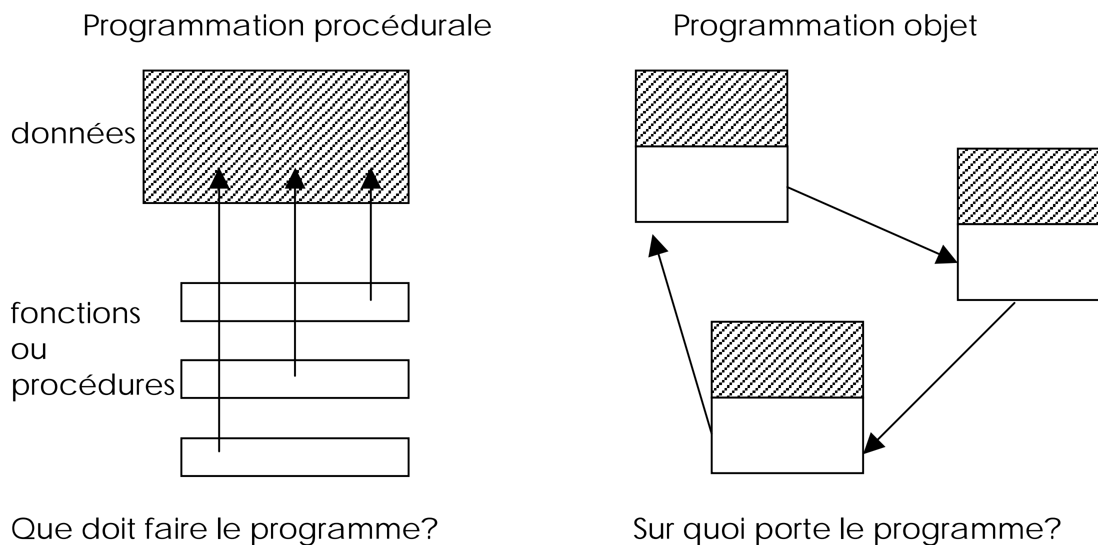
Approche orientée-objet

L'approche orientée-objet induit une nouvelle culture du développement logiciel. Elle nécessite une **rupture avec les pratiques de programmation traditionnelles** (procédurales).
Dans l'approche procédurale (fonctionnaliste) les données sont séparées des fonctions et procédures qui les utilisent.
Dans l'approche objet **les données et les traitements concernant ces données sont regroupés dans des entités appelées objets**. Un programme objet est un réseau d'objets qui communiquent par l'envoi de messages pour réaliser un traitement.



I. Rappel sur la programmation procédurale

A. Les types

Le type d'une variable détermine un ensemble de valeurs possibles pour la variable et les opérations qu'il est possible de faire sur cette variable. Par exemple, l'opération modulo ne peut se faire que sur les variables de type entier, la multiplication ne peut se faire que sur les types numériques (réel et entier)…

Le type d'une variable peut
- soit être un type primitif (de base) appartenant au langage lui-même (entier, réel, chaîne, caractère, booléen).
- soit être défini par l'utilisateur.

Un type défini par l'utilisateur permet de représenter des variables structurées, de type enregistrement, composées d'autres variables appelées champs.

Un type est une sorte de moule qui sert à créer des variables. Chaque variable est un exemplaire, une occurrence de type.

ex:
Soit une structure de donnée permettant de mémoriser les informations concernant un compte bancaire (simplifié)

```
Tcompte : enregistrement
    numéro: entier
    nom : chaîne
    solde : réel
Finenregistrement
```

Une fois un type utilisateur déclaré, on peut déclarer des variables de ce type

```
Var
cpte1, cpte2 : Tcompte
```

L'accès aux champs des variables structurées se fait grâce à l'opérateur point '.'
Ainsi, on peut valoriser ces champs

```
        cpte1. nom ← "Dupond"
        cpte1.numéro ← 1032
        cpte1.solde ← 0
```

On pourrait aussi déclarer des variables de type Tcompte dans le tas. Dans ce cas, on ne déclare pas directement la variable mais un pointeur (parfois appelé référence) sur cette variable. Ensuite seulement, la variable est créée dans le tas par l'opérateur nouveau. L'accès au champ se fait ensuite à travers le pointeur qui est déréférencé.

```
Var
   pcpte : pointeur sur Tcompte
Début
   pcpte ← nouveau Tcompte
   pcpte -> nom ← "Dupond"          //ou (*pcpte).nom = "Dupond"
   pcpte -> numéro ← 1032
   pcpte -> solde ← 0
```

B. Traitements sur les données : sous-programmes

En programmation procédurale, si nous voulons effectuer des traitements sur les variables, nous pouvons définir des procédures et fonctions utilisant ces variables. Ces sous-programmes sont définis en dehors des types de données qu'ils manipulent.
Les variables manipulées doivent être passées en paramètre des sous-programmes qui les manipulent.

