LA PROGRAMMATION ORIENTEE OBJET (POO)

INTRODUCTION

La programmation classique telle qu'étudiée au travers des langages procédurales comme le C, le Pascal… définie un programme comme étant un ensemble de données sur lesquelles agissent des procédures et des fonctions. Les données constituent la partie passive du programme. Les procédures et les fonctions constituent la partie active.

Dans cette approche données et procédures sont traitées indépendamment les unes des autres sans tenir compte des relations étroites qui les unissent.

Les langages objets sont nés pour améliorer la conception et le développement des logiciels. Ils sont fondés sur la connaissance d'une seule catégorie d'entités informatiques : les objets.

Un objet incorpore des aspects statiques et dynamiques au sein d'une même notion. La Programmation Orientée Objet(POO) est une méthode d'implémentation dans laquelle les programmes sont organisés sous formes de collections coopératives d'objets, dont chacun représente une instance d'une classe quelconque et dont toutes les classes sont membres d'une hiérarchie de classes unis à travers des relations d'héritage.

Ainsi dans l'approche orienté objet un algorithme sera essentiellement vu comme un ensemble d'objets auxquels l'utilisateur envoie des messages et qui s'en envoient pendant le fonctionnement.

Chapitre 1: LES NOTIONS PRELIMINAIRES

I. LES FONCTIONS PREDEFINIES

Certains traitements ne peuvent être effectués par un algorithme. C'est par exemple le calcul du sinus d'un angle : pour en obtenir une valeur approchée il faudrait appliquer une formule complexe.
Pour faciliter les traitements mais aussi éviter aux programmeurs de tout reprendre à chaque fois depuis le début, tous les langages de programmation propose un certain nombre de fonctions .Certaines sont indispensables, elles permettent d'effectuer les traitements qui seraient sans elle impossibles. D'autres servent à soulager le programmeur en lui épargnant des longs algorithmes.

1) Structure générale des fonctions

Reprenons l'exemple du sinus. Les langages proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrirons :
A ⇦ Sin(35)
Une fonction est donc constituée de trois parties :

- le **nom** proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage,
- deux parenthèses, une ouvrante, une fermante,
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des **arguments**, ou des **paramètres**. Certaines fonctions exigent un seul argument, d'autres deux, etc. et d'autres encore aucun. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (logique, c'est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain **type**, et qu'il faut respecter ces types.

2) Les fonctions de manipulation de texte (chaine de caractères)

Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères.

Longueur d'une chaîne

Syntaxe : `longueur(ch)`
Cette fonction retourne (renvoie) le nombre de caractères d'une chaîne de caractères.
Exemple : `longueur("Bonjour")`
Cette fonction retourne 7

La fonction de concaténation de chaînes

La concaténation est la juxtaposition de deux chaînes (ou plus) afin d'en former une seule.
Syntaxe : `concat(ch1, ch2)`
Cette fonction retourne une chaîne formée par la concaténation de ch1 et de ch2. La chaîne résultat est formée de ch1 suivi de ch2.
Exemple :

```
concat("Bonjour"," Monsieur")
concat("Bonjour","Monsieur")
concat("Bonjour"," ","Monsieur")
```

Le résultat de la première instruction est : Bonjour Monsieur.
Le résultat de la deuxième instruction est : BonjourMonsieur.
Le résultat de la troisième instruction est : Bonjour Monsieur.

La fonction de copie de chaînes

Syntaxe :

```
Copie(ch1, position, n)
```

Cette fonction (recopie une partie de la chaîne ch1 à partir de la position précisée par l'expression entière position, en limitant la recopie au nombre de caractères précisés par l'entier n.

Exemple :

```
Copie("Bonjour",4,4)
```

La fonction de comparaison de chaînes

Syntaxe :

```
comp(ch1, ch2)
```

Cette fonction compare deux chaînes de caractères en utilisant l'ordre des caractères définis par le code ASCII. Elle fournit une valeur entière définie comme étant :

- positive si ch1 > ch2
- nulle si ch1 = ch2. C'est-à-dire si ces deux chaînes contiennent exactement la même suite de caractères.
- négative si ch1 < ch2

Exemple :

```
comp("bonjour", "monsieur")
```

Cette fonction retourne une valeur négative.

La fonction de recherche dans une chaîne

Il existe des fonctions de recherche dans une chaîne de caractères de l'occurrence d'un caractère ou d'une autre chaîne de caractères.

Syntaxe :

```
recherche(ch1, caractère)
```

Cette fonction dans la chaîne ch1, la première position où apparaît le caractère mentionné.

```
recherche(ch1, ch2)
```

Cette fonction recherche dans la chaîne ch1, la première occurrence complète de la chaîne ch2. Elle renvoie un nombre correspondant à la position de la chaîne ch2 dans la chaîne ch1. Si la chaîne ch2 n'est pas comprise dans la chaîne ch1, la fonction renvoie zéro.

Exemple :

```
recherche("bonjour monsieur","jour")
```

Cette fonction retourne 4.

3) Les fonctions mathématiques

Les fonctions mathématiques prédéfinies permettent la réalisation d'un traitement mathématique sur des données numériques.

Fonction	Description	Exemple	Résultat
Abs(nombre)	Retourne la valeur absolue d'un nombre	x ← Abs(-12)	x = 12
Ent(nombre)	Retourne la partie entière d'un nombre	x ← Ent(12.3)	x = 12
Cos(angle)	Retourne une valeur spécifiant le cosinus d'un angle	x ← Cos(0)	x = 1
Sin(angle)	Retourne une valeur spécifiant le sinus d'un angle	x ← Sin(0)	x = 0
Tan(angle)	Retourne une valeur contenant la tangente d'un angle	x ← Tan(0)	x = 0

Sqrt(nombre)	Retourne une valeur spécifiant la racine carrée d'un nombre	x ← Sqrt(4)	x = 2
Alea()	Retourne un nombre aléatoire compris entre 0 (inclus) et 1 (exclu)	x ← alea()	0 =< x < 1

La fonction ***Alea*** renvoie un nombre réel aléatoire compris entre 0 et 1. Pour obtenir une valeur entière comprise entre min et max (avec min < max), on peut utiliser la formule suivante :

```
Ent((max-min+1)*Alea() + min)
```

Exemple :Générer un nombre aléatoire X compris entre 5 et 10.
Solution :

```
X ← Ent(6 * Alea() + 5)
```

Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction Chr), et une autre pour la conversion inverse (fonction Asc) :

Asc("N") vaut 78

Chr(63) vaut "?"

II. PROCEDURES ET FONCTIONS

INTRODUCTION

Lorsqu'un programme est long, il a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement, il est irréaliste d'écrire son code d'un seul tenant. En fait, on décompose le programme en plusieurs parties plus petites, on donne un nom à chacune de ces parties, et on les assemble pour former le programme final. C'est le principe de la **programmation modulaire**, qui repose sur l'écriture de sous-programmes.
Un sous-programme est, comme son nom l'indique, un petit programme réalisant un traitement particulier qui s'exécute à l'intérieur d'un autre programme. Le corps du programme s'appelle alors la **procédure principale ou programme principal.**
Les sous-programmes sont utilisés pour deux raisons essentielles :
- **quand un même traitement doit être réalisé plusieurs fois** dans un programme (ou qu'il est utilisé dans plusieurs programmes): on écrit un sous-programme pour ce traitement et on l'appelle à chaque endroit où l'on en a besoin. On évite ainsi de réécrire plusieurs fois le code du traitement.
- **pour organiser le code**, améliorer la conception et la lisibilité des gros programmes. En effet, le découpage en sous-programmes permet de traiter séparément les difficultés.
Certains sous-programmes ont déjà été écrits et peuvent être utilisés directement dans n'importe quel programme. Ce sont des **sous-programmes ou fonctions standards ou prédéfinis**
Mais les sous-programmes prédéfinis ne suffisent pas pour découper un gros programme : le programmeur est amené à écrire le code de ses propres sous-programmes.
Il existe deux sortes de sous-programmes : **les procédures et les fonctions.**
Nous allons étudier d'abord les procédures simples, puis les fonctions.

A. LES PROCEDURES

Une procédure est un ensemble d'instructions regroupées sous un nom, qui réalise un traitement particulier dans un programme lorsqu'on l'appelle.
La notion de procédure permet de distinguer dans un programme :

- la définition d'une procédure : c'est l'écriture de son nom et de la séquence d'instructions qui la composent,
- l'appel d'une procédure, qui s'effectue en écrivant uniquement son nom : c'est une instruction qui a pour effet d'exécuter la séquence d'instructions associée au nom de la procédure.

Une procédure peut, de plus, avoir des paramètres qui peuvent être des paramètres d'entrée, de sortie ou à la fois d'entrée et de sortie. Un paramètre est un nom donné à une variable manipulée par la procédure.

1. Définition de procédure

Comme un programme, une procédure possède un nom, des variables, des instructions, un début et une fin.
Mais contrairement à un programme, un sous-programme ne peut pas s'exécuter indépendamment d'un autre programme.
Syntaxe de la **DEFINITION** d'une procédure:

```
Procedure nomProcedure
Var …//declaration des variables
DEBUT//debut de la procedure
//instructions de la procedure
Fin Procedure//fin de la procedure
```

Exemple

```
Procédure ligneEtoile( )
Var
nbe: entier ;
Début
Pour nbe de 1 à 10 Faire
Afficher '*' ;
FinPour
Afficher '\n' ;
Fin Procedure
```

Cette procédure permet d'afficher une ligne de 10 étoiles puis passe à la ligne.

