Introduction à la Programmation Orientée Objet

25/07/2004

Par Hdd34 (Programmation générale)

Apprendre simplement la Programmation Orientée Objet

Avant Propos

Dans ce tutoriel vous apprendrez à manier la Programmation Orientée Objet, ou POO pour les intimes, dans le cadre du langage Pascal Orienté Objet.
Loin d'être aussi complexe qu'elle peut le laisser transparaître, la POO peut se maîtriser rapidement au

point de ne plus pouvoir s'en passer.

Si la programmation dite *procédurale* est constituée de procédures et fonctions sans liens particuliers agissant sur des données dissociées pouvant mener rapidement à des difficultés en cas de modification de la structure des données, la **programmation objet**, pour sa part, tourne autour d'une unique entité : l'*objet*, offrant de nouvelles perspectives, et que je vous invite à découvrir de suite...

Attention !
Borland a longtemps employé le nom de *Pascal Objet* pour **Delphi**. Celui-ci a été récemment renommé *langage Delphi*. Nous n'aborderons pas dans ce tutoriel une approche spécifique à **Delphi**. Nous nous orienterons plus vers une approche générale du Pascal, **tous compilateurs Pascal confondus** sitôt que ceux-ci supportent la Programmation Orientée Objet, comme c'est le cas pour **Turbo Pascal**, **FreePascal**, **GNU Pascal**, ..., et bien sûr **Delphi**.

1. Vue d'ensemble de la POO

Avant de rentrer plus avant dans le sujet qui nous intéresse, nous allons commencer par poser un certain nombre de bases.

1.1. L'objet

Il est impossible de parler de Programmation Orientée Objet sans parler d'*objet*, bien entendu. Tâchons donc de donner une définition aussi complète que possible d'un *objet*.
Un *objet* est avant tout une **structure de données**. Autrement, il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. En cela, rien ne distingue un *objet* d'une quelconque autre structure de données. La principale différence vient du fait que l'*objet* **regroupe les données et les moyens de traitement de ces données**.

Un *objet* rassemble de fait deux éléments de la programmation *procédurale* :

Les **champs** :

Les *champs* sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un *champ* peut posséder un type quelconque défini au préalable : nombre, caractère, ..., ou même un type objet.

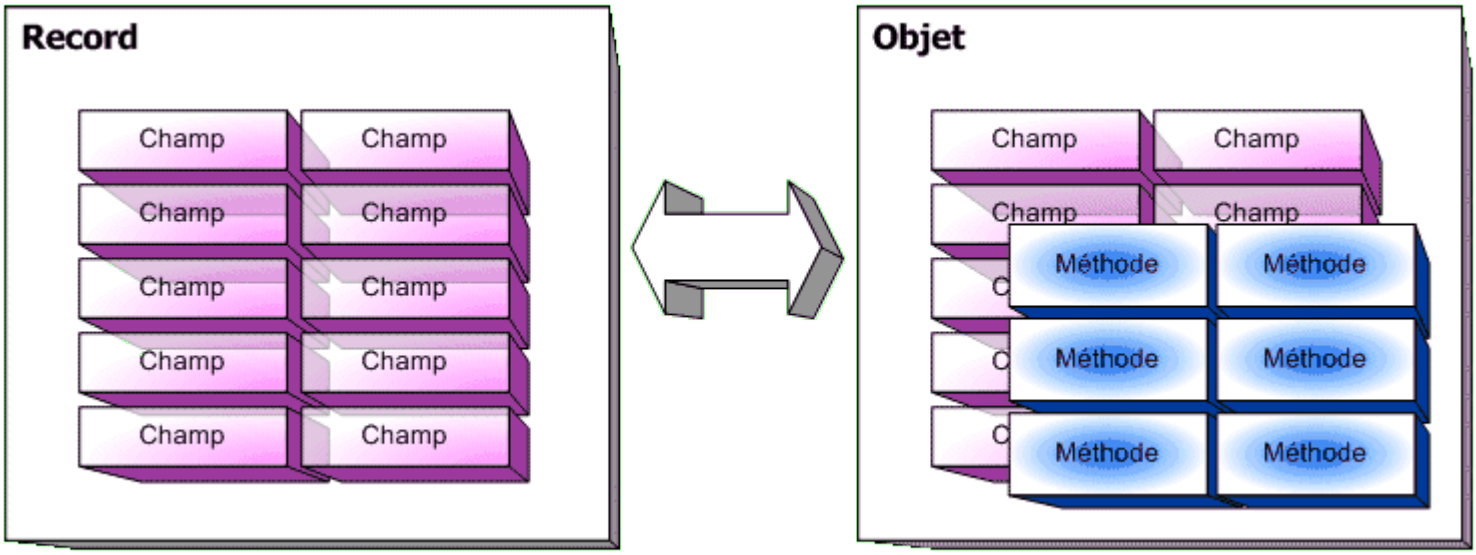
Les **méthodes** :

Les *méthodes* sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données.

Les champs et les méthodes d'un objets sont ses **membres**.

Si nous résumons, un *objet* est donc un type servant à stocker des données dans des *champs* et à les gérer au travers des *méthodes*.
Si on se rapproche du Pascal, un objet n'est donc qu'une extension évoluée des *enregistrements* (type **record**) disposant de procédures et fonctions pour gérer les champs qu'il contient.

On notera souvent les membres d'un objet **Objet.Membre** de façon à lever toute ambiguïté quant au propriétaire du membre considéré.



1.2. Objet et classe

Avec la notion d'**objet**, il convient d'amener la notion de **classe**. Cette notion de **classe** n'est apparue dans le langage Pascal qu'avec l'avènement du langage Delphi et de sa nouvelle approche de la Programmation Orientée Objet. Elle est totalement absente du Pascal standard.

Ce que l'on a pu nommer jusqu'à présent *objet* est, pour Delphi, une **classe d'objet**. Il s'agit donc du type à proprement parler. L'**objet** en lui-même est une **instance de classe**, plus simplement un exemplaire d'une classe, sa représentation en mémoire.
Par conséquent, on déclare comme type une *classe*, et on déclare des variables de ce type appelées des *objets*.

Si cette distinction est à bien prendre en considération lors de la programmation en Delphi, elle peut toutefois être totalement ignorée avec la plupart des autres compilateurs Pascal. En effet, ceux-ci ne s'appuient que sur les notions d'**objet** et d'**instance d'objet**.

Nous adopterons par conséquent ici ce point vue, qui simplifie le vocabulaire et la compréhension.

On pourra remarquer que FreePascal pour sa part définit une **classe** comme un **"pointeur vers un objet ou un enregistrement"**.