Traitements effectués sur des comptes:

// Initialiser un compte (solde initialisé à 0):
Procédure Init(S UnCompte: Tcompte, E UnNuméro : entier, E Nom : chaîne)
// Créditer un compte:
Procédure Créditer (E/S UnCompte: Tcompte, E Montant : entier)
// Débiter un compte:
Procédure Débiter(E/S UnCompte: Tcompte, E Montant: entier)
// Connaître le solde d'un compte:
Fonction Solde (E UnCompte: Tcompte): réel

L'organisation des programmes procéduraux conduit à une séparation des données et des traitements.

La programmation objet au contraire permet de regrouper les données et les traitements sur ces données dans une seule entité appelée Classe

II. Notions d'objet et de classe

Un objet est un élément identifiable du monde réel qui est soit concret (une voiture, un stylo), soit abstrait (une entreprise, le temps).
Un objet est caractérisé par :

- ce qu'il est (c'est à dire les **données** sur lui même, son **état**)
- par ce qu'il sait faire (son **comportement**)

Un objet fait partie d'une catégorie d'objets appelée **classe**.
La classe est le type de l'objet.

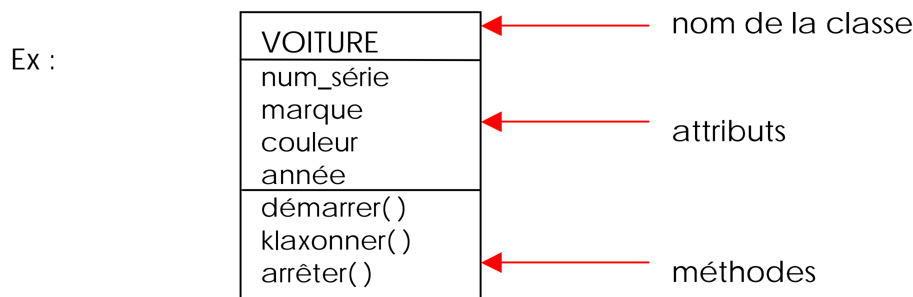
Différentes manières de dire la même chose:
Un objet est une **instance** de classe.
Un objet est une variable dont le type est sa classe. Réciproquement, on peut dire qu'une variable est un objet dont la classe est son type !
Un objet est un exemplaire d'une classe. La classe est comme un moule, à partir duquel on peut faire des gâteaux: les objets.

En programmation, une classe peut être considérée comme un cas particulier de structure (au sens du C/C++) composée de 2 catégories de champs:

- les **attributs** (champs de données ou propriétés représentant l'état des objets)
- les **méthodes** (fonctions ou procédures applicables aux objets, qui permettent son comportement)

Représentation d'une classe en notation **UML** (unified modeling language)
Une classe se représente comme une entité du modèle entité-association, sauf que la partie basse est divisée en deux parties : l'une pour les attributs, l'autre pour les méthodes.

Ex :



Le concept de classe se rapproche de la notion d'entité du modèle entité-association (MCD). Mais la notion de classe est un peu différente parce que l'entité ne s'intéresse qu'aux données (la structure), alors que la classe englobe la structure et les procédures de manipulation de ces données.

Notation algorithmique

```
Type
Classe Voiture
    attributs privés
        num_série : chaine
        marque : chaine
        couleur : chaine
        année : entier
    méthodes publiques
        procédure démarrer( )
        procédure klaxonner ( )
        procédure arrêter ( )
FinClasse
```

Une classe se déclare comme un type enregistrement auquel on ajoute le prototype des méthodes applicables aux objets de cette classe.

Lorsqu'une classe est déclarée, on peut déclarer des objets de cette classe. La syntaxe ne pose pas de problème particulier. Dans cet exemple, on déclare deux objets de type Voiture.

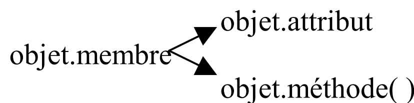
```
Var
   ma_voiture : Voiture          //nom de l'objet : Classe
   autre_voiture: Voiture
```

Accès aux attributs et aux méthodes

En programmation classique, on accède aux champs d'un enregistrement en utilisant l'opérateur point.

enregistrement.champ

En programmation objet, on accède aux champs (ou **membres**) de la même manière, avec l'opérateur point. La seule différence est que les champs d'un objet sont soit des attributs, soit des méthodes.



Remarque: Pour distinguer l'accès à un attribut de l'accès à une méthode, on met toujours des parenthèses après une méthode, même s'il n'y a aucun paramètre.