2. Appel d'une procédure

Pour déclencher l'exécution d'une procédure dans un programme, il suffit de l'appeler, c'est-à-dire d'indiquer son nom suivi de parenthèses.

```
Programme RectangleEtoile
Var nlignes: entier ;
        i : entier ;
Début
Afficher "Ce programme dessine un rectangle d'étoiles. Combien voulez-vous de lignes?" ;
Saisir nlignes ;
Pour i de 1 à nlignes Faire
ligneEtoile( ) ;
FinPour
Fin
```

Lorsque le processeur rencontre l'appel d'une procédure, il arrête momentanément l'exécution du programme appelant pour aller exécuter les instructions de la procédure. Quand il a terminé l'exécution de la procédure, le processeur reprend l'exécution du programme appelant là où il s'était arrêté.

Une procédure peut être appelée soit par un programme, soit par un autre sous-programme (qui lui-même a été appelé). Les appels de sous-programmes peuvent s'imbriquer autant qu'on le désire.

3. Notions de variables locales et de paramètres

Les variables déclarées dans une procédure ne sont pas utilisables dans le programme appelant et inversement, les variables déclarées dans le programme appelant ne sont pas utilisables dans les procédures.
Chaque programme et sous-programme a son propre espace de variables, inaccessible par les autres. On dit que **les variables sont LOCALES**. (On verra qu'il existe des variables globales, mais elles sont très peu utilisées).
Une question qui se pose alors est de savoir comment procédures et programmes vont pouvoir communiquer des données. Par exemple, on pourrait vouloir que le programme principal communique à la procédure combien d'étoiles afficher par ligne. Cela est possible grâce aux paramètres.
Un paramètre est une variable particulière qui sert à la communication entre programme appelant et le sous-programme.
Exemple :
Dans notre exemple, nous allons mettre le nombre d'étoiles par lignes en paramètre.
Pour cela, nous indiquons entre parenthèses la déclaration du paramètre (qui est une variable de la procédure !). La valeur de cette donnée est communiquée à l'appel, par le programme appelant.

```
Procédure ligneEtoile(nombre : entier ) //sous-programme
Var
cpt : entier ;
Début
Pour cpt de 1 à nombre Faire
Afficher "*";
FinPour
Fin Procedure

Programme RectangleEtoile //programme appelant
Var
nlignes, netoiles: entier ; //nombre de lignes et nombre d'étoiles par ligne
i : entier ;
Début
Afficher "Ce programme dessine un rectangle d'étoiles." ;
Afficher "Combien voulez-vous d'étoiles par ligne?" ;
Saisir netoiles ;
Afficher "Combien voulez-vous de lignes?" ;
Saisir nlignes ;
Pour i de 1 à nlignes Faire
ligneEtoile(netoiles) ;
FinPour
Fin
```

Fonctionnement du passage des paramètres

Lors de l'appel de la procédure, la valeur de la variable netoiles passée en argument est copiée dans le paramètre formel nombre (qui est une variable). La procédure effectue alors le traitement avec la variable nombre qui a bien la valeur voulue: celle de netoiles.
La procédure n'utilise par directement la variable netoile : elle utilise sa valeur, qu'elle a recopiée dans sa propre variable-paramètre.

Remarque importante sur l'ordre et le nombre des paramètres
Lorsqu'il y a plusieurs paramètres dans la définition d'une procédure, il faut absolument qu'il y en ait le même nombre à l'appel et que l'ordre soit respecté (car la copie se fait dans l'ordre).

Les paramètres réels et formels
Il est primordial de bien distinguer les paramètres qui se trouvent dans l'en-tête d'une procédure, lors de sa définition et les paramètres (ou arguments) qui se trouvent placés entre parenthèses lors de l'appel.

- Les paramètres placés **dans la définition** d'une procédure sont les **paramètres formels**. Ils servent à décrire le traitement à réaliser par la procédure indépendamment des valeurs traitées. Les paramètres formels sont des variables locales à la procédure, et à ce titre ils sont **déclarés** dans l'entête de la procédure.
- Les paramètres placés **dans l'appel** d'une procédure sont les **paramètres réels ou effectifs**.

Ils contiennent effectivement les valeurs sur lesquelles sera effectué le traitement de la procédure. Lors de l'appel, leur valeur est recopiée dans les paramètres formels correspondants. Un paramètre effectif peut être soit une variable du programme appelant, soit une valeur littérale, soit le résultat d'une expression.

Passage par valeur(en entréé(E) donnée(D),input(I)):

Dans ce type de passage, le paramètre formel reçoit uniquement une copie de la valeur du paramètre effectif. La valeur du paramètre effectif ne sera jamais modifiée.
Exemple :
Soit l'algorithme suivant :

```
Algorithme Passage_par_valeur
Variables N : entier
//Déclaration de la procédure P1
Procédure P1(A : entier)
Début
   A ← A * 2
   Afficher(A)
FinProcedure
//Algorithme principal
Début
  N ← 5
  P1(N)
  Afficher(N)
Fin
```

Passage par référence ou par adresse (entréé/Sortie(E/S)
donnée/Resultat(D/R),input/output(I/O) :
Dans ce type de passage, la procédure utilise l'adresse du paramètre effectif. Lorsqu'on utilise

l'adresse du paramètre, on accède directement à son contenu. La valeur de la variable effectif sera donc modifiée.
Les paramètres passés par adresse sont précédés du mot clé Var.
Exemple : Reprenons l'exemple précédent

```
Algorithme Passage_par_référence
Variables N : entier
//Déclaration de la procédure P1
Procédure P1 (Var A : entier)  //ou Procédure P1 (e/s A : entier)
Début
   A ← A * 2
   Afficher(A)
FinProc
//Algorithme Principal
Début
    N ← 5
    P1(N)
    Afficher(N)
Fin
```

B. LES FONCTIONS

Les fonctions sont des sous-programmes qui retournent un et un seul résultat au programme appelant. De ce fait, les fonctions sont appelées pour récupérer une valeur, alors que les procédures ne renvoient aucune valeur au programme appelant.
L'appel des fonctions est différent de l'appel des procédures :
L'appel d'une fonction doit obligatoirement se trouver à l'intérieur d'une instruction (affichage, affectation,…) qui utilise sa valeur.
Le résultat d'une fonction doit obligatoirement être retourné au programme appelant par l'instruction **Retourne ou Renvoyer**.
Syntaxe générale

Fonction *nom*(*déclaration des paramètres*) : *type de la valeur retournée*
Var*//variables locales*
Début
//traitement
Retourne *valeur à retourner*
FinFonction

Exemple :

```
Fonction factoriel (n: entier) : entier
/*Cette fonction retourne le factoriel du nombre n passé en paramètre*/
Var
i : entier ;
fact : entier ;
Début
fact <-1 ;
Pour i de 1 à n faire
fact <- fact * i ;
FinPour
Retourne fact ;
```

FinFonction

Exercice 6.3 :
Créer un programme qui se sert de la fonction factoriel de l'exemple pour calculer le factoriel d'un nombre saisi par l'utilisateur.
Exercice 6.4
Écrivez une fonction qui renvoie la somme de cinq nombres fournis en argument.

III. LES STRUCTURES

Une structure est une suite finie d'objets de types différents. Ce mécanisme permet de grouper un certain nombre de variables de types différents au sein d'une même entité. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contigües en mémoire. Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur. Par exemple, pour une adresse, on a besoin du numéro et du nom de la rue.

1) Déclaration d'une structure

Lors de la déclaration de la structure, on indique les champs de la structure, c'est-à-dire le type et le nom des variables qui la composent:

nomStructure : structure

variable1 : type1

..

variableN : typeN

finstructure

Exemple

sEtudiant : structure
nom : chaine
moy : entier
finstructure

2) Définition d'une variable de type structure

La définition d'une variable structurée suit la syntaxe suivante :

Nom_Variable_Structuree :Nom_Structure

Nom_Structure représente le nom d'une structure que l'on aura préalablement déclarée.
Nom_Variable_Structuree est le nom que l'on donne à la variable structurée.

Exemple : adresse1 : adresse

A,B :etudiant

Il va de soi que, comme dans le cas des variables on peut définir plusieurs variables structurées en les séparant avec des virgules:

3) Accès aux champs d'une variable de type structure

Pour accéder à un champ d'une variable structure on utilise la syntaxe :

Nom_Variable.Nom_Champ;

Ainsi, pour affecter des valeurs à la variable *Pierre* (variable de type *structuture etudiant* définie précédemment), on pourra écrire:

Pierre.Age ←18;

Pierre.Sexe ← 'M';

4) Tableaux de structures

Etant donné qu'une structure est composée d'éléments de taille fixes, il est possible de créer un tableau ne contenant que des éléments du type d'une structure donnée.
struct Nom_Structure Nom_Tableau[Nb_Elements];

Exemple
classes[1..20] :etudiant ;
classes[1].Age=16 ;

