TD Algorithme et programmation

J.-F. Remm & J.-F. Berdjugin & J.-F. Dufayard
The Djeff's Team

3 avril 2006

Table des matières

1 Introduction

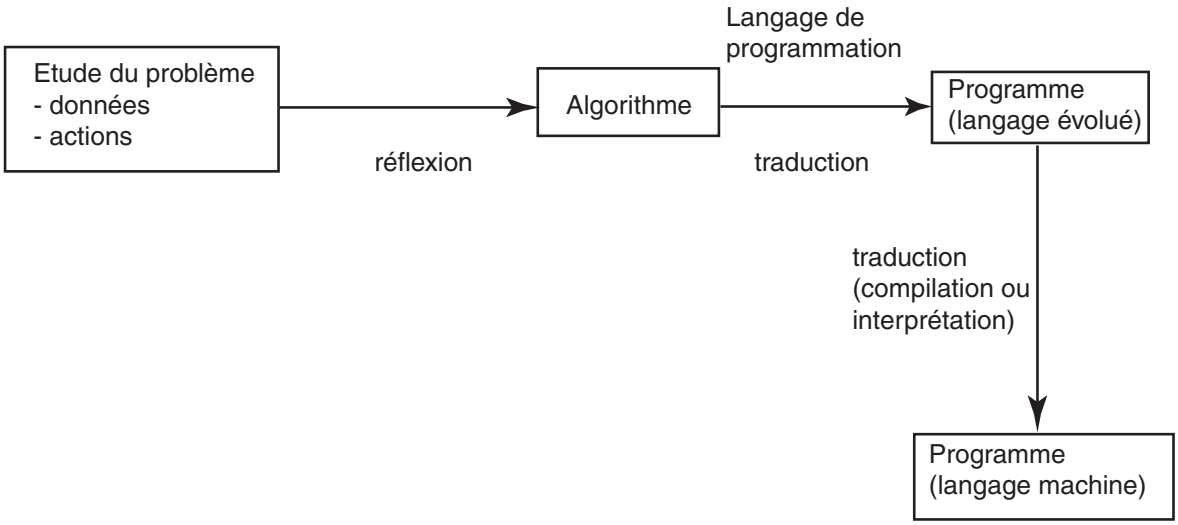

Fig. 1 – Principe

Définition 1 (Algorithme (Petit robert))
Suite finie séquentielle de règles que l'on applique à un nombre fini de données, permettant de résoudre des classes de problèmes semblables.

Définition 2 (Algorithme)
Calcul, enchaînement des actions nécessaires à l'accomplissement d'une tâche.

Définition 3 (Algorithme (Vaccari))
Transformation d'un problème en une suite ordonnée d'opérations comportant un nombre fini d'étapes permettant de résoudre le problème

Définition 4 (Programme)
Suite d'instructions données à l'ordinateur et codées dans un langage de programmation compréhensible (après traduction) par l'ordinateur.

Définition 5 (Instruction)
On distingue deux types d'instructions :

1. des instructions de traitement de l'information (voir section 3 page 4)
2. des instructions destinées à commander le déroulement du programme (voir section 4 page 8)

2 Structure générale

2.1 Préambule

Syntaxe

Pseudo langage	Java	Commentaire

Algo *nomAlgo*
déclarations
Début
instructions
Fin
	class nomAlgo
	{
	public static void main(String[] args)
	{
	déclarations
	instructions
	}
	}	

2.2 Variable – constante

Définition 6 (Donnée)
Toute **donnée** est soit une **variable**, soit une **constante**.

Définition 7 (Variable – constante)
Une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme. À l'opposé, une **constante** va conserver la même valeur tout au long du traitement. Chaque donnée (variable ou constante) est définie par :

- un **identificateur** : c'est le nom que l'on lui donne
- un **type** (voir section 2.3)
- une **valeur**

2.3 Type

Définition 8 (Type)
La **valeur** d'une variable est codé en binaire. À une **nature** (numérique, caractère,...) de l'information mémorisée correspond généralement une manière de coder cette information. Le type d'une variable permet d'associer entre eux, la **nature** des informations, le **codage** mais aussi les **limites** et **opérations** associés.

Définition 9 (Opérateur)
Un opérateur est un symbole indiquant une opération à effectuer. Les opérateurs sont bien souvent binaires – reliant deux opérandes –, mais peuvent aussi être unaires ou ternaires. Un opérateur est bien souvent associé à un type de données. On peut classer la plupart des opérateurs dans trois grandes familles :

1. opérateurs **arithmétiques** : donnent un résultat numérique à partir d'opérandes numériques (addition, soustraction,...)
2. opérateurs **relationnels** : donnent un résultat logique à partir d'opérandes numériques (plus grand que, égal, ...)
3. opérateurs **logiques** : donnent un résultat logique à partir d'opérandes logiques (et logiques, ...)

Exemple 1
3 + 3 : +, opérateur binaire d'addition
a `ET` b : `ET`, opérateur binaire
-c : - négation unaire
`NON`(a `OU` b) : `NON`, opérateur unaire.

En algorithmique, nous utiliserons les types suivants :

2.3.1 `booléen`

valeurs : `vrai` ou `faux`

opérateurs : `ET`, `OU` et `NON`

2.3.2 `entier` ou `réel`

valeurs : n'importe quelle valeur entière ou réelle

exemple : 18 -3,5

opérateurs : addition `+`, soustraction `-`, multiplication `*`, division `/`, modulo (reste de division entière) `%`, comparaison $>, \geq, \leq, <, =, \neq$

2.3.3 `caractère` ou `chaîne` de caractère

valeurs : lettres, chiffres, ...

exemple : "bonjour", 'a'

opérateur : concaténation `+`

2.3.4 `vide`

valeurs : aucunes

opérateur : aucun

3 Instruction

3.1 Déclaration

L'utilisation d'une variable ou d'une constante sera TOUJOURS précédé d'une **déclaration**. Cette déclaration permet :

- d'associer formellement un identificateur à un type
- d'éviter l'utilisation multiple d'un nom pour deux données différentes (double déclaration).

Syntaxe

Pseudo langage	Java	Commentaire

Algo Decl
var A,B : entier
const C ← 0.5 : réel
Début
 	
Fin
	public class Decl
	{
	public static void main(String[] args)
	{
	int A, B;
	final double C = 0.5;
	}
	}
		`A` et `B` sont des variables de type entier, `C` est un constante de type réel.

3.2 Commentaires

Syntaxe

Pseudo langage	Java	Explication
// commentaire	`// commentaire`	le commentaire n'est **pas** exécuté

3.3 Affectations

Définition 10 (Affectation)
Une affectation $x \leftarrow expr$ est une instruction qui permet de spécifier qu'au moment de son exécution, la variable x recevra comme nouvelle valeur la valeur de l'**expression** `expr` spécifiée en partie droite de l'instruction.

Définition 11 (Expression)
Une **expression** permet de désigner le calcul d'une "nouvelle" valeur à partir d'autres valeurs et d'opérations. Les valeurs utilisés dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales, ...

Exemple 2
3 + 27
x * 3 // si x est une variable numérique
...

