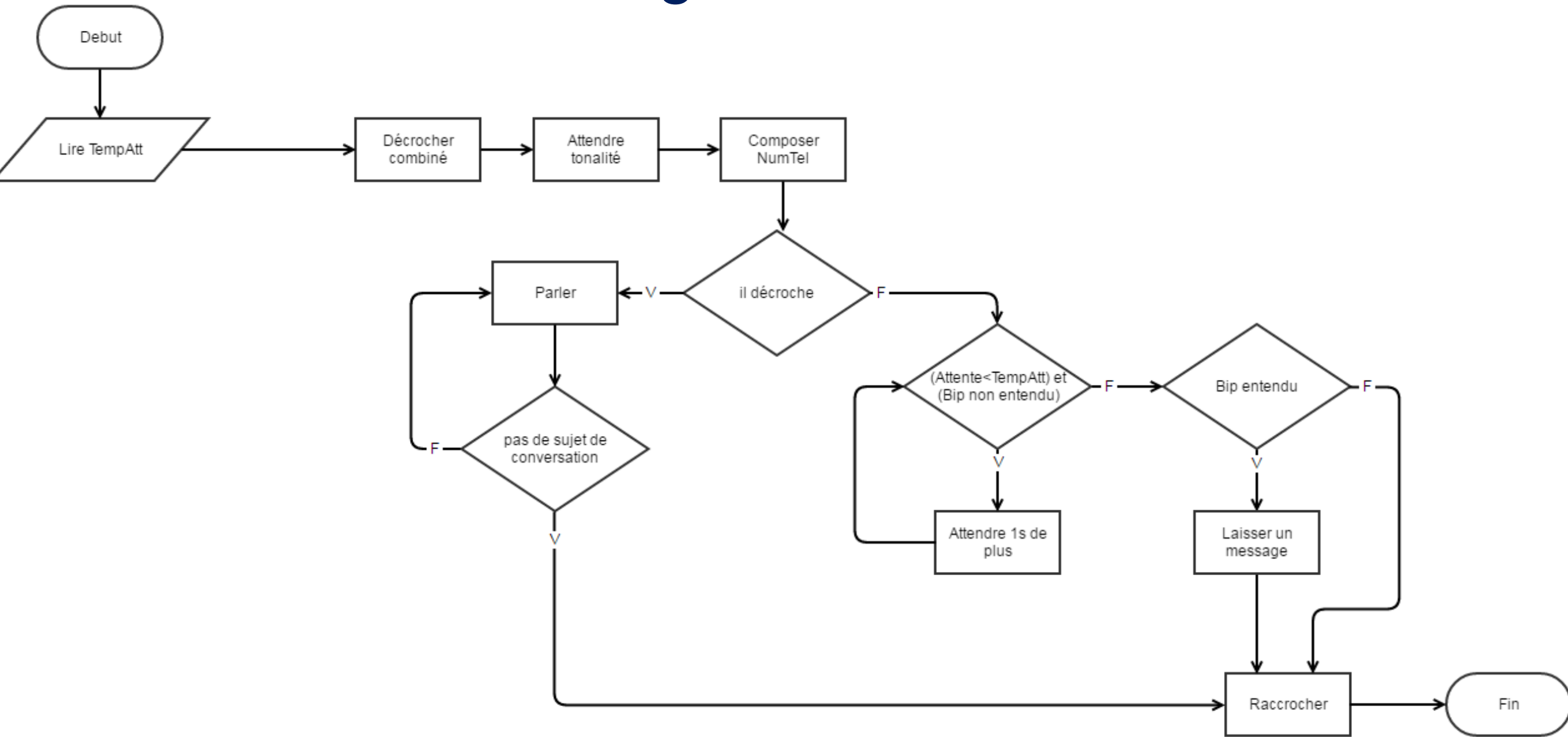
**FIN SI**

Raccrocher

### FIN



# Correction de l'ordinogramme



# ORDINOGRAMME

## ➤ *Remarque importante :*

- Ce type de représentation peut parfaitement convenir dans le cadre de l'expression structurée de la pensée.
- l'ordinogramme servait à l'analyse ascendante (c'est à dire du détail vers le général).
  - C'est pourquoi ce formalisme est considéré plutôt comme un bon outil de communication – et non comme un outil de conception si l'analyse de la pensée n'est pas structurée

# PSEUDO-CODE

- C'est une représentation écrite des éléments constitutifs d'un algorithme.
- **Chaque élément** constituant l'algorithme **est défini par un terme représentatif et non ambigu.**
- pseudo-code = éléments d'un langage → décrit l'algorithme  
Compréhensible par l'homme

Par opposition

- langage de programmation = code → transcrit l'algorithme en instructions exécutables par une machine.  
Accessible qu'au développeur, qu'à l'ordinateur



# PSEUDO-CODE

- L'expérience prouve qu'à court ou moyen terme, c'est l'utilisation très bien définie d'un petit nombre d'opérations élémentaires de représentation des algorithmes qui simplifie leur expression, facilitant ainsi tant leur conception et leur mise au point, que leur maintenance.
- En 1966, Böhm et Jacopini ont démontré qu'il est possible de décrire tout processus algorithmique en n'utilisant que :
  - l'enchaînement séquentiel,
  - et la boucle « Tant que ».

# Éléments de pseudo-code

- Il n'existe pas à l'heure actuelle de norme internationale syntaxique du pseudo-code.
- Nous en proposerons un exemple qui respecte l'esprit du langage

<b>Alternative</b>	SI <condition>              traitement 1 SINON              traitement 2 FIN-SI
<b>Itération</b>	TANT QUE <condition>              Traitement FIN-TANT-QUE  REPETER              Traitement JUSQU'A <condition>

# Éléments de pseudo-code

<b>Affectation de valeur</b>	$A \leftarrow B$ $A \leftarrow B / 100$
<b>Lecture d'une valeur</b> (du clavier, du disque...)	LIRE A
<b>Ecriture d'une valeur</b> (à l'écran, sur le disque, ...)	AFFICHER A (à l'écran) ECRIRE A (sur le disque)

## ➤ **Remarques :**

- « FIN-SI » clôture toujours l'alternative « SI », « FIN-TANT-QUE » la répétitive « TANT QUE »
- un trait vertical relie SI ... SINON ... FIN-SI, TANT QUE ... FIN-TANT-QUE, REPETER ... JUSQU'A
- les traitements imbriqués sont écrits en retrait

# Structure générale d'un algorithme informatique

- Tout algorithme informatique est composé de **3 parties** :
    - Un **entête** qui sert notamment à le nommer et préciser son utilité.
    - Une **partie déclarative** (ou **dictionnaire des données**) qui recense et définit tous les objets qu'il utilise. Chaque donnée doit y être déclarée en précisant :
      - son identificateur,
      - son type,
      - éventuellement sa longueur,
      - s'il s'agit d'une donnée constante, sa valeur.
    - Une **partie exécutable**, le « **corps** », qui indique les actions à exécuter avec les objets décrits
- ➔ Remarque : Des (\* commentaires \*) peuvent et **doivent être** intercalés pour la clarté du programme

# Exemple : MODULE DEVINETTE

(\* Entête\*)

(\* L'utilisateur doit deviner un nombre secret : \*)  
(\* S'il le trouve, on lui dit en combien de coups ; s'il abandonne, on divulgue le secret \*)

(\* Le dictionnaire des données\*)

CONSTANTE    secret            entier = 723

VARIABLES    jouer            caractère    1            (\* données saisies \*)

              nombre            entier

              nb\_coups          entier            (\* donnée calculée \*)

# Exemple : MODULE DEVINETTE

(\* Le corps\*)

nb\_coups  $\leftarrow$  0

REPETER

(\* Demander s'il veut (re)jouer \*)

ECRIRE « Voulez-vous deviner mon nombre secret ? »

ECRIRE « Tapez O pour oui, n'importe quelle autre touche pour non »

LIRE jouer

SI jouer = 'O'

ECRIRE « Tentez votre chance : »

LIRE nombre

nb\_coups  $\leftarrow$  nb\_coups + 1

SI nombre < secret

ECRIRE « Trop petit ! »

SINON SI nombre > secret

ECRIRE « Trop grand ! »

FIN-SI

FIN-SI

FIN-SI

JUSQU'A nombre = secret OU jouer  $\neq$  'O'

SI nombre = secret

ECRIRE « Vous avez trouvé en », nb\_coups, « coups »

SINON

ECRIRE « Mon nombre secret est », secret

FIN-SI

FIN-MODULE-DEVINETTE



# Exercices



1. Ecrire un algo d'un programme qui échange la valeur de deux variables.
  - Exemple, si  $a = 2$  et  $b = 5$ , le programme donnera  $a = 5$  et  $b = 2$
2. Ecrire un algo qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant.
  - Faire en sorte que des libellés apparaissent clairement.
3. Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne.
  - En cas de réponse supérieure à 20, on fera apparaître un message : **Plus petit !** et inversement, **Plus grand !** si le nombre est inférieur à 10.

# Exercice

- Ecrire un algo d'un programme qui échange la valeur de deux variables.
- Exemple, si  $a = 2$  et  $b = 5$ , le programme



```
DEBUT  
(* Déclarations *)  
    VARIABLES a entier  
                b entier  
                t entier  
  
(* le corps *)  
    a = 2  
    b = 5  
    t ← a  
    a ← b  
    b ← t  
  
FIN
```

# Exercice

- Ecrire un algorithme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant.
  - Faire en sorte que des libellés apparaissent clairement.



```
VARIABLES nb,pht,ttva,pttc numerique
DEBUT
  Ecrire "Entrez le prix hors taxes : "
  Lire pht
  Ecrire "Entrez le nombre d'articles : "
  Lire nb
  Ecrire "Entrez le taux de TVA : "
  Lire ttva
   $pttc \leftarrow nb * pht * (1 + ttva)$ 
  Ecrire " Le prix toutes taxes est : ", pttc
FIN
```

# Exercice

- Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne.
  - En cas de réponse supérieure à 20, on fera apparaître un message : **Plus petit !** et inversement, **Plus grand !** si le nombre est inférieur à 10.



```
VARIABLES N entier
```

```
DEBUT
```

```
  N ← 0
```

```
  TANT QUE N < 10 ou N > 20
```

```
    Ecrire "Entrez un nombre entre 10 et 20"
```

```
    Lire N
```

```
    SI N < 10 ALORS
```

```
      Ecrire " Plus grand ! "
```

```
    SINON
```

```
      Ecrire " Plus petit ! "
```

```
    FIN-SI
```

```
  FIN-TANT-QUE
```

# EXERCICE : L'ENQUETE COMPARATIVE

- Cahier d'exercices.