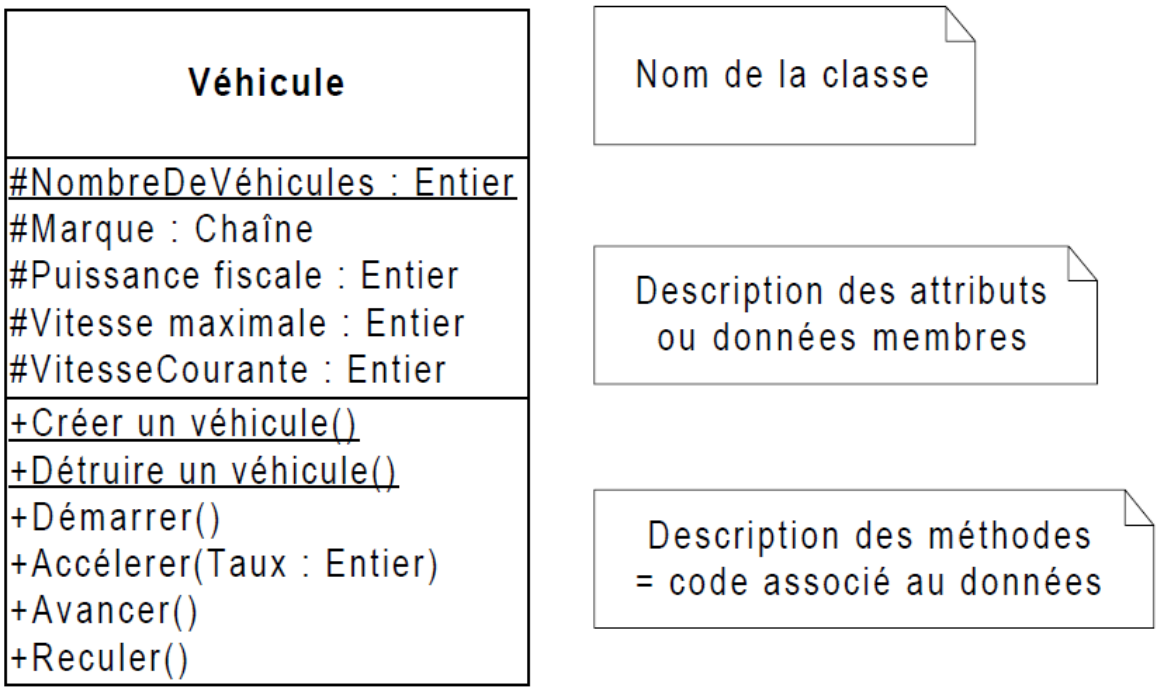
Chapitre 2: PRESENTATION DE LA NOTION OBJET

Le monde dans lequel nous vivons est constitué d'objets. Les méthodes actuelles de développement informatique permettent de manipuler le concept d'objets. La Programmation Orientée Objet(POO) consiste à modéliser un ensemble d'éléments du monde réel en un ensemble d'entités informatiques appelés objet. Cela est plus proche du monde réel, dans lequel tous les objets disposent d'attributs auxquels sont associés des activités.

1) Définitions d'un objet

Un *objet* est une entité cohérente rassemblant des données et du code travaillant sur ses données. Une *classe* peut être considérée comme un moule à partir duquel on peut créer des objets. En fait, on considère plus souvent que les classes sont les *descriptions* des objets, lesquels sont des *instances* de leur classe.

Pourquoi ce vocabulaire ? Une classe décrit la structure interne d'un objet : les données qu'il regroupe, les actions qu'il est capable d'assurer sur ses données. Un objet est un état de sa classe. Considérons par exemple la modélisation d'un véhicule telle que présentée par la figure suivante :



Légende :
: Protégé
- : Privé
+ : public

Figure 1.1 Représentation de la classe Véhicule

La classe Vehicule possède les attributs NombreDeVehicules, Marque puissance fiscale , vitesse maximale, vitessecourante et des methodes creerUnvehicule(), detriureUnVehicule(), demarrer(), accelerer(), avancer() et reculer().

2) LES OBJETS, ATTRIBUTS ET METHODES

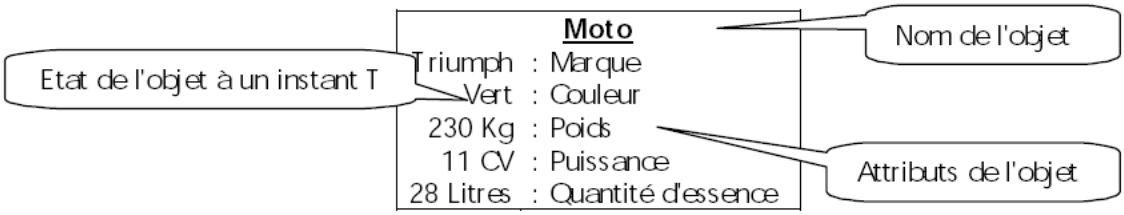
La principale difficulté d'un algorithme par objet sera dans la détermination des bons objets. Ceci constitue l'objectif de l'étape d'analyse et conception qui précède la mise en place d'une application informatique.

a) La notion d'objet

L'informaticien et le non informaticien peuvent avoir un langage commun basé sur ce concept. Un objet au sens informatique du terme permet de désigner une représentation "abstraite" d'une chose concrète du monde réel ou virtuel. Un objet présente les 3 caractéristiques suivantes :

Objet = État + Comportement

L'état d'un objet définit la valeur des données (ou attributs), par exemple dans le cas d'un objet Moto, celui-ci pourrait être caractérisé par les attributs suivants : la marque, la couleur, le poids, la puissance, la quantité d'essence... Ce que l'on représente graphiquement par :



L'état de l'objet peut être amené à changer durant son cycle de vie. Par exemple, la quantité d'essence et le poids de la moto varient en permanence lorsque celle-ci roule.

• **Le comportement** d'un objet indique toutes les compétences de celui-ci et décrit les actions et les réactions qu'il peut avoir. Chaque élément de base du comportement est appelé opération. Les opérations d'un objet sont déclenchées suite à une stimulation externe de l'utilisateur qui appuie sur un bouton ou du programmeur qui appelle une opération. Il existe un lien très étroit entre le comportement d'un objet et son état.

b) La notion de classe

On appelle *classe* la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet.

Une classe est composée de deux parties:

- **Les attributs** ou *données membres*: données représentant l'état de l'objet,
- **Les méthodes** ou *fonctions membres*: opérations applicables aux objets.

c) Attributs

Les attributs d'un objet sont l'ensemble des informations se présentant sous forme de variable et permettant de représenter l'état de l'objet.

d) Méthodes

Une méthode est une fonction ou procédure liée à un objet qui est déclenchée à la réception d'un message particulier : la méthode déclenchée correspond strictement au message reçu. La liste des méthodes définies au sein d'un objet constitue l'interface de l'objet pour l'utilisateur.

e) Message

Un message est une demande d'activation d'une méthode envoyé à un objet. Les messages auquel l'objet peut répondre sont en fait les intitulés des méthodes qu'il peut déclencher : les identificateurs et le type des paramètres nécessaires.

3) Création d'un objet

La figure suivante montre l'opération d'instanciation de la classe en 2 objets différents :

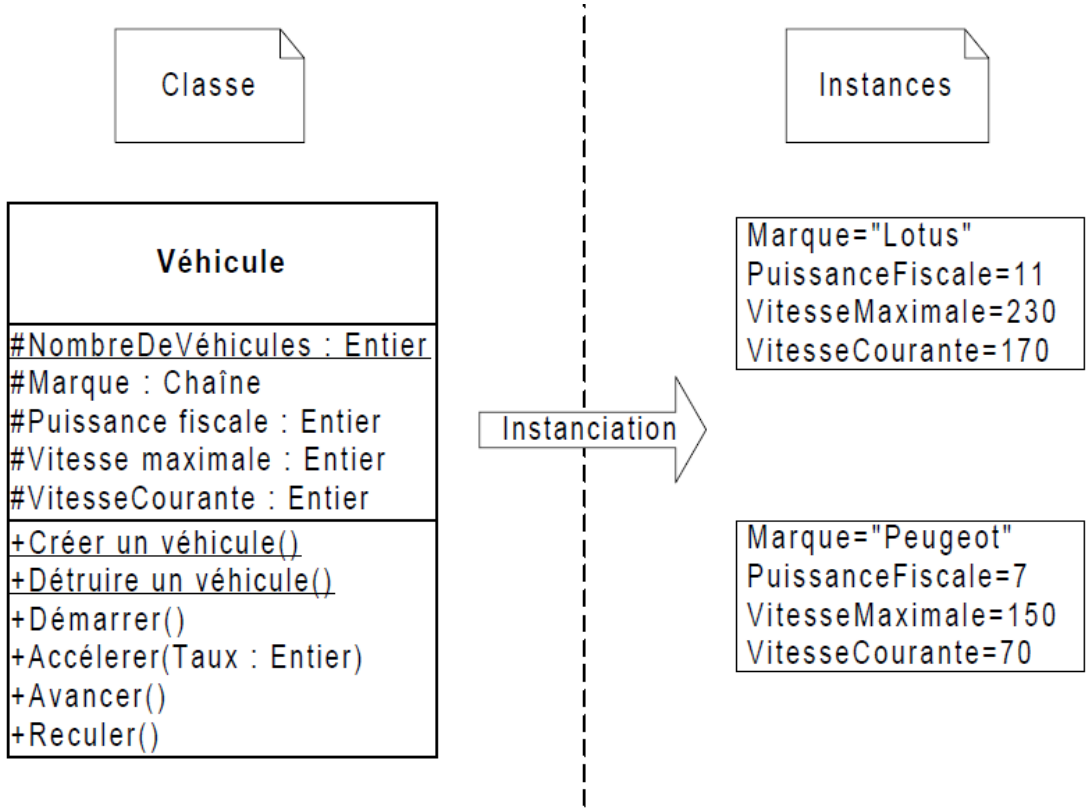


Figure 1.2 Instanciation d'une classe en deux objets

4) Le principe d'encapsulation

Ce principe prône les idées suivantes :

- Un objet rassemble en lui même ses données (les attributs) et le code capable d'agir dessus (les méthodes),
- Abstraction de données : la structure d'un objet n'est pas visible à l'extérieur, son interface est constituée de messages invocables par un utilisateur. La réception d'un message déclenche l'exécution de méthodes.
- Abstraction procédurale : du point de vue de l'extérieur l'invocation d'un message est une opération atomique. L'utilisateur n'a aucun élément d'information sur la mécanique

interne mise en œuvre. Par exemple, il ne sait pas si le traitement requis a demandé l'intervention de plusieurs méthodes ou même la création d'objets temporaires ...