Syntaxe

Pseudo langage	Java	Commentaires
identif ← *expr*	`identif = expr;`	

Exercice 1
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect1
var A : entier
var B : entier
Début
A ← 5
B ← A+1
A ← 2

Fin

Exercice 2
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect2
  var A : entier
  var B : entier
Début
  A ← 2
  A ← A+1
Fin

Exercice 3
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect3
  var A : entier
  var B : entier
Début
  B ← A+1
  A ← 2
Fin

Exercice 4
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect4
  var A : entier
  var B : entier
Début
  A+5 ← 3
Fin

Exercice 5
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect5
  var A : entier
  var B : entier
Début
  A ← 5
  B ← A+4
  A ← A+1
  B ← A-4
Fin

Exercice 6
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect6
  var A : entier
  var B : entier
  var C : entier

Début
A ← 3
B ← 10
C ← A+B
B ← A+B
A ← C
Fin

Exercice 7
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect7
var A : entier
var B : entier
Début
A ← 5
B ← 7
A ← B
B ← A
Fin

Exercice 8
Donner le résultat de l'exécution de l'algorithme suivant :

Algo Affect8
var A : entier
var B : entier
Début
A ← 5
B ← 7
B ← A
A ← B
Fin

Exercice 9
Écrire un algorithme permettant d'inverser deux variables.

3.4 Entrées/Sorties

Un programme a généralement besoin de communiquer avec l'extérieur :
- affichage de résultats sur un écran
- demande à l'utilisateur de fournir une donnée
- ...

Dans la réalité les instructions d'**écriture** et de **lecture** se font sur des périphériques. Une lecture spécifie qu'une nouvelle valeur d'un variable doit être lue sur un périphérique (typiquement à l'aide du clavier). Une lecture a comme avantage sur une affectation le fait de rendre le programme indépendant des données. I.e. le même programme peut s'exécuter sur des données différentes. Une écriture permet d'écrire sur un périphérique (typiquement l'écran) la valeur d'une variable, d'une constante ou d'une expression.

Syntaxe

Pseudo langage	Java	Commentaires
Lire (*variable*) Ecrire (*variable*)	lecture : compliquée écriture : `System.out.println(`*variable*`)`	voir tp 15 pour la lecture ...

4 Structure de contrôle

Jusqu'à maintenant, les instructions s'exécutaient dans l'ordre de leur apparition. Or une des caractéristiques essentielle de l'ordinateur (que l'on retrouve dans le comportement humain) est de pouvoir :

- faire un choix (qui dépend par exemple d'une saisie de l'utilisateur)
- répéter plusieurs fois la même série d'instructions

Définition 12 (Structure de contrôle)
Une structure de contrôle est une instruction destinée à commander le déroulement du programme.

4.1 Enchaînement/Séquence

Définition 13 (Enchaînement/Séquence)
Structure de base, qui permet d'enchaîner deux instructions.

Est-il besoin d'en dire plus ?

4.2 Conditionnelles

Définition 14 (Conditionnelle)
Appelée aussi choix, cette structure permet d'exécuter une série d'instructions plutôt qu'une autre en fonction du résultat d'un test (expression booléenne).

Syntaxe

Pseudo langage	Java	Commentaires

Si (*condition*) **Alors**
instructions1
Sinon
instructions2
Fin Si
	if (condition)
	{
	instructions1
	}
	else
	{
	instructions2
	}
		conditionnelle est une expression s'évaluant en valeur **booléenne**
		on peut omettre la clause `else`.

Exercice 10
Écrire un algorithme qui lit deux valeurs entière et affiche le maximum des deux.

Exercice 11 (Calcul de remise (1))
A la caisse d'un supermarché (bien connu à l'Isle d'abeau), nous bénéficions d'une remise de 1% sur le `montant` de nos achat lorsque celui-ci dépasse 300 euros. Écrire un algorithme qui après lecture du montant initialement du, affiche le montant à payer.

Remarques :
- maladroit d'avoir plus d'une instruction d'écriture du résultat.
- maladroit d'avoir une clause `sinon`
- inutile d'avoir recours à une autre variable.

Exercice 12 (Calcul de remise (2))
Même exercice avec :
- 1% de remise pour un achat compris entre 300 et 750 euros
- 2% au delà de 750 euros

Exercice 13
Lire trois valeurs entières A, B et C. Afficher le maximum des trois

Exercice 14 (Calcul d'une facture d'électricité)
Trouver le prix à payer sachant qu'une facture inclut une somme de 4 euros de frais fixes et que s'ajoute un prix en fonction de la consommation :
- 0,1 euro/kWH pour les 100 premiers kilowatts heures
- 0,07 euro/kWH pour les 150 suivants
- 0,04 euro/kWH au delà

Exercice 15 (Estimation du prix de revient d'un véhicule)
Il existe un barème pour l'évaluation du prix de revient kilométrique des véhicules. Écrire un algorithme effectuant le calcul de ce `prix` en fonction de `nb`, nombre de kilomètres parcourus .
Règles :

puissance fiscale / nb de km	5CV	6CV
jusqu'à 5000	n1 * 0,43 (= p1)	n * 0,47
de 5001 à 20000	(n2 * 0,23) + p1 (=p2)	(n * 0,27) + 1000
au delà de 20000	(n3 * 0,28) + p2	n * 0,32

où `n` est le nombre total de kilomètres parcourus, `n1` le nombre de kilomètres parcourus entre 0 et 5000, `n2` le nombre de kilomètres parcourus entre 5001 et 20000 et `n3` le nombre de kilomètre parcourus au delà de 20000. Exemple : si j'ai parcouru 8500 km, `n`=8500, `n1`=5000, `n2`=3500 et `n3`=0.

Exercice 16 (Le monnayeur)
Un distributeur qui rend de la monnaie doit rendre en priorité les pièces les plus grosses. En supposant que la machine rends des jetons de 5, 2 et 1 unités et qu'elle doit vous rendre `nb` unités, écrire un algorithme qui simule le rendu. On suppose que la caisse de départ de la machine est **illimitée**. I.e. il y a toujours assez de jetons en caisse pour le rendu.

Exercice 17
Même exercice avec une caisse de départ **limitée**

4.3 Boucle – structure itérative

Les boucles sont des structures de données permettant de répéter plusieurs fois une séquence d'instructions. Cette répétition est bien souvent **conditionnelle** (c'est à dire qu'elle dépend d'une condition).

4.3.1 Tant que

On répète une série d'instructions tant qu'une condition est vrai.

Syntaxe

Pseudo langage	Java	Commentaires

Tant que (*condition*) **Faire**
instructions
Fin Tant que
	while (condition)
	{
	instructions
	}
		condition est une expression **booléenne** (dont la valeur est vrai ou faux). Tant que la condition est vrai, on répète `instructions`

4.3.2 Répéter/Jusqu'à

Syntaxe

Pseudo langage	Java	Commentaires

Répéter
instructions
Jusqu'à (*condition*)
	do
	{
	instructions
	}
	while (!condition) ;
		condition est une expression s'évaluant en valeur **booléenne**
		on répète `instructions` jusqu'à ce que la condition devienne vrai.