Ex :
soit ma_voiture un objet de type voiture
Pour accéder à sa couleur on écrit ma_voiture.couleur
Pour appeler la méthode démarrer on écrit ma_voiture.démarrer( )

III. Différences et points communs entre l'approche objet et l'approche procédurale

1. Au niveau des déclarations:

programmation objet

```
Type
Classe Voiture
   attributs privés
      num_série : chaine
      marque : chaine
      couleur : chaine
      année : entier
   méthodes publiques
      procédure démarrer( )
      procédure klaxonner ( )
      procédure arrêter ( )
FinClasse
```

programmation procédurale

```
Tvoiture: enregistrement
      num_série : chaine
      marque : chaine
      couleur : chaine
      année : entier
Finenreg

//sous-programmes de manipulation
procédure démarrer(une_voiture : Tvoiture )
procédure klaxonner (une_voiture : Tvoiture)
procédure arrêter (une_voiture : Tvoiture )
```

Différences au niveau du regroupement

- en programmation objet, les données et les traitements sont regroupés dans une classe
- en programmation non objet, les données sont regroupées dans un enregistrement mais les traitements sont déclarés à part

Différences au niveau des paramètres

- les méthodes ne prennent pas en paramètre l'objet qu'elles manipulent. En effet, une méthode est toujours appelée par un objet et la méthode appelée "sait" toujours quel objet l'a appelée
- les sous-programmes prennent en paramètre le nom de la structure à manipuler.

2. Au niveau de l'utilisation des traitements

En **programmation procédurale**, pour faire démarrer ma_voiture, il aurait fallu appeler la procédure démarrer avec le paramètre ma_voiture

démarrer(ma_voiture)

En **programmation objet**, pour faire démarrer ma_voiture, c'est l'objet ma_voiture qui fait appel à sa méthode démarrer

ma_voiture.demarrer( )

En programmation objet, tous les traitements sont des méthodes, et sont donc appelées par un objet.

Cas particulier du programme principal:

- en C++ objet , la fonction main( ) est une fonction "à part", qui n'appartient à aucune classe. C'est une exception à la programmation objet
- en Java, en revanche, même la fonction main( ) est une méthode d'une classe. Cette classe est le point d'entrée du programme et peut ensuite faire appel à d'autres classe pour réaliser un programme complet.

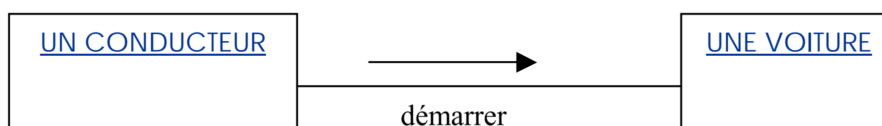
Exemple complet : voir l'annexe 1

IV. Les messages et les événements

Dans l'approche orientée-objet, toutes les fonctions et procédures sont intégrées dans les objets. De ce fait, toutes les actions sont réalisées par des objets. Le plus souvent, les traitements font appel à plusieurs objets différents. Il est alors nécessaire que les objets communiquent entre eux. La communication entre objets passe par l'**envoi de messages**. Un message est la demande d'un service ou plus techniquement c'est tout simplement l'**appel d'une méthode** pour un objet donné.

Par exemple, pour démarrer une voiture, il faut un objet voiture, mais il faut aussi qu'un objet conducteur demande à la voiture de démarrer. L'objet conducteur demande à la voiture de démarrer : on dit qu'il lui envoie un message.

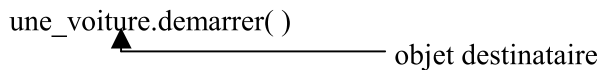
En UML, l'envoi d'un message entre deux objets se représente ainsi



- Un **objet** est représenté par un rectangle, son nom est souligné:
- Le **message** est représenté par une flèche orientée de l'émetteur au récepteur et par le nom du message.
- Le trait liant les deux objets symbolise une **relation**.

Le message sera contenu dans une méthode de la classe Conducteur (expéditeur) et aura la forme suivante:

une_voiture.demarrer( )



A la réception de ce message, l'objet voiture va exécuter la méthode démarrer.

Un objet peut **exécuter une méthode** à la réception :

- soit d'un **message** venant d'un autre objet (ou du programme principal)
- soit d'un message venant du système d'exploitation (exemple : clic sur un bouton) : on appelle ce genre de message **événement**

V. Abstraction et encapsulation de données

1. Principe d'abstraction

C'est un principe qui permet de masquer la complexité des objets à l'utilisateur. C'est grâce à l'abstraction qu'un utilisateur peut utiliser un objet sans savoir comment il fonctionne (ex: un enfant sait se servir d'un téléviseur, malgré la complexité de cet appareil). Grâce à l'abstraction, la façon dont un objet est construit n'a pas d'influence sur la façon dont on l'utilise. Par exemple, c'est grâce à l'abstraction qu'une voiture diesel se conduit comme une voiture essence, alors que la technologie du moteur est très différente dans les deux cas.

Pour respecter le principe d'abstraction, un objet possède une interface, visible et manipulable par l'utilisateur de l'objet et une implémentation, c'est-à-dire la façon dont l'objet est construit, qui n'est pas visible pas l'utilisateur.