Dans les versions canoniques du paradigme objet, les services d'un objet ne sont invocables qu'au travers de messages (méthode), lesquels sont individuellement composés de:

- Un nom,
- Une liste de paramètres en entrée,
- Une liste de paramètres en sortie.

On précise trois modes d'accès aux attributs et méthodes d'un objet :
- Le mode **public** avec lequel les éléments seront accessibles directement par l'objet lui-même ou par d'autres objets. Il s'agit du niveau le plus bas de protection.
- Le mode **privé** avec lequel les éléments de l'objet seront inaccessibles à partir d'autres objets : seules les méthodes de l'objet pourront y accéder. Il s'agit du niveau le plus fort de protection.
- Le mode **protégé :** avec lequel les attributs de l'objet seront accessibles par les méthodes de l'objet et les classes filles(Héritage).

5) Attribut d'instance, attribut de classe, méthode d'instance, méthode de classe

Dans le modèle précédent, un véhicule est représenté par une chaîne de caractères (sa *marque*) et trois entiers : la *puissance fiscale*, la *vitesse maximale* et la *vitesse courante*. Toutes ces données sont représentatives d'un véhicule particulier, autrement dit, chaque objet véhicule aura sa propre copie de ses données : on parle alors d'attribut *d'instance*. L'opération *d'instanciation* qui permet de créer un objet à partir d'une classe consiste précisément à fournir des valeurs particulières pour chacun des attributs d'instance.

En revanche, considérons l'attribut Nombre de véhicules chargé de compter le nombre de véhicules présents à un moment donné dans la classe. Il est incrémenté par l'opération Créer un véhicule et décrémenté par l'opération Détruire un véhicule. C'est un exemple typique d'attribut partagé par l'ensemble des objets d'une même classe. Il est donc inutile et même dangereux (penser aux opérations de mise à jour) que chaque objet possède sa copie propre de cet attribut, il vaut mieux qu'ils partagent une copie unique située au niveau de la classe. On parle donc d'attribut de classe.

Le même raisonnement s'applique directement aux méthodes. En effet, de la même manière que nous avions établi une distinction entre attributs d'instance et attributs de classe, nous allons différencier méthodes d'instances et méthodes de classe.

Prenons par exemple la méthode Démarrer. Il est clair qu'elle peut s'appliquer individuellement à chaque véhicule pris comme entité séparée. En outre, cette méthode va clairement utiliser les attributs d'instance de l'objet auquel elle va s'appliquer c'est donc une méthode d'instance.

Considérons désormais le cas de la méthode Créer un véhicule. Son but est de créer un nouveau véhicule, lequel aura, dans un second temps, le loisir de positionner des valeurs initiales dans chacun de ces attributs d'instance. Si nous considérons en détail le processus permettant de créer un objet, nous nous apercevons que la première étape consiste à allouer de

la mémoire pour le nouvel objet. Hors cette étape n'est clairement pas du ressort d'un objet : seule la classe possède suffisamment d'informations pour la mener à bien : la création d'un objet est donc en partie une méthode de classe. Notons également qu'au cours de cette étape, l'objet recevra des indications additionnelles, telles que, par exemple, une information lui indiquant à quelle classe il appartient. En revanche, considérons la phase d'initialisation des attributs. Celle-ci s'applique à un objet bien précis : celui en cours de création. L'initialisation des attributs est donc une méthode d'instance.

Nous aboutissons finalement au constat suivant : la création d'un nouvel objet est constituée de deux phases :

Une phase du ressort de la classe : allouer de la mémoire pour le nouvel objet et lui fournir un contexte d'exécution minimaliste.

Une phase du ressort de l'objet : initialiser ses attributs d'instance.

Si ces deux phases sont clairement séparées dans un langage tel que l'objective C, elles ne le sont plus en C++ ou en Java qui agglomèrent toute l'opération dans une méthode spéciale, ni vraiment méthode d'instance, ni méthode de classe : le constructeur.

Remarque

La liste des messages auxquels est capable de répondre un objet constitue son interface : c'est la partie publique d'un objet. Tout ce qui concerne son implémentation doit rester caché à l'utilisateur final : c'est la partie privée de l'objet. Bien entendu, tous les objets d'une même classe présentent la même interface.

6) La création d'une classe

Une classe est un type de données abstrait caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.

Les inscriptions précises permettent de créer les objets définis en deux temps :

- La création de la classe,
- La création effective et utilisable de l'objet comme instance au quelle on peut envoyer des messages.

a) La déclaration d'une classe

La création de la classe contient la déclaration de la classe (des attributs et méthodes) et la définition des méthodes.

La syntaxe de déclaration d'une classe est la suivante :

Classe « nomDeLaclasse »

Debut

Déclaration des attributs

Signature des methodes

Fin

Exemple de la classe Compte :

La déclaration de la classe compte revient à définir son numéro et son solde et à préciser comment accéder à ces informations.

La classe compte possède les deux attributs numéro et solde et quatre méthodes :

- La méthode créer(chaine,réel), qui permet de donner des valeurs aux attributs,
- La méthode debiter (montant :réel), qui diminue le solde,
- La méthode crediter(montant :réel) qui augmente le solde,
- La méthode SoldeCourant() qui récupère le solde courant d'un compte.

Syntaxe de la classe Compte :

```
Classe Compte
    DEBUT
        Privé :
            Numero :chaine de caracteres
            Solde :réel
        Public :
            Procedure  Créer(chaine,réel)
            Procedure  Debiter(montant :réel)
            Procedure  Crediter(montant :réel)
            Fonction Solde_courant() :réel
    FIN
```

Remarque :

La précision d'une méthode notamment le type de ses arguments et du type de données de retour s'appelle sa signature.

7) Déclaration et création d'une instance d'une classe

Pour utiliser un objet d'une classe dans un programme on doit créer une instance de la classe c'est à dire une variable du type de la classe. Les instances indiquent qu'un identifiant est un objet de la classe et à ce titre se réfère à la classe et possède ses attributs et méthodes. La création d'une instance de classe se fait en deux étapes :

- La déclaration de l'instance,
- La création effective de l'instance.

a) La déclaration de l'instance

nomInstance :nomClasse

Exemple :

Cpte :Compte

b) La création effective de l'instance

La création effective de l'instance est effectuée grâce à la méthode de construction appelée **Constructeur et** à l'utilisation de **new.**

Dans l'exemple de la classe Compte c'est la méthode créer () qui remplace le constructeur.

Cpte=new Compte()

8) Accès aux éléments de la classe

Une fois qu'un objet est crée on peut accéder aux éléments de la classe par

nomObjet.nomMethode() pour une méthode

nomObjet.attribut pour un attribut

Exemple :Cpte.creer("0123123891",175000)

Cpte.Debiter(50000)

9) Représentation algorithmique de la classe compte

La création de la classe constitue la déclaration de la classe et la rédaction des corps des méthodes.

Exemple de la classe Compte :

- **Déclaration de la classe**

```
Classe Compte
    DEBUT
        Privé :
            Numero :chaine de caracteres
            Solde :réel
        Public :
            Procedure Créer(chaine,réel)
            Procedure  Debiter(montant :réel)
            Procedure  Crediter(montant :réel)
            Fonction SoldeCourant() :réel
    FIN
```

- Rédaction des corps des méthodes (définition des méthodes)

```
    Procedure Compte ::Créer(num :chaine,sol :réel)
        Début
            Numero<-num
            solde<-sol
        Fin
    Procedure Compte ::debiter(mtant :réel)
        Début
            solde<-solde-mtant
        Fin
```

```
Procedure Compte ::crediter(mtant :réel)
        Début
                solde<-solde+mtant
        Fin
Fonction Compte ::SoldeCourant() :réel
        Début
                Retourner solde
        Fin
```

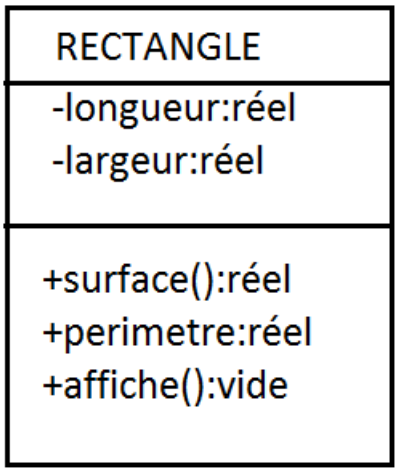
Exemple d'algorithme de manipulation

```
Algorithme manipCompte
Utilise Compte
Var  cp :Compte
Debut
        Cp=new Compte()
        Cp.creer("12112121212A ",150000) ;
        Afficher" le solde initial du compte est",Cp.SoldeCourant()
        Cp.crediter(40000)
        Afficher" le solde courant du compte est",Cp.SoldeCourant()
        Cp.debiter(20000) ;
        Afficher" le solde courant du compte est",Cp.SoldeCourant()
FIN
```

10) Constructeur et destructeur

Un constructeur est une méthode particulière d'une classe qui permet, lors de la création effective d'un objet, instance de la classe, d'initialiser ces attributs. Cette méthode porte le nom de la classe et ne retourne pas de valeur.