La différence essentielle entre ces deux structures itératives est le moment où est évalué le test. L'usage de "répéter" fait passer au moins **une fois dans la boucle** ; la condition de terminaison n'est évalué qu'à l'issue du passage. À l'opposé, avec "tant que" la condition de terminaison est évalué avant toute entrée dans la boucle. Il peut donc n'y avoir **aucune exécution** de la boucle.

Exemple 3

Algo Boucle1
  var a,b : entier
Début
  Répéter
    Lire(a)
    b ← a*a
    Ecrire(b)
  Jusqu'à (a = 0)
Fin

Exercice 18
Exécuter l'algorithme suivant.

```
Algo Boucle2
  var a : entier
Début
  a ← 5
  Tant que (a > 0) Faire
    Ecrire(a)
    a ← a - 2
  Fin Tant que
Fin
```

Exercice 19
Lire un caractère et l'afficher jusqu'à ce que l'on saisisse `'y'`. Réaliser deux versions cet algorithme :

1. une version avec affichage du `'y'` final
2. une version sans affichage du `'y'` final

Exercice 20
Écrire un algorithme qui affiche les `n` premiers entiers (de 1 à `n`).

Exercice 21
Écrire un algorithme qui affiche la somme des `n` premiers entiers

Exercice 22 (Remboursement d'emprunt)
Calculer le nombre d'années nécessaires au remboursement d'un emprunt à taux d'intérêt fixe et dont le remboursement annuel est fixe également. (Attention, le remboursement de la première année doit être strictement supérieur à l'intérêt payé la première année)

Exercice 23 (Codage)
Coder un nombre *nb* écrit en base 10 en base *base* (*base* < 10). Rappel : il faut faire les divisions successives de *nb* par *base* jusqu'à obtenir un quotient nul.
NB : on formate le résultat par une concaténation de chaînes.

Exercice 24
Écrire un algorithme qui fasse deviner un nombre entier `aTrouver` en donnant des indications (trop grand, trop petit) avec `nbEssai` autorisé. Il faut obtenir un affichage final `Gagné !` ou `Perdu !`.

Exercice 25 (Décomposition en facteurs premiers)
Décomposer un `nombre` en nombre premiers. Essayer les divisions du nombre par les tous les entiers (à partir de 2) et faire afficher simplement les différent diviseur. N.B. On effectue les divisions du nombre par les différents entiers, qu'ils soient premiers ou no, de toute façon, un nombre qui n'est pas premier ne pourrait diviser car tous ses diviseurs (plus petit que lui) auraient précédemment divisé le nombre.

Exercice 26 (Décomposition en facteurs premiers (subsidiaire))
Même exercice mais avec affichage des puissances.

4.3.3 Pour

Syntaxe

Pseudo langage	Java	Commentaires

Pour *var* **de** *debut* **à** *fin*
Faire
instructions
Fin Pour
	for(int var=debut ;var<=fin ;var++)
	{
	instructions
	}
	
	for(int var=debut ;var>=fin ;var−−)
	{
	instructions
	}
		la boucle est répétée pour *var* valant successivement *debut*, *debut+1*, ..., *fin* ou pour *var* valant successivement *debut*, *debut-1*, ..., *fin*

Exercice 27
Écrire un algorithme qui affiche la somme des `n` premiers entiers

Exercice 28 (Placement d'argent)
Si l'on place `somme` euros au 1 janvier de l'année `anDepot` à `taux`% (en accumulant les intérêts) , quelle va être la somme présente sur le compte le 1 janvier de de l'année `anRetrait` ?

4.4 Fonction – procédure

Un algorithme, si on lui conservait sa structure monolithique, pourrait comporter plusieurs centaines de lignes et deviendrait rapidement illisible. Un bon algorithme (et le futur programme issu de la traduction de l'algorithme) se doit d'être compris rapidement.

Il faut donc modulariser l'écriture d'algorithmes de manière à rendre un algorithme :
- court
- clair
- compréhensible
- cohérent

Ces qualités permettront de rendre un algorithme réutilisable, maintenable, ...

Définition 15 (Sous-algorithme/sous-programme)
Un sous programme est un ensemble d'instructions, calculant un certain nombre de résultat en fonction d'un certain nombre de données. On appelle **argument** ou **paramètre** les données et résultats du sous programme.

Définition 16 (Définition d'un sous-programme)
La **définition** d'un sous-programme comporte :
- une **signature** ; l'indication de son nom, du nombre et types des paramètres à fournir et éventuellement type du résultat,
- un **corps** ; la description des actions à enchaîner.

Les paramètres sont représentés par des noms quelconques appelés **paramètres formels**.

Définition 17 (Appel d'un sous-programme)
L'**appel** (ou exécution) d'un sous-programme est l'invocation du sous-programme par appel de son nom en fournissant des **données** (constantes, valeurs de variables, résultats d'un autre sous-programme,...) en nombre et type requis. Les données fournies sont appelées **paramètres effectifs**.

Définition 18 (Mode de transmission des paramètres (non exhaustif))
- passage par **valeur** : c'est la valeur (donc une copie) de la donnée qui est transmise. La donnée est transmise en lecture seule.
- passage par **référence** (ou par adresse) : c'est l'adresse de la donnée qui est transmise. La donnée est transmise en lecture/écriture.

N.B. 1 En java, aussi bien les types primitifs que les types références (objets et tableaux) sont passés par valeur. Cependant "extérieurement", comme c'est la références qui est passée, cela s'apparente presque à un passage par adresse. C'est à dire que l'état d'un objet peut être changé par l'application de méthodes s'appliquant à celui-ci, mais la référence à l'objet, contenue dans la variable, reste la même.

Définition 19 (Fonction)
Une **fonction** est un sous programme qui transmet un résultat. Il est nécessaire d'avoir un moyen de spécifier quelle est la valeur de ce résultat.

Définition 20 (Procédure)
Une **procédure** est un sous programme qui ne transmet aucun résultat. Pour nous, une procédure sera un type de fonction particulière, dont le résultat est d'un type de donnée particulier : `vide`.

Définition 21 (Méthode)
Une **méthode** est une fonction ou procédure associée à une classe d'objets particulière qui ne peut être appelée que par l'envoi d'un message. Un méthode est définie dans une classe (voir section 5.1 page 15).