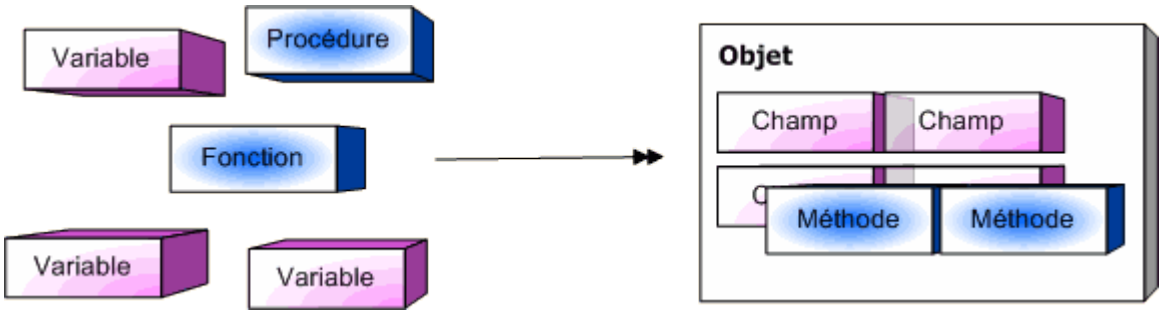
1.3. Les 3 fondamentaux de la POO

La Programmation Orientée Objet est dirigée par 3 fondamentaux qu'il convient de toujours garder à l'esprit : **encapsulation**, **héritage** et **polymorphisme**. Houlà ! Inutile de fuir en voyant cela, car en fait, il ne cachent que des choses relativement simples. Nous allons tenter de les expliquer tout de suite.

1.3.1. Encapsulation

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.
L'**encapsulation** introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.
Si l'**encapsulation** est déjà une réalité dans les langages procéduraux (comme le Pascal non objet par exemple) au travers des unités et autres librairies, il prend une toute nouvelle dimension avec l'**objet**.
En effet, sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion *interne* de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'**encapsulation** permet donc de masquer un certain nombre de champs et méthodes tout en laissant visibles d'autres champs et méthodes. Nous verrons ceci un peu plus loin.

Pour conclure, l'**encapsulation** permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.



1.3.2. Héritage

Si l'encapsulation pouvait se faire manuellement (grâce à la définition d'une unité par exemple), il en va tout autrement de l'**héritage**. Cette notion est celle qui s'explique le mieux au travers d'un exemple. Considérons un objet *Bâtiment*. Cet objet est pour le moins générique, et sa définition reste assez vague. On peut toutefois lui associer divers champs, dont par exemple :

Les murs
Le toit
Une porte
L'adresse
La superficie

On peut supposer que cet objet *Bâtiment* dispose d'un ensemble de méthodes destinées à sa gestion. On pourrait ainsi définir entre autres des méthodes pour :

Ouvrir le Bâtiment
Fermer le Bâtiment
Agrandir le Bâtiment

Grâce au concept d'**héritage**, cet objet *Bâtiment* va pouvoir donner naissance à un ou des *descendants*. Ces descendants vont tous bénéficier des caractéristiques propres de leur *ancêtre*, à savoir ses champs et méthodes. Cependant, les descendants conservent la possibilité de posséder leur propres champs et méthodes. Tout comme un enfant hérite des caractéristiques de ses parents et développe les siennes, un objet peut hériter des caractéristiques de son ancêtre, mais aussi en **développer de nouvelles**, ou bien encore se **spécialiser**.

Ainsi, si l'on poursuit notre exemple, nous allons pouvoir créer un objet *Maison*. Ce nouvel objet est toujours considéré comme un *Bâtiment*, il possède donc toujours des murs, un toit, une porte, les champs *Adresse* ou *Superficie* et les méthodes destinées par exemple à *Ouvrir le Bâtiment*.
Toutefois, si notre nouvel objet est toujours un *Bâtiment*, il n'en reste pas moins qu'il s'agit d'une *Maison*. On peut donc lui adjoindre d'autres champs et méthodes, et par exemple :

Nombre de fenêtres
Nombre d'étages
Nombre de pièces
Possède ou non un jardin
Possède une cave

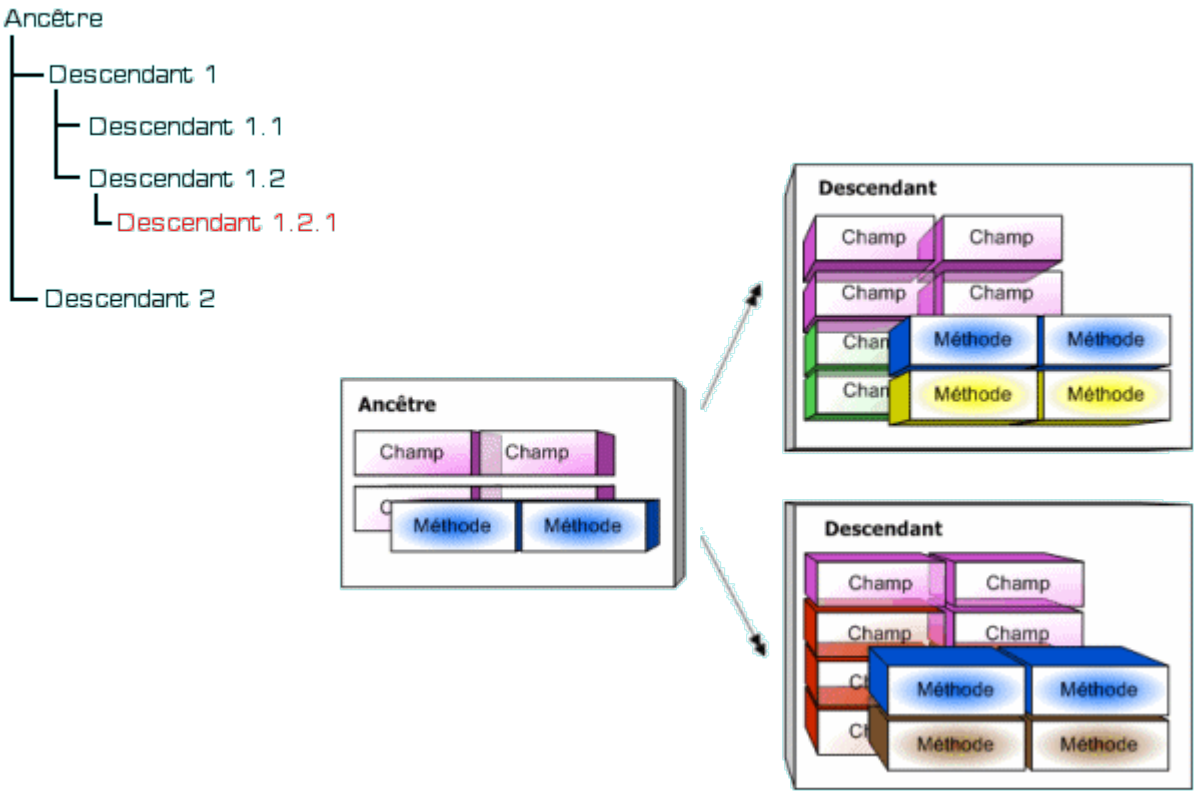
Notre *Bâtiment* a ici bien évolué. Il s'est **spécialisé**. Avec notre *Maison*, nous sommes allés plus avant dans les détails, et elle est à même de nous offir des services plus évoluées. Nous avons complété ce qui n'était qu'un squelette.

Ce processus d'héritage peut bien sûr être répété. Autrement dit, il est tout à fait possible de déclarer à

présent un descendant de *Maison*, développant sa spécialisation : un *Chalet* ou encore une *Villa*. Mais de la même manière, il n'y a pas de restrictions théoriques concernant le nombre de descendants pour un objet. Ainsi, pourquoi ne pas déclarer des objets *Immeuble* ou encore *Usine* dont l'ancêtre commun serait toujours *Bâtiment*.

Ce concept d'**héritage** ouvre donc la porte à un nouveau genre de programmation.

On notera qu'une fois qu'un champ ou une méthode est définie, il ou elle le reste pour tous les descendants, quel que soit leur degré d'éloignement.

