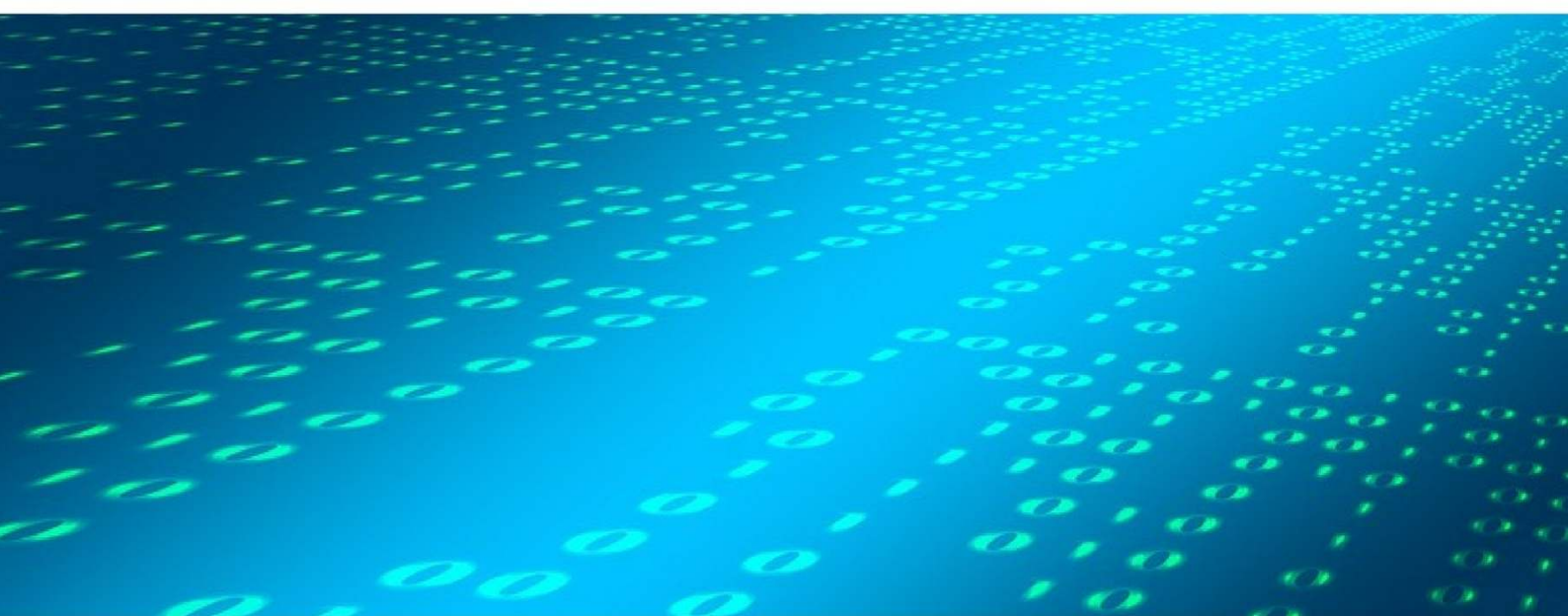




PROGRAMMATION ORIENTÉE OBJET EN C#



BAPTISTE PESQUET

Table of Contents

Introduction	1.1
Initiation à la POO	1.2
Principaux concepts objets	1.3
Gestion des objets en mémoire	1.4
La relation d'association	1.5
La relation d'héritage	1.6
Compléments sur l'écriture de classes	1.7
Gestion des exceptions	1.8

Programmation orientée objet en C#

Ce livre est un support de cours à l'[Ecole Nationale Supérieure de Cognitique](#).



Résumé

Ce livre est une introduction à la **programmation orientée objet** (POO). Il présente de manière progressive et illustrée les concepts essentiels et les savoir-faire associés :

- Ecriture et utilisation de classes.
- Ajout de constructeurs et d'accesseurs, encapsulation.
- Gestion des objets en mémoire (notion de référence d'objet).
- Mise en relation de classes : association et héritage.
- Redéfinition et surcharge de méthodes, membres statiques.
- Gestion des exceptions.