Syntaxe

Pseudo langage

```
Fonction nomFonc(param1 : type1 [, param2 : type2] ) : typeRetour
  instructions
  Renvoie (resultat) // si nécessaire
Fin Fonction
Algo Appel

Début
  nomFonc(val1,val2)
Fin
```

Java

```
class Appel
{
    static typeRetour nomFonc (type1 param1,
                               type2 param2)
    {
        instructions
        return (resultat) ;
    }

    public static void main(String[] args)
    {
        nomFonc(val1,val2) ;
    }
}
```

Exemple 4
Définition de la fonction `carre` et son appel.

```
Fonction carre (nombre : entier) : entier
  var res : entier
  res ← nombre * nombre
  Renvoie(res)
Fin Fonction
```

```
Algo ExAppelCarre
  var r : réel
Début
  Lire(r)
  Ecrire(carre(r))
Fin
```

Exercice 29 (Calcul d'impôts)
Les impôt sur le revenu d'un contribuable dépendent :

- du nombre de personnes présentes dans le foyer (0,5 pour les deux premiers enfant, 1 pour les suivant ou un adulte) ; le nombre de **parts**
- du **revenu** du contribuable

Soit la fonction suivante :

```
Fonction calculImpot (part : entier, revenu : réel) : réel
  var quotientFamilial : réel
  var impot : réel
  quotientFamilial ← revenu/part
  Si (quotientFamilial < 5000) Alors
    impot ← 0
  Sinon Si (quotientFamilial < 10 000) Alors
    impot ← revenu * 0.075 - 375 * part
  Sinon Si (quotientFamilial < 15 000) Alors
    impot ← revenu * 0.2 - 1625 * part
  Sinon Si (quotientFamilial < 25 000) Alors
    impot ← revenu * 0.3 - 3125 * part
  Sinon Si (quotientFamilial < 40 000) Alors
    impot ← revenu * 0.4 - 5625 * part
  Sinon Si (quotientFamilial < 50 000) Alors
    impot ← revenu * 0.45 - 7625 * part
  Sinon
    impot ← revenu * 0.5 - 10125 * part
  Fin Si
  Renvoie(impot)
Fin Fonction
```

Écrire un algorithme qui utilise cette fonction pour calculer le montant des impôts dus par un contribuable ayant 3 parts et un revenu de 24 000 euros et pour un contribuable ayant une part et un revenu de 20 000 euros.

Exercice 30 (Perimètre)
Ecrire une fonction prenant un réel (*rayon*) en paramètre qui renvoie la valeur du périmètre d'un cercle de ce rayon. Puis écrire l'algorithme qui appelle la fonction et

affiche le périmètre d'un cerce dont le rayon est donné par l'utilisateur.

Exercice 31 (factorielle)
Écrire une fonction qui prend un entier n en paramètre et renvoie le calcul de factorielle n.
Rappel :

$$n! = n \times n - 1 \times ... \times 1$$

Exercice 32 (Convertion)
Écrire une fonction prenant deux entiers (*base* et *nombre*) en paramètres qui renvoie (sous forme d'une chaîne de caractères) le résultat de la conversion de *nombre* écrit en base 10 vers une base *base* ($base < 10$). Puis écrire l'algorithme qui appelle la fonction.

5 Structures de données

Définition 22 (Structure de données)
On appelle **structure de données** l'association d'un ou plusieurs noms et d'un ensemble de données auxquelles ce ou ces noms permettent d'accéder.

5.1 Classe et Objet

Nous ne prétendons pas ici refaire la théorie des langages à objets. Cependant nous allons préciser certaines notions. Une **classe** est une description statique d'une famille d'objets ayant même structure et même comportement. Les deux aspects sont donc :

- la donnée d'une composante structurelle (non dynamique) : **variables d'instances** (ou champs, ou attributs, ou propriétés) qui caractérisent l'état d'un objet pendant l'exécution d'un programme.
- la donnée d'une composante dynamique : procédures ou fonctions appelées **méthodes**. Les méthodes manipulent les variables d'instances.

La classe est donc un plan de construction, permettant d'obtenir des objets tous semblables (à l'instar d'un véhicule automobile d'une marque et d'un modèle particulier) mais tous différents (chaque véhicule possède sa couleur, son immatriculation,... le fait de repeindre un véhicule ne modifie pas la couleur des autres).

Un objet est une entité indépendante (dont la structure est connue de lui seul). Pour agir sur un objet, il faut utiliser les méthodes de l'interface de celui-ci. Cette utilisation passe par l'envoi d'un message (qui peut être vu comme une requête) à l'objet.

Dans la suite, nous allons définir et utiliser des objets .

5.1.1 Définition

NomClasse
variableInstance : type
NomClasse() nomMethode(param1 : type1, param2 : type2) : typeRetour

Syntaxe

Pseudo langage	Java

Classe NomClasse
var variableInstance : type
...
Methode NomClasse()
... // le constructeur qui porte le nom de la classe
Fin Methode
Methode nomMethode(param1 : type1, param2 : type 2) : typeRetour
 	
Fin Methode
Fin Classe
	public class NomClasse
	{
	type variableInstance ;
	
	public NomClasse()
	{
	}
	
	public typeRetour nomMethode(type1 param1
	, type2 param2)
	{
	}
	}

Exemple 5
Une personne est connue par son nom, son prénom, sa taille, son poids, son âge, ... Il est possible de connaître le nom, le prénom d'une personne. On peut changer de nom (ex : après un mariage) mais pas de prénom et comparer l'âge de deux personnes. La modélisation UML de cette classe vous est donné figure 2.

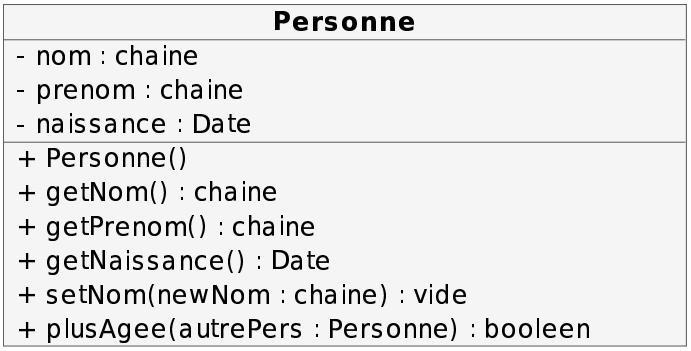

FIG. 2 – Classe personne

5.1.2 Utilisation

L'utilisation d'une classe se fait en trois temps :

1. déclaration d'une variable du type objet,
2. création (ou instantiation) de l'objet,
3. utilisation des méthodes.

Syntaxe

Pseudo langage	Java

Algo Utilisation
var p1 : Personne
// DÉCLARATION
Début
p1 ← nouveau Personne ("Martin","Martin", nouveau Date(...))
// CRÉATION – INSTANTIATION
p1.getNom()
// UTILISATION DE MÉTHODES
Fin
	public class Utilisation
	{
	public static void main(String[] args)
	{
	Personne p1 ;
	p1 = **new** Personne ("Martin","Martin" ,**new** Date(12/02/85)) ;
	System.out.println("le nom est : " + p1.getNom()) ;
	}
	}

Exercice 33 (Vehicule)
Un véhicule se démarre, s'accélère, se freine, s'arrête, ... On peut consulter la valeur de la vitesse (comme sur un compteur de voiture). On peut comparer la vitesse de deux véhicules. Il est préférable de n'actionner le démarreur que si la voiture n'est pas en marche... Proposer un diagramme de classe UML reflétant la situation.