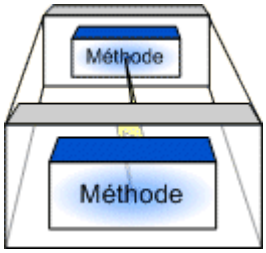


1.3.3. Polymorphisme

Le terme **polymorphisme** est certainement celui que l'on appréhende le plus. Mais il ne faut pas s'arrêter à cela. Afin de mieux le cerner, il suffit d'analyser la structure du mot : *poly* comme *plusieurs* et *morphisme* comme *forme*. Le **polymorphisme** traite de la capacité de l'objet à posséder *plusieurs formes*.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir **redéfinir une méthode** afin de la réécrire, ou de la **compléter**.
On voit donc apparaître ici ce concept de **polymorphisme** : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.
Le **polymorphisme**, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet *Véhicule* et ses descendants *Bateau*, *Avion*, *Voiture* possédant tous une méthode *Avancer*, le système appellera la fonction *Avancer* spécifique suivant que le véhicule est un *Bateau*, un *Avion* ou bien une *Voiture*.

Attention !
Le concept de **polymorphisme** ne doit pas être confondu avec celui d'*héritage multiple*. En effet, l'héritage multiple - non supporté par le Pascal standard - permet à un objet d'hériter des membres (champs et méthodes) de plusieurs objets à la fois, alors que le **polymorphisme** réside dans la capacité d'un objet à modifier son comportement propre et celui de ses descendants au cours de l'exécution.



2. Différents types de méthodes

2.1. Constructeurs et destructeurs

Parmi les différentes méthodes d'un objet se distinguent deux types de méthodes bien particulières et remplissant un rôle précis dans sa gestion : les **constructeurs** et les **destructeurs**.

2.1.1. Constructeurs

Comme leur nom l'indique, les **constructeurs** servent à **construire l'objet en mémoire.** Un **constructeur** va donc se charger de mettre en place les données, d'associer les méthodes avec les champs et de créer le *diagramme d'héritage* de l'objet, autrement dit de mettre en place toutes les liaisons entre les ancêtres et les descendants.

Il faut savoir que s'il peut exister en mémoire plusieurs instances d'un même type objet, autrement dit plusieurs variables du même type, seule **une copie des méthodes** est conservée en mémoire, de sorte que chaque instance se réfère à la même zone mémoire en ce qui concerne les méthodes. Bien entendu, les champs sont distincts d'un objet à un autre.
De fait, seules les données diffèrent d'une instance à une autre, la "machinerie" reste la même, ce qui permet de ne pas occuper inutilement la mémoire.
Certaines remarques sont à prendre en considération concernant les **constructeurs**.

> Un objet peut **ne pas avoir de constructeur** explicite. Dans ce cas, c'est le compilateur qui se charge de créer de manière **statique** les liens entre champs et méthodes.
> Un objet peut avoir **plusieurs constructeurs** : c'est l'utilisateur qui décidera du constructeur à appeler. La présence de constructeurs multiples peut sembler saugrenue de prime abord, leur rôle étant identique. Cependant, comme pour toute méthode, un constructeur peut être *surchargé*, et donc effectuer diverses actions en plus de la construction même de l'objet. On utilise ainsi généralement les constructeurs pour **initialiser les champs** de l'objet. A différentes initialisations peuvent donc correspondre différents constructeurs.
> S'il n'est pas nécessaire de fournir un constructeur pour un objet statique, il devient **obligatoire** en cas de **gestion dynamique**, car le *diagramme d'héritage* ne peut être construit de manière correcte que lors de l'exécution, et non lors de la compilation.

2.1.2. Destructeurs

Le **destructeur** est le pendant du constructeur : il se charge de **détruire l'instance de l'objet**. La mémoire allouée pour le *diagramme d'héritage* est libérée. Certains compilateurs peuvent également se servir des destructeurs pour éliminer de la mémoire le code correspondant aux méthodes d'un type d'objet si plus aucune instance de cet objet ne réside en mémoire.

Là encore, différentes remarques doivent être gardées à l'esprit :

> Tout comme pour les constructeurs, un objet peut **ne pas avoir de destructeur**. Une fois encore, c'est le compilateur qui se chargera de la destruction **statique** de l'objet.

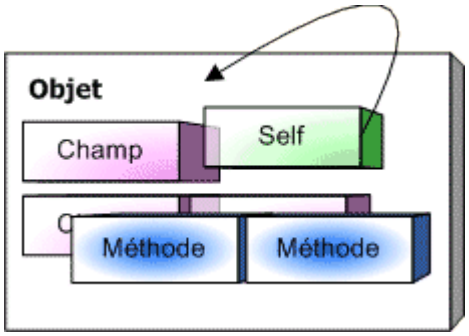
Un objet peut posséder **plusieurs destructeurs**. Leur rôle commun reste identique, mais peut s'y ajouter la destruction de certaines variables internes pouvant différer d'un destructeur à l'autre. La plupart du temps, à un constructeur distinct est associé un destructeur distinct.
En cas d'utilisation **dynamique**, un **destructeur s'impose** pour détruire le diagramme créé par le constructeur.

2.2. Pointeur interne

Très souvent, les objets sont utilisés de manière **dynamique**, et ne sont donc créés que lors de l'exécution. Si les méthodes sont toujours communes aux instances d'un même type objet, il n'en est pas de même pour les données.

Il peut donc se révéler indispensable pour un objet de pouvior se référencer lui-même. Pour cela, toute instance dispose d'un **pointeur interne** vers elle-même.

Ce pointeur peut prendre différentes appellations. En **Pascal**, il s'agira du pointeur **Self**. D'autres langages pourront le nommer *this*, comme le C++, ...



2.3. Méthodes virtuelles et méthodes dynamiques

2.3.1. Méthodes virtuelles

2.3.1.1. Principe

Une méthode dite **virtuelle** n'a rien de fictif ! Il s'agit d'une méthode dont la résolution des liens est effectuée dynamiquement. Voyons ce que cela signifie.

Comme nous le savons déjà, toute méthode est susceptible d'être surchargée dans un descendant, de manière à être écrasée ou complétée. Par conséquent, toute méthode surchargée donne lieu à création d'une nouvelle section de code, et donc à une nouvelle adresse en mémoire.
De plus, tout objet possède un lien vers la table des méthodes de ses ancêtres : le diagramme d'héritage. De fait, tout type objet est directement lié à ses types ancêtres. Autrement dit, si nous reprenons l'exemple du début, l'objet *Maison* peut être assimilé à un *Bâtiment*.

Considérons à présent la méthode *Ouvrir* d'un *Bâtiment*. Celle-ci consiste à ouvrir la porte principale.
A présent, surchargeons cette méthode pour l'objet *Maison*, de sorte que la méthode *Ouvrir* non seulement ouvre la porte principale, mais également les volets de notre *Maison*.

Déclarons maintenant une instance **statique** de *Bâtiment*, et appelons cette méthode *Ouvrir*. Lors de la création de l'exécutable, le compilateur va vérifier le type d'instance créé. Le compilateur lie alors notre appel à celui de *Bâtiment.Ouvrir* (la méthode *Ouvrir* de l'objet *Bâtiment*), en toute logique. Il ne se pose aucun problème.