## Indications

- Réfléchir aux variables nécessaires avant de se lancer.



```
(* VARIABLES externes *)
prixA (* prix du produit dans le supermarché A *)
prixB (* prix du même produit dans le supermarché B *)

(* Attention: prixA=0 et prixB=0 indiquent la FIN de liste)

(* VARIABLES internes *)
totalA (* somme totale des prixA *)
totalB (* somme totale des prixA *)

totalMini (* somme totale des prix les moins chers *)
```

# Corrigé : L'ENQUETE COMPARATIVE

**Remarque :** les variables précédemment citées doivent être typées.

## 1) (\* Partie « exécutable » \*)

(\* Saisir les prix par produit, au fur et à mesure calculer les 3 totaux \*)

totalA  $\leftarrow$  0

totalB  $\leftarrow$  0

totalMini  $\leftarrow$  0

Saisir prixA, prixB

TANT QUE prixA  $\neq$  0 OU prixB  $\neq$  0

SI prixA > 0 ET prixB > 0

totalA  $\leftarrow$  totalA + prixA

totalB  $\leftarrow$  totalB + prixB

SI prixA < prixB

totalMini  $\leftarrow$  totalMini + prixA

SINON totalMini  $\leftarrow$  totalMini + prixB

FIN-SI

Saisir prixA, prixB

## 2)

SINON (\* Traitement des erreurs de saisie : on resaisit prixA ou prixB ou les 2 \*)

SI prixA < 0

Saisir prixA

SINON RIEN

FIN-SI

SI prixB < 0

Saisir prixB

SINON RIEN

FIN-SI

SI (prixA = 0 ET prixB  $\neq$  0) OU (prixB = 0 ET prixA  $\neq$  0)

Saisir prixA, prixB

SINON RIEN

FIN-SI

FIN-TQ



# Corrigé : L'ENQUETE COMPARATIVE

## 3) (\* Partie « exécutable » suite \*)

(\* Afficher le montant du « panier idéal » \*)

Ecrire « Le montant du panier idéal est : », totalMini

(\* Dire quel est le supermarché globalement le moins cher \*)

SI totalA < totalB

Ecrire « C'est le 1<sup>er</sup> supermarché le plus intéressant »

SINON SI totalA > totalB

Ecrire « C'est le 2<sup>ème</sup> supermarché le plus intéressant »

SINON Ecrire « Les 2 supermarchés se valent »

FIN-SI

FIN-SI

**FIN-MODULE enquête**

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**

# ALGEBRE DE BOOLE



# PRESENTATION

- L'Algèbre de Boole est une méthode de raisonnement qui présente deux intérêts importants :
  - elle permet de représenter par une notation simple et précise toutes les propositions impliquées dans un raisonnement,
  - elle permet de remplacer un raisonnement logique par un calcul et de bénéficier des automatismes que celui-ci permet.

# GENERALITES

- Soit  $E$  un ensemble d'éléments  $e_i$  et une propriété  $p$ .
- Chaque élément  $e_i$ , a ou n'a pas la propriété  $p$ .
- A chaque  $e_i$ , on associe une variable, dite variable de Boole, qui vaut :
  - 1 si  $p$  est vraie,
  - 0 si  $p$  est fausse.

# GENERALITES

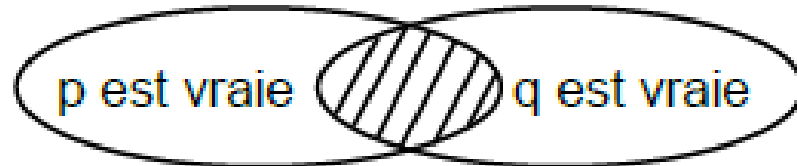
## ➤ Exemple :

- E est l'ensemble des concepteurs-développeurs d'un service informatique, e est un élément de cet ensemble, p est la propriété « avoir déjà travaillé sur un ordinateur z/OS ». Appelons « a » la variable de Boole correspondante.
- « a » est bien une variable, car sa valeur est fonction de l'individu considéré :
  - 1 s'il a travaillé sur z/OS,
  - 0 s'il n'a pas travaillé sur z/OS.



# LA CONJONCTION "ET"

- Considérons maintenant deux propriétés de chacun des éléments du même ensemble. Par exemple, connaître z/OS et connaître JAVA.
- Nous avons alors un autre sous-ensemble qui est l'ensemble des Concepteurs-Développeurs ayant déjà travaillé sur JAVA. Cette propriété sera notée  $q$ , la variable de Boole associée étant «  $b$  ».



- Dans la zone hachurée, les propriétés  $p$  **et**  $q$  sont vraies, et là seulement. On peut encore dire que dans cette zone les variables  $a$  **et**  $b$  sont égales à 1, et là seulement.

# LA CONJONCTION "ET"

- La table de vérité permet de recenser tous les cas de combinaison possibles

<b>a</b>	<b>b</b>	<b>a ET b</b>
1	1	1
0	1	0
1	0	0
0	0	0

- Par convention, la conjonction **ET** est souvent notée « **x** » ou « **.** »

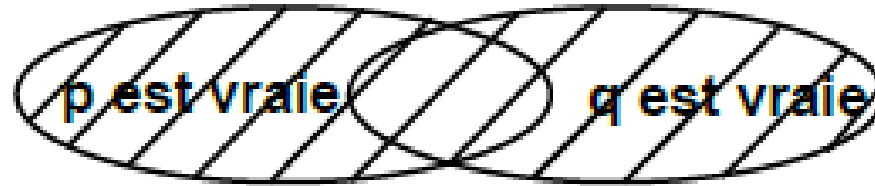
**a x b**

ou

**a . b**

# LA CONJONCTION "OU"

- Nous considérons toujours les deux mêmes propriétés des éléments du même ensemble.



- Dans la zone hachurée, les propriétés  $p$  ou  $q$  sont vraies : chacun des points de cet ensemble représente un concepteur-développeur qui possède au moins une des deux propriétés énoncées (et peut-être les deux) : il s'agit d'un OU **inclusif**.

# LA CONJONCTION "OU"

➤ Table de vérité :

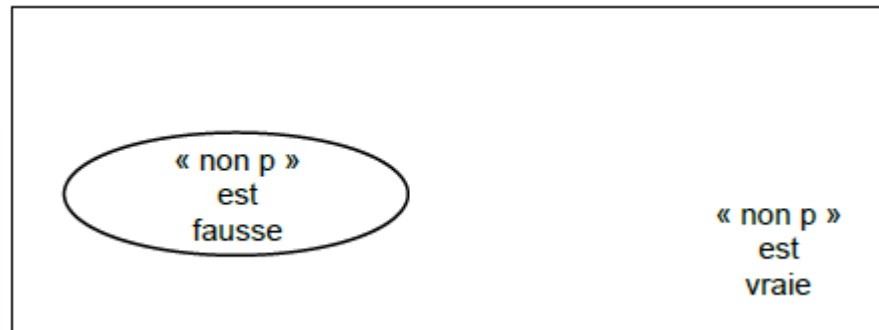
<b>a</b>	<b>b</b>	<b>a OU b</b>
1	1	1
0	1	1
1	0	1
0	0	0

➤ Par convention, la conjonction **OU** est souvent notée « **+** »

$$a + b$$

# LA CONJONCTION "NON"

- A chaque propriété  $p$  nous associons la propriété « non  $p$  ». Nous dirons que « non  $p$  » est vraie quand «  $p$  » est fausse et réciproquement.
- Dans notre exemple « non  $p$  » est la propriété « ne jamais avoir travaillé sur z/OS ».



- Par convention, **NON**( $p$ ) est souvent notée  $\overline{p}$

# RELATIONS FONDAMENTALES

$$a \cdot a = a$$

$$1 \cdot a = a$$

$$0 \cdot a = 0$$

$$\frac{\quad}{a \cdot a = 0}$$

$$\frac{\frac{\quad}{\quad}}{a = a}$$

$$a + a = a$$

$$1 + a = 1$$

$$0 + a = a$$

$$\frac{\quad}{a + a = 1}$$



# RELATIONS FONDAMENTALES

## ➤ ASSOCIATIVITE

$$(a + b) + c = a + (b + c)$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

## ➤ COMMUTATIVITE

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

# RELATIONS FONDAMENTALES

## ➤ DISTRIBUTIVITE

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

## ➤ ABSORPTION

$$a + (a \cdot b) = a$$

$$a \cdot (a + b) = a$$

$$\overline{a + (a \cdot b)} = \overline{a + b}$$

# RELATIONS FONDAMENTALES

## ➤ Loi de MORGAN

$$\overline{a \text{ ou } b} = \overline{a} \text{ et } \overline{b}$$

$$\overline{a \text{ et } b} = \overline{a} \text{ ou } \overline{b}$$

➤ Exemple : La négation de  
est

$$(A = B) \text{ ET } (C = D)$$

$$(A \neq B) \text{ OU } (C \neq D)$$

# CONCLUSION

- Nous avons défini ci-dessus les règles de calcul qui permettent de simplifier les expressions booléennes.
- Nous avons donc la possibilité d'exprimer une propriété à partir de propriétés plus « atomiques », puis par une réduction des termes semblables, une application des identités remarquables, etc. Nous obtiendrons une nouvelle expression de la propriété souvent plus maniable.
- Nous avons remplacé le cheminement logique de la pensée par un mécanisme de calcul

# Exercice

- traduire la situation pratique suivante en équation logique :
  - « *On décroche un téléphone quand on décide d'appeler quelqu'un ou lorsque le téléphone sonne et qu'on décide de répondre* ».



```
Décrocher = (Sonnerie ET Décision de répondre)  
OU Décision d'appeler  
a = "Sonnerie"  
b = "Décision de répondre"  
c = "Décision d'appeler"  
  
d = a.b + c
```

# Exercice

## ➤ Dessiner la table de vérité de la fonction décrocher

a="Sonnerie" b="Décision de répondre"

c="Décision d'appeler" d=résultat



Table de vérité de décrocher			
a	b	c	d
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

La table indique une situation absurde : **quand on décide d'appeler quelqu'un et que le téléphone sonne sans qu'on ait envie de répondre, on décrocherait quand même.**

Proposer une modification **d2** de l'équation pour corriger ce cas



$$d2 = \bar{a}.c + a.b.$$



# Exercice

- Un bon élève s'interroge s'il est sage de sortir un soir. Il doit décider en fonction de quatre variables :
  - $a$  = il a assez d'argent
  - $b$  = il a fini ses devoirs
  - $c$  = le transport en commun est en grève
  - $d$  = l'automobile de son père est disponible
- Ecrivez l'équation logique montrant a quelle condition l'élève pourra sortir

Cet élève pourra sortir si :

1) il a assez d'argent donc  $a = \text{vrai}$

2) il a fini ses devoirs, donc  $b = \text{vrai}$

3) le transport en commun n'est pas en grève, donc  $c = \text{faux}$  **ou** si l'automobile de son père est disponible, donc  $d = \text{vrai}$

L'expression logique de sortir en fonction de l'état des variables  $a$ ,  $b$ ,  $c$  et  $d$  peut donc s'écrire ainsi :

$$\text{Sortir} = a \cdot b \cdot (\bar{c} + d)$$



# Exercice

- Les scenarios possibles pour décrocher un téléphone sont :
- 1) il décide d'appeler : ( $\bar{a} . \bar{b} . c = 1$ )
  - 2) il décide de répondre ou décide d'appeler ( $\bar{a} . b . c = 1$ )
  - 3) sonnerie et il décide de répondre ( $a . b . \bar{c} = 1$ )
  - 4) sonnerie et il décide d'appeler et il décide de répondre ( $a . b . c = 1$ )

- Ce qui donne l'équation suivante :  $d2 = \bar{a} . \bar{b} . c + \bar{a} . b . c + a . b . \bar{c} + a . b . c$

- Trouver une équation minimaliste de d2



$$\begin{aligned} d2 &= \bar{a} . c (\bar{b} + b) + a . b (\bar{c} + c) \\ &= \bar{a} . c + a . b \end{aligned}$$

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**

# COMPLEMENTS ALGORITHMIQUES



# DONNEES

- **Types de données prédéfinis :**
- Dans tout langage de programmation, il existe des types de données prédéfinis.
- Les plus courants sont
  - Numériques,
  - Alphanumériques
  - Booléens



# DONNEES : Le type numérique

- Dans quel cas déclare-t-on une variable en numérique ?
- Uniquement quand cette variable est **destinée à des opérations arithmétiques**.
- Une donnée numérique peut contenir une valeur entière ou décimale, signée ou non. Sa longueur doit indiquer le nombre maximal de chiffres (« digits ») pouvant figurer avant et après la virgule.
- Par exemple, la déclaration :
  - Compteur NUMERIQUE 5
    - indique que cette variable pourra contenir 99999 ou -123 mais pas 123456



# DONNEES : Le type numérique

- En ce qui concerne les réels, l'interprétation dépendra du langage

Montant NUMERIQUE 7,2

- variable pouvant contenir 7 digits au total **dont** 2 décimales, c'est-à-dire 12345,67 ou -123 mais pas 12345678 ni 1,234

OU

- variable pouvant contenir 7 digits pour la partie entière **plus** 2 décimales.  
Exemple : 1234567,89

# DONNEES : type caractère

- Une donnée de type caractère (encore appelé type « alphanumérique ») peut contenir des lettres, des chiffres et tout autre symbole que l'on peut obtenir au clavier.
- Ce type est également **ordonné** par une table où sont recensées toutes les valeurs :
  - table ASCII sur la plupart des systèmes
  - table EBCDIC dans le monde IBM
- C'est pourquoi :
  - 'a' < 'b'
  - 'a' <> 'A'
  - '9' > '8' ...