Exemple: Un téléviseur est un objet très complexe. Pourtant, un enfant de 3 ans sait l'utiliser car les commandes à connaître sont très simples: allumer, éteindre, changer de chaîne, augmenter ou baisser le son. L'utilisateur ne sait pas comment est fait un téléviseur mais cela ne l'empêche pas de l'utiliser correctement. C'est le principe de l'abstraction. L'interface d'un téléviseur est son écran bien-entendu mais aussi la télécommande et les boutons se trouvant sur le poste.

L'interface d'une classe est constituée de la signature[1] de ses méthodes publiques

2. L'encapsulation

Ce principe découle directement du principe d'abstraction. Un objet ne peut être manipulé que via les méthodes qui lui sont associées lors de sa création. **La structure interne de l'objet (ses attributs) est inaccessible directement: il faut passer par des méthodes qui constituent l'interface**. De ce fait, aucun utilisateur non autorisé ne pourra malencontreusement modifier l'objet d'une façon interdite. Les attributs, qui ne sont pas accessibles d'un autre objet, sont qualifiés de **privés**. Les méthodes, qui peuvent être appelées par d'autres objet (par l'envoi d'un message) sont qualifiées de **publiques**.

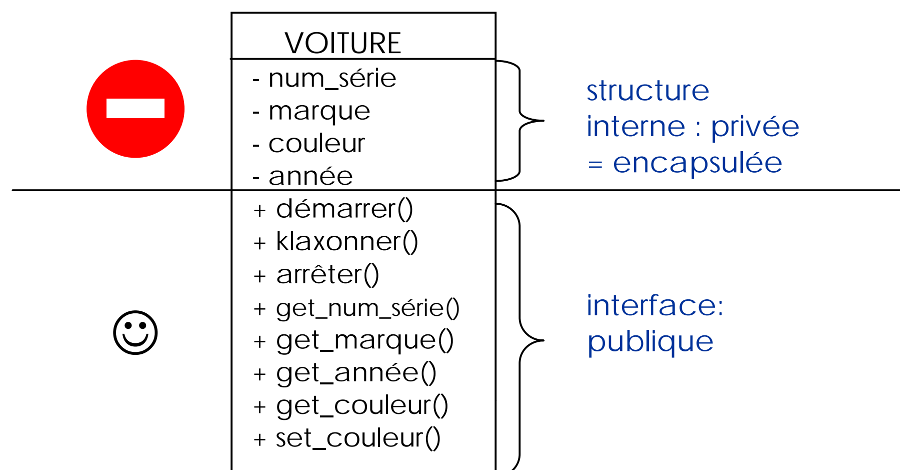
Ainsi, pour éviter que les utilisateurs ne cassent les composants (ou se fassent mal avec), les téléviseurs sont contenus dans une boite et les composants sont donc inaccessibles sans démonter l'appareil.

[1] Rappel: la signature d'une méthode est constituée de son nom ainsi que du nombre et du type de ses paramètres.

En programmation, c'est la même chose: l'encapsulation empêche que des bugs ne viennent accidentellement modifier les données d'un objet. L'encapsulation permet d'augmenter la fiabilité, la sécurité et l'intégrité des programmes.

L'accès direct aux attributs ne pouvant pas se faire directement, comment y accéder en cas de besoin? Tout simplement en définissant des méthodes qui permettent d'obtenir l'information voulue. Ces méthodes (publiques) permettant d'accéder aux attributs (privés) d'un objet sont appelés **accesseurs**. Il existe deux types d'accesseurs : les accesseurs en lecture (qui commencent souvent par get) et les accesseurs en écriture (qui commencent souvent par set).

Exemple et notation UML:



Notation :

L'indicateur privé est symbolisé par le signe -.

L'indicateur public est symbolisé par le signe +

L'accès direct au numéro de série, à la marque, à la couleur et à l'année d'une voiture ne peut pas se faire directement. On est obligé de passer par les accesseurs. Seul l'attribut couleur peut être modifié par l'appel de l'accesseur en écriture set_couleur. Les autres attributs ne peuvent pas être modifiés et c'est tant mieux car ce sont des attributs qui ne doivent jamais être changés.

☞ EXEMPLE:
Var
une_voiture: Voiture
…

☹
~~la_couleur ← une_voiture.couleur~~
~~une_voiture.couleur ← "vert"~~
~~une_voiture.marque ← Renault~~

☺
la_couleur ← une_voiture. get_couleur( )
une_voiture.set_couleur("vert")
impossible de modifier la marque d'une voiture

Cas particuliers:
Dans la majorité des cas, les attributs sont privés et les méthodes sont publiques. Mais il est possible de déclarer aussi des méthodes privées, utilisables seulement par les autres méthodes de la classe.
En Java comme en C++, on peut aussi déclarer des attributs publics (accessibles par d'autres objets). Mais dans ce cas, le principe de l'encapsulation est violé: ce n'est plus de la programmation objet pure.