Exemple :



Représentation de la classe Rectangle

```
Classe Rectangle
DEBUT
        Privé
```

```
                Longueur :réel
                Largeur :réel
        Public
Rectangle (réel, réel)
Fonction surface () : réel
Fonction périmètre () : réel
Procédure affiche ()
FIN

Rectangle :: Rectangle(lo :réel,lar :réel)
DEBUT
        Longueur<-lo
        Largeur<-lar
FIN

Fonction  Rectangle :: Surface() :réel
DEBUT
        Retourner(longueur*largeur)
FIN

Fonction  Rectangle :: périmetre() :réel
DEBUT
        Retourner(2*(longueur+largeur))
FIN
Procédure  Rectangle ::affiche ()
DEBUT
Afficher(“rectangle de longueur : “,longueur,“ et de largeur “,largeur)
FIN
Algorithme  testRectangle
Var rect :rectangle
DEBUT
Rect=new Rectangle(12,5.6)
Afficher rect.surface()
Afficher rect.perimetre()
Rect.affiche()
FIN
```

Inversement, dans la plupart des langages une méthode est exécutée automatiquement quand l'instance devient hors de portée. Cette méthode qui est appelée avant la libération de l'espace mémoire associé à l'objet est appelée ***destructeur***.

11) Les accesseurs et les mutateurs

➢ La protection des données membres

Les données membres portant l'étiquette *privé* ne peuvent pas être manipulées directement par les fonctions membres des autres classes. Ainsi, pour pouvoir manipuler ces données membres, on doit prévoir des méthodes spéciales portant l'étiquette *public*, permettant de manipuler ces données.

- Les fonctions membres permettant d'accéder aux données membres sont appelées **accesseurs**, parfois *getter* (appellation d'origine anglophone).

- Les fonctions membres permettant de modifier les données membres sont appelées **mutateurs**, parfois *setter* (appellation d'origine anglophone).

➢ **La notion d'accesseur**

Un accesseur est une fonction membre permettant de récupérer le contenu d'une donnée membre. Un accesseur, pour accomplir sa fonction :

- doit avoir comme type de retour le type de la variable à renvoyer,
- ne doit pas nécessairement posséder d'arguments.
  Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom de l'accesseur par le préfixe *Get* :
  Exemple :

```
Fonction Rectangle ::getLongueur() :réel
DEBUT
        Retourner longueur
FIN
Fonction Rectangle ::getLargeur() :réel
DEBUT
        Retourner largeur
FIN
```

➢ **La notion de mutateur**

Un mutateur est une méthode (procédure membre) permettant de modifier le contenu d'une donnée membre. Un mutateur, pour accomplir sa fonction :

- doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre,
- ne doit pas nécessairement renvoyer de valeur (procédure).

Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom du mutateur par le préfixe *Set*.

Exemple

```
Procédure Rectangle ::setLongeur(l :réel)
DEBUT
        Longueur<-l
FIN
Procédure Rectangle ::setLargeur(l :réel)
DEBUT
        Largeur<-l
FIN
```

12) Attribut et méthode statique

Le mot clé **static**, utilisé pour un attribut(ou une méthode), permet d'indiquer que cet attribut (ou méthode) est commun à tous les objets de la classe concernée : il s'agit d'un attribut de la classe elle-même, et si on modifie cet attribut pour un objet donné, il sera modifié pour tous les objets de la classe (puisque c'est le même).

```
classe Voiture
        Debut
                prive
                        static  nombre = 0:entier ;
                        id:entier  ;
                public
                        Voiture(id:entier)
                        Static fonction getNombre():entier
        Fin
```

Un **attribut de classe** peut être accédé par n'importe quel nom de référence associé à un objet de cette classe, ou par le nom de la classe elle-même.

13) La référence sur l'objet courant : this

Ce mot désigne l'adresse de l'objet invoqué. Il est *utilisable uniquement* au sein d'une méthode. Le mot clé *this* permet de désigner l'objet dans lequel on se trouve, c'est-à-dire que lorsque l'on désire faire référence dans une méthode à l'objet dans lequel elle se trouve, on utilise *this*. C'est une référence sur l'objet courant.
L'objet courant *this* est en réalité une variable système qui permet de désigner l'objet courant.
this peut être utile :

- Lorsqu'une variable locale (ou un paramètre) "cache", en portant le même nom, un attribut de la classe.
- Pour déclencher un constructeur depuis un autre constructeur.

```
  Procédure Rectangle ::setLargeur(largeur :réel)
DEBUT
        This.largeur<-largeur
FIN
```

Chapitre 3 : HERITAGE et POLYMORPHISME

I. L'HERITAGE

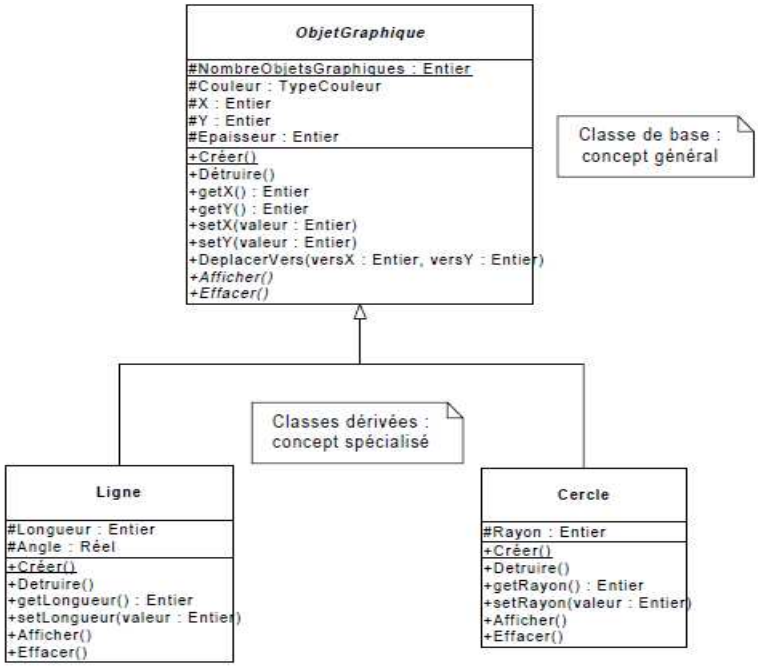
1) Définitions

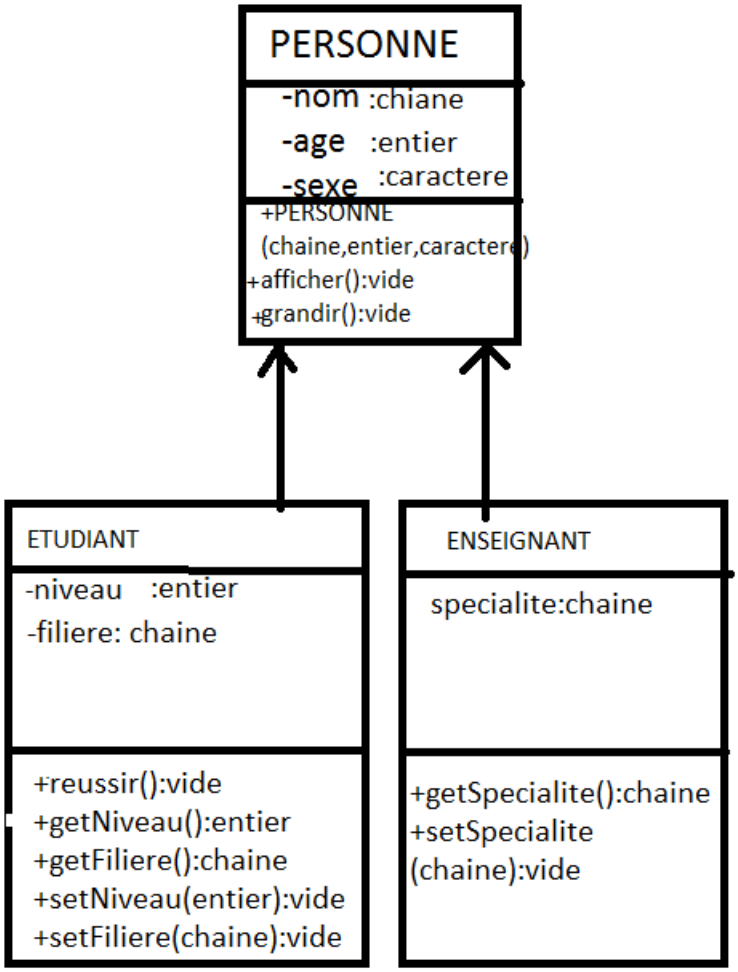
L'héritage est le second des trois principes fondamentaux du paradigme orienté objet (encapsulation, héritage, polymorphisme). Il est chargé de traduire le principe naturel de Généralisation / Spécialisation.
En effet, la plupart des systèmes réels se prêtent à merveille à une classification hiérarchique des éléments qui les composent. Le principe d'héritage est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.
En termes de concepts objets cela se traduit de la manière suivante :

On associe une classe au concept le plus général, nous l'appellerons *classe de base* ou *classe mère* ou *super - classe*.
Pour chaque concept spécialisé, on dérive une classe du concept de base. La nouvelle classe est dite *classe dérivée* ou *classe fille* ou *sous-classe*
On parle également de relation *est-un* pour traduire le principe de l'héritage.
L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Grâce à l'héritage, les objets d'une classe ont accès aux données et aux méthodes de la classe parente et peuvent les étendre.

2) Graphe d'héritage

Le graphe d'héritage d'un ensemble de classes est la structure représentant la manière dont ces classes sont liées les unes aux autres par héritage. Ce graphe constitue une hiérarchie.