Exercice 34 (Adresse IP)
En vous aidant de vos connaissances en réseaux, donner une modélisation UML et sa traduction en classe de la notion d'adresse IP. On veut pouvoir manipuler des adresses de la forme : 192.156.19.0/24 Pour ceci, nous allons supposer l'existence d'un type `binaire` permettant de manipuler (plus facilement) des nombres binaires. Les caractéristiques de ce type sont les suivantes :

valeurs : n'importe quelle valeur codé en binaire

exemple : 10010001

opérateurs : addition `+`, soustraction `-`, multiplication `*`, division `/`, modulo `%`, comparaison $>, \geq, \leq, <, =, \neq$, `ET` (et bit à bit), `OU` (ou bit à bit), `NON` (négation bit à bit), `(entier)` transtypage vers entier.

Exemple 6 (Compte bancaire)
Un compte bancaire qui est détenu par un titulaire peut se débiter, se créditer. On peut consulter le solde d'un compte bancaire. Une modélisation UML de cette situation vous est donné figure 3.

Exercice 35 (Cuisine)
Le restaurant propose différents plats à sa clientèle. Un plat, qui appartient à une catégorie (entrée, dessert, ...), est préparé par une des équipes de cuisine. Un plat est désigné par son nom, sa durée de préparation, sa durée de cuisson et sa liste d'ingrédients. Il peut être préparé plusieurs fois. Les cuisiniers ne font partie que d'une équipe de cuisine. Chaque équipe a un responsable parmi les cuisiniers qui la composent. Proposer un diagramme de classe UML reflétant la situation.

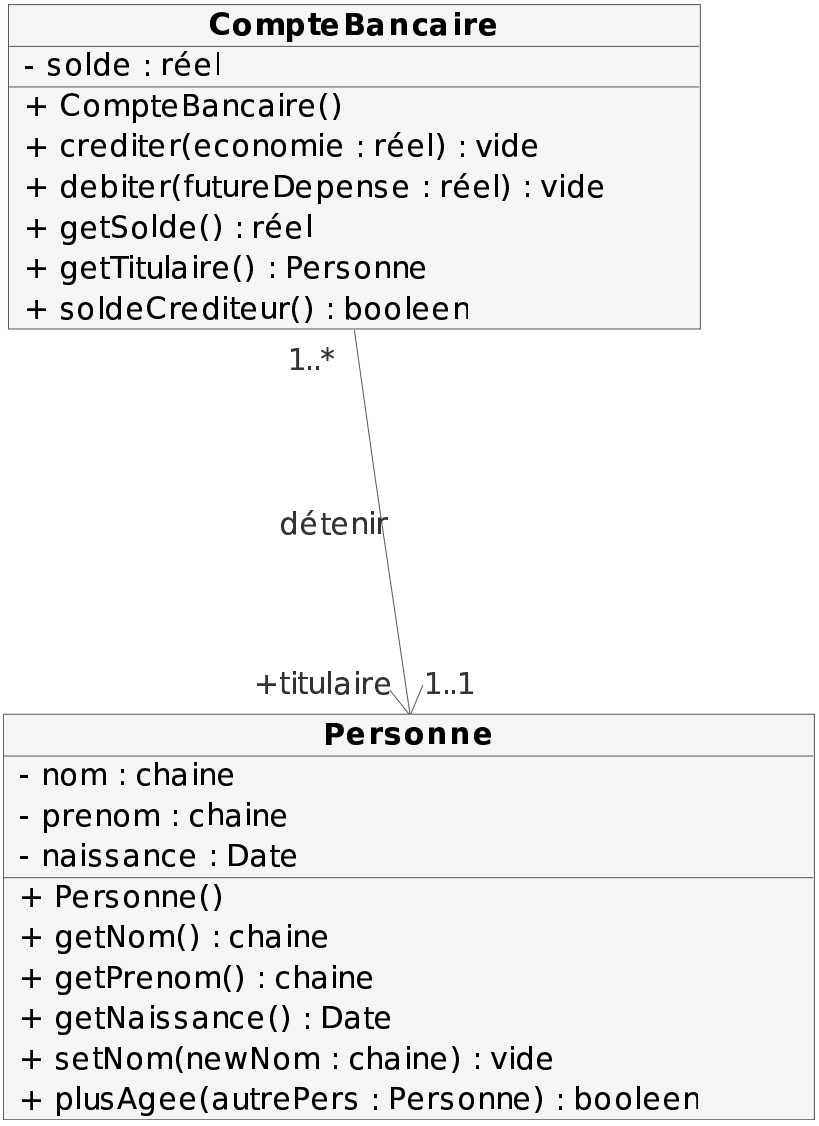

FIG. 3 – Compte bancaire

5.2 Tableau

Définition 23 (Tableau)
Un tableau est une structure de donnée permettant de représenter une collection de valeurs. Une valeur est repérée par un **nom** et un (ou plusieurs) **indices(s)**. Les indices ont des valeurs appartenant à un ensemble fini, en général un sous-ensemble des entiers.

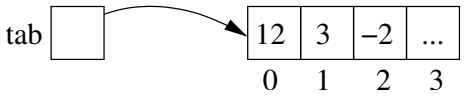

FIG. 4 – Exemple de tableau

N.B.1 Selon les langages de programmation, le dimensionnement du tableau peut se faire de manière :

statique : la dimension est spécifiée lors de la déclaration

dynamique : la dimension est spécifiée à l'exécution

Pour nous, le dimensionnement sera réalisé à l'aide de la méthode de création du tableau.

N.B.2 De même, certains langage vont autoriser un redimensionnement du tableau (perl), d'autres non (java).

5.2.1 Définition

Nous supposons l'existence d'un classe paramétrée `Tableau` dont la modélisation UML et le squelette de la classe sont donnée

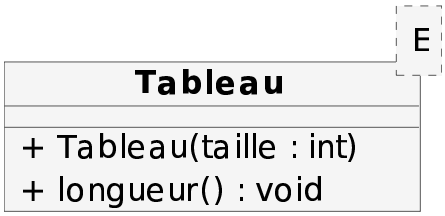

FIG. 5 – Classe Tableau

Classe Tableau<E>
  ...
  Methode Tableau<E>(taille : entier)
    ...
  Fin Methode
  Methode longueur() : entier
    ...
  Fin Methode
Fin Classe

5.2.2 Utilisation

Syntaxe

Pseudo langage	Java

var tabEntier : Tableau<entier> ; // déclaration
tabEntier ← nouveau Tableau<entier>(10) ; // dimensionnement
tabEntier[2] ← 3 ; //accès en écriture
Ecrire(tabEntier[2]) ; //accès en lecture
Ecrire(tabEntier.longueur()) ; //accès à la longueur
	int[] tabEntier ; *// déclaration*
	tabEntier = **new int**[10] ; *// dimensionnement*
	tabEntier[2] = 3 ; *//accès en écriture*
	System.out.println(tabEntier[2]) ; *//accès en lecture*
	System.out.println(tabEntier.length) ; *//accès à la longu*

Exemple 7

```
Algo Exemple1
  var voyelle : Tableau<caractère>
  var i : entier
Début
  voyelle ← nouveau Tableau<caractère>(6)
  voyelle[0] ← 'a'
  voyelle[1] ← 'e'
  voyelle[2] ← 'i'
  voyelle[3] ← 'o'
  voyelle[4] ← 'u'
  voyelle[5] ← 'y'
  Pour i de 0 à voyelle.longueur()-1  Faire
    Ecrire (voyelle[i])
  Fin Pour
Fin
```