VI. Implementation: définition des méthodes

Nous avons vu jusqu'à présent comment déclarer une classe et un objet, comment accéder aux attributs et aux méthodes d'un objet. Il nous reste à voir comment implémenter, c'est-à-dire coder les différentes méthodes.

Pour cela, basons-nous sur la classe Tcompte de l'annexe.
Nous allons implémenter ses méthodes

```
Procédure   Tcompte :: Créditer(E: Montant : réel)
Début
   solde ← solde + Montant
FinProc
```

Lorsque l'on définit une méthode, il faut spécifier à quelle classe elle appartient. Pour cela nous utiliserons l'opérateur double deux points ::

solde n'est ni une variable locale, ni un paramètre. Mais qu'est-ce donc alors??
En fait, solde représente l'attribut solde correspondant à l'objet de type Tcompte qui reçoit le message Créditer.

Cet objet peut être désigné explicitement par le mot clé this. This est remplacé au moment de l'appel par l'objet auquel est adressé l'appel.

```
Procédure   Tcompte :: Créditer(E: Montant : réel)
Début
   this.solde ← this.solde + Montant
FinProc
```

this est la plupart du temps implicite. Nous verrons cependant dans un TP qu'il est parfois utile de le mentionner explicitement.

Fonctionnement

```
Var  cpte1: Tcompte
…
cpte1.créditer(1000)     //message créditer envoyé à l'objet cpte1
```

Cet appel va entraîner l'exécution de l'instruction cpte1.solde ← cpte1.solde + Montant

<u>Implémentation des autres méthodes</u>

```
Procédure Tcompte::Débiter(E Montant: réel)
Début
   solde ← solde – Montant
FinProc

Fonction Tcompte::Solde( ): réel          //accesseur en lecture
Début
  retourne (solde)
FinFonct

Fonction Tcompte::Numéro( ): entier
Début
   retourne (numéro)
FinFonct
```

```
Fonction Tcompte :: Nom( ):chaîne
Début
   retourne (nom)
FinFonct

Procédure Tcompte :: Change_Nom( nv_nom: chaîne)   //accesseur en écriture
Début
    nom ← nv_nom
FinProc
```

VII. Exercice d'entrainement

I) Modification de la classe Tcompte
Ajoutez une procédure affiche( ) qui affiche l'état d'un compte

II) Mise en oeuvre de la classe Tcompte

1) Créez un programme principal qui déclare et initialise 3 objets de la classe Tcompte avec les données suivantes
   - Dupond, numéro de compte 1 et solde initial de 200F
   - Durand, numéro de compte 2 et solde initial 40F
   - Lajoie, numéro de compte 3 et solde initial 20000F

2) Affichez l'état de chacun des comptes
3) Augmentez chaque compte de 10%
4) Lajoie donne 5000F à Dupond et le reste à Durand

VIII. Les Constructeurs

Jusqu'à présent, les objets étaient initialisés par une méthode spécifique (Init) qui avait pour rôle de charger tout ou partie des attributs. Dans ce cas, un objet doit être déclaré puis initialisé. Mais si l'utilisateur (le programmeur qui utilise l'objet) oublie d'appeler la méthode d'initialisation, aucune erreur ne sera détectée par le compilateur. L'objet aurait alors des valeurs d'attributs indéterminées, ce qui pose des problèmes de robustesse.
En fait, la programmation objet a pour but de construire des programmes fiables. Il existe donc un mécanisme qui permet d'éviter les problèmes dus aux valeurs indéterminées, en rendant l'initialisation automatique à la déclaration de l'objet. Ce mécanisme est fondé sur les constructeurs.

Un constructeur est une méthode particulière qui est appelée automatiquement à chaque "création" d'objet (soit par déclaration, soit par allocation dynamique dans le tas). Cette méthode permet d'initialiser l'objet créé avec des valeurs valides.

Par convention, un constructeur porte le même nom que la classe et ne renvoie rien. En algorithmique, on fera précéder cette méthode du mot clé constructeur.

Exemple 1:

```
Classe personne
  attributs privés:
        nom: chaîne
        prenom : chaîne
        age : entier
  méthodes publiques:
        Constructeur personne( unNom : chaîne, unPrenom: chaîne, unAge: entier )
        …
FinClasse
```

```
Constructeur personne ( unNom : chaîne, unPrenom: chaîne, unAge: entier )
Début
  nom ← unNom
  prenom ← unPrenom
  age ← unAge
Fin
```

Instancier un objet avec son constructeur
Pour créer un objet dans la pile, il paraît naturel d'écrire:

```
p : personne
```

mais ceci est inexact car le constructeur, qui est appelé lors de la déclaration d'un objet, possède deux paramètres. Il faut donc que ces paramètres soient passés d'une manière ou d'une autre au constructeur lors la déclaration. On va donc utiliser la notation suivante:

```
Var
    p : personne("Dupont", "Toto", 20)
```

Et pour créer un objet dans le tas, il faut aussi lui passer le nombre de paramètres requis par le constructeur:

```
Var
  ppers : pointeur sur personne
Début
```

ppers ← **nouveau** personne("Durand", "Titi", 19)

Il n'est pas possible de créer un objet (par déclaration dans la pile ou allocation dans le tas) sans lui passer le nombre d'argument du constructeur.