3) Ecriture déclarative

```
Classe Personne
        Debut
                Privé
                        Nom :chaine
                        Age :entier
                        Sexe :caractere
                Public
                        Personne(chaine,entier,caractere)
                        Procedure Afficher()
                        Procedure grandir()
        Fin

Classe Etudiant hérite de Personne
        Début
                Privé
                        Niveau :entier
                        Filiere :chaine
                Public
                        Procedure reussir()
                        Procedure setNiveau()
```

```
                    Procedure setFiliere()
                    Fonction getNiveau() :entier
                    Fonction getFiliere() :chaine
        Fin
```

4) Héritage multiple

Certains langages comme le C++ permettent d'utiliser l'héritage multiple.
Il permet de créer des classes dérivées à partir de plusieurs classes de base.

5) Les méthodes virtuelles (redéfinition des méthodes)

Il existe des cas ou le type de l'instance qui appelle une méthode n'est pas connu des la compilation. Il ne sera déterminé qu'à l'exécution du programme. On peut souhaiter alors redéfinir les méthodes à différents niveau de la hiérarchie de la classe pour que la méthode effectivement mise en œuvre soit celle du type réel de l'instance au moment de l'exécution. Dans ce cas on dit que les méthodes sont virtuelles. On parle aussi de la redéfinition des méthodes dans les classes filles.

Il est donc possible de redéfinir dans une classe fille une méthode déjà définie dans la classe mère.

```
Classe Personne
        Debut
                Privé
                        Nom :chaine
                        Age :entier
                        Sexe :caractere
                Public
                        Personne(chaine,entier,caractere)
                        Procedure Afficher()
                        Procedure grandir()
        Fin

Classe Etudiant hérite de Personne
        Début
                Privé :
                        Niveau :entier
                        Filiere :chaine
                Public:
                        Etudiant(chaine,entire,caractere,entier,chaine)
                        Procedure afficher()
                        Procedure reussir()
                        Procedure setNiveau()
                        Procedure setFiliere()
                        Fonction getNiveu() :entier
                        Fonction getFiliere() :chaine
        Fin
Etudiant ::Etudiant(n :chaine,a :entier,s :caractere,ni :entier,f chaine) :Personne(n,a,s)
        DEBUT
                Niveau<-ni
                Filiere<-f
```

```
        FIN
Procedure Etudiant ::afficher()
        DEBUT
        Pesonne.afficher()
        Afficher “est un etudiant de niveau“,niveau,“ et de la filiere“,filiere
        FIN
Procedure Etudiant ::reussir()
        DEBUT
        Si niveau<5 alors niveau<-niveau+1
        Finsi
        FIN
Procedure Etudiant ::setNiveau(n :entier)
        DEBUT
        niveau<-n
        FIN
Procedure Etudiant ::setFiliere(f:chaine)
        DEBUT
        filiere<-f
        FIN
Fonction Etudiant ::getNiveau() : entier
        DEBUT
        Retourner niveau
        FIN
Fonction Etudiant ::getFiliere() : chaine
        DEBUT
        Retourner filiere
        FIN
```

6) La référence à la classe mère : super

Le mot clé **super** permet d'accéder à la version non redéfinie des attributs et méthodes de la super-classe. Il permet aussi d'appeler un des constructeurs de la classe mère, avec ou sans paramètres.

```
Etudiant ::Etudiant(n :chaine,a :entier,s :caractere,ni :entier,f chaine) :
        DEBUT
                super(n,a,s)
                Niveau<-ni
                Filiere<-f
        FIN
Procedure Etudiant ::afficher()
        DEBUT
        super.afficher()
        Afficher “est un etudiant de niveau“,niveau,“ et de la filiere“,filiere
        FIN
```

II. LE POLYMORPHISME

1) Définition

Le polymorphisme est le troisième des trois grands principes sur lequel repose le paradigme objet. Comme son nom l'indique le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes. Selon les langages, le polymorphisme pourra s'exprimer sur l'ensemble des classes d'un système alors que d'autres le confinent aux classes appartenant à une même hiérarchie.

2) La redefinition

Grâce à la redéfinition, il est possible de redéfinir une méthode dans des classes héritant d'une classe de base. Par ce mécanisme, une classe qui hérite des méthodes d'une classe de base peut modifier le comportement de certaines méthodes héritées pour être adaptées aux besoins de la classe fille

Nous allons démontrer la puissance du polymorphisme au travers de l'exemple classique des classes d'objets graphiques. Un document dessin peut être vu comme une collection d'objets graphiques qui va contenir des cercles, des rectangles, des lignes, ou toute autre sorte d'objet graphique qui pourrait dériver de la classe `ObjetGraphique`. En effet, une référence sur un objet d'une classe spécialisée peut toujours être affectée à une référence sur un objet d'une classe généraliste.

Si nous voulons dessiner un dessin tout entier, il nous faudra appeler la méthode Afficher pour chacun des objets contenus dans le dessin. Hors, nous avons pris soin de conserver la même signature pour les différentes méthodes Afficher de tous les objets appartenant à la hiérarchie `d'ObjetGraphique` : c'est la condition Sine Qua Non de l'utilisation du polymorphisme.

Le polymorphisme de la méthode `Afficher` garantit que la bonne méthode sera appelée sur chaque objet.

Classe ObjetGraphique
Début
Privé
Message :Chaine
Public
ObjetGraphique(m :chaine)
Procedure affiche()
finClasse
classe Ligne hérite de ObjetForme
Debut
Privé
Longueur :reel
public
Ligne(m :chaine,l :réel)
Procedure affiche()
finClasse
classe Rectangle herite de ObjetGraphique
Debut
Prive
Longueur :reel
Largeur :reel

Public
Rectangle(m :chaine,long :réel,larg :reel)
Procedure affiche()
finClasse

Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque : il s'agit du **polymorphisme d'héritage (la redéfinition)**.
Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objet, en les masquant par une interface commune (qui est la classe de base).
Imaginons un jeu d'échec comportant des objets
roi, *reine*, *fou*, *cavalier*, *tour* et *pion*, héritant chacun de l'objet *piece*.
La méthode *mouvement()* pourra, grâce au polymorphisme d'héritage, effectuer le mouvement approprié en fonction de la classe de l'objet référencé au moment de l'appel. Cela permettra notamment au programme de dire *piece.mouvement* sans avoir à se préoccuper de la classe de la pièce.

3) Une forme faible de polymorphisme : la surcharge

La surcharge est un mécanisme fréquemment proposé par les langages de programmation orientés objet et qui permet d'associer au même nom de méthode / fonction / procédure différentes signatures.
Par exemple, on pourrait proposer deux signatures différentes pour la méthode `Afficher` :
Pas d'argument si l'on désire utiliser le périphérique d'affichage par défaut
Spécification d'un périphérique en argument.

Le polymorphisme *paramétrique* rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.

La surcharge ou surdéfinition des méthodes consiste à définir des méthodes *différentes* portant *le même nom.* Dans ce cas, il faut les différencier par le type et/ou le nombre des arguments.
La surcharge permet de définir des méthodes portant le même nom, mais acceptant des paramètres de type différents et/ou en nombre différent. En fonction du type et du nombre des paramètres passés lors de l'appel, c'est une version ou une autre de la méthode qui sera effectivement appelée.

Attention : Le type de retour d'une méthode ne permet pas de différencier deux méthodes portant le même nom (si c'est la seule différence).

Ainsi, on peut par exemple définir plusieurs
méthodes homonymes *addition()* effectuant une somme de valeurs.

- La méthode *addition(entier, entier) :entier* pourra retourner la somme de deux entiers
- La méthode *addition(reel, reel)* :reel pourra retourner la somme de deux flottants
- La méthode *addition(chaine, chaine) : chaine* pourra définir au gré de l'auteur la somme de deux caractères

```
Classe Calcul
Début
Public
Fonction static addition(entier,entier) :entier
Fonction static addition(reel,reel) :reel
Fonction static addition(chaine,chaine) :chaine
finClasse
```

```
Classe Rectangle
DEBUT
        Privé
                Longueur :réel
                Largeur :réel
        Public
Rectangle (réel, réel)
Rectangle ()
Fonction surface () : réel
Fonction périmètre () : réel
Procédure affiche () : vide
FIN

Rectangle :: Rectangle(lo :réel,lar :réel)
DEBUT
        Longueur<-lo
        Largeur<-lar
FIN
Rectangle :: Rectangle()
DEBUT
        Longueur<-0
        Largeur<-0
FIN
```

Chapitre 4 : LES CLASSES ABSTRAITES, INTERFACE, LIEN ET AGREGATION

I. Les classes abstraites

1) Définition

Une classe est dite abstraite si elle ne peut pas être directement instanciée, elle peut ne pas fournir d'implémentation pour certaines de ces méthodes qui sont dites méthodes abstraites. Bien entendu, étant incomplète, une classe abstraite ne peut avoir d'instance. Il appartient donc à ces classes dérivées de définir du code pour chacune des méthodes abstraites. On parlera alors de classes *concrètes* ; les classes concrètes étant les seules à même d'être instanciées.

Une méthode est dite abstraite lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (*c'est à dire* on connaît sa signature (déclaration) mais pas sa définition).

On ne peut instancier une classe abstraite : elle est vouée à se spécialiser. Une classe abstraite peut très bien contenir des méthodes concrètes.

Une classe abstraite pure ne comporte que des méthodes abstraites. En programmation orientée objet, une telle classe est appelée une interface.

Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef *abstrait à* son nom.