# DONNEES : type caractère

- La longueur d'une chaîne de caractères indique le nombre maximal de caractères que la chaîne peut comporter.
- La plupart des langages de programmation ont défini pour **opérations de type caractère** :
  - calculer la longueur réelle d'une chaîne (syntaxe : `long(chaîne)` )
  - « concaténer » 2 chaînes (mettre bout à bout)
  - « extraire » une sous-chaîne (isoler un sous-ensemble d'une chaîne)

# DONNEES : type booléen

- Un booléen n'admet que 2 valeurs qui s'excluent mutuellement :
  - **vrai / faux**,
  - **0 / 1** (0 signifiant « faux » et 1 signifiant « vrai »).
  
- Par exemple, la déclaration :  
`Fin-fichier booléen`
  - Si fin-fichier = 0 → ce n'est pas la fin de fichier,
  - Si fin-fichier = 1 → c'est la fin de fichier.

# DONNEES : Les tableaux

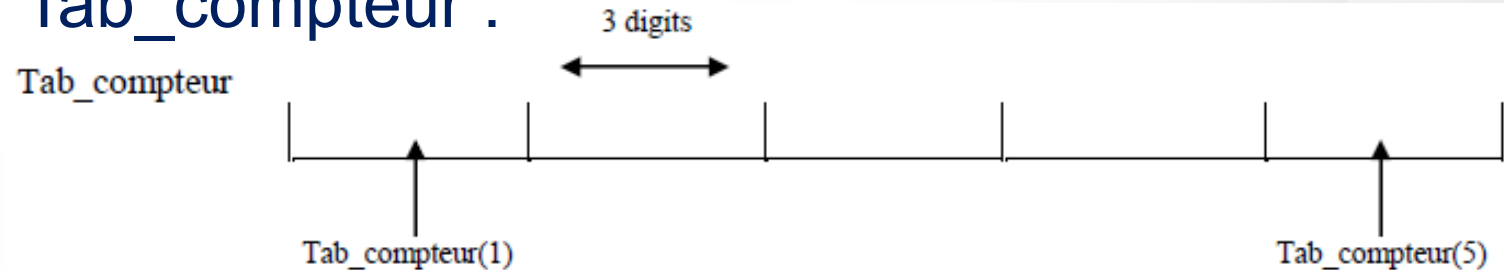
- Un tableau est une donnée composée d'un **nombre défini d'éléments** :
  - **de même type et**
  - **de même longueur.**
- Chaque **élément** (ou « **poste** ») est identifié par un numéro : son « **indice** ».
- Selon les langages de programmation, le nombre d'éléments est indiqué soit directement, soit en précisant les bornes de l'indice

# DONNEES : Les tableaux

- Par exemple, la déclaration :

```
Tab_compteur [ 5 ] NUMERIQUE 3
```

- indique que le tableau Tab\_compteur est composé de 5 postes, chacun déclaré en numérique de 3.
- Représentation du tableau Tab\_compteur :



- `Tab_compteur(1)`, `Tab_compteur(5)` sont maintenant des identificateurs d'entiers, qui se manipulent comme des identificateurs habituels, sauf que l'indice peut être remplacé par une variable numérique, ce qui va permettre d'automatiser certains traitements



# DONNEES : Les tableaux

- **Attention** : Le développeur doit **empêcher les débordements de tableau** car les problèmes engendrés sont imprévisibles.
  - Il est donc interdit de trouver dans un programme `Tab_compteur(l)` avec  $l = 0$  ou, dans notre exemple, avec  $l$  supérieur à 6 !
- Il est à noter que certains langages démarrent leur comptage à 0.
  - Dans ce cas, dans notre exemple les limites seraient `Tab_compteur(0)` à `Tab_compteur(4)`.

# DONNEES : Les tableaux

- Exemple : Opérer une remise de 10% sur le montant de 50 factures

Dictionnaire des données :

Nb_max	NUMERIQUE	=	50
No_fac	NUMERIQUE		2
Tab_montants [Nb_max ]	NUMERIQUE		10,2

Corps :

.../...

No\_fac ← 1

REPETER

	Tab_montants (No_fac)	←	Tab_montants (No_fac) * 0,9
	No_fac	←	No_fac + 1

JUSQU'à No\_fac > Nb\_max

# DONNEES : Les tableaux

- Beaucoup de langages de programmation admettront des tableaux à **2 dimensions**, certains même à **N dimensions** (en permettant de construire des tableaux de tableaux).
- Certains permettent de mettre en oeuvre des tableaux « dynamiques » (dont le nombre maximum de postes n'est pas une constante).

# Exercice

- Donnez un algorithme qui assure le tri d'un tableau à **10** éléments
- Regardez le lien suivant :
  - <https://www.youtube.com/watch?v=Ns4TPTC8whw>
- En bon français, nous pourrions décrire le processus de la manière suivante :
  - Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
  - Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel est le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.



```
VARIABLES Tab[10] entier
           posmini entier
```

**DEBUT**

```
Tab ← {5,3,9,2,1,8,0,4,7,6}
```

```
i ← 0
```

```
(*boucle principale : le point de départ se décale à chaque tour*)
```

**REPETER**

```
(*on considère provisoirement que t(i) est le plus petit élément*)
```

```
posmini ← i
```

```
(*on examine tous les éléments suivants*)
```

**REPETER**

```
  j ← i+1
```

```
  Si t(j) < t(posmini) Alors
```

```
    posmini ← j
```

**Finsi**

```
  j ← j+1
```

**JUSQUA** j = 11

```
(* A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer la permutation.*)
```

```
temp ← t(posmini)
```

```
t(posmini) ← t(i)
```

```
t(i) ← temp
```

```
(* On a placé correctement l'élément numéro i, on passe à présent au suivant.*)
```

```
i ← i + 1
```

**JUSQU'À** i = 10

**FIN**

# Structures de données

- Il est possible dans la plupart des langages informatiques de déclarer des variables qui ont **un sens dans leur globalité**, mais qui ont aussi une signification lorsque l'on s'intéresse aux **caractéristiques élémentaires qui la composent**.
- Quelle est la différence entre une structure de données et un tableau ?
- Une **structure de données** contient des variables de **nom** et de **type différents**
- Un **tableau** contient des postes de **même nom, même type et même longueur**.



# Structures de données

- Par exemple, le numéro de sécurité sociale à 13 chiffres représente l'identifiant unique d'un assuré social. Mais...
  - le premier chiffre représente le sexe de l'assuré ;
  - les quatre suivants, l'année puis le mois de sa naissance ;
  - les deux suivants son département de naissance ;
  - etc.
- Pour pouvoir manipuler aussi bien l'identifiant de l'assuré que les quatre chiffres précisant sa date de naissance, plutôt que de déclarer deux variables physiquement distinctes et qu'il faudra valoriser en parallèle, on peut indiquer la décomposition de la variable de la manière suivante :
- |          |            |   |
|----------|------------|---|
| VARIABLE | no_sécu    |   |
| • sexe   | CARACTERE  | 1 |
| • année  | NUMERIQUE  | 2 |
| • mois   | NUMERIQUE  | 2 |
| • départ | CARACTERES | 2 |
| • ville  | CARACTERES | 3 |
| • no_seq | NUMERIQUE  | 3 |

# Structures de données

- Nous pouvons améliorer la description de notre structure en imaginant que le mois et l'année vont aussi être manipulés en bloc (à des fins d'affichage par exemple) et que le département, la ville et le numéro séquentiel n'ont par contre pas besoin d'être détaillés puisqu'ils ne seront pas exploités indépendamment des autres caractères du numéro de sécurité sociale. La structure deviendra donc :

- |          |            |            |   |
|----------|------------|------------|---|
| VARIABLE | no_sécu    |            |   |
| •        | sexe       | CARACTERE  | 1 |
| •        | date_naiss |            |   |
| —        | année      | NUMERIQUE  | 2 |
| —        | mois       | NUMERIQUE  | 2 |
| •        | reste      | CARACTERES | 8 |

# Structures de données

- Dans beaucoup de langages de programmation, on peut définir au préalable un nouveau type, en indiquant pour chacun de ses membres son type et sa dimension ; puis déclarer une (ou des) variable(s) ayant ce nouveau type.
- Exemple :

```
TYPE noss
  sexe          NUMERIQUE  1
  date_naiss    NUMERIQUE  2
  année         NUMERIQUE  2
  mois          NUMERIQUE  2
  reste         NUMERIQUE  8

TYPE employe
  no_sécu       noss
  nom           CARACTERES  20
  prénom       CARACTERES  15
  adresse
    libellé1    CARACTERES  40
    libellé2    CARACTERES  40
    cod_post    CARACTERES   5
    ville       CARACTERES  20

VARIABLE Var_emp1  employe
VARIABLE Var_emp2  employe
```

# Structures de données

- Quelle que soit la technique de déclaration, il faudra, pour accéder aux membres eux-mêmes, utiliser une notation préfixée (ou notation “pointée”) :

```
Var_emp1.nom ← 'Untel'
```

```
Var_emp2.adresse.cod_post ← '75010'
```

- Enfin, il est toujours possible de déclarer un **tableau de structures de données** :

```
Tab_emp [10] employe
```

- Puis, pour l'utiliser :

```
Tab_emp(3).no_secu.sexe ← 1
```

# Objets

- Il est possible dans les langages informatiques orientés objets de déclarer des variables qui ont comme type des classes.
  - La **classe** est une définition, un modèle caractérisant un ensemble d'éléments du système d'information.
  - Ces définitions existent en amont des algorithmes et peuvent être définies par des concepteurs-développeurs et/ou faire parties du langage objet.
- En plus de mémoriser des données, que l'on nomme des attributs, la classe stocke aussi les traitements possibles à effectuer, que l'on nomme des méthodes.
  - **L'objet** est un représentant de la classe que l'on stocke en variable, à l'identique des variables de type booléen, numérique...

# Objets

- Par exemple nous disposons d'une classe Date contenant :
  - en attributs : le jour, le mois, l'année et le siècle.
  - En méthode, cette classe dispose d'un traitement nommé `retourneLibelle` nous permettant d'obtenir le libellé du mois (sous forme de caractères).

```
La classe Date {  
* attributs *  
    siècle NUMERIQUE 2  
    année NUMERIQUE 2  
    mois NUMERIQUE 2  
    jour NUMERIQUE 2  
* méthode *  
    Alphanumérique retourneLibelle()  
}
```

# Objets

- On peut donc déclarer une variable `dateNaissance` de type `Date`, puis l'initialiser et la manipuler

```
Date dateNaissance
```

```
dateNaissance ← 01072016
```

```
dateNaissance.jour ← 31
```

- On peut déclarer une variable `libelle` de type caractères puis l'initialiser par la variable `dateNaissance` en utilisant la méthode *retourneLibelle()*

```
Alphanumerique libelle
```

```
Libelle ← dateNaissance.retourneLibelle()
```

- Libelle prend la valeur Juillet.