☞ Remarque: Il est possible d'écrire une classe sans écrire son constructeur. Dans ce cas, le compilateur génère un constructeur implicite, qui ne prend aucun paramètres et qui initialise tous les attributs à la valeur par défaut de leur type (0 pour les numériques, NUL pour les pointeurs, "" pour les chaînes, etc)

Exemple 2:

```
Classe fraction
   Attributs privés
         numérateur : entier
         dénominateur : entier
   Méthodes publiques
         Constructeur fraction(n: entier, d: entier)
         Procédure Affiche ( )
FinClasse
```

```
Constructeur fraction(n: entier, d: entier)
Début
    numérateur ← n
    Tantque d ≠ 0 Faire
         Afficher "Erreur: le dénominateur ne peut pas être égal à 0. Veuillez saisir une nouvelle valeur"
         Saisir d
    FinTantque
    dénominateur ← d
Fin
```

```
Procédure fraction :: Affiche( )
Début
   Afficher "\n", n, "\" , d
Fin
```

Pour créer un objet fraction:

```
Var
         madivision : fraction(12 , 7)
```

Pour afficher madivision:

```
         madivision.Affiche( )              // donne 12/7 à l'écran
```

☞ Remarques:

- Un constructeur est appelé juste après l'allocation d'un emplacement mémoire pour l'objet.
- Un constructeur est utilisé à 95% pour effectuer des <u>initialisations</u> avec les valeurs qu'il reçoit en paramètre. Mais le traitement réalisé par un constructeur peut être beaucoup plus élaboré. Il n'y a aucune restrictions quand aux instructions qui peuvent être réalisées par un constructeur (excepté qu'un constructeur ne peut rien retourner). Par exemple, un constructeur peut vérifier que les valeurs des attributs passées en paramètre sont valides (et permettre une correction de l'erreur le cas échéant)
- Un constructeur qui ne possède aucun paramètre est appelé <u>"constructeur par défaut"</u>
- La création et l'initialisation par le constructeur sont regroupées dans une seule instruction que l'on appelle <u>instanciation</u>

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Multiplicité des constructeurs

Rappel :
La **signature** d'un sous-programme ou d'une méthode comprend son nom, son type de renvoi ainsi que le nombre et le type de ses paramètres. Deux méthodes différentes peuvent porter le même nom à condition qu'elles n'aient pas la même signature.

Il est possible de déclarer plusieurs constructeurs différents pour une même classe, afin de permettre plusieurs manière d'initialiser un objet. Les constructeurs diffèrent alors par leur signature.

Exemple 2 (suite):
Classe fraction
  Attributs privés
    numérateur : entier
    dénominateur : entier
  Methodes publiques
    Constructeur fraction(n: entier, d: entier)
    Constructeur fraction(nb : entier)
    Procédure Affiche ( )
FinClasse

Constructeur fraction(n: entier, d: entier) //premier constructeur
Début
  numérateur ← n
  Tantque d ≠ 0 **Faire**
    Afficher "Erreur: le dénominateur ne peut pas être égal à 0. Veuillez saisir une nouvelle valeur"
    Saisir d
  FinTantque
  dénominateur ← d
Fin

Constructeur fraction(nb: entier) //deuxième constructeur
Début
  numérateur ← n
  dénominateur ← 1
Fin
…

Remarque:
Dans ce dernier cas, comme le numérateur et le dénominateur sont de même type, on ne pourrait pas définir un troisième constructeur permettant de passer seulement le dénominateur en paramètre. En effet, un tel constructeur aurait la même signature que le précédent (rappelons que le nom des paramètres formels ne fait pas partie de la signature)

~~**Constructeur** fraction(d: entier)~~
~~**Début**~~
~~numérateur ← 1~~
~~dénominateur ← d~~
~~**Fin**~~

Au moment de l'instanciation, le choix du constructeur appelé se fait en fonction du nombre et du type des paramètres effectifs: le constructeur exécuté est celui dont la signature est cohérente avec les paramètres.

fraction(10, 3) → appel du premier constructeur (10/3)
fraction(5) → appel du deuxième constructeur (5/1)