Il est possible de ne pas définir une méthode virtuelle dans une classe : elle devra alors obligatoirement être définie dans les classes dérivées. On dit qu'une telle méthode est une méthode virtuelle pure(ou une méthode abstraite). Lorsqu'une classe contient une méthode abstraite, elle doit être déclarée classe abstraite.

2) Le but des classes abstraites

Le but des classes abstraites est de définir un cadre de travail pour les classes dérivées en proposant un ensemble de méthodes que l'on retrouvera tout au long de l'arborescence. Ce mécanisme est fondamental pour la mise en place du **polymorphisme**. En outre, si l'on considère la classe Véhicule, il est tout à fait naturel qu'elle ne puisse avoir d'instance : un véhicule ne correspond à aucun objet concret mais plutôt au concept d'un objet capable de démarrer, ralentir, accélérer ou s'arrêter, que ce soit une voiture, un camion ou un avion.

Prenons l'exemple des formes géométriques. C'est un bon exemple, on peut imaginer un programme où on veut calculer la somme des surfaces de plusieurs pièces de formes diverses. Le parcours de cet ensemble peut se faire à l'aide d'une référence sur une classe Forme, à la condition que toutes les classes décrivant des objets susceptibles d'apparaitre dans la collection soient dérivées de Forme. Pour que le calcul de surface puisse être fait par l'intermédiaire d'une référence sur Forme, il faut que cette classe dispose d'une méthode surface(). Toutefois le calcul de la surface d'une forme n'a pas de sens: On sait calculer par exemple la surface d'un rectangle ou d'un disque en fonction de leurs dimensions, mais la notion de forme est trop abstraite pour permettre la définition du corps de la méthode surface. On peut donc définir la classe de base ainsi :

```
classe abstraite Forme
DEBUT
```

```
public :
  fonction abstraite surface() :réel //une méthode abstraite
FIN
```

Bien que la classe Forme ne comporte qu'une seule méthode abstraite, elle n'est pas inutile puisqu'elle va permettre de mettre en œuvre le polymorphisme.
On peut avoir des classes filles suivantes :

La classe Cercle,

```
classe Cercle hérite de Forme
        Debut
        protégé :
         rayon :reel
        public :
        Cercle( r :reel)
        Fonction  surface():réel
        Fin
Cercle ::Cercle(r :reel)
Debut
        Rayon←r
Fin
Fonction Cercle ::surface() :reel
Var surf :reel
Debut
        surf←3.14*rayon*rayon
        retourner surf
Fin
```

La classe rectangle

```
classe Rectangle hérite de Forme
        Début
        Protégé :
                Largeur :reel
                Longueur :reel
        public:
        Rectangle( lg :reel, larg :reel)
        Fonction  surface():réel
        Fin

Rectangle ::Rectangle( lg :reel, larg :reel)
Début
Longueur←lg
Largeur←larg
Fin
Fonction Rectangle :: surface() :reel
        Debut
        retourner longueur*largeur
        Fin
```

Dans un programme principal on pourra manipuler les objets de ces classes :

```
Programme test
Var  tableau F[0..1] :Forme
```

```
C :Cercle
R :Rectangle
Surf :reel
Debut
 C=new Cercle(4.5)
 R=new Rectangle(4,5)
 Surf←0
 F[0]←C
F[1]←R
 Pour i de 1 à 2 faire
Surf←surf+F[i].surface()
finPour
afficher"la somme des surfaces est ",surf
Fin
```

Ainsi le rapport entre ces deux classes Rectangle et Cercle est qu'elle dispose chacune de la méthode surface avec la même signature. Elles sont toutes deux des redéfinitions de la méthode abstraite déclarée dans la classe Forme dont héritent Cercle et Rectangle. La seule raison d'être de la classe Forme est de permettre d'établir ce rapport syntaxique qui va par la suite nous permettre d'appeler ces deux méthodes très différentes comme une seule et unique méthode.
Le fait que Forme est une classe abstraite à deux avantages :

- il résout le problème du corps de la méthode Forme::surface() ,
- et indique clairement que forme n'existe que pour permettre l'utilisation du polymorphisme.

II. Les interfaces

1) Principe

Le langage Java offre un type de classe un peu particulier que l'on appelle Interface. Une interface est une classe abstraite dont toutes les propriétés sont des constantes de classe (publiques, statiques et finales), et toutes les méthodes sont par défaut abstraites (donc à redéfinir dans les classes qui « héritent » de cette interface). On retrouve la notion d'interface aussi en C# qui s'est inspiré de Java. Dans d'autres langages, la notion d'interface est parfois aussi présente mais avec des variantes et un autre nom.

2) Notion d'implémentation

Une classe n'hérite pas d'une interface : elle l'implémente. Implémenter une interface suppose respecter le modèle qu'elle impose (à travers ses méthodes abstraites) et permet d'accéder à ses constantes. Contrairement à l'héritage qui doit être unique (excepté en C++), l'implémentation peut être multiple : une classe peut implémenter plusieurs interfaces, donc respecter plusieurs modèles.
Une interface est par contre autorisée à hériter de plusieurs autres interfaces.
Exemples

```
interface Forme
public :
// ces méthodes sont abstraites
aire() : reel
```

```
perimetre() : reel
classe Carre implemente Forme
prive :
cote : reel
public :
Carre(cote : reel)
aire() : reel
perimetre() : reel
Classe Cercle implemente Forme
prive :
rayon : reel
public :
Cercle(rayon : reel)
aire() : reel
perimetre() : reel
```

Voici le contenu des 2 méthodes redéfinies dans la classe Carre :

```
// calcul de l'aire
Carre::aire() : reel
debut
retourner (cote * cote)
fi n
// calcul du périmètre
Carre::perimetre() : reel
debut
retourner (4*cote)
fi n
```

Voici le contenu des 2 méthodes redéfinies dans la classe Cercle :

```
// calcul de l'aire
Cercle::aire() : reel
debut
retourner (Math.PI * rayon * rayon)
fi n
// calcul du périmètre
Cercle::perimetre() : reel
debut
retourner (2 * Math.PI * rayon)
fin
```

Du coup, si vous créez un tableau basé sur l'interface Forme, vous allez pouvoir accéder aux aires et périmètres de tous les objets du tableau sans vous inquiéter du type de chaque objet.

```
// déclaration de la collection
colForme [1..10]:  Forme
// un objet pour récupérer les objets de la collection
uneForme : Forme
debut
...
colForme[1]←(new Carre(50))
...
// affichage des aires de chaque objet de la collection
pour i de 1 10 faire
// uneForme est du type de l'objet récupéré (Carre ou Cercle)
affi cher colForme[i].aire()
finpour
...
```

```
fin
```

III. LIEN ET AGREGATION

On regroupe sous cette notion les différents liens pouvant exister entre les objets au cours d'une application: l'instanciation, l'héritage, la référence et la composition.

1) L'instanciation (Lien : 'instance de')

Dans le monde réel, les objets sont classés en catégories suivant leurs caractéristiques (propriétés). Par exemple toutes les 'tasses de café' peuvent être caractérisée par une couleur, une quantité de boisson et une température, de même que toutes les tasses peuvent être remplies, vidées et lavées.

Ainsi avant de commencer à créer des objets, il est nécessaire de définir d'abord leurs classes d'appartenances. La définition d'une classe consiste à lui donner un nom unique et à définir les attributs et méthodes formant ses propriétés.

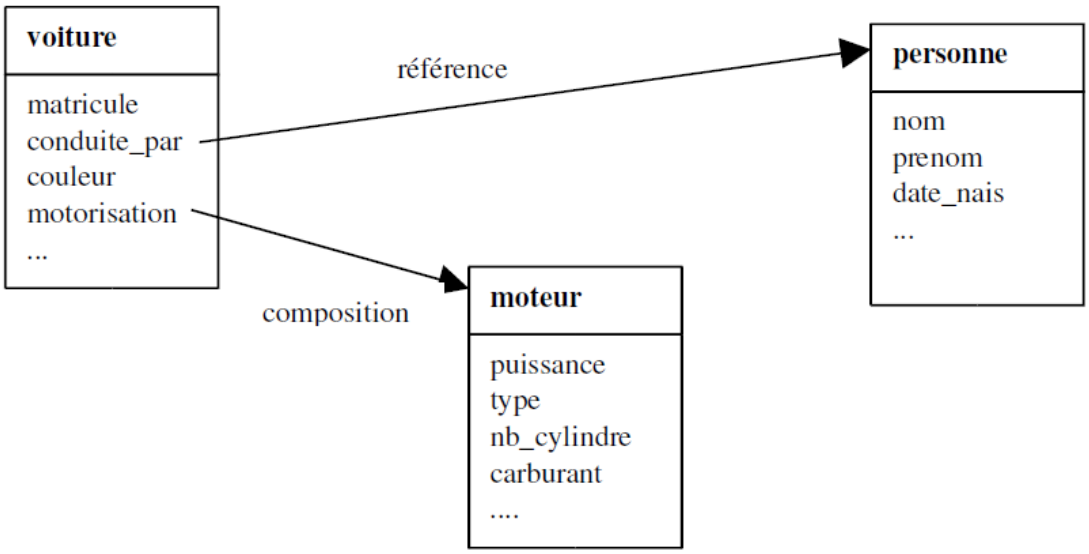
2) L'héritage (Lien : 'est-un')

Lorsqu'on déclare une nouvelle classe C, on a la possibilité de spécifier que cette nouvelle classe doit hériter des propriétés d'une ou de plusieurs classes mères. Ainsi les nouvelles propriétés définies, seront rajoutées à celles héritées pour former un nouveau concept plus spécifique que ceux déjà définis dans les classes mères. La classe C est alors une spécialisation de ses classes mères. Ces dernières sont appelées les super-classes de C.