Considérons à présent un autre exemple : déclarons une variable **dynamique** destinée, en principe, à recevoir un objet *Bâtiment*. Comme nous l'avons vu juste avant, l'objet *Maison* est **compatible** avec

l'objet *Bâtiment*. Comme nous travaillons en dynamique, nous nous servons de **pointeurs**. De fait, je peux très bien décider, avec cette variable pointant vers un objet *Bâtiment*, de déclarer une instance de type *Maison* : le compilateur n'opposera aucune réticence.
Si nous résumons, nous avons donc une variable de type officiel *pointeur vers Bâtiment* et contenant en réalité une *Maison*.
Appelons alors notre méthode *Ouvrir*. Comme nous avons une *Maison*, il faut que l'on ouvre les volets. Or, si nous exécutons notre programme, les volets resteront clos. Que s'est-il passé ?

Lors de la création du programme, le compilateur s'est arrêté sur notre appel à *Ouvrir*. Ayant déclaré un objet *Bâtiment*, le compilateur **ignore tout du comportement du programme lors de son exécution**, et par conséquent ignore que la variable de type *pointeur vers Bâtiment* contiendra à l'exécution un objet *Maison*. De fait, il effectue une liaison vers *Bâtiment.Ouvrir* alors que nous utilisons une *Maison* !

La solution, vous l'aurez compris, réside dans l'utilisation des **méthodes virtuelles**. Grâce à celles-ci, la résolution des liens est effectuée **dynamiquement**, autrement dit **lors de l'exécution**. Ainsi, si nous déclarons notre méthode *Ouvrir* comme **virtuelle**, lors de la création du programme, le compilateur n'effectuera aucune liaison statique devant notre appel. Ce n'est que lors de l'exécution, au moment de l'appel, que la liaison va s'effectuer. Ainsi, au moment où l'on désirera appeler *Ouvrir*, notre programme va interroger son **pointeur interne** pour déterminer son type. Bien évidemment, cette fois-ci, il va détecter une instance de *Maison*, et l'appel se fera donc en direction de *Maison.Ouvrir*. Les volets s'ouvrent...

Vous aurez noté toute l'importance des méthodes virtuelles. D'une manière générale, sitôt qu'une méthode est susceptible d'être surchargée, il faut la déclarer comme virtuelle.

Attention !
Les **constructeurs** des **objets** ne seront **jamais déclarés comme virtuels**, car c'est toujours le bon constructeur qui est appelé. Le caractère virtuel est donc inutile et sera même signalé comme une erreur par le compilateur.
Par contre, les **destructeurs** seront **toujours déclarés comme virtuels** car souvent surchargés.

Il n'en est pas de même pour les **classes** qui elles peuvent s'appuyer sur le principe de *constructeur virtuel*. C'est notamment le cas de **Delphi** avec les références de classes à propos desquelles la documentation donne plus de précisions.
Vous pouvez aussi consulter les tutoriels suivants :
* Cours sur la POO de Frédéric Beaulieu
* Cours sur les métaclasses de Laurent Dardenne

2.3.1.2. Constructeurs et Table des Méthodes Virtuelles

Afin de pouvoir appeler la méthode approprié au moment souhaité, un objet doit s'appuyer sur une liste de ses méthodes virtuelles : la **VMT** ou **V**irtual **M**ethods **T**able, la Table des Méthodes Virtuelles. Cette table est mise en place par les **constructeurs** d'un objet. Tout objet possède sa propre VMT, conservant toujours un lien avec la VMT de son ancêtre.

Lorsqu'un appel à une méthode virtuelle est effectuée, l'objet recherche dans sa VMT s'il trouve la méthode recherchée. Si c'est le cas, alors il utilise l'adresse enregistrée et exécute la méthode. Sinon, il parcourt la VMT de son ancêtre direct et ainsi de suite jusqu'à l'ancêtre le plus éloigné dans la hiérarchie. De même, lorsque qu'une méthode surchargée fait appel à la méthode ancêtre, alors une recherche est effectuée en partant cette fois-ci de la VMT du premier ancêtre.

La VMT est détruite par un **destructeur** lorsque celle-ci n'a plus lieu d'exister.

Si jamais on utilise une méthode **virtuelle** sans avoir appelé au préalable un **constructeur**, le caractère virtuel ne sera pas pris en compte et les résultats seront imprévisibles.

2.3.2. Méthodes dynamiques

Après les méthodes virtuelles, on se demande ce que l'on a pu inventer de pire ! Rassurez-vous, rien du tout. Les méthodes **dynamiques** ne sont en fait que des méthodes virtuelles.
Leur particularité réside dans le fait qu'elles sont **indexées**. Autrement dit, chaque méthode **dynamique**

possède un numéro unique pour l'identifier.

Il convient de les comparer aux méthodes virtuelles :

> **Avantage :** Les méthodes dynamiques consomment moins de mémoire
> **Inconvénient :** La gestion interne des méthodes dynamiques est plus complexe, et donc plus lente

Par conséquent, on préfèrera toujours les méthodes virtuelles, sauf si de nombreuses méthodes virtuelles doivent être déclarées, auquel cas on se reportera aux méthodes dynamiques.

2.4. Méthodes abstraites

Une méthode **abstraite** est une méthode qu'il est **nécessaire de surcharger**. Elle ne possède donc **pas d'implémentation**. Ainsi, si on tente d'appeler une méthode **abstraite**, alors une erreur est déclenchée.

Bien entendu, il convient lors de la surcharge d'une telle méthode de ne pas faire appel à la méthode de l'ancêtre...

Les méthodes **abstraites** sont généralement utilisées lorsque l'on bâtit un squelette d'objet devant donner lieu à de multiples descendants devant tous posséder un comportement analogue. On pourra prendre notamment l'exemple de l'objet **TStream** et de tous ses descendants.

3. Visibilité

De par le principe de l'encapsulation, afin de pouvoir garantir la protection des données, il convient de pouvoir *masquer* certaines données et méthodes internes les gérant, et de pouvoir laisser *visibles* certaines autres devant servir à la gestion publique de l'objet. C'est le principe de la **visibilité**.

3.1. Champs et méthodes publics

Comme leur nom l'indique, les champs et méthodes dits **publics** sont accessibles depuis tous les descendants et dans tous les modules : programme, unité, ...

On peut considérer que les éléments **publics** n'ont pas de restriction particulière.

Les méthodes publiques sont communément appelées **accesseurs** : elles permettent d'accéder aux champs d'ordre *privé*.
Il existe des accesseurs en *lecture*, destinés à récupérer la valeur d'un champ, et des accesseur en *écriture* destinés pour leur part à la modification d'un champ.
Il n'est pas nécessaire d'avoir un accesseur par champ privé, car ceux-ci peuvent n'être utilisés qu'à des fins internes.
très souvent, les accesseurs en *lecture* verront leur nom commencer par *Get* quand leurs homologues en *écriture* verront le leur commencer par *Set* ou *Put*.


Les **constructeurs** et les **destructeurs** éventuels d'un objet devront bénéficier de la visibilité **publique**, sans quoi un programme externe ne pourrait pas les appeler !


Attention !
Un champ ne devra être public **que si sa modification n'entraîne pas de changement dans le comportement de l'objet**. Dans le cas contraire, il faut passer par une méthode. Modifier un champ "manuellement" et ensuite appeler une méthode pour informer de cette modification est une **violation du principe d'encapsulation**.