# ENCHAINEMENTS : Choix multiple

- Il s'agit d'un ensemble de traitements **exclusifs**, conditionnés par la valeur d'une variable commune :

```
CHOIX SELON variable
|
|  valeur_1      traitement 1
|  valeur_2      traitement 2
|  valeur_3      traitement 3
|  ...
|  AUTREMENT     traitement N
|
FIN-CHOIX
```

- Cette construction est équivalente à plusieurs alternatives **imbriquées** : si une même valeur est citée plusieurs fois, seul le traitement face à sa première apparition est exécuté.

```
SI variable = valeur_1
|
|  traitement 1
|
SINON SI variable = valeur_2
|
|  traitement 2
|
|  SINON SI variable = valeur_3
|  |
|  |  traitement 3
|  |
|  |  SINON
|  |  |
|  |  |  traitement N
|  |  |
|  |  |  FIN-SI
|  |  FIN-SI
|  FIN-SI
FIN-SI
```

# ENCHAINEMENTS : Itération fixe

- Cette boucle est utile quand on connaît le nombre de fois où l'on voudra exécuter un traitement

```
POUR indice DE valeur_init À valeur_finale
|
|   Bloc d'instructions
|
FIN-POUR
```

- La valeur finale est déterminée une seule fois, à l'entrée de la boucle.
- L'indice reçoit la valeur initiale, puis varie à chaque tour par pas de +1 (par défaut).

- Exemple : Lire un livre

```
Nb_pages ← 100
POUR no_page DE 1 À Nb_pages
|
|   Lire la page
|
FIN-POUR
```

Aurait dû  
s'écrire – si la  
boucle POUR  
n'existait pas :

```
Nb_pages ← 100
No_page ← 1
TANT QUE no_page ≤ Nb_pages
|
|   Lire la page
|   No_page ← No_page + 1
|
FIN-TQ
```

# Débranchement inconditionnel

- Une des structures de commandes répandues de l'algorithmique est le branchement inconditionnel.
- À l'exécution d'un programme, lorsqu'une instruction a été exécutée, le registre d'adresse d'instruction contient l'adresse de l'instruction suivante.
- Pour modifier cette adresse, c'est-à-dire pour rompre l'exécution séquentielle, beaucoup de langages informatiques proposent une instruction permettant de se débrancher à un point quelconque du source, repéré par une « étiquette ».
- En pseudo-code, cette instruction a la forme suivante :

```
A ← B  
  
ALLER À Suite  
  
...  
Suite  
  
B ← C
```

# Débranchement inconditionnel

- Toutes les études faites sur les structures de programme montrent que l'instruction ALLER À est inutile et plutôt néfaste.
- La polémique qui s'est développée à propos de cette instruction vient du fait que certains langages parmi les plus utilisés (BASIC, COBOL) ne possédaient pas les instructions correspondant aux différentes structures de commande de l'algorithmique, notamment la répétitive, et que dans ces conditions, le ALLER À était nécessaire pour les traduire.
- Si le ALLER À est parfois nécessaire, pourquoi vouloir l'éliminer ?
  - Parce qu'il **facilite un raisonnement suivant le libre cours de la pensée qui, par essence, n'est pas structurée**

# Débranchement inconditionnel

- En éliminant les « ALLER À » et en fournissant un petit nombre de structures de base, l'algorithmique va favoriser un mode de raisonnement structuré, standardisé, permettant d'améliorer la productivité du programmeur et de faciliter la mise au point ainsi que les futures maintenances.
- Toutefois nous ne supprimerons pas totalement les instructions ALLER À car, dans le cadre des langages dont nous disposons, cela entraîne parfois la répétition inutile de conditionnements tout au long d'un module.
  - Nous n'admettons donc exclusivement qu'une seule forme de ALLER À :
    - le **ALLER À descendant**, se branchant **exclusivement à l'intérieur** du même module.
  - Toute autre utilisation sera exclue et nous réfléchirons sur son utilité à chaque fois que le cas se présentera, car, répétons-le, son utilisation peut toujours être évitée.

# EXERCICE : LE PALINDROME

➤ Cahier d'exercices





# Corrigé : LE PALINDROME

## (\* Partie déclarative \*)

```
CONST      MaxCar numérique    = 80
VAR        Phrase [MaxCar] caractères
          Gauche    numérique  2  (* position du caractère à gauche *)
          Droite    numérique  2  (* position du caractère à droite )
          Palindrome booléen
          Trouvé    booléen    (* il y a au moins un caractère pris en compte *)
```

## (\* Partie exécutable \*)

### (\* Préparation \*)

```
ECRIRE « Donnez votre phrase »
LIRE Phrase
```

### (\* Init Traitement: déterminer si c'est un palindrome ou non, indiquer s'il existe au moins un caractère pris en compte \*)

```
Gauche ← 1
Droite ← longueur_reelle(phrase)
Palindrome ← vrai
Trouvé ← faux
```

TANT QUE gauche <= droite ET palindrome

SI ( Phrase(Gauche) < 'a' OU Phrase(Gauche) > 'z' )  
ET ( Phrase(Gauche) < 'A' OU Phrase(Gauche) > 'Z' )  
ET ( Phrase(Gauche) < '0' OU Phrase(Gauche) > '9' )  
(\* ce n'est pas un caractère à prendre en compte \*)  
Gauche ← Gauche + 1

SINON

SI idem avec Phrase(Droite)  
(\* ce n'est pas un caractère à prendre en compte \*)  
Droite ← Droite - 1

SINON

Trouvé ← vrai

SI Phrase(Gauche) = Phrase(Droite)  
Gauche ← Gauche + 1  
Droite ← Droite - 1

SINON

Palindrome ← faux

FIN-SI

FIN-SI

FIN-SI

FIN-TQ

# Corrigé : LE PALINDROME

(\* Résultat : indiquer si c'est un palindrome ou non, ou rien \*)

ECRIRE phrase

SI NON Trouvé

ECRIRE « Je ne peux conclure »

SINON

SI palindrome

ECRIRE « est un palindrome »

SINON

ECRIRE « N'est PAS un palindrome »

FIN-SI

FIN-SI

**FIN-MODULE Palindrome**

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**

FICHIERS



# FICHIERS

- Un fichier est un ensemble d'informations stocké sur un (ou plusieurs) disque(s), délimité par un label de début et un label de fin.
- Un fichier peut stocker aussi bien des données que des programmes.
- De l'organisation des informations au sein du fichier, découlent différents modes d'accès possibles.



# FICHIERS : Types d'organisation

## ➤ Les fichiers texte

- Ce sont ceux que vous pouvez manipuler avec un éditeur ou un traitement de texte. Dans de tels fichiers, les informations sont stockées sous forme de chaînes de caractères ; leur contenu est donc lisible.
- Ces fichiers sont structurés par lignes (en général délimitées par le caractère « Retour chariot » suivi du caractère « fin de ligne »).
- Cette organisation empêche de demander au système, par exemple, de lire la 3ème ligne. En effet, les lignes n'ont a priori pas de taille fixe, on ne peut donc présupposer de l'emplacement du début d'une ligne particulière – sauf à parcourir le fichier ligne à ligne, ce qui est peu performant.

# FICHIERS : Fichiers séquentiels

- Ces fichiers contiennent des données ayant toutes le même type, le plus souvent, regroupées en **enregistrements** (records).
- Les données de ces fichiers sont enregistrées physiquement dans **l'ordre chronologique** de leur écriture et seront restituées, en lecture, dans le même ordre.
- Lorsqu'un fichier est organisé en enregistrements, il faudra, à l'utilisation, en indiquer la structure (le « masque »).
- Si ces enregistrements sont de format fixe (de même longueur), il sera possible d'accéder directement à un élément par son numéro d'ordre dans le fichier.

# FICHIERS : Fichiers séquentiels indexés

- Cette fois, on associe un fichier « **index** » au fichier séquentiel typé, qui permettra de retrouver rapidement un élément dans le fichier séquentiel :
- On enregistre dans ce fichier index uniquement les valeurs de la (ou des) donnée(s) servant de critère de recherche (la « clé »), accompagnées de l'adresse physique du ou des enregistrements correspondants dans le fichier séquentiel.
- Un index est en général un arbre binaire, structure permettant une recherche dichotomique (bien plus rapide qu'une recherche séquentielle).

# FICHIERS : Les bases de données

- Un système de gestion de bases de données est un logiciel qui assure notamment l'interface entre utilisateurs (ou développeurs) et fichiers physiques.
- Ceux-ci n'ont donc plus à connaître la structure et l'emplacement des fichiers utilisés (seul le DBA(\*) s'en préoccupe) et peuvent se concentrer sur une vision purement logique des données

# FICHIERS : Utilisation

- Quelle que soit l'organisation du fichier, la procédure générale sera la même :

1. **Déclaration du nom du fichier** : celui qu'on utilisera au sein du programme pour le référencer.

- C'est lors de cette première étape qu'on indiquera l'**organisation physique** du fichier (texte, séquentiel, accès direct...) et le **mode d'accès** choisi (séquentiel, indexé...).
- Remarque : Il restera à le mettre en correspondance avec le nom réel du fichier sur le disque (et son emplacement exact).

# FICHIERS : Utilisation

**2. Ouverture du fichier** qui doit préciser la méthode d'accès choisie. Sont possibles pour les fichiers en organisation séquentielle (Sequential Access Method ou « SAM ») :

- **En création** Le fichier n'existe pas, son « ouverture » provoque l'écriture du *label de début* (“ BOF ” ou “ TOF ” pour Bottom, ou Top Of File). Le traitement consistera à remplir le fichier, c'est donc une ouverture en écriture.
- **En extension** Le fichier existe, on souhaite y ajouter des informations à *la suite* de celles qui y sont déjà.
- **En lecture** Le fichier existe, on veut juste le lire.

# FICHIERS : Utilisation

## 3. Opérations de lecture / écriture

- Un ordre de lecture - ou d'écriture - dans un fichier, lit - ou écrit - un élément dans le fichier et déplace un pointeur (ou « curseur ») sur l'élément suivant.

## 4. Fermeture du fichier

- Provoque l'écriture du *label de fin de fichier* ( " EOF " pour End Of File) s'il avait été ouvert en écriture ; déconnecte le fichier du programme.



# FICHIERS : Exemples d'utilisation : façon Cobol

```
MODULE manip_fichier
```

```
(* * * Recopier dans un fichier les employés habitant à Paris * * *)
```

```
(* Déclarations *)
```

```
    FIC fic_employes (* Nom interne du fichier *)
```

```
    Séquentiel ACCES séquentiel (* Organisation et mode d'accès *)
```

```
    FIC fic_parisiens
```

```
    Séquentiel ACCES séquentiel
```

```
(* Il faut une variable par fichier : le "masque" *)
```

```
    VARIABLE employé
```

```
    Matricule caractères 5
```

```
    Nom caractères 20
```

```
    Adresse caractères 80
```

```
    Codpost numérique 5
```

```
    Ville caractères 20
```

```
    Salaire numérique 7,2
```

```
    ...
```

# FICHIERS : Exemples d'utilisation : façon Cobol - suite

```
VARIABLE parisien
  Matricule caractères 5
  Nom caractères 20
  Adresse caractères 80
  Codpost numérique 5
  Ville caractères 20
  Salaire numérique 7,2

VARIABLE fin_fic_employes (* Et un booléen pour gérer la fin du fichier *)

(* Partie exécutable *)
OUVRIR fic_employes EN LECTURE
OUVRIR fic_parisiens EN CREATION (* Ecriture du label de début de parisiens *)
...
```

# FICHIERS : Exemples d'utilisation : façon Cobol - suite

```
(* Parcours séquentiel du fichier employes *)
LIRE fic_employes (* Lecture du 1er employé *)
  A LA FIN fin_fic_employes ← vrai

TANT QUE NON fin_fic_employes
  SI emp.codpost >= 75000 ET emp.codpost <= 75999
    parisien ← employé
    ECRIRE fic_parisiens (* Ajout d'un parisien *)
  FIN-SI
  LIRE fic_employes (* Lecture du prochain employé *)
FIN-TANT-QUE

FERMER fic_employes
FERMER fic_parisiens (* Ecriture du label de fin de parisiens *)
FIN-MODULE manip_fichier
```

**MODULE manip\_fichier**

(\* \* \* Recopier dans un fichier les employés habitant à Paris \* \* \*)

(\* Déclarations \*)

```
FIC fic_employes (* Nom interne du fichier *)
Séquentiel ACCES séquentiel (* Organisation et mode d'accès *)
FIC fic_parisiens
Séquentiel ACCES séquentiel
TYPE ENREG (* Les 2 fichiers auront la même structure *)
```

(\* Partie exécutable \*)

```
OUVRIR fic_employes EN LECTURE
OUVRIR fic_parisiens EN CREATION (* Ecriture du label de début de parisiens *)
(* Parcours séquentiel du fichier employes *)
LIRE ( fic_employes, emp ) (* Lecture du 1er employé *)
TANT QUE NON FIN_DE_FICHER (fic_employes)
    SI emp.codpost <= 75000 ET emp.codpost >= 75999
        ECRIRE ( fic_parisiens, emp ) (* Ajout d'un parisien *)
    FIN-SI
    LIRE ( fic_employes, emp ) (* Lecture du prochain employé *)
FIN-TANT-QUE
FERMER fic_employes
FERMER fic_parisiens (* Ecriture du label de fin de parisiens *)
```

**FIN-MODULE manip\_fichier**

# Exercice

- On travaille avec le fichier du carnet d'adresses en champs de largeur fixe.