Exercice 36

Exécuter l'algorithme suivant :

```
Algo Exercice1
  var carre : Tableau <entier>
  var i : entier
Début
  carre ← nouveau Tableau<entier>(4)
  carre[0] ← 2
  carre[1] ← 5
  carre[2] ← 3
  carre[3] ← 10
  Pour i de 1 à carre.longueur()-1 Faire
    carre[i] ← carre[i] * carre[i]
  Fin Pour
  Pour i de 0 à carre.longueur()-1  Faire
    Ecrire(carre[i])
  Fin Pour
Fin
```

Exercice 37

Exécuter l'algorithme suivant :

```
Algo Exercice2
  var nb : Tableau <entier>
  var i : entier
Début
  nb ← nouveau Tableau<entier>(6)
  nb[0] ← 1
  Pour i de 1 à nb.longueur()-1 Faire
    nb[i] ← nb[i-1] + 2
  Fin Pour
  Pour i de 0 à nb.longueur()-1  Faire
    Ecrire(nb[i])
  Fin Pour
Fin
```

Exercice 38
Exécuter l'algorithme suivant :

```
Algo Fibonacci
  var suite : Tableau <entier>
  var i : entier
Début
  suite ← nouveau Tableau<entier>(6)
  suite[0] ← 1
  suite[1] ← 1
  Pour i de 2 à suite.longueur()-1  Faire
    suite[i] ← suite[i-1] + suite[i-2]
  Fin Pour
  Pour i de 0 à suite.longueur()-1  Faire
    Ecrire(suite[i])
  Fin Pour
Fin
```

Exercice 39
Écrire un algorithme permettant d'obtenir la somme des élément d'un tableau `tab` .

Exercice 40
Même exercice, mais sous forme d'une fonction prenant un tableau d'entiers en paramètre et renvoyant un entier.

Exercice 41 (Maximum)
Écrire une fonction qui prend un tableau d'entiers en paramètre et détermine le plus grand élément de celui-ci.

Exercice 42 (Position du maximum)
Écrire une fonction qui prend un tableau d'entiers en paramètre et détermine la position du plus grand élément de celui-ci.

Exercice 43 (Position du maximum (2))
Même exercice, mais en indiquant les bornes min et max de recherche du maximum.

Exercice 44 (Inversion)
Écrire une procédure qui permet d'inverser un tableau.

Exercice 45 (Suppression)
Écrire une fonction qui supprime l'élément à la position `pos` dans un tableau `tab` de `E`.

Exercice 46 (Insertion)
Écrire une fonction qui insère un élément elt de type `E` dans un tableau `tab` de `E` à une position donnée. On suppose que :
- soit il reste un position de libre dans le tableau (à la fin)
- soit on perd le dernier élément

Exercice 47 (Fusion)
Écrire une fonction qui prend en paramètre deux tableaux et renvoie le tableau résultant de leur concaténation

Exercice 48 (Recherche séquentielle dans un tableau non trié)
Écrire une fonction qui prend en paramètre un tableau et un valeur et qui renvoie la position (première trouvée) de la valeur ou -1 si cette valeur n'a pas été trouvée dans le tableau.

Exercice 49 (Recherche séquentielle dans un tableau trié)
On peut arrêter la recherche quand la valeur recherché ne peut plus être trouvée. On suppose le tableau trié de manière croissante. le `type` des éléments du tableau doit être doté d'un ordre total (i.e. $\forall$ elt1, elt2 : `type` elt1$\geq$ elt2 ou elt1 $<$ elt2)

Exercice 50 (Recherche dichotomique dans un tableau trié)
Le principe de cette recherche est très astucieux (et économique). Algo en $\log(n)$

Exercice 51 (Tri par échange)
Écrire une fonction qui trie un tableau (d'entier) donné en paramètre. Le principe de l'algorithme est :
- de rechercher la position du maximum sur les cases de 0 à n (n initialement à longueur-1)
- d'échanger le maximum et la case n
- de diminuer n et recommencer jusqu'à ce que n soit égal à 1.

Tri en n^2.

Exercice 52 (Tri rapide)
Algorithme en $n\log(n)$

5.3 Pile

Définition 24 (Pile)
Une pile est une structure de données de type LIFO (*Last In First Out*). Ce qui signifie que l'élément inséré en dernier dans une pile est le premier à en être extrait. L'analogie avec une pile d'assiette nous amène facilement à définir l'interface de la classe `Pile` (voir figure 6).

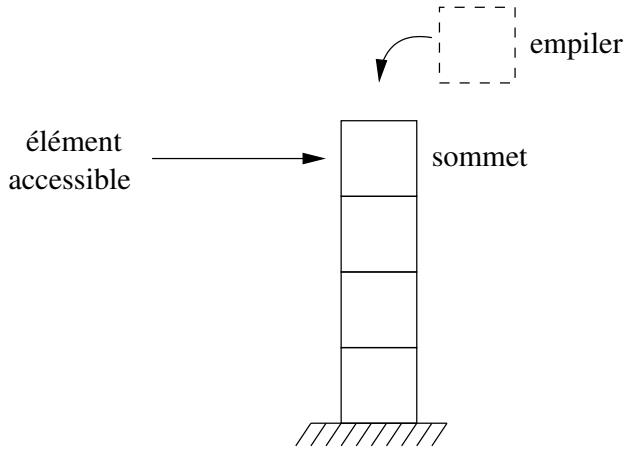

FIG. 6 – Pile

Interface de `Pile<E>`	**Implantation**
Pile() // constructeur	var elts : Tableau<E>
sommet() : E	var sommet : entier
estVide() : booléen	
empiler(val : E) : vide	
depiler() : E	
nbElement() : entier	

Les piles sont souvent utilisés sans que l'on s'en rendre compte. Parmi les utilisations courantes des piles, on peut citer :

1. les appels successifs de méthodes ou de fonctions.
2. la réalisation d'une fonctionnalité "annuler" dans un logiciel.

5.4 Liste

Définition 25 (Liste)
Une **liste** est une suite (éventuellement vide) d'éléments. On peut insérer, supprimer n'importe où dans la liste.

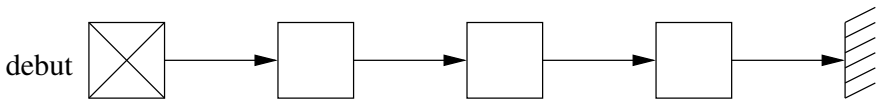

FIG. 7 – Liste simplement chaînée

5.4.1 Implantation Cellule

5.4.2 Implantation Liste

Exercice 53 (Définition)
Seul le début est nécessaire.

Exercice 54 (Constructeur)
Initialisation du début avec la cellule "fictive".

Exercice 55 (Vide)
Une méthode pour savoir si la liste est vide.

Exercice 56 (Premier)
Accès au premier élément de la liste.

Exercice 57 (Dernier)
Accès au dernier élément de la liste.