3.1. Champs et méthodes privés

La visibilité **privée** restreint la portée d'un champ ou d'une méthode au **module où il ou elle est déclaré(e)**. Ainsi, si un objet est déclaré dans une unité avec un champ privé, alors ce champ ne pourra être accédé qu'à l'intérieur même de l'unité.

Cette visibilité est à bien considérer. En effet, si un descendant doit pouvoir accéder à un champ ou une méthode privé(e), alors ce descendant doit nécessairement être déclaré dans le **même module que son ancêtre**.

Généralement, les **accesseurs**, autrement dit les méthodes destinées à modifier les champs, sont déclarés comme **privés**.

3.1. Champs et méthodes protégés

La visibilité **protégé** correspond à la visibilité **privé** excepté que tout champ ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où il se situent.

Cette visibilité est souvent à préférer à la visibilité privée, cependant elle n'est pas supportée par tous les compilateurs. Par exemple, **Turbo Pascal ne la reconnaît pas**.

4. Le Pascal Objet

Tous les éléments de la Programmation Orientée Objet énoncés jusqu'ici sont bien entendu supportés par le Pascal Objet. Nous allons voir à présent comment les implémenter.

4.1. Déclaration d'un objet

4.1.1. Déclaration simple

Avant de pouvoir utiliser la POO, il convient de savoir déclarer un objet. Pour ce faire, on a recours au mot réservé **object**. Un objet étant une *structure de données*, il sera donc toujours déclaré à l'intérieur d'un bloc **type**.

L'objet le plus basique que l'on puisse déclarer est tout simplement l'objet vide, qui n'effectue rien et ne contient rien. On le déclare comme ceci :

```
type
  TObjetVide = object
  end;
```

On remarquera que la déclaration d'un objet se terminera toujours par un **end;**.
De plus, on adopte très souvent en Pascal une convention de notation pour les objets : leur nom commence toujours par **T**, comme *type*. Nous respecterons toujours cette convention dans ce tutoriel, et nous parlerons donc des objets *TMaison* ou encore *TChose*.

Attention !
Sur les compilateurs de nouvelle génération, le mot réservé **object** sera remplacé par le mot réservé **class**.

```
type
  TObjetVide = class
  end;
```

Dans la suite de ce tutoriel, nous adopterons l'utilisation de mot réservé **object**. Celui-ci sera à remplacer en fonction du compilateur utilisé (**Delphi** par exemple).

4.1.2. Déclarations imbriquées

Il est parfois nécessaire de déclarer des objets qui s'utilisent mutuellement. On peut ainsi prendre l'exemple de deux objets, *TParent* et *TEnfant*, le parent ayant la nécessité de connaître la liste de ses

enfants, et l'enfant la nécessité de connaître son parent.
Intervient alors le problème de la déclaration imbriquée de deux objets. Si l'on tente de les déclarer comme ceci :

```
type
  TParent = object
    Enfant: TEnfant;
  end;

  TEnfant = object
    Parent: TParent;
  end;
```

Le compilateur va déclencher une erreur indiquant qu'il connaît pas encore *TEnfant* lorsqu'il tente d'analyser la structure de *TParent*.
La solution au problème passe **nécessairement par l'instanciation dynamique** des deux objets (voir le paragraphe concerné). On va donc déclarer deux pointeurs vers les deux types considérés et *seulement ensuite*, on déclarera les objets eux-même, ceci dans **le même bloc type** :

```
type
  { Déclaration des pointeurs }
  PParent = ^TParent;
  PEnfant = ^TEnfant;

  { Déclaration des objets utilisant les pointeurs }
  TParent = object
    Enfant: PEnfant;
  end;

  TEnfant = object
    Parent: PParent;
  end;
```

Grâce à cette méthode, plus aucune erreur n'est déclenchée, car lorsque le compilateur va analyser *TParent*, il aura déjà eu connaissance de l'existance de *PEnfant*. Ceci n'est bien évidemment possible que parce que le compilateur accepte la déclaration **prématurée** de pointeurs avant le type vers lequel ils pointent.


Sous **Delphi**, l'intanciation étant automatiquement **dynamique**, ce problème ne se pose pas, et pour résoudre le problème, on se contente d'**annoncer la classe** avec une **déclaration partielle** :

```
type
  { Déclarations partielles }
  TParent = class;
  TEnfant = class;

  { Déclaration des objets utilisant les classes }
  TParent = class
    Enfant: TEnfant;
  end;

  TEnfant = class
    Parent: TParent;
  end;
```

4.2. Champs et méthodes

Un objet vide ne présentant pas d'intérêt majeur, il pourrait être intéressant de savoir comment lui ajouter des champs et des méthodes.


Attention !
Dans un objet, il convient de **toujours déclarer les champs AVANT les méthodes**. Si jamais un champ était déclaré après une méthode, alors le compilateur générerait une erreur (généralement, le compilateur indique qu'il attend un **end** au niveau du champ mal placé).
Comme à l'intérieur d'un enregistrement de type **record**, les champs se déclarent comme de simples variables, sans réutiliser le mot réservé **var** à l'intérieur de la déclaration de l'objet. Tous les types peuvent être utiliséspour un champ. Ainsi, les exemples suivants sont tous valides :

```
type
  TChose = object
    Entier: Integer;
    Chaine: string;
    Fichier: file;
  end;
type
  TBidule = object
    Enr: record
           A, B: Integer;
         end;
    Entier: Integer;
  end;
type
  TTruc = object
    Obj1: TChose;
    Obj2: object
            Champ: Char;
          end;
  end;
```

On pourra ainsi accéder aux champs de la manière suivante :

```
var
  Bidule: TBidule;
  Truc: TTruc;

begin
  ...
  Bidule.Enr.A := ...;
  ...
  Truc.Obj1.Entier := 0;
  Truc.Obj2.Champ := 'X';
  ...
end.
```

Les méthodes se déclarent à la suite des champs. Le schéma théorique actuel de description d'un objet serait ainsi :

```
type
  TObjet = object
    Champ;
     ...
    Champ;
    Methode;
    ...
    Methode;
  end;
```

4.2.1. Procédures et fonctions

Les méthodes - procédures et fonctions - se déclarent à l'intérieur de l'objet comme on le ferait dans la partie **interface** d'une unité :

```
type
  TObjet = object
    Champs;
    ....
    procedure Methode1(Params: ParamType);
    function  Methode2(Params: ParamType): RetType;
  end;
```

Une fois ces méthodes déclarées, il faut écrire le code source associé. On complète leur déclaration **en-dehors** de la déclaration de l'objet. Le nom de la méthode est alors **précédé du nom de l'objet suivi d'un point**. Un exemple expliquant aussi bien, voici donc ce que donnerait l'ajout d'une méthode *Methode1* :

```
type
  TObjet = object
    procedure Methode1(Param: ParamType);
  end;
```

```
procedure TObjet.Methode1(Param: ParamType);
begin
  ...
end;
```
Comme il s'agit d'une déclaration de type **forward**, on peut éventuellement omettre les paramètres lorsque l'on complète la déclaration d'une méthode :

```
type
  TObjet = object
    procedure Methode1(Param: ParamType);
  end;


procedure TObjet.Methode1;
begin
  ...
end;
```
Ce code et le code précédent sont tous deux parfaitement identiques. On choisira donc de privilégier soit la facilité de lecture, soit la simplicité d'écriture.