- Ecrivez un algorithme pour saisir l'adresse d'un individu qui sera



**Variables** Nom \* 20, Prénom \* 17, Tel \* 10, Mail \* 20, Lig en Caractère

**Debut**

**Ecrire** "Entrez le nom : "

**Lire** Nom

**Ecrire** "Entrez le prénom : "

**Lire** Prénom

**Ecrire** "Entrez le téléphone : "

**Lire** Tel

**Ecrire** "Entrez le Mail : "

**Lire** Mail

Lig ← Nom & Prénom & Tel & Mail

**Ouvrir** "Adresse.txt" sur 1 **pour Ajout**

**EcrireFichier** 1, Lig

**Fermer** 1

**Fin**

# LA PERSISTANCE : Définition

- La persistance des données et des états d'un programme font référence aux mécanismes de sauvegarde et de restauration des données.
- Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus.
- Ces informations de reprise peuvent être enregistrées sur disque, éventuellement sur un serveur distant quand il s'agit d'un système de gestion de base de données (SGBD).

# LA PERSISTANCE : Le modèle des SGBDR

- Les systèmes de bases de données relationnelles (SGBDR) apparues dès la fin des années 1970 ont comme particularité une gestion performante des sauvegardes, sans perte d'informations après reprise lors d'un plantage.
- C'est ce qui a permis à ce modèle de s'implanter dans les entreprises pendant plus de 40 ans.
  - Implantation due aussi au fait qu'il existait un langage standard de conception de base et manipulation des données **nommé SQL** (Strutured Query Language).



# LA PERSISTANCE : Le modèle des SGBDR :

## Présentation du modèle relationnel

- Les bases de données relationnelles permettent de faire évoluer le schéma des données sans modifier les programmes existants.
- Elles sont entièrement intégrées aux architectures distribuées et proposent de nombreuses fonctionnalités liées au développement et à l'exploitation d'applications client-serveur.
- Elles sont conçues pour aller vite en mise à jour et assurent un mode transactionnel, qui permet un retour en arrière s'il n'y a pas de validation.
- L'élément principal d'une base relationnelle est la **table**.
  - La table est un ensemble de colonnes fixes (de type alphanumérique, numérique, ...) contenant des lignes représentant les données stockées par l'entreprise.
  - Chaque ligne est identifiée de manière unique par une **clé primaire**. Il est possible d'accéder à ces données via différents index.

# LA PERSISTANCE : Le modèle des SGBDR :

## Exemples de tables du modèle relationnel

### ➤ Exemple d'accès aux données d'un SGBDR (SGBD Relationnel)

- Soit les tables :
  - EMPLOYE (MLE, NOM, PRENOM, SEXE)
  - AFFECTATION (MLE, CODPRO, DATDEB)
  - DEPT (CODEPT, DESIGN, DG)

MLE [DECIMAL(10 , 0)]	NOM [CHAR(20)]	PRENOM [CHAR(20)]	SEXE [CHAR(1)]
25	LORENT	CATHERINE	F
53	BLASQUEZ ...	NICOLAS	M
54	HERNANDEZ ...	ANTOINE	M
55	MARCIER	JACQUES	M
56	GALLET	BRUNO	M
100	DUCHE	SYLVIE	F
103	SOUPINACIO ...	JESUS	M
218	FONTAINE ...	JEAN-PIERRE	M
273	CHICHE	GLADYS	F
286	ROUVRAIS ...	PHILIPPE	M
303	SALAMI	JUSTIN	M
304	ALPHANDERY ...	ALPHONSE	M
333	PONCHEL	VINCENT	M
633	BALLARD	GASTON	M
652	BIBER	ALBERT	M
672	DUVAL	CHRISTINE	F

MLE [DECIMAL(10 , 0)]	CODPRO [DECIMAL(10 , 0)]	DATDEB [DATE]
25	6	01/04/96
53	120	02/02/97
54	120	02/02/97
55	120	02/02/97
56	6	02/02/97
100	12	01/04/96

CODEPT [DECIMAL(10 , 0)]	DESIGN [CHAR(20)]	DG [CHAR(20)]
2	CONSEIL	DUPONT
3	INDUSTRIE	PEREZ
6	ETHNOS	ROQUE ...
12	FORMATION	MARTIN
120	COMPTABILITE	MIRAN

(\* \* \* Rechercher les employés  
habitant à Paris \* \* \*)

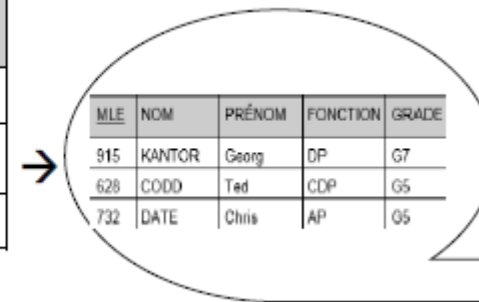
**SELECT** mle, nom, prenom, sexe  
**FROM** ingénieur **WHERE** ville = 'PARIS'

# LA PERSISTANCE : Le modèle des SGBDR : Principes de la journalisation

- Les modifications apportées dans les bases peuvent être conservées dans un journal.
  - Les informations sauvegardées dans le journal sont les valeurs des lignes des tables avant et après la modification.
  - Cela permet de maintenir **l'intégrité des données** et de ne pas perdre d'informations en cas de plantage important avec endommagement des disques et obligation de répartir des sauvegardes journalières.

Exemple : La table Salarié contient 3 lignes, ces informations ont été sauvegardées, le soir, sur un support externe.

SALARIÉ	MLE	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	AP	G5



MLE	NOM	PRÉNOM	FONCTION	GRADE
915	KANTOR	Georg	DP	G7
628	CODD	Ted	CDP	G5
732	DATE	Chris	AP	G5

# LA PERSISTANCE : Le modèle des SGBDR : Principes de la journalisation

- Dans la journée, l'utilisateur effectue des modifications sur la base.
  - Il ajoute le salarié 10 Dupont Pierre DP G7 et modifie la fonction du salarié 732 avec la valeur CDP.

SALARIÉ	<u>MLE</u>	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	<del>AP</del> CDP	G5
	10	Dupont	Pierre	DP	G7

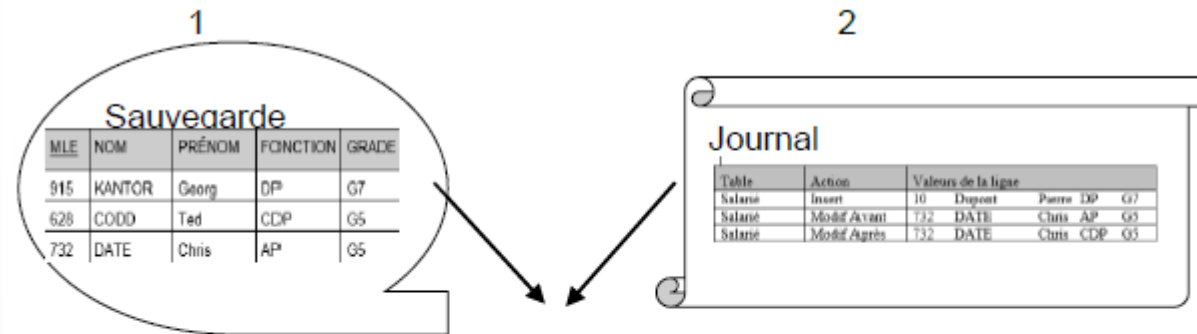
- Le journal (qui n'est pas une table) contiendra :

Table	Action	Valeurs de la ligne				
Salarié	Insert	10	Dupont	Pierre	DP	G7
Salarié	Modif Avant	732	DATE	Chris	AP	G5
Salarié	Modif Après	732	DATE	Chris	CDP	G5

- Ainsi, nous sommes capables à partir du journal de refaire la modification (lecture avant), et/ou **de défaire la modification** (lecture arrière)

# LA PERSISTANCE : Le modèle des SGBDR : Principes de la journalisation

- En cas de plantage, on restaure la table Salarié à partir du support externe, (elle ne contient que les 3 lignes du départ) et à **partir du journal**, on refait les mises à jour (Insertion et Modification Après).
- Il n'y a pas de perte de données entre la sauvegarde de la veille et le travail effectué mais perdu suite au plantage !



Récupération de la table des salariés et des mises à jour de la journée

SALARIÉ	MLE	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	CDP	G5
	10	Dupont	Pierre	DP	G7



# LA PERSISTANCE : Le modèle des SGBDR : Principes de la journalisation

- La journalisation n'est pas obligatoire mais fortement conseillée !
- Des points de validation (dits COMMIT) et d'invalidation (ROLLBACK) peuvent être placés dans le journal pour délimiter des transactions (détail des transactions dans le point suivant). (Mises à jour de plusieurs lignes de plusieurs tables).
  - Dans ce cas, on dit que la base est **sous contrôle de validation**.
- Pour les Commit et Rollback et pour la gestion des contraintes d'intégrités référentielles, la journalisation est obligatoire.

# LA PERSISTANCE : Les transactions :

## Définition

- Une transaction est une unité logique de travail qui comprend un ensemble d'ordres SQL indissociables, exécutés par un seul utilisateur. Elle fait passer la base de données **d'un état cohérent à un autre état cohérent.**
- Si le moindre problème intervient au cours de l'exécution d'une transaction, toutes les mises à jour qui ont été faites à partir du début de la transaction peuvent être annulées.



# LA PERSISTANCE : Les transactions :

## Propriétés d'une transaction

- On utilise l'acronyme ACID pour désigner les propriétés d'une transaction. Une transaction assure :
  - L'**Atomicité** des instructions, qui sont considérées comme indissociables les unes des autres. Soit elles arrivent toutes à terme, soit elles sont toutes annulées.
  - La **Cohérence** de la base de données, qui passe d'un état cohérent à un autre état cohérent.
  - L'**Isolation** des opérations ayant lieu au cours d'une transaction, par le mécanisme des verrous
  - La **Durabilité** des mises à jour, une fois qu'une validation est faite à l'issue d'une transaction, il n'est plus possible de demander l'annulation des mises à jour.