Les exemples de code sont écrits en C#, mais sont facilement transposables à d'autres langages orientés objets comme Java.

Pré-requis

La lecture de ce livre nécessite une connaissance des bases de la programmation :

- Notions de variable et de type.
- Structures conditionnelles et itératives (boucles).
- Programmation modulaire (sous-programmes) et passage de paramètres.
- Tableaux et listes.

Contributions

Ce livre est publié sous la licence Creative Commons [BY-NC-SA](#). Son code source est disponible sur [GitHub](#). N'hésitez pas à contribuer à son amélioration en utilisant les *issues* pour signaler des erreurs et les *pull requests* pour proposer des ajouts.



Merci d'avance et bonne lecture !

Initiation à la programmation orientée objet

L'objectif de ce chapitre est de découvrir ce qu'est la programmation orientée objet.

Les exemples de code associés sont [disponibles en ligne](#).

Introduction

Pourquoi utiliser la programmation orientée objet ?

La programmation de logiciels a connu il y a quelques années le passage d'une ère artisanale à une ère industrielle. Des logiciels de plus en plus complexes doivent être réalisés dans des délais de plus en plus courts, tout en maintenant le meilleur niveau de qualité possible. Comment faire pour répondre à ces exigences contradictoires ? La solution passe par l'emploi de techniques de développement adaptées, comme l'illustre l'extrait ci-dessous.

(...) Il est clair que le monde du logiciel ne progresse pas aussi vite que celui du matériel. Qu'ont donc les développeurs de matériel 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 leurs 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 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. (...)"

David Chappel, "Au coeur de ActiveX et OLE", 1997

La programmation orientée objet, souvent abrégée **POO**, permet de concevoir une application sous la forme d'un ensemble de briques logicielles appelées des **objets**. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues.

La POO facilite la conception de programmes par réutilisation de composants existants, avec tous les avantages évoqués plus haut. Elle constitue le standard actuel (on parle de *paradigme*) en matière de développement de logiciels.

Histoire de la programmation orientée objet

Les concepts de la POO naissent au cours des années 1970 dans des laboratoires de recherche en informatique. Les premiers langages de programmation véritablement objets ont été **Simula**, puis **Smalltalk**.

A partir des années 1980, les principes de la POO sont appliqués dans de nombreux langages comme **Eiffel** (créé par le Français Bertrand Meyer), **C++** (une extension objet du langage **C** créé par le Danois Bjarne Stroustrup) ou encore **Objective C** (une autre extension objet du C utilisé, entre autres, par l'iOS d'Apple).

Les années 1990 ont vu l'avènement des langages orientés objet dans de nombreux secteurs du développement logiciel, et la création du langage **Java** par la société Sun Microsystems. Le succès de ce langage, plus simple à utiliser que ses prédécesseurs, a conduit Microsoft à riposter en créant au début des années 2000 la plate-forme **.NET** et le langage **C#**, cousin de Java.

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés : Java et C# bien sûr, mais aussi **PHP** (à partir de la version 5), **VB.NET**, **PowerShell**, **Python**, etc. Une connaissance minimale des principes de la POO est donc indispensable à tout informaticien, qu'il soit développeur ou non.

REMARQUE : la POO s'accompagne d'un changement dans la manière de penser les problèmes et de concevoir l'architecture des applications informatiques. C'est ce qu'on appelle l'analyse (ou la modélisation) orientée objet. Son principal support est le langage de modélisation **UML**.

Première approche de la POO

La notion d'objet

Quand on utilise la POO, on cherche à représenter le domaine étudié sous la forme d'objets. C'est la phase de **modélisation orientée objet**.

Un **objet** est une entité qui représente (*modélise*) un élément du domaine étudié : une voiture, un compte bancaire, un nombre complexe, une facture, etc.