4.2.2. Constructeurs

Un constructeur se déclare exactement comme une autre méthode. Il n'y a pas d'ordre particulier, et on peut très bien intercaler des constructeurs au milieu des autres méthodes. Le mot réservé **procedure** (ou **function**) sera ici remplacé par le mot réservé **constructor** :

```
type
  TObjet = object
    constructor Init(Param: ParamType);
  end;


constructor TObjet.Init;
begin
  ...
end;
```
L'usage veut que le constructeur principal d'un objet soit appelé *Init* ou *Create*. Il n'y a aucune obligation dans ce domaine, mais on essaie la plupart du temps de suivre cette convention.

Turbo Pascal utilise pour tous les objets standard la "convention *Init*", et **Delphi** pour sa part utilise la "convention *Create*".

Il est possible de faire **échouer un constructeur**, par exemple si une opération nécessaire à l'initialisation de l'objet s'est mal déroulée.
En fonction du compilateur, soit on déclenchera une **exception** (**Delphi**), soit on fera appel à la procédure **Fail** :

```
type
  TObjet = object
    constructor Init;
   end;

constructor TObjet.Init;
begin
  ...
  { Fait échouer le constructeur }
  if ... then Fail;
end;
```

4.2.3. Destructeurs

De même que pour les constructeurs, on se servira ici du mot réservé **destructor** :

```
type
```

```
  TObjet = object
    destructor Done;
  end;


destructor TObjet.Done;
begin
  ...
end;
```

Ici encore, l'usage veut que l'on appelle les destructeurs *Done* ou *Destroy*. De plus, la plupart du temps, un destructeur n'aura pas de paramètres. Toutefois, le contraire est tout à fait autorisé par le compilateur.

Turbo Pascal utilise pour tous les objets standard la "convention *Done*", et **Delphi** pour sa part utilise la "convention *Destroy*".

4.2.4. Méthode virtuelles et dynamiques

Si vous désirez déclarer une méthode **virtuelle**, alors vous devez ajouter à la déclaration de la méthode (procédure, fonction, constructeur ou destructeur) le mot réservé **virtual** suivi d'un point virgule. **virtual** ne devra pas être repris lorsque vous complèterez le code de la méthode :

```
type
  TObjet = object
    procedure Methode(Param: ParamType); virtual;
  end;

procedure Methode(Param: ParamType);
begin
  ...
end;
```

Les méthodes **dynamiques** sont également appelées méthodes **virtuelles indexées**. En effet, rien ne les distingue des méthodes virtuelles si ce n'est qu'elles possèdent un index, un numéro pour les identifier.
La déclaration d'une méthode dynamique varie d'un compilateur à un autre. Sur les compilateurs plus anciens, elles seront déclarées comme ceci :

```
type
  TObjet = object
    procedure Methode(Param: ParamType); virtual IndexUnique;
  end;

procedure Methode(Param: ParamType);
begin
  ...
end;
```

Où *IndexUnique* représente un entier **unique** permettant d'identifier la méthode virtuelle.

Sur les compilateurs plus récents, l'index est géré automatiquement, et il suffit pour déclarer une méthode dynamique de remplacer le mot réservé **virtual** par le mot réservé **dynamic** :

```
type
  TObjet = class
    procedure Methode(Param: ParamType); dynamic;
  end;

procedure TObjet.Methode(Param: ParamType);
begin
  ...
end;
```

Attention !
Si votre compilateur utilise le mot réservé **object** (comme **Turbo Pascal**), et si une méthode est déclarée avec le mot réservé **virtual** alors toutes les méthodes surchargées devront aussi être déclarées avec **virtual**.
Il en sera de même avec les méthodes dynamiques.

Si par contre votre compilateur utilise le mot réservé **class**, les méthodes surchargées devront être déclarées avec le mot réservé **override**. Consultez la documentation de votre compilateur

pour plus de détails.

4.2.5. Méthodes abstraites

La déclaration d'une méthode abstraite dépend du compilateur utilisé. Si vous utilisez un compilateur Pascal d'ancienne génération, alors, bien qu'une méthode abstraite ne doit en théorie pas posséder d'implémentation, celle-ci se déclarera comme toute autre méthode, et dans le corps de la méthode, on ajoute un appel à la procédure **Abstract;** :

```
type
  TObjet = object
    procedure MethodeAbs;
  end;

procedure TObjet.MethodeAbs;
begin
  { C'est une méthode abstraite }
  Abstract;
end;
```


Les nouveaux compilateurs utilisent le mot réservé **abstract**, et suppriment logiquement le corps de la méthode :

```
type
  TObjet = class
    { C'est une méthode abstraite }
    procedure MethodeAbs; abstract;
  end;
```

4.2.6. Paramètre implicite Self

Dès lors qu'une méthode est appelée, le compilateur lui fournit *implicitement*, autrement dit de manière non visible, un paramètre supplémentaire, **Self**, que vous n'avez donc pas à gérer.

Le paramètre **Self** représente l'instance de l'objet en cours, et possède donc le même type que l'objet. Ainsi, si un objet possède un champ nommé *Toto*, alors les deux codes suivants seront identiques :

```
procedure TObjet.Methode;
begin
  Toto := 1;
end;
procedure TObjet.Methode;
begin
  Self.Toto := 1;
end;
```

Le paramètre **Self** sert de manière interne à l'objet pour garantir un appel correct des méthodes virtuelles.
Pour le programmeur, **Self** peut aussi servir à vérifier qu'une instance d'un même type d'objet est déjà en mémoire et, par exemple, interdire la création d'une nouvelle instance :

```
type
  TObjet = object
    constructor Init;
  end;

var
  Instance: TObjet;

constructor TObjet.Init;
begin
  if Instance <> Self then Fail;
  ...
end;

begin
  Instance.Init;
  ...
end.
```

Le paramètre **Self** est toujours transmis **en premier** à la méthode. Ceci peut avoir son importance lors de l'utilisation de l'assembleur.

4.3. Visibilité

La visibilité de champs et méthodes s'indique grâce à divers mots réservés :

Visibilité publique : **public**
Visibilité privée : **private**
Visibilité protégée : **protected**

Le spécificateur de visibilité doit être placé avant l'ensemble des champs et méthodes devant bénéficier de cette visibilité :

```
type
  TObjet = object
  private
    ChampPriv;
    ...
    MethodePriv;
    ...
  public
    ChampPub;
    ...
    MethodePub;
    ...
  end;
```

Les spécificateurs peuvent être placés dans un ordre quelconque, et apparaître plusieurs fois si nécessaire, bien que cette possibilité soit rarement utilisée.

Rappel
La visibilité **protégée** n'est pas disponible sous **Turbo Pascal**.

4.4. Héritage

4.4.1. Déclaration

Un objet en Pascal ne peut hériter que d'un seul ancêtre. Afin de spécifier celui-ci, on spécifie son nom entre parenthèses après le mot réservé **object** :

```
type
  TObjet = object(TAncetre)
    ...
  end;
```

Sitôt qu'un ancêtre est déclaré pour un objet, ce dernier peut accéder à tous les champs et méthodes de son ancêtre **sans avoir à les redéclarer** (en fonction de la visibilité).

Attention !
Si vous désirez surcharger une méthode **virtuelle**, alors vous devez la redéclarer **exactement comme elle était déclarée dans l'ancêtre**. Notamment, vous ne pourrez pas ajouter ou supprimer de paramètres à sa déclaration.