# LA PERSISTANCE : Algorithme d'accès au monde relationnel

## **MODULE accès à la base en Lecture**

**(\* Déclarations du serveur contenant le schéma de la base \*)**

(\* Charger le pilote d'accès à la base (= les définitions du fournisseur du SGBDR permettant de travailler avec le SGBDR \* )

URL caractères 256

NomBase numérique 10

USER caractères 10

VARIABLE Resultat (\* Une seule variable suffit \*)

**(\* Partie exécutable \*)**

SE CONNECTER à la base

CREER UNE REQUETE SQL DE LECTURE

EXECUTER LA REQUETE SUR LE SERVEUR

RECUPERER LE RESULTAT DU SERVEUR (Resultat) (\* Tous les employés \*)

TANT QUE NON FIN\_DE\_FICHER (Resultat)

    LIRE ( 1 ligne du Resultat ) (\* Lecture du prochain employé \*)

    TRAITER la ligne

FIN-TANT-QUE

FERMER la connexion

**FIN-MODULE accès base**

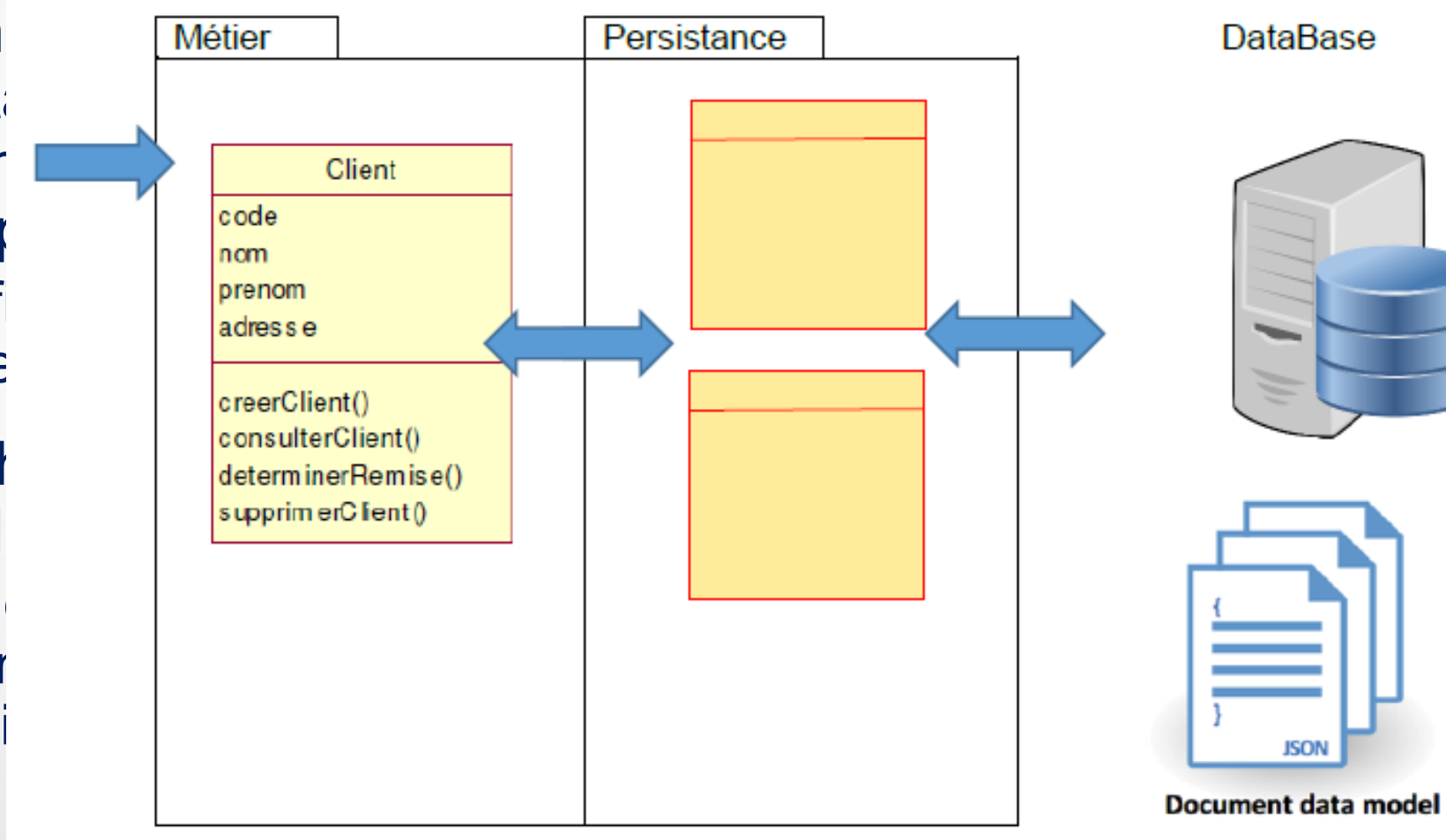
# LA PERSISTANCE : Les transactions :

## Début et fin d'une transaction

- Certains SGBD (comme SQL Server), ont des ordres spécifiques pour démarrer une transaction. Sinon, Le début d'une transaction est toujours implicite. Une transaction **débute** :
  - Dès la première commande de mise à jour rencontrée ; INSERT, UPDATE ou DELETE,
  - Dès la fin de la transaction précédente
- Une transaction se **termine** :
  - sur un ordre de validation de transaction : COMMIT,
  - sur un ordre d'annulation de transaction : ROLLBACK,
  - lors de la fin du processus qui a débuté la transaction : fin normale de session utilisateur avec déconnexion, ou fin anormale d'une session utilisateur (sans déconnexion)
  - sur utilisation de commandes SQL de DDL (Data Definition Language) ou de DCL (Data Control Language).
- Les modifications apportées aux données au cours d'une transaction ne seront visibles dans les transactions concurrentes que lorsque celle-ci sera terminée avec validation des changements.

# LA PERSISTANCE : Impact de la persistance pour le concepteur-développeur

- Le concepteur-développeur doit être sûr que la persistance est gérée correctement.
  - Il suffit de valider la persistance.
- En technologie de base de données.
  - On introduit le terme de Relational Database.



La persistance (ou la persistance peut être abandonnée) est une notion qui est utilisée dans la gestion de la base de données. La plupart des mondes :

# Exercice

- Ecrire un algorithme qui supprime des individus dont le mail est invalide (pour considérer que sont invalides les mails qui ne contiennent qu'une seule arobase ou plus d'une arobase).



## Structure Bottin

Nom en Caractère \* 20

Prénom en Caractère \* 15

Tel en caractère \* 10

Mail en Caractère \* 20

## Fin Structure

Tableau MesPotes() en Bottin

Variable MonPote en Bottin

Variables i, j en Numérique

## Debut

(\*On recopie "Adresses" dans MesPotes en testant le mail...\*)

Ouvrir "Adresse.txt" sur 1 pour Lecture

i ← -1

Tantque Non EOF(1)

LireFichier 1, MonPote

nb ← 0

i ← 1

REPETER

Si Mid(MonPote.Mail, i, 1) = "@" Alors

nb ← nb + 1

FinSi

i ← i + 1

JUSQUE'A i > Len(MonPote.Mail)

Si nb = 1 Alors

i ← MesPotes.LEN + 1

Redim MesPotes(i)

MesPotes(i) ← MonPote

FinSi

FinTantQue

Fermer 1

(\*On recopie ensuite l'intégralité de Fic dans "Adresse« \*)

Ouvrir "Adresse.txt" sur 1 pour Ecriture

j ← 0

REPETER

EcrireFichier 1, MesPotes(j)

j ← j + 1

JUSQUE'A j = i

Fermer 1

Fin

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**



# MODULARITE

## « COMMENT DIVISER POUR MIEUX REGNER »





# MODULARITE : PRESENTATION

- Souvent des tâches analogues sont nécessaires à plusieurs endroits de votre développement : **même traitement** exécuté, mais avec des données différentes (les « paramètres » ou « arguments »).
- Un module est une partie d'un algorithme (ou d'un programme) à laquelle on donne **un nom**, qui servira à appeler son exécution.
- Cette technique a plusieurs intérêts : elle fait correspondre la structure de votre algorithme avec votre analyse, permettant ainsi une lecture « descendante » - ce qui facilite tant la mise au point que l'évolution de votre logiciel.
- De plus, lorsque les traitements sont semblables, vous n'écrirez qu'une seule fois le module correspondant.

# MODULARITE : PRESENTATION

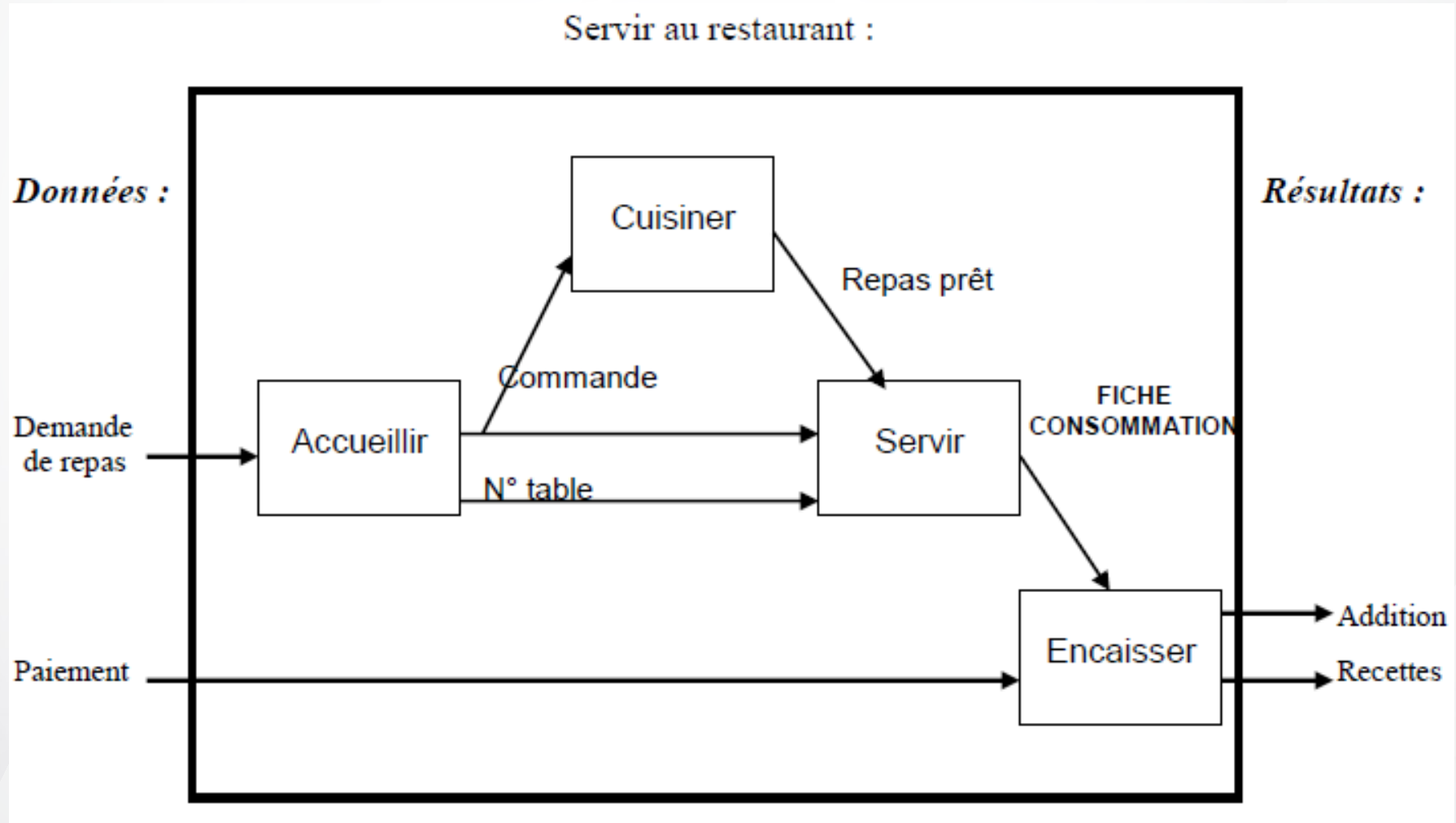
- Vous pouvez même ainsi vous constituer une bibliothèque d'outils généralistes (modules d'accès aux fichiers, de contrôle et de mise en forme des dates...), qui accélèrera vos développements ultérieurs.
- Enfin, la réalisation des modules peut parfaitement être confiée à **plusieurs personnes.**
- Pour définir un module - procédure, fonction ou méthode(\*) - il est donc important de **délimiter** logiquement **l'action** encore plus ou moins complexe qu'il représente (le rôle, la fonction qu'on lui attribue) et **d'indiquer les données** nécessairement connues au début, les données obtenues en résultat

# MODULARITE : FONDEMENTS SYSTEMIQUES

- Mais ces modules, qui seront ensuite examinés isolément, ne sont pas indépendants de tout contexte : ils y puisent des données, transmettent des résultats. Bref, à chaque étape de l'analyse, se pose également la question du **partage des informations** :
- Des données du problème sont inutiles pour la résolution de certains modules. Inversement, certaines données ne seront utiles que pour un module et encombreraient la vue d'ensemble.
- Evidemment, plus le nombre d'informations échangées entre les différents modules est important, plus il est difficile à contrôler : mieux vaut chercher à le minimiser.