Exercice 58 (Insertion en tête)
On insère une nouvelle valeur en tête de liste

Exercice 59 (Insertion en queue)
On insère une nouvelle valeur en queue de liste

Exercice 60 (Supression en tête)
On supprime la valeur en tête de liste

Exercice 61 (Suppression en queue)
On supprime la valeur en queue de liste

Exercice 62 (Taille)
Renvoie le nombre d'éléments de la liste que nous recomptons à chaque fois. N.B. Il existe bien entendu, une meilleures solution qui consiste à rajouter une variable d'instance incrémentée à chaque insertion et décrémentée à chaque suppression dans la liste.

Exercice 63 (Accès)
Renvoie l'élément à une position quelconque.

Exercice 64 (Insertion)
Insère un élément à une position quelconque.

Exercice 65 (Supression)
Supprime l'élément à une position quelconque.

5.4.3 Implantation Modifiée pour itérer

L'idée est de gérer un curseur que l'on peut avancer, reculer. De plus on peut insérer, supprimer, accéder l'élément à la position du curseur. L'insertion et la suppression demandant d'agir une cellule avant, nous allons positionner le curseur **une position avant** celle "affichée" (voir figure 8).

Exercice 66 (Définition)
Il faut rajouter une variable pour le curseur.

Exercice 67 (Constructeur)
Rajouter dans le constructeur, l'initialisation du curseur.

Exercice 68 (Démarrer)
La méthode `demarrer()` permet de remettre le curseur en première position.

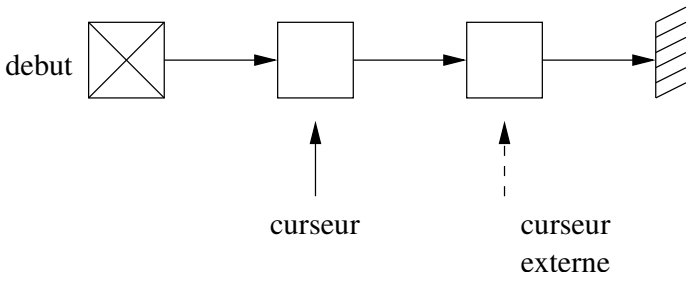

FIG. 8 – Principe du curseur

Exercice 69 (Hors bornes)
La méthode `aElement()` permet de savoir si le curseur est sur une position qui comporte un élément. Cette méthode renvoie faux en cas de liste vide ou si le curseur est "au delà" du dernier élément. Ce dernier cas arrive en cas de suppression du dernier élément ou si on veut insérer en queue.

Exercice 70 (Accès)
Renvoie l'élément à la position courante.

Exercice 71 (Insertion)
Insère un élément à la position courante

Exercice 72 (Supression)
Supprime l'élément à la position courante. Ne fait rien si le curseur est en fin de liste

Exercice 73 (Avance)
Avance le curseur d'une position. Ne fait rien si on ne peut plus avancer

Exercice 74 (Recul)
Recule le curseur d'une position Ne fait rien si on ne peut plus reculer. Pour reculer, on utilise une cellule temporaire, qu'on initialise au début et qu'on fait avancer. Quand le suivant de la cellule temporaire est égal au curseur, la cellule temporaire est le nouveau curseur.

5.4.4 Utilisation

Exercice 75 (Exécution d'un algorithme)
Exécuter l'algorithme suivant :

```
Algo TestListe
  var l : Liste <entier>
Début
  l ← nouvelle Liste()
  l.insererQueue(1)
  l.insererTete(2)
  l.insererTete(3)
  l.demarrer()
  l.inserer(4)
  l.inserer(5)
```

```
    l.avancer()
    l.inserer(6)
    l.demarrer()
    Tant que (l.aElement()) Faire
      Ecrire (l.element())
      l.avancer()
    Fin Tant que
  Fin
```

Exercice 76 (Suppression)
Écrire une fonction prenant en paramètre une liste d'entier et qui supprime les éléments négatifs.

Exercice 77 (Lecture interactive)
On lit interactivement des entier au clavier jusqu'à lire -1. Écrire un algorithme qui rempli une liste avec les valeurs saisies. Avantage : on ne connaît pas d'avance la taille de la liste.

5.5 Récursivité

Définition 26 (Recursivité)
On appelle récursivité le fait pour un sous-programme (méthode, fonction ou procédure selon le contexte) de s'appeler au moins une fois. La vision récursive s'oppose bien souvent à la vision itérative.

Exemple 8 (Factorielle)
Le calcul d'une factorielle peut s'exprimer de deux manières :
- $n! = n \times n-1 \times ... \times 1$
- $n! = n * (n-1)!$

La première version permet l'implantation d'une fonction itérative (voir exercice 31). La deuxième permet une implantation récursive de la fonction factorielle.

```
  Fonction factorielle (nombre : entier) : entier
    var res : entier
    Si (nombre = 1 OU nombre = 0) Alors
      res ← 1
    Sinon
      res ← factorielle(nombre-1)
    Fin Si
    Renvoie(res)
  Fin Fonction
```

Une des difficultés d'un algorithme récursif est d'en garantir la terminaison.

Exercice 78 (Fibonacci)
Écrire une fonction (récursive) permettant de calculer un élément de la suite de Fibonacci donnée par la définition par récurrence suivante :
- $U_0 = 1$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

Exercice 79 (Recherche dichotomique dans un tableau trié)
Réécrire une fonction de recherche dichotomique récursive.

Exercice 80 (Aspirateur de site)
On suppose l'existence d'un type `URL` dotée de deux méthodes :
- analyse() : tableau [URL] qui renvoie dans un tableau tous les liens disponibles dans la page. Cela corresponds aux valeurs des attributs suivants : <a href=.., <img src=..., <link rel="stylesheet" href=..., ...
- recupere() : vide qui enregistre l'URL

5.6 Graphe – Arbre

Ceci est une présentation basique de la théorie des arbres et des graphes. Les définitions et exemples présentés sont à destination d'un publique intéressé par la conception Internet et multimédia en général.

5.6.1 Graphe

Définition 27 (Graphe)
Un **graphe** est un ensemble fini d'entités appelées **sommets**, reliés les uns aux autres par un ensemble fini de liens appelés **arêtes**.

Exemple 9 (Graphe)
Les graphes servent souvent à modéliser des données concrètes afin de les traiter ou de les présenter avec l'outil mathématique ou informatique. Il est par exemple possible de modéliser le réseau autoroutier français. Créons un graphe dont les sommets seront les villes, et les arêtes seront les autoroutes reliant ces villes. Une version extrêmement simplifiée de ce graphe vous est donnée figure 9 :

Ce graphe nous apprend par exemple qu'il est possible de rallier Grenoble à Paris par autoroute, et qu'il est indispensable de passer soit par Lyon, soit par Bordeaux. Si nous décidions de porter des informations kilométriques sur les arêtes du graphe, nous pourrions déduire qu'il est de loin plus court de passer par Lyon que de passer par Bordeaux.

Ce graphe nous apprend encore qu'il est impossible de rallier Strasbourg à Bayonne uniquement par autoroute.