Objet = état + actions

Cette équation signifie qu'un objet rassemble à la fois :

- des **informations** (ou données) qui le caractérisent.
- des **actions** (ou traitements) qu'on peut exercer sur lui.

Imaginons qu'on souhaite modéliser des comptes bancaires pour un logiciel de gestion. On commence par réfléchir à ce qui caractérise un compte bancaire, puis on classe ces éléments en deux catégories :

- les informations liées à un compte bancaire.
- les actions réalisables sur un compte bancaire.

En première approche, on peut considérer qu'un compte bancaire est caractérisé par un **titulaire**, un **solde** (le montant disponible sur le compte) et utilise une certaine **devise** (euros, dollars, etc). Les actions réalisables sur un compte sont le dépôt d'argent (**crédit**) et le retrait (**débit**).

On peut regrouper les caractéristiques de notre objet "compte bancaire" dans ce tableau.

Informations	Actions
titulaire, solde, devise	créditer, débiter

DEFINITION : un **objet** se compose **d'informations** et **d'actions**. Les actions utilisent (et parfois modifient) les informations de l'objet.

- L'ensemble des informations d'un objet donné est appelée son **état**.
- L'ensemble des actions applicables à un objet représente son **comportement**.

REMARQUE : les actions associées à un objet s'expriment généralement sous la forme de verbes à l'infinitif (*créditer*, *débiter*).

A un instant donné, l'état d'un objet "compte bancaire" sera constitué par les valeurs de son titulaire, son solde et sa devise. L'état de l'objet "compte bancaire de Paul" sera, sauf exception, différent de l'état de l'objet "compte bancaire de Pierre".

La notion de classe

Nous venons de voir que l'on pouvait représenter un compte bancaire sous la forme d'un objet. Imaginons que nous voulions gérer les différents comptes d'une banque. Chaque compte aura son propre titulaire, son solde particulier et sa devise. Mais tous les comptes

auront un titulaire, un solde et une devise, et permettront d'effectuer les mêmes opérations de débit/crédit. Chaque objet "compte bancaire" sera construit sur le même modèle : ce modèle est appelée une **classe**.

DEFINITION : une **classe** est un **modèle d'objet**. C'est un nouveau type créé par le programmeur et qui sert de modèle pour tous les objets de cette classe. Une classe spécifie les informations et les actions qu'auront en commun tous les objets qui en sont issus.

Le compte en banque appartenant à Jean, dont le solde est de 450 euros, est un compte bancaire particulier. Il s'agit d'un objet de la classe "compte bancaire". En utilisant le vocabulaire de la POO, on dit que l'objet "compte bancaire de Jean" est une **instance** de la classe "compte bancaire".

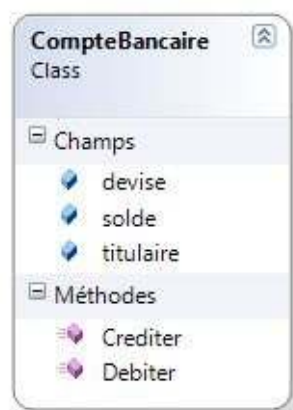
ATTENTION : ne pas confondre **objet** et **classe**. Une classe est un type abstrait (exemple : un compte bancaire en général), un objet est un exemplaire concret d'une classe (exemple : le compte bancaire de Jean).

Un objet est une variable particulière dont le type est une classe.

Représentation graphique

Afin de faciliter la communication entre les programmeurs n'utilisant pas le même langage de programmation objet, il existe un standard de représentation graphique d'une classe. Ce standard fait partie de la norme UML (*Unified Modeling Language*). On parle de **diagramme de classe**.

Voici la représentation de notre classe "compte bancaire" réalisée par Visual Studio.



Voici la même classe représentée sous la forme d'un diagramme conforme au standard UML.



Quel que soit l'outil utilisé, on observe que la classe est décomposée en deux parties :

- les **champs**, qui correspondent aux informations de la classe : *devise*, *solde* et *titulaire*.
- les **méthodes**, qui correspondent aux actions réalisables: *créditer*, *débiter* et *décrire*.