Par contre, si vous **redéfinissez une méthode**, alors vous pourrez très bien modifier ses paramètres comme vous le désirez si celle-ci n'est **pas virtuelle**.

```
type
  TAncetre = object
    Champ: Integer;
    procedure ModifChamp(Valeur: Integer); virtual;
    procedure ARedefinir;
  end;
```

```
procedure TAncetre.ModifChamp(Valeur: Integer);
begin
  Champ := Valeur;
end;

procedure TAncetre.ARedefinir;
begin
  ...
end;


type
  TDescendant = object(TAncetre)
    procedure ModifChamp(Valeur: Integer); virtual;
    procedure ARedefinir(ParamSup: Integer);
  end;

{ On surcharge la méthode virtuelle ModifChamp en la remplaçant par une autre }
procedure TDescendant.ModifChamp(Valeur: Integer);
begin
  { On accède au champ sans le redéclarer dans TDescendant }
  Champ := Valeur + 1;
end;

{ La méthode a été redéfinie avec de nouveaux paramètres }
procedure TDescendant.ARedefinir(ParamSup: Integer);
begin
  ...
end;
```

4.4.2. Surcharge et appel de l'ancêtre

La plupart du temps, lorsque que l'on surcharge une méthode, le but n'est pas d'**écraser l'ancienne**, mais de la **compléter** de façon à apporter de nouvelles fonctionnalités. Il est donc nécessaire de pouvoir appeler la méthode ancêtre.
Deux cas de figure sont alors à envisager.

Il n'est pas nécessaire de surcharger ou de redéfinir une méthode ! Ainsi, si un objet ne surcharge pas une méthode, c'est celle du premier ancêtre la définissant ou la surchargeant qui sera appelée.
De fait, il n'est pas nécessaire pour un objet de réécrire un constructeur (ou un destructeur) si celui de son ancêtre suffit à son initialisation.

4.4.2.1. Ancêtre direct

Généralement, on appelle l'ancêtre direct, celui dont on hérite en première main. Pour appeler la méthode ancêtre, on utilise alors le mot réservé **inherited** devant le nom de la méthode, à l'endroit on l'on désire effectuer l'appel :

```
type
  TAncetre = object
    Champ: Integer;
    procedure ModifChamp(Valeur: Integer); virtual;
  end;

procedure TAncetre.ModifChamp(Valeur: Integer);
begin
  Champ := Valeur;
end;


type
  TDescendant = object(TAncetre)
    Temp: Integer;
    procedure ModifChamp(Valeur: Integer); virtual;
  end;
```

```
procedure TDescendant.ModifChamp(Valeur: Integer);
begin
  { On effectue diverses actions }
  Inc(Valeur);
  { On appelle l'ancêtre }
  inherited ModifChamp(Valeur);
  { On peut effectuer d'autres actions }
  Temp := Champ;
end;
```
La méthode ancêtre peut être appelée autant de fois que désiré, et on peut effectuer un nombre quelconque d'opérations avant et/ou après ce ou ces appels.

4.4.2.2. Ancêtre indirect

Dans certains cas particuliers, il peut être nécessaire d'appeler un ancêtre plus éloigné dans la hiérarchie : par exemple l'ancêtre de l'ancêtre direct. Dans ce cas, il faut faire explicitement appel à la méthode en faisant précéder son nom par le nom de l'ancêtre suivi d'un point. Dans ce cas, les méthodes surchargées apparaissant entre l'objet et l'ancêtre appelé seront **ignorées**.

```
type
  TPremier = object
    ...
    procedure Methode; virtual;
  end;

procedure TPremier.Methode;
begin
  ...
end;


type
  TSecond = object(TPremier)
    ...
    procedure Methode; virtual;
  end;

procedure TSecond.Methode;
begin
  ...
end;


type
  TDernier = object(TSecond)
    ...
    procedure Methode; virtual;
  end;

procedure TDernier.Methode;
begin
  ...
  { On appelle TPremier.Methode : TSecond.Methode est ignorée }
  TPremier.Methode;
end;
```

4.4.3. Valeur de retour d'un constructeur

Lorsque que le constructeur d'un ancêtre est appelé avec **inherited**, celui-ci renvoie une valeur booléenne indiquant si la construction de l'objet s'est effectuée sans erreur.
Ainsi, si la construction a échoué dans l'ancêtre (par un appel à *Fail* notamment), alors il convient de faire échouer également la construction du descendant, celui-ci ne pouvant fonctionner sans le support de son ancêtre. Dès lors, il faudra aussi faire un appel à *Fail*.

Généralement, on adopte la syntaxe suivante :

```
constructor TDescendant.Init(...);
begin
  if not inherited Init(...) then Fail;
  ...
end;
```

4.4.4. Ordre d'appel de l'ancêtre dans les constructeurs et destructeurs

S'il est possible de faire appel à **inherited** où on le souhaite et le nombre de fois désiré dans une méthode, les constructeurs et les destructeurs sont régis par des règles plus strictes qui **ne seront pas vérifiées par la compilateur**. Elles sont donc d'autant plus importantes.

> **Constructeurs :**
> D'une manière générale, le constructeur ancêtre devra être appelé **en premier lieu**, aucune autre instruction ne devant précéder cet appel.
> Cependant, il est possible d'effectuer diverses opérations en vue, par exemple, de transmettre une valeur en paramètre au constructeur ancêtre. Néanmoins, on s'abstiendra de **toute modification de la valeur d'un champ** avant l'appel à l'ancêtre, car cette valeur serait sujette à modification.

```
constructor TDescendant.Init(...);
begin
  { Première opération du constructeur : l'appel de l'ancêtre }
  if not inherited Init(...) then Fail;
  ...
end;
```

> **Destructeur :**
> De façon logique, l'appel au destructeur ancêtre devra se faire **en dernier**, une fois tous les champs du descendant nécessitant un traitement particulier désalloués proprement. En effet, une fois un destructeur appelé, il est impossible de prévoir la valeur d'un champ.

```
destructor TDescendant.Done;
begin
  ...
  { Dernière opération du destructeur : l'appel de l'ancêtre }
  inherited Done;
end;
```

4.5. Instanciation d'un objet

Une fois l'objet déclaré - autrement dit le type défini - il reste à créer une ou plusieurs instances de celui-ci, avec des variables : c'est l'**instanciation**, ou plus simplement la **création**. Il en existe de deux sortes : l'instanciation **statique** et l'instanciation **dynamique**.

4.5.1. Instanciation statique

L'instanciation **statique** est certainement la plus simple à mettre en oeuvre, mais aussi celle à éviter le plus possible. Elle consiste à simplement déclarer une variable du type objet comme on déclarerait n'importe quelle variable :

```
type
  TObjet = object
    ...
  end;

var
  MonObjet: TObjet;
```

Si l'objet possède un **constructeur**, celui-ci devra être appelé avant toute autre méthode, et de même, si l'objet possède un **destructeur**, il devra être appelé en dernier.

```
type
  TObjet = object
```

```
    ...
    constructor Init(...);
    destructor Done;
  end;

constructor TObjet.Init(...);
begin
  ...
end;

destructor TObjet.Done;
begin
  ...
end;


var
  MonObjet: TObjet;

begin
  ...
  MonObjet.Init(...);
  ...
  MonObjet.Methode;
  ...
  MonObjet.Done;
  ...
end.
```


L'instanciation **statique** présente plusieurs inconvénients, le principal concernant la mémoire. En effet, en mode réel sous DOS, seul 64 Ko de mémoire sont disponibles pour les variables statiques. Il en résulte donc une limitation drastique concernant le nombre d'objets en mémoire et leur taille.