# MODULARITE : FONDEMENTS

## SYSTEMIQUES



# MODULARITE : TYPES DE MODULES

- Il existe deux formes traditionnelles de modules :
  - Une **fonction** nomme une **valeur** résultant d'un calcul (arithmétique, booléen...). Elle renvoie donc toujours une valeur unique et équivaut à une opération : son appel peut figurer dans une expression.
    - Exemples :

```
Variable_entiere ← arrondi(v_réelle)
```

```
SI premiers_caractères(mot, 3) = 'ABC' ...
```
  - Une **procédure** désigne une **action** complexe et peut ne retourner aucune, une ou plusieurs valeurs.
  - Son appel équivaut à une nouvelle instruction.
    - Exemples :

```
Controler_saisie
```

```
Saisir(nombre, msg_question, val_mini, val_maxi)
```

# MODULARITE : TYPES DE MODULES

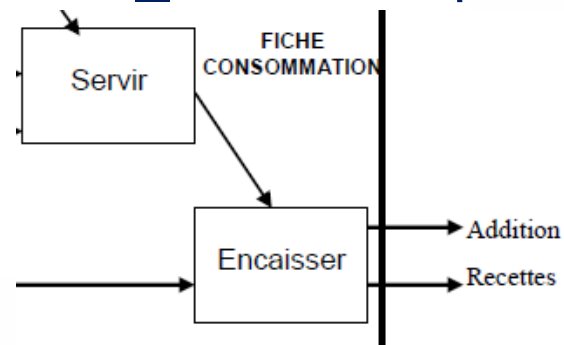


- Il est toujours possible d'écrire une fonction sous forme de procédure (attention : pas l'inverse).
  - C'est une affaire de style.
- **Attention** : Certains langages ne connaissent pas ces notions de fonctions ou procédures, mais différencient simplement celles de « modules » (internes au programme) et de « sous-programmes » (externes au programme).



# MODULARITE : TRANSMISSION DE PARAMETRES

- Il existe différents modes de transmission des arguments :
  - Les **paramètres en entrée** : le module n'utilise ces données qu'en tant que renseignements et ne les modifie pas.
    - Exemples : la fiche consommation (exemple p.4) est en entrée du module « encaisser » ; le mot dont on isole les 1ers caractères, et le nombre de caractères à isoler ; msg\_question, val\_mini, val\_maxi de la procédure « Saisir »

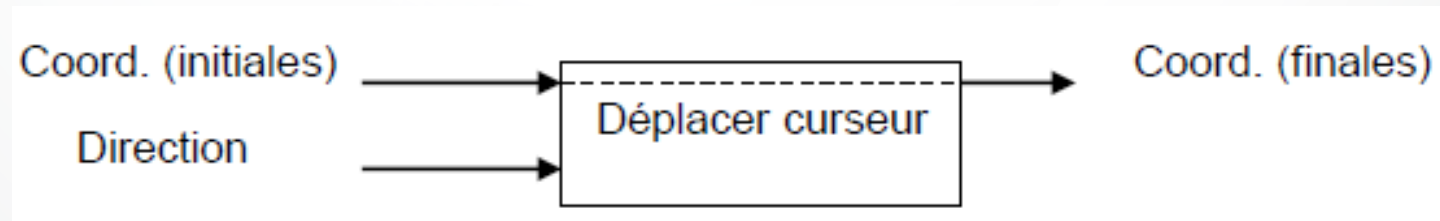


- *Remarque* : Traditionnellement, les paramètres des fonctions sont toujours des paramètres d'entrée : le seul résultat attendu étant émis par la donnée renvoyée par la fonction.



# MODULARITE : TRANSMISSION DE PARAMETRES

- Les **paramètres en sortie** sont entièrement « calculés » par le module – y compris leur valeur initiale.
  - Par exemple : la commande émise par le module « accueillir » ; le nombre saisi
- Les **paramètres en entrée/sortie** ou « **entrées modifiées** » : cette fois la communication est bilatérale.
  - Par exemple, les coordonnées d'un curseur à déplacer :



# MODULARITE : DECLARATIONS / APPELS DE MODULES

- Comme d'habitude, il faudra déclarer ce nouvel objet (le module) avant de pouvoir l'appeler.
- La déclaration d'une fonction a le même aspect que celle d'une procédure, mais il faut en plus :
  - Dans son entête, déclarer le type de la donnée qu'elle représente (la valeur retournée)
  - Dans sa partie exécutable, placer au moins une instruction « RENVOIE(valeur) »
- Dans tous les cas, au moment de la définition d'un module, les paramètres sont « **formels** » puisqu'on ignore encore sur quelles données le module va réellement opérer ; ces paramètres ne peuvent qu'en être des modèles.

# MODULARITE : DECLARATIONS / APPELS DE MODULES

- Au moment de l'appel, il s'agit de donner les paramètres « **effectifs** » qui se substitueront aux paramètres formels lors de l'exécution du module.
- Les deux listes doivent correspondre :
  - même nombre d'arguments,
  - mêmes types,
  - cités dans le même ordre.



*Attention* : Certains langages ne mettent pas en oeuvre ces techniques de transmission de paramètres (voir l'exemple plus loin).

# MODULARITE : Exemples procédure

```
PROCEDURE P_Saisir(en sortie : nombre entier,  
                    en entrée : msg_question caractères,  
                    val_mini entier,  
                    val_maxi entier)
```

```
    ECRIRE msg_question
```

```
    LIRE nombre
```

```
    TANT QUE nombre < val_mini OU nombre > val_maxi
```

```
        ECRIRE « Attention, erreur de saisie. Recommencez »
```

```
        ECRIRE msg_question
```

```
        LIRE nombre
```

```
    FIN-TANT-QUE
```

```
FIN-PROCEDURE
```

```
MODULE xxx
```

```
    CONSTANCE MAX NUMERIQUE 2 = 30
```

```
    VARIABLE nb NUMERIQUE 3
```

```
    .../...
```

```
    P_Saisir (nb, « Donnez le nombre d'articles », 1, MAX )
```

```
FIN-MODULE xxx
```

**Appel de la procédure**

# MODULARITE : Exemples fonction

```
FONCTION F_Saisir(  msg_question caractères,  
                    val_mini entier,  
                    val_maxi entier) RENVOIE un entier
```

```
Nombre : entier
```

```
ECRIRE msg_question
```

```
LIRE nombre
```

```
TANT QUE nombre < val_mini OU nombre > val_maxi  
    ECRIRE "Attention, erreur de saisie. Recommencez"  
    ECRIRE msg_question  
    LIRE nombre
```

```
FIN-TANT-QUE
```

```
RENVOIE (nombre)
```

```
FIN-FONCTION
```

```
MODULE xxx
```

```
    CONSTANCE MAX = 30 .../...
```

```
    SI F_Saisir (« Donnez le nombre d'articles », 1, MAX ) = 10  
        .../...
```

```
    FIN SI
```

```
FIN-MODULE xxx
```

**Appel de la fonction**

# MODULARITE : IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS

- Chaque module pouvant avoir besoin d'objets qui lui sont propres (constantes, types, variables, mais aussi procédures et fonctions), on aboutit à une construction imbriquée qui détermine différents **niveaux de visibilité** des identificateurs(\*) :
- Les identificateurs déclarés dans un module (paramètres formels compris) sont « **locaux** » : ils ne sont connus qu'à l'intérieur de celui-ci – pas des modules de même niveau, ni de niveau supérieur.
  - Par contre, un module connaît les identificateurs « **globaux** », déclarés au(x) niveau(x) supérieur(s).
  - Lorsqu'il y a homonymie d'identificateurs dans deux blocs de niveaux différents, le dernier masque les premiers (la référence concerne l'objet déclaré dans le bloc le plus proche en remontant l'imbrication).

Entête du programme principal

Constante zero

Module1 (param1)

Module11(param11)

Variable onze

Partie exécutable

Module12(param12)

Variable douze

Partie exécutable

Variable un

Partie exécutable

Module2 (param2)

Variable deux

Partie exécutable

Variable zerobis

Partie exécutable du programme principal

# MODULARITE : IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS

- Ainsi, dans ce schéma :
- Le module principal connaît :
    - zero, module1, module2 et zerobis
  - Le module1 connaît :
    - zero, module1, param1, module11, module12, un
  - Le module11 connaît :
    - zero, module1, param1, module11, param11, onze
  - Le module12 connaît :
    - zero, module1, param1, module12, param12, douze
  - Le module2 connaît :
    - zero, module2, param2, deux

Entête du programme principal

Constante zero

Module1 (param1)

Module11(param11)

Variable onze

Partie exécutable

Module12(param12)

Variable douze

Partie exécutable

Variable un

Partie exécutable

Module2 (param2)

Variable deux

Partie exécutable

Variable zerobis

Partie exécutable du programme principal



# MODULARITE : IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS



- Puisqu'un module peut accéder aux variables globales, il peut paraître redondant de les définir comme des arguments. Mais comment pourrez-vous utiliser ce module dans un autre contexte ?
- De plus, en paramétrant systématiquement les données partagées, vous obtiendrez une véritable description fonctionnelle du module, rien qu'en consultant son entête.

# MODULARITE : IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS

- Certains langages n'admettent ni procédure, ni fonction, ni passage de paramètres. Dans ce cas, le module appelant :

```
MODULE Saisir
  ECRIRE msg_question
  LIRE nombre
  TANT QUE nombre < MAX OU nombre > MIN
  |   ECRIRE "Attention, erreur"
  |   ECRIRE msg_question
  |   LIRE nombre
  FIN-TANT-QUE
FIN-MODULE
```

```
MODULE xxx
  CONSTANTE MAX = 30
  CONSTANTE MIN = 30
  VARIABLE msg_question CARACTERES 30
  VARIABLE nombre NUMERIQUE 3
  VARIABLE nb_articles NUMERIQUE 3
  VARIABLE nb_fournisseurs NUMERIQUE 3
  .../...
  msg_question ← "Donnez le nombre d'articles"
  APPEL Saisir
  nb_articles ← nombre
  .../...
  msg_question ← "Donnez le nombre de fournisseurs"
  APPEL Saisir
  nb_fournisseurs ← nombre
FIN-MODULE xxx
```

# LES FONCTIONS PRÉDÉFINIES

- Tout langage de programmation propose ainsi un certain nombre de **fonctions** ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs – et pénibles - algorithmes.
- **Les fonctions de texte**
  - Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères
  - Tous les langages, je dis bien tous, proposent peu ou toutes les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

# LES FONCTIONS PRÉDÉFINIES

- **Len(chaine)** : renvoie le nombre de caractères d'une chaîne
  - `Len("Bonjour, ça va ?")`                      vaut              16
  - `Len("")`    vaut              0
- **Mid(chaine,n1,n2)** : renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long.
  - `Mid("Zorro is back", 4, 7)`                      vaut              "ro is b"
  - `Mid("Zorro is back", 12, 1)`                      vaut              "c"
- Ce sont les deux seules fonctions de chaînes réellement indispensables.
- Remarque : Mid peut aussi dans certains langages se nommait SUBSTRING

# LES FONCTIONS PRÉDÉFINIES

- Pour nous épargner des algorithmes fastidieux, les langages proposent également :
  - **Left(chaine,n)** : renvoie les n caractères les plus à gauche dans chaîne.
    - `Left("Et pourtant...", 8)`                      vaut                      `"Et pourt"`
  - **Right(chaine,n)** : renvoie les n caractères les plus à droite dans chaîne
    - `Right("Et pourtant...", 4)`                      vaut                      `"t..."`
  - **Trouve(chaine1,chaine2)** : renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.
    - `Trouve("Un pur bonheur", "pur")`                      vaut                      4
    - `Trouve("Un pur bonheur", "techno")`                      vaut                      0

# LES FONCTIONS PRÉDÉFINIES

- Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction **Asc**), et vise versa (fonction **Chr**) :

- `Asc ("N")`

vaut 78

- `Chr (63)`

vaut "?"