DEFINITION : une **méthode** représente une action réalisable sur un objet de la classe dans laquelle cette méthode est définie. Une méthode représente une partie du comportement d'une classe.

En résumé

- La POO consiste à programmer en utilisant des **objets**.
- Un objet modélise un élément du domaine étudié (exemples : un compte bancaire, une voiture, un satellite, etc).
- Un objet est une **instance** de **classe**. Une classe est un type abstrait, un objet est un exemplaire concret de cette classe.
- Une classe regroupe des **informations** et des **actions**.
- Les informations sont stockées sous la forme de **champs**. Les champs décrivent l'**état** d'un objet.
- Les actions réalisables sur un objet sont représentés par des **méthodes**. Elles expriment ce que les objets peuvent faire, leur **comportement**.

Les informations stockées par un objet peuvent être appelées **champs**, **attributs** ou encore **propriétés**. Attention, ce dernier terme est ambigu en C# (voir plus loin).

Programmer avec des objets

Ecriture d'une classe

Une fois la modélisation d'une classe terminée, il est possible de la traduire dans n'importe quel langage de programmation orienté objet.

Voici la traduction en C# de la classe `CompteBancaire` .

```
public class CompteBancaire
{
    public string titulaire;
    public double solde;
    public string devise;

    public void Crediter(double montant)
    {
        solde = solde + montant;
    }
    public void Debiter(double montant)
    {
        solde = solde - montant;
    }
}
```

La définition d'une classe commence par le mot-clé `class`. On retrouve ensuite la définition des champs (attributs) et des méthodes de la classe. On remarque que les méthodes utilisent (et modifient) les valeurs des attributs.

Une méthode est une sorte de mini-programme. On peut y déclarer des variables locales et y utiliser tous les éléments de programmation déjà connus (alternatives, boucles, etc).

On remarque un nouveau mot-clé : `public`, sur lequel nous reviendrons ultérieurement.

Utilisation d'une classe

En utilisant le modèle fourni par la classe, il est possible de créer autant d'objets que nécessaire. Différents objets d'une même classe disposent des mêmes attributs et des mêmes méthodes, mais les valeurs des attributs sont différentes pour chaque objet. Par exemple, tous les comptes bancaires auront un solde, mais sauf exception, ce solde sera différent pour chaque compte.

Le programme C# ci-dessous utilise la classe `CompteBancaire` définie plus haut pour créer le compte de Jean et y effectuer deux opérations.

```
static void Main(string[] args)
{
    CompteBancaire comptePierre;           // déclaration d'un nouvel objet
    comptePierre = new CompteBancaire();    // instantiation de cet objet

    // affectations de valeurs aux attributs
    comptePierre.titulaire = "Pierre";
    comptePierre.solde = 0;
    comptePierre.devise = "euros";

    // appels des méthodes
    comptePierre.Crediter(300);
    comptePierre.Debiter(500);

    string description = "Le solde du compte de " + comptePierre.titulaire +
        " est de " + comptePierre.solde + " " + comptePierre.devise;
    Console.WriteLine(description);
}
```

A la fin du programme, l'attribut *solde* de l'objet `comptePierre` contient la valeur -200.



Déclaration et instantiation

On remarque que la création de l'objet `comptePierre` se fait en deux étapes :

1. déclaration de l'objet
2. instantiation de l'objet

```
CompteBancaire comptePierre;           // déclaration d'un nouvel objet
comptePierre = new CompteBancaire();    // instantiation de cet objet
// ...
```

La **déclaration** permet de créer une nouvelle variable, appelée `comptePierre` dans l'exemple ci-dessus. A ce stade, aucune réservation de mémoire n'a eu lieu pour cet objet. Il est donc inutilisable.

Le type de la variable `comptePierre` est `CompteBancaire`. Le type d'un objet est sa classe.