L'instanciation **statique** a été **supprimée sur les compilateurs récents** comme **Delphi**, où la gestion est obligatoirement dynamique, mais en gardant une syntaxe équivalente à la syntaxe statique (suppression notamment de l'utilisation explicite des pointeurs).

4.5.2. Instanciation dynamique

L'instanciation **dynamique** est certainement la plus utilisée et celle permettant de stocker en mémoire le plus d'objets et d'information. Elle suppose bien entendu une connaissance des **pointeurs**.

Afin de pouvoir utiliser le dynamique, il faut commencer par déclarer un pointeur vers le type objet que l'on projette d'utiliser. L'usage veut que cette déclaration se fasse conjointement avec celle de l'objet en lui-même. De même, on choisit généralement de faire figurer la déclaration du pointeur juste avant celle de l'objet.
Le nom du pointeur pourra reprendre celui de l'objet, en remplaçant le **T** initial par un **P**. Par exemple, si l'on considère l'objet *TObjet*, alors le pointeur vers l'objet sera nommé communément *PObjet*.

```
type
  PObjet = ^TObjet;
  TObjet = object
    ...
  end;
```

4.5.2.1. Allocation

L'allocation mémoire de l'objet et son initialisation se font généralement **en même temps** à l'aide de la fonction standard **New**. Deux formes peuvent être utilisées :

New(VarObjet, Constructeur);
Avec cette syntaxe, on utilise **New** en tant que procédure en faisant apparaître en tant que deuxième paramètre le constructeur avec ses éventuels paramètres.

```
var
  Objet: PObjet;
```

```
begin
  New(Objet, Init(...));
  ...
end.
```

VarObjet := New(PointeurTypeObjet, Constructeur);
Cette syntaxe utilise la syntaxe étendue de **New** en l'utilisant en tant que fonction. On fait toujours apparaître en deuxième le constructeur affecté de ses divers paramètres. La différence vient du fait que l'on spécifie clairement de quel type l'objet doit être. Cette syntaxe permet plus de liberté en ce que l'on peut très bien affecter à un pointeur quelconque (un type descendant par exemple) un autre type objet.
De plus, cette syntaxe se rapproche beaucoup de celle adoptée par les compilateurs plus récents comme **Delphi**. On pourra donc légitimement la préférer.

```
var
  Objet: PObjet;

begin
  Objet := New(PObjet, Init(...));
  ...
end.
```


Il est tout à fait possible d'allouer dans un premier temps l'objet comme n'importe quel pointeur avec *New(Objet);* et ensuite d'appeler le constructeur.
Néanmoins, cette pratique est obsolète et on l'évitera au possible.


On n'oubliera pas lors de l'appel aux méthodes que l'on utilise dorénavant un pointeur. Le symbole ^ est donc de mise :

```
var
  Objet: PObjet;

begin
  ...
  Objet^.Methode(...);
  ...
end.
```


Attention !
Comme on travaille à présent avec des **pointeurs**, il faut vérifier que l'objet est bien alloué. On peut ainsi tester l'égalité avec **nil** ou bien utiliser la fonction interne **Assigned** :

```
var
  Objet: PObjet;

begin
  Objet := New(PObjet, Init(...));
  if Objet <> nil then
  begin
    ...
    Dispose(Objet, Done);
  end;
end.
```

Ou :

```
var
  Objet: PObjet;

begin
  Objet := New(PObjet, Init(...));
  if Assigned(Objet) then
  begin
    ...
    Dispose(Objet, Done);
  end;
end.
```


Les compilateurs récents comme **Delphi** utilisent la syntaxe **VarObjet := TypeObjet.Constructeur;**, les pointeurs étant intégrés directement à la déclaration de la classe. Ils sont donc implicites. De plus, on se servira de blocs **try**...**finally** pour protéger son code :

```
type
  TObjet = class
    constructor Create(...);
```

```
  end;


var
  Objet: TObjet;

begin
  Objet := TObjet.Create(...);
  try
    ...
  finally
    ...
  end;
end;
```

4.5.2.2. Désallocation

Tout comme pour l'allocation, la désallocation d'effectue généralement en même temps que l'appel du destructeur de l'objet à l'aide de la procédure **Dispose**. Le destructeur est alors passé avec ses éventuels paramètres comme deuxième paramètre de la procédure.

```
var
  Objet: PObjet;

begin
  ...
  Dispose(Objet, Done(...));
end.
```


Tout comme pour les constructeurs, il est possible d'appeler le destructeur pour n'appeler que par la suite **Dispose** séparément. Cette pratique devra toutefois être évitée car obsolète et parfois source d'erreurs.


Delphi n'appelle pas directement le destructeur mais fait appel à la procédure **Free** :

```
var
  Objet: TObjet;

begin
  Objet := TObjet.Create(...);
  try
    ...
  finally
    TObjet.Free;
  end;
end.
```

4.6. L'objet générique TObject

Turbo Pascal et la plupart des autres compilateurs définissent un objet générique servant de base à tous les objets de la bibliothèque standard : c'est **TObject**. Cet objet, placé dans l'unité **Objects**, est déclaré comme ceci :

```
type
  PObject = ^TObject;
  TObject = object
    constructor Init;
    destructor Done; virtual;
    procedure Free;
  end;
```

Init :
Le constructeur de **TObject**, en plus d'effectuer son rôle habituel (mise en place du diagramme d'héritage, VMT, ...), **met à zéro tous les champs** . Cette opération n'est en effet pas automatique pour un objet quelconque, et, par conséquent, tous les objets ne dérivant pas de **TObject** voient le contenu de leurs champs **indéfini** au moment de l'initialisation.
Done :
Le destructeur **Done** n'effectue rien ! Son seul intérêt réside dans le fait qu'il est déclaré comme

virtuel. Par conséquent, tous les descendants devront implémenter un destructeur virtuel, ce qui permet d'éviter toute erreur due à l'oubli de la sorte.
Free :
La méthode **Free** permet de se dispenser de l'appel à **Dispose**. En effet, en appelant **Free**, l'objet se libère *tout seul*.

On pourra donc prendre pour habitude de dériver tous ses objets de base de **TObject**, car il fournit une architecture de base pratique et importante pour la compatibilité avec les objets de la bibliothèque standard. L'allocation et la désallocation d'un tel objet deviendront alors :

```
uses
  Objects;

type
  PChose = ^TChose;
  TChose = object(TObject)
  private
    ...
  public
    ...
    constructor Init(...);
    destructor Done; virtual;
  end;

constructor TChose.Init(...);
begin
  if not inherited Init then Fail;
  ...
end;

destructor TChose.Done;
begin
  ...
  inherited Done;
end;

...


var
  Chose: PChose;

begin
  Chose := New(PChose, Init(...));
  if Assigned(Chose) then
  begin
    ...
    Chose^.Free;
  end;
end.
```


Delphi impose par défaut *TObject* comme ancêtre si aucun ancêtre n'est spécifié lors de la déclaration d'une classe. Ainsi, les deux exemples suivants sont équivalents :

```
type
  TObjet = class
    ...
  end;
type
  TObjet = class(TObject)
    ...
  end;
```

Conclusion

Vous avez appris ici les bases de la Programmation Orientée Objet et vous êtes à présent à même de construire vous-même vos propres programmes objets.
Seule la pratique permettant de faire des progrès, à vos claviers !