- A moins de programmer avec un langage un peu particulier, comme le C, qui traite en réalité les chaînes de caractères comme des tableaux, on ne pourrait pas se passer des deux fonctions **Len** et **Mid** pour traiter les chaînes.



# Exercice

- Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces (et c'est déjà un petit peu moins bête).



```
Variable Bla en Caractère
Variables Nb, i en Entier
Debut
    Ecrire "Entrez une phrase : "
    Lire Bla
    Nb ← 0
    Pour i ← 1 à Len(Bla)
        Si Mid(Bla, i , 1) = " " Alors
            Nb ← Nb + 1
        FinSi
    fin Pour
    Ecrire "Cette phrase compte ", Nb + 1, " mots"
Fin
```

# Exercice

- Ecrivez une fonction qui purge une chaîne d'un caractère, la chaîne comme le caractère étant passés en argument. Si le caractère spécifié ne fait pas partie de la chaîne, celle-ci devra être retournée intacte. Par exemple :
- `Purge("Bonjour","o")` renvoie "Bnjour"
  - `Purge("J'ai horreur des espaces", " ")` renvoie "J'ai horreur desespaces"
  - `Purge("Moi, je m'en fous", "a")` renvoie "Moi, je m'en fous"



```
Fonction PurgeSimple(a, b)
  Variable Sortie en Caractère
  Variable i en Numérique
  Début
    Sortie ← ''
    Pour i ← 1 à Len(a)
      Si Mid(a, i, 1) <> b Alors
        Sortie ← Sortie & Mid(a, i, 1)
      FinSi
    fin Pour
  Renvoyer Sortie
FinFonction
```

# SOMMAIRE

- **Chapitre 1 : INTRODUCTION : QUELQUES DEFINITIONS**
- **Chapitre 2 : CONSTITUANTS D'UN ALGORITHME**
- **Chapitre 3 : FORMALISME**
- **Chapitre 4 : ALGEBRE DE BOOLE**
- **Chapitre 5 : COMPLEMENTS ALGORITHMIQUES**
- **Chapitre 6 : FICHIERS**
- **Chapitre 7 : MODULARITE**
- **Chapitre 8 : LES TESTS**

# LES TESTS



# TYPES DE TESTS

- Nous ne distinguerons ici que deux **types de tests** :
  - Les tests « **unitaires** », ou tests de module, qui ont pour objet de contrôler que le module est réalisé conformément à ses spécifications.
  - Les tests « **d'intégration** », l'assemblage des composants du logiciel pouvant lui-même être source d'erreurs si les liens entre les différents modules n'ont pas été préalablement correctement définis.
- Ceux-ci sont particulièrement importants lorsque la programmation est événementielle (intégration, enchaînement de fenêtres).



# TECHNIQUES DE CONCEPTION

- Deux familles de **techniques de conception** de tests existent :
  - L'approche « **boîte noire** » permettant de réaliser des tests fonctionnels : on s'intéresse uniquement au comportement externe du module (description de ses entrées / sorties).
  - L'approche « **boîte blanche** » pour des tests structurels : cette fois, on veut contrôler non plus ce que fait le module mais la manière dont il le fait, ce qui implique de connaître sa logique interne (instructions, débranchements, conditions...).



# ETAPES DE CONCEPTION DES TESTS

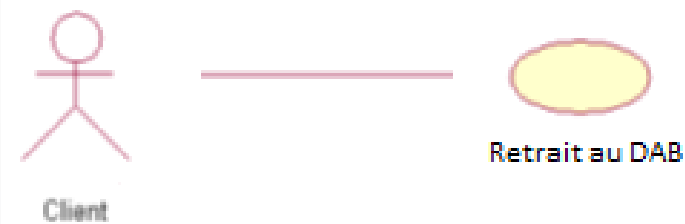
- La phase de tests peut être décomposée en **plusieurs étapes** :
- 1. Préparation des tests :
  - Définir les situations à tester. En théorie, elles doivent représenter l'exhaustivité des cas possibles pour le programme.
  - Déterminer les valeurs des données d'entrée permettant de contrôler ces situations.
  - Décrire explicitement et entièrement les résultats attendus à l'issue de l'exécution.
- 2. Exécution des tests et comparaison des résultats obtenus avec ceux attendus
- 3. Correction :
  - Localisation et définition de la nature exacte des défauts.
  - Correction des défauts. Et dans ce cas, retour au 2 : refaire tous les tests pour s'assurer qu'une « correction » n'a pas tout cassé ! « **non régression** » jusqu'à obtenir le résultat attendu.
- Il faut évidemment recommencer ces 2 dernières étapes jusqu'à ce qu'il n'y ait plus d'erreur !

# ETAPES DE CONCEPTION DES TESTS

- Dans **les technologies objet**, il est défini de mettre en œuvre **des cas d'utilisation ou Use Case**.
- Les cas d'utilisation sont des descriptions textuelles, utilisées pour détecter et consigner les besoins. Ils montrent les interactions fonctionnelles entre les acteurs et le système à l'étude.
- **Le scénario** est un ensemble d'actions qui permet d'atteindre l'objectif du cas d'utilisation.
  - Il existe plusieurs scénarii pour un cas d'utilisation et on identifie :
    - Le scénario normal ou principal : séquence d'actions la plus fréquente dans le cas où tout se passe bien.
    - Les scénarii alternatifs : séquences d'actions qui permettent d'aller du début à la fin normale du cas d'utilisation et différentes du cas normal.
    - Les scénarii d'exceptions : séquences d'actions qui ne permettent pas de terminer normalement le cas d'utilisation.
- Dans sa description au niveau le plus fin et en faisant intervenir les instances des objets impliqués, les scénarii obtenus à ce niveau sont les scénarii utilisés pour la **recette utilisateur**.
- **Les utilisateurs détiennent la connaissance métier et peuvent donc fournir des jeux d'essais** qui seront utilisés par le concepteur développeur.
  - Les cas d'utilisation sont intégrés aux outils de développements. Et sont réutilisés pour les tests de non régression.

# ETAPES DE CONCEPTION DES TESTS

➤ **Exemple de scénarii et de test d'un cas d'utilisation** pour un client d'une banque souhaitant retirer de l'argent au DAB (Distributeur Automatique de Billet)



- Test fourni par l'utilisateur :
- 1- Le client Dupont de la banque insère sa carte n°14571254.

2- Il tape son code 1025.

3- Le système valide.

4- Il tape le montant demandé : 100 Euros.

Cas d'utilisation	Retrait d'argent à un DAB
Début du cas d'utilisation	Le client insère sa carte de crédit dans le DAB
Fin du cas d'utilisation	Le client obtient l'argent demandé
Acteurs	Le client, le système interbancaire
Scénarii	
Scénario normal	« tout est normal »
	1 Le client insère sa carte
	2 Le client tape son code confidentiel et valide
	3 Le système accepte
	4 Le client tape son montant et valide
	5 Le système autorise la somme demandée
	6 Le client retire sa carte, son argent et son reçu
Scénario alternatif 1	« le client se trompe une fois dans la saisie de son code »
	1.. 2 idem
	A 1 Le système refuse
	A 2 Le client tape à nouveau son code et valide.
	3.. 6 Idem.
Scénario alternatif 2	« le crédit disponible autorisé est inférieur à la somme demandée »
	1.. 4 idem
	B 1 Le système avertit le client qu'il ne peut pas lui donner la somme désirée
	B 2 Le client tape un nouveau montant et valide
	5.. 6 idem
.../...	
Scénario d'exception 1	« le client se trompe 3 fois dans la saisie de son code »

# EXERCICE : LA COMPETITION INTERNATIONALE

- Cahier d'exercices
- Indications : on utilisera des modules



# Correction : LA COMPETITION INTERNATIONALE

## (\* Partie déclarative \*)

FICHIERS

paysA séquentiel ACCES séquentiel

paysB séquentiel ACCES séquentiel

resultat séquentiel ACCES séquentiel

VAR

perfA, perfB numériques 3 dont 1 décimale

Notes : ouvrir, fermer, fusionner sont des modules sans argument.

Recopier est un modules avec 3 arguments

## (\* Partie exécutable \*)

ouvrir\_fics

fusionner (\* jusqu'à ce qu'un des 2 fichiers soit entièrement parcouru \*)

(\* Recopier la fin de l'autre fichier \*)

SI FIN\_DE\_FICHER(paysA)

recopier(« paysB », perfB, « resultat »)

SINON

recopier(« paysA », perfA, « resultat »)

FIN\_SI

fermer\_fics



# Correction : LA COMPETITION INTERNATIONALE

Notes : ecrire-lire est un module avec 3 arguments

## **PROCEDURE ouvrir\_fics**

(\* Données globales : fichiers du module competition \*)

```
OUVRIR paysA  EN LECTURE
OUVRIR paysB  EN LECTURE
OUVRIR resultat EN CREATION
```

**FIN ouvrir\_fics**

## **PROCEDURE fermer\_fics**

(\* Données globales : fichiers du module competition \*)

```
FERMER paysA
FERMER paysB
FERMER resultat
```

**FIN fermer\_fics**

## **PROCEDURE fusionner**

(\* Données globales : du module competition \*)

```
LIRE(paysA, perfA)
```

```
LIRE(paysB, perfB)
```

```
TANT QUE  NON ( FIN_DE_FICHER(paysA) ) ET
           NON ( FIN_DE_FICHER(paysB) )
```

```
    SI perfA < perfB
```

```
        ecrire_lire (« paysA », perfA, « resultat »)
```

```
    SINON
```

```
        ecrire_lire (« paysB », perfB, « resultat »)
```

```
    FIN_SI
```

```
FIN_TANT_QUE
```

**FIN fusionner**



# Correction : LA COMPETITION INTERNATIONALE

**/\* Outils \*/**

## **PROCEDURE recopier**

**(En entrée : fic\_source chaine, (\* le nom du fichier \*))**

**En entrée modifiée : enregistrement numérique,**

**En entrée : fic\_cible chaine)**

**TANT QUE NON FIN\_DE\_FICHER(fic\_source)**

**ecrire\_lire (fic\_source, enregistrement, fic\_cible)**

**FIN\_TANT\_QUE**

**FIN recopier**

## **PROCEDURE ecrire\_lire**

**(En entrée : fic\_source chaine,**

**En entrée modifiée : enregistrement numérique,**

**En entrée : fic\_cible chaine)**

**ECRIRE (fic\_cible, enregistrement)**

**LIRE (fic\_source, enregistrement)**

**FIN ecrire\_lire**