Exercice sur les constructeurs

Voilà une classe produit, un programme principal qui l'utilise ainsi que sa sortie d'écran.
- Retrouver les constructeurs manquants (déclaration et implémentation).
- Ecrivez l'implémentation de la fonction prix_ttc et complétez le programme principal pour que s'affiche en plus le prix toutes taxes de la trousse.
(l'implémentation des autres classes n'est pas demandée)

```
Classe produit
  attributs privés
    libellé : chaîne
    code_tva : caractère
    prix_achat : réel
    stock : entier
  méthodes publiques
    …  //constructeurs manquants
    Procédure getlibellé( )
    Procédure changePrix(nvprix: réel)
    Procédure EntréeStock(quantité : entier)
    Procédure SortieStock(quantité : entier)
    Fonction prix_ttc( )
    Procédure Aff_tout( )
FinClasse

Procédure produit :: Aff_tout( )
Début
   Afficher libellé
   Selon code_tva Faire
        h : Afficher "taux de tva 19.6%"
        b : Afficher "taux de tva 5.5%"
        x : Afficher "taux de tva non déterminé"
   FinSelon
   Si prix_achat = 0
          Alors Afficher "prix d'achat non déterminé"
          Sinon Afficher "prix d'achat : ", prix_achat
    FinSi
    Si stock = 0
          Alors Afficher "aucun produit en stock"
          Sinon Afficher "quantité en stock : ", stock
    Finsi
FinProc

…  //Implémenter les constructeurs et la fonction prix_ttc

Programme testproddduit          /*programme principal*/
Var
  prod1: produit ("stylo")
  pprod2: pointeur sur produit
Début
  Afficher "Saisir le libellé du produit"
  Saisir le_lib
  Afficher "Saisir le code tva du produit (h pour 19.6, b pour 5.5 et x pour inconnu)"
  Saisir le_code
  Afficher "Saisir le prix d'achat du produit"
  Saisir le_prix
  Afficher "Saisir la quantité du produit en stock"
  Saisir le_stock
```

```
  pprod2 ← nouveau produit(le_lib, le_code,  le_prix, le_stock)
  Afficher "Voici l'état des deux produits instanciés"
  prod1.Aff_tout( )
  pprod2->Aff_tout( )
Fin
```

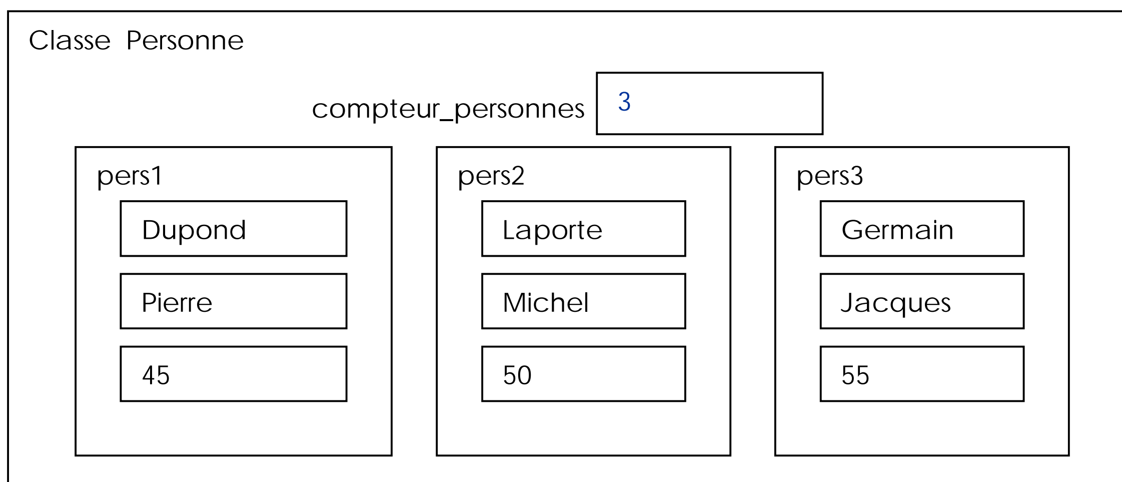
Sortie d'écran (les valeurs en italique sont saisies par l'utilisateur)

Saisir le libellé du produit
trousse
Saisir le code tva du produit (h pour 19.6, b pour 5.5 et x pour inconnu)
19.6
Saisir le prix d'achat du produit
20.30
Saisir la quantité du produit en stock
10
Code tva inconnu. Tapez la lettre correspondant à votre choix: h pour 19.6, b pour 5.5, ou x pour inconnu
votre choix: *h*
Voici l'état des deux produits instanciés
stylo
taux de tva non déterminé
prix d'achat non déterminé
aucun produit en stock
trousse
taux de tva : 19.6
prix d'achat : 20.30
quantité en stock : 10

IX. Les membres statiques (membres de classe)

A. Les attributs statiques

Chaque objet d'une classe possède sa propre version de chacun des attributs de la classe. Autrement dit, un objet ne partage pas ses attributs avec les autres objets de sa classe. Pourtant, dans certains cas, il est nécessaire de mémoriser une donnée commune à tous les objets d'une classe (par exemple, le nombre d'objet appartenant à la classe). Dupliquer la donnée dans chacun des objets serait gâcher du temps et de l'espace mémoire. C'est pourquoi il existe des propriétés spéciales, les **attributs statiques ou attributs de classe**, qui ne sont crées qu'en un seul exemplaire commun à tous les objets de la classe.