5.6.2 Arbres et arborescences

Définition 28 (Arbre)
Un **arbre** est un graphe sans cycle (c'est-à-dire qu'il existe au plus un seul chemin entre deux sommets) et connexe (c'est-à-dire qu'il existe au moins un chemin entre deux sommets).

Définition 29 (Noeud – branche)
Dans le cas particulier des arbres, un sommet est appelé **noeud** et une arête est appelée **branche**.

Définition 30 (Arborescence)
Une **arborescence** est un arbre particulier, dont les branches sont orientées (c'est-à-dire qu'elle sont à sens unique, contrairement aux autoroutes), et qui possède un noeud appelé **racine** duquel sont accessibles tous les autres noeuds.

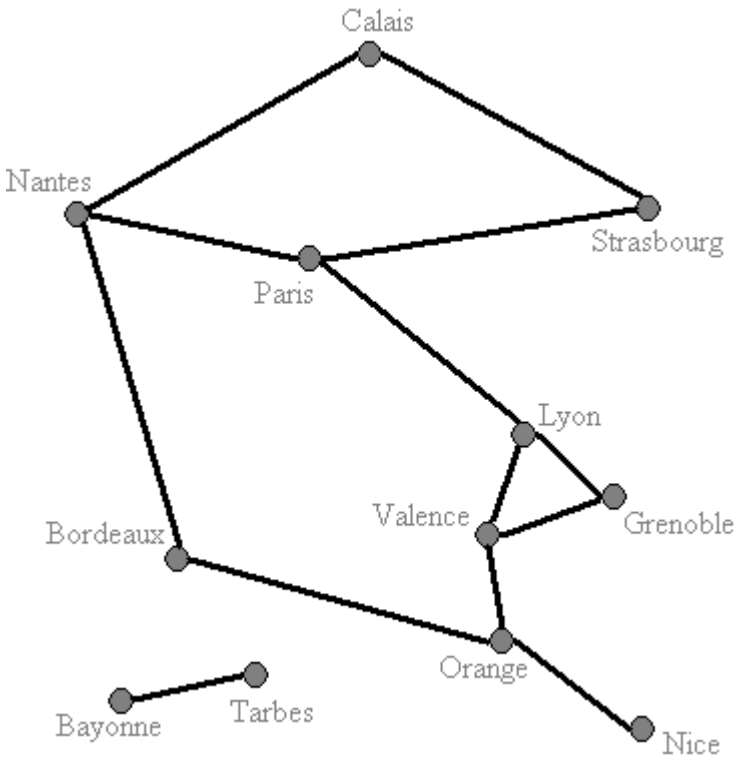

FIG. 9 – Exemple de graphe

Définition 31 (Père – fils)
Les noeuds directement accessibles depuis un noeud N sont appelés fils de N. Le noeud depuis lequel est accessible un noeud N est appelé père de N.

Dans le langage courant, on emploi souvent le mot arbre pour désigner des arborescences.

Exemple 10 (Arbre)
Un document html peut être représenté sous forme d'arbre. En considérant les éléments comme les noeuds de l'arbre, et l'inclusion d'un élément dans un autre comme les branches, la figure 10 présente une partie de l'arbre d'un document html.

La racine de l'arbre est le noeud "html". Un tableau est toujours contenu dans le body : ainsi "table" est un fils de "body" et "body" est le père de "table".

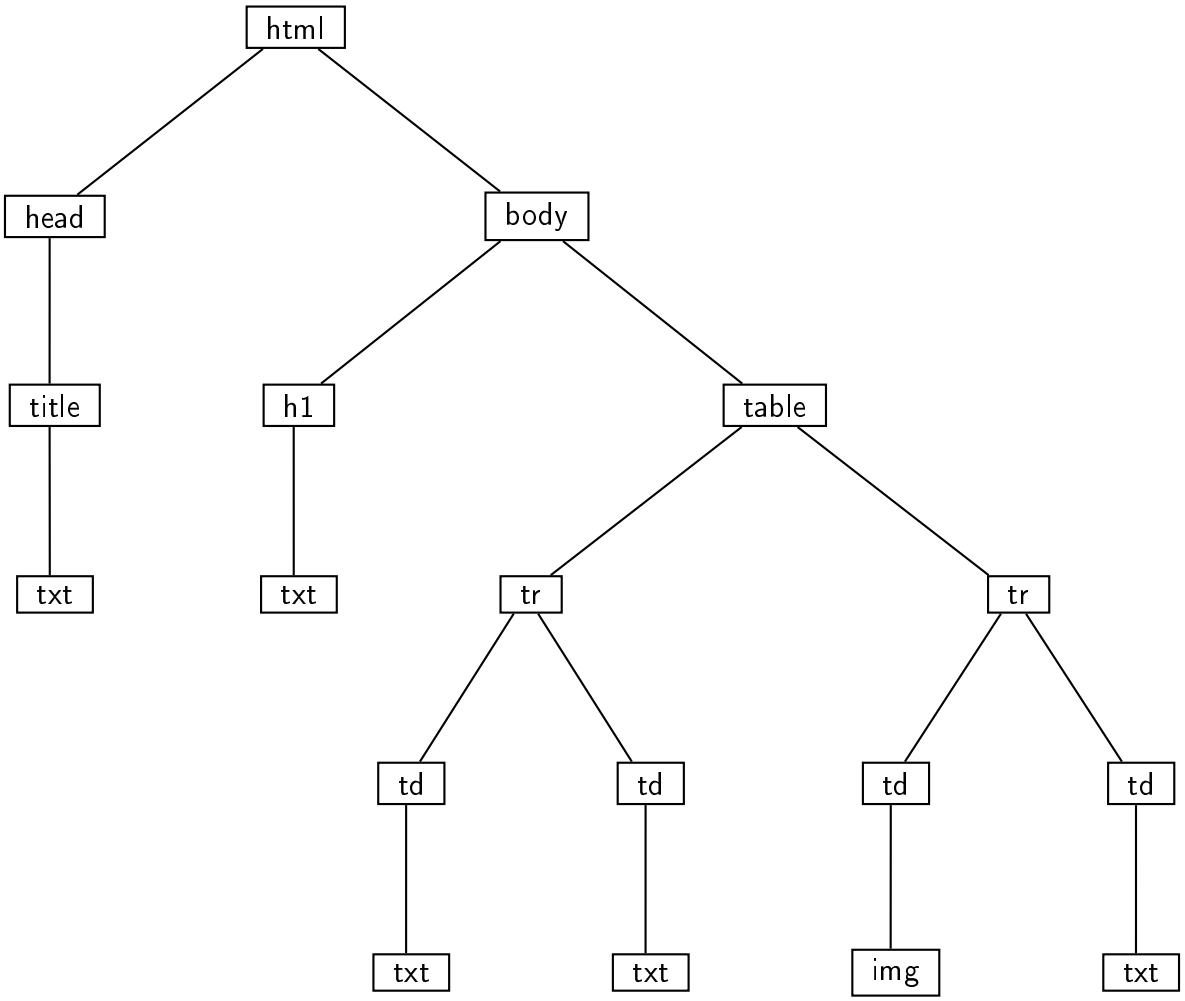

FIG. 10 – Exemple d'arbre