3) Les objets complexes (Lien : 'référence-à' et 'composé-de')

Quand on veut représenter un objet du monde réel, on est souvent confronté au problème de la complexité du concept réel, car beaucoup d'objets courants, sont inter-dépendants et/ou composés les uns par rapport aux autres. Par exemple un objet voiture est composé d'un grand nombre d'objets d'autres types comme un moteur, des roues, une carrosserie, un système de freinage, une boite à vitesse, etc ... Chacun de ces objets (par exemple le moteur) est lui-même composé d'une multitude d'autres sous-objets (des pistons, des soupapes, des joints, des boulons, ...). Ce type d'objet aussi complexe soit-il est manipulé de la même manière qu'un objet simple (comme un entier ou une chaîne de caractères).Pour pouvoir modéliser des objets complexes, on a la possibilité, lors de la déclaration d'une classe, de définir des attributs ayant comme type une classe déjà définie. Cette manière de procéder permet d'établir des associations (liens de référence ou de composition) entre plusieurs classes.

Un lien de composition peut exister par exemple entre la classe voiture et la classe moteur. Alors qu'entre cette même classe voiture et la classe personne il y a un lien de référence, pour indiquer qu'une voiture est conduite par une personne.



Un lien de composition est un lien de référence sémantiquement plus riche. Par exemple si v est un objet de la classe voiture, composé du moteur m et conduite par la personne p. L'existence de l'objet m dépend de l'objet v, car c'est un sous-objet ou encore un objet composant. Par contre l'objet p ne fait pas partie de l'objet v, il ne fait pas partie de ses objets composants. Donc si on décide de détruire l'objet v (par exemple la voiture en question ne fait plus partie du parc automobile de l'entreprise) tous ses composants doivent aussi être détruits (son moteur m) de la base de données, par contre son conducteur p reste non affecté par cette opération (il conduira un autre véhicule par exemple).

4) Agrégation et composition

Agrégation : A a B

quand : A contient un objet de type B en attribut et au moins un envoi de message.

Agrégation c'est comme une association, au niveau du code. C'est un cas particulier en fait, ou conceptuellement B est agrégé à A, il y a une relation contenu / contenant, aggrégé / aggrégeant, une idée de possession. La seule différence entre les deux c'est la sémantique du lien. Un homme est associé à sa femme (association) et il possède un compte en banque (aggrégation).

Il y a donc aussi une référence vers l'objet, car c'est une association, et toujours envois de message.

Exemple : soient les classes Voiture et Roue, une voiture a besoin dans sa définition de roues, mais ses roues peuvent être changées, et donc les roues ont une existence en dehors de la voiture. La voiture possède des roues, si la voiture va à la casse on peut en récupérer les roues pour les mettre sur une autre voiture.

Différence association / aggrégation : Un homme est associé à sa femme (association) et il possède un compte en banque (aggrégation).

Composition(Agrégation forte)

quand : A contient un objet de type B en attribut et crée éventuellement un emplacement mémoire pour cet objet (par un new).

Ce qui est important pour la composition ce n'est pas tellement qu'il créé l'objet, mais c'est surtout que si l'objet de type A disparaît, celui de B disparaît aussi. L'objet B n'a plus d'existence sans celui de A.

Donc, au niveau du code, il y aura en effet souvent (presque toujours même) A qui créera l'objet de type B (mais ça peut également arriver pour une aggrégation, il faut bien créer les objets à un moment ou un autre, par exemple lorsqu'on créé une voiture on va lui créer des roues, mais c'est une aggrégation faible car on peut les changer par la suite, et les roues peuvent exister sans la voiture). Ce qui est important donc, c'est que l'objet de type A est le seul à posséder ce référent vers l'objet de type B, de telle sorte que s'il meurt, l'objet de type B qui le compose meurt aussi.

Exemple : soient les classes Arbre et Feuille. Quand on créé un arbre, on en créé les feuilles, l'arbre est composé d'un tronc, de racines,… et de feuilles. Si l'arbre meurt, on ne sait pas en récupérer les feuilles pour les mettre sur un autre arbre -> elles meurent aussi.

Chapitre 5 : LES TABLEAUX, LES CLASSSES TECHNIQUES, CLASSE METIER ET CLASSE COLLECTIONS

I. LES TABLEAUX

Un tableau est un objet ayant un nombre d'éléments fixe et pouvant contenir des objets ou des types primitifs. Dans certains langages comme Java, un tableau est lui-même un objet.
Un tableau peut contenir des primitifs tous du même type : {1,2,3} ou {'a','b','c'}.
Un tableau peut également contenir des objets de mêmes types.

1. Déclaration simple de tableau d'objet

Dans certains langages orienté objet comme le C++ les tableaux d'objet sont définis comme les tableaux de types primitifs.

lesPièces[1..100] : tableau de PièceNonAgréée ou

lesPièces[100] : tableau de PièceNonAgréée ou

tableau F[10] :Forme

et pour accéder aux éléments du tableau :

F[1].surface() ;

2. Tableau comme Objet

Dans certains langages Orienté Objet comme Java le tableau est un objet :

Syntaxe de déclaration

Tableau de 5 entiers primitifs :
entier[] monTableau=new entier[5];
ou
entier monTableau[]=new entier[5];

La première notation est de loin la meilleure car de cette façon, nous visualisons bien que *monTableau* est de type tableau d'entiers primitifs. Dans la seconde notation, les crochets sont mis à la fin de la référence pour assurer une continuité avec les normes du C.

II. CLASSE METIER /CLASSE TECHNIQUE

Il existe deux grandes catégories de classes. Vous avez eu l'occasion de les découvrir lors des séquences précédentes. Certaines classes comportent des informations : elles Représentent une personne, un objet, un lieu… Ce sont les classes « métiers ». D'autres classes apportent des outils pour faciliter certaines manipulations : elles permettent de manipuler les curseurs, les dates… Ceux sont les classes « techniques ».
Classe métier : classe qui permet de décrire un objet qui a une vie propre.
Classe technique : classe « boite à outils » qui permet de faciliter certaines manipulations.

Exemple :

Classe Date
Attributs prives :
annee : Entier
mois : Entier
jour : Entier
Methode a portee classe :
fonction aujourdhui() : Date *// renvoie la date du jour*
// Exemple d'appel : an ←Date.aujourdhui().année()
// la variable entière an reçoit l'année de la date du jour
Methodes publiques :
fonction annee() : Entier *// renvoie l'année*
fonction mois() : Entier *// renvoie le mois*
fonction jour() : Entier *// renvoie le jour*
fonction difference(uneDate : Date) : Entier

fin classe

III. Les vecteurs (collections)

Lorsque l'on crée un tableau, il faut spécifier sa taille qui est fixe. Certains langages fournissent des objets très utilisé comme le vecteur (classe *java.util.Vector en Java*) ou collection. L'utilisation d'un vecteur (collection) plutôt qu'un tableau est souvent avantageuse lorsque la taille finale du tableau n'est pas connue à l'avance. Un vecteur est un tableau dynamique. La première cellule d'un vecteur peut avoir l'index zéro.

Une collection définit un groupe d'objets, sur lequel des opérations de lecture, insertion, suppression, déplacement et remplacement peuvent être effectuées.
Parmi les classes techniques, il existe un ensemble de classes spécialement dédiées à la manipulation d'un ensemble d'objets. Ces classes, appelées Collections, permettent de manipuler des ensembles d'objets avec les caractéristiques suivantes :
• les objets ne sont pas placés de façon contiguë dans la mémoire ;
• le programmeur ne s'occupe pas de la façon dont sont mémorisés les objets ;
• la taille de la collection est variable et s'adapte au nombre d'objets qu'elle contient ;
• l'ajout et la suppression d'objets ne nécessitent aucun décalage ;
• l'accès à un objet peut se faire séquentiellement ou par indice, voire par une clé prédéfinie (suivant le type de collection) ;
• certaines collections offrent des possibilités supplémentaires, comme la recherche, le tri…

Pourquoi « classe générique » ?

Une classe « générique » est une classe qui manipule un type non défini à l'avance (donc « générique ») et qui sera précisé au moment de l'instanciation de la classe.
Les collections sont génériques car effectivement elles permettent de manipuler un ensemble d'objets d'un type précis, mais qui ne sera défini qu'au moment de la création de la collection (

Exemple :

Classe Collection
// Il s'agit d'une classe générique.

Méthodes publiques
fonction cardinal() : Entier
// retourne le nombre d'éléments de la collection
fonction existe(unObjet : Objet) : Booléen
// teste si unObjet fait partie de la collection
fonction index(unObjet : Objet) : Entier
// retourne l'index de l'objet passé en paramètre dans la collection,
// le premier objet de la collection a pour index 1
fonction donnerObjet(unIndex : Entier) : Objet
// retourne l'objet qui se trouve à l'index passé en paramètre
procédure ajouterObjet(unObjet : Objet)
// ajoute unObjet à la collection
procédure remplacerObjet(unIndex : Entier, unObjet : Objet)
// remplace, dans la collection, l'objet figurant à l'index passé en paramètre
// par l'objet passé en paramètre
procédure retirerObjet(unObjet : Objet)
// supprime de la collection l'objet passé en paramètre
procédure vider( )
// vide le contenu de la collection

Fin Classe Collection

// Pour instancier une collection
uneCollection : Collection de MaClasse *// la collection instanciée contiendra des objets de la //classe MaClasse*