Les attributs statiques, s'il en existe, seront déclarés au début de la section des attributs et sont suivis du mot clé souligné statique.

Classe Personne
attributs privés
compteur_personnes : entier **statique**
Nom: chaîne
Prenom: chaîne
Age: entier
méthodes publiques
…

Pour accéder à un attribut statique, on utilise généralement le nom de la classe.
ex:
Afficher "Le nombre de personnes est de ", Personne.compteur_personnes

Remarque
on peut aussi utiliser n'importe quel objet de la classe car tous les objets de la classe ont accès à l'attribut statique. Cependant, pour des raisons de lisibilité, ceci n'est guère conseillé.

☞ Exemple de manipulation d'un attribut statique
Pour compter le nombre d'objets d'une classe, il suffit de placer l'incrémentation de la variable statique à l'intérieur du constructeur des objets. Ainsi, à chaque fois qu'un nouvel objet est créé, la variable statique est incrémentée de 1. Une variable statique est toujours initialisée à 0 par défaut. Il est cependant conseillé de l'initialiser dès la déclaration

Classe Personne
attributs privés
compteur_personnes : entier **statique ← 0**
Nom: chaîne
Prenom: chaîne
Age: entier
méthodes publiques
Constructeur Personne(unNom, un Prenom, unAge)
…
FinClasse

Constructeur Personne (unNom, un Prenom, unAge)
Début
Nom ← unNom
Prenom ← unPrenom
Age ← unAge
compteur_personnes ← compteur_personnes + 1
Fin

Application:

Compléter la classe produit et son implémentation pour qu'elle permette de trouver et d'afficher:

- le nombre de produits instanciés
- le prix du produit le plus cher (hors taxe)

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

B. Les Méthodes statiques

Un attribut statique existe en dehors de toute instance de la classe. De la même manière, **une méthode statique peut être appelée indépendamment de tout objet**. Pour invoquer une méthode statique, on utilise le nom de la classe et non celui d'un objet de la classe. Une méthode statique peut donc être appelée alors qu'aucun objet de la classe n'existe.
Comme **une méthode statique** existe en dehors de tout objet, elle **ne peut manipuler que les attributs statiques et les autres méthodes statiques de sa classe**. Elle ne peut utiliser directement ni les attributs d'instances (non statiques), ni les méthodes d'instances (non statiques)[2].

X. Les avantages de l'approche objet

Etant donné les coûts liés au déploiement et à la maintenance des logiciels, les informaticiens ont toujours cherché à développer des méthodes d'analyse et de conception pour permettre un développement rapide, fiable et efficace. L'approche orientée-objet est née dans ce but. Elle offre des potentialités intéressantes face à la complexité croissante des nouveaux systèmes informatiques:
- réutilisabilité
- modularité
- prototypage
- maintenabilité

A. La réutilisabilité

Méditez cet extrait de "Au cooeur de ActiveX et OLE" de David Chappel

" Au cours des 35 dernières années, les concepteurs de matériel informatique sont paasés de machines de la taille d'un hangar à des ordinateurs portables légers basés sur de minuscules microprocesseurs. Au cours des mêmes années, les développeurs de logiciels sont passés de l'écriture de programmes en assembleur et en COBOL à l'écriture de programmes écrire plus grands en C et C++. On pourra parler de progrès (bien que cela soit discutable", mais il est clair que le monde du logiciel ne progresse pas aussi vite que le celui du matériel. Qu'ont donc les développeurs de matériels que les développeurs de logiciels n'ont pas?

La réponse est donnée par les composants. Si les ingénieurs en matériel électronique devaient partir d'un tas de sable à chaque fois qu'ils conçoivent un nouveau dispositif, si leur première étape devait toujours consister à extraire le silicium pour fabriquer des circuits intégrés, ils ne progresseraient pas bien vite. Or, un concepteur de matériel construit toujours un système à partir de composants préparés, chacun chargé d'une fonction particulière et fournissant un ensemble de services à travers des interfaces définies. La tâche des concepteurs de matériel est considérablement simplifiée par le travail de leur prédécesseurs.

La réutilisation est aussi une voie vers la création de meilleurs logiciels. Aujourd'hui encore, les développeurs de logiciels en sont toujours à partir d'une certaine forme de sable et à suivre les mêmes étapes que les centaines de programmeurs qui les ont précédés. Le résultat est souvent excellent, mais il pourrait être amélioré. La création de nouvelles applications à partir de composants existants, déjà testés a toutes les chances de produire un code plus fiable. De plus, elle peut se révéler nettement plus rapide et plus économique, ce qui n'est pas moins important."

[2] Néanmoins, si elle prend des objets en paramètre, elle pourra utiliser leurs attributs et leurs méthodes d'instance.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com