L'instanciation utilise l'opérateur `new`, qui permet de réserver une zone mémoire spécifique pour l'objet. On dit que l'objet est instancié par l'opérateur `new`. Sans cette étape indispensable, l'objet déclaré ne peut pas être utilisé.

DEFINITION : l'**instanciation** est l'opération qui consiste à créer un nouvel objet à partir d'une classe, au moyen de l'opérateur `new` .

Il est courant de rassembler sur une même ligne déclaration et instanciation d'un objet.

```
CompteBancaire comptePierre = new CompteBancaire(); // déclaration et instanciation d
'un nouvel objet
// ...
```

ATTENTION : ne pas confondre **instanciation** et **initialisation** :

- instancier, c'est créer un nouvel objet (opérateur `new`)
- initialiser, c'est donner une valeur initiale à quelque chose (opérateur `=`)

Ajout d'une méthode

Comme il est fastidieux de construire la description d'un compte bancaire à chaque fois qu'on en a besoin, on voudrait ajouter une opération de description du compte. Cette action est réalisée par une nouvelle méthode nommée *Decrire* ajoutée à la classe `CompteBancaire` . La description est renvoyée sous la forme d'une chaîne de caractères (`string`).

```
public class CompteBancaire
{
    // ...

    // Renvoie la description d'un compte
    public string Decrire()
    {
        string description = "Le solde du compte de " + titulaire + " est de " + sold
e + " " + devise;
        return description;
    }
}
```

Instanciation de plusieurs objets

Voici un autre programme qui gère les comptes de Pierre et de Paul. Il permet à l'utilisateur de saisir un montant qui sera débité à Pierre et crédité à Paul.

```
static void Main(string[] args)
{
    CompteBancaire comptePierre = new CompteBancaire();
    comptePierre.titulaire = "Pierre";
    comptePierre.solde = 500;
    comptePierre.devise = "euros";

    CompteBancaire comptePaul = new CompteBancaire();
    comptePaul.titulaire = "Paul";
    comptePaul.solde = 150;
    comptePaul.devise = "euros";

    Console.Write("Entrez le montant du transfert : ");
    double montantTransfert = Convert.ToDouble(Console.ReadLine());
    comptePierre.Debiter(montantTransfert);
    comptePaul.Crediter(montantTransfert);

    Console.WriteLine(comptePierre.Decrire());
    Console.WriteLine(comptePaul.Decrire());
}
```



Principaux concepts objets

L'objectif de ce chapitre est de présenter les concepts essentiels de la programmation orientée objet.

Les exemples de code associés sont [disponibles en ligne](#).

Constructeur

Reprenons l'exemple de la classe `CompteBancaire` du chapitre précédent.



Tout compte a nécessairement un titulaire, un solde initial et une devise lors de sa création. On aimerait pouvoir instancier un objet de la classe `CompteBancaire` en définissant directement les valeurs de ses attributs. Pour cela, nous allons ajouter à notre classe une méthode particulière : le **constructeur**.

```
public class CompteBancaire
{
    public string titulaire;
    public double solde;
    public string devise;

    // Constructeur
    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    // ...
}
```

Rôle

DEFINITION : le **constructeur** est une méthode spécifique dont le rôle est de construire un objet, le plus souvent en initialisant ses attributs.

ATTENTION : le nom du constructeur doit être identique au nom de la classe, et sa définition ne comporte pas le mot-clé `void`.

L'utilisation d'un constructeur se fait au moment de l'instanciation de l'objet (opérateur `new`), en passant en paramètres les futures valeurs des attributs de l'objet créé.

```
// déclaration et instanciation d'un nouvel objet en utilisant son constructeur
CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");

// appels de méthodes
comptePierre.Crediter(300);
comptePierre.Debiter(500);
Console.WriteLine(comptePierre.Decrire());
```

Constructeur par défaut

Lorsqu'une classe ne définit aucun constructeur (comme dans l'exemple du chapitre précédent), un constructeur par défaut sans aucun paramètre est implicitement créé. Il n'a aucun comportement mais son existence permet d'instancier des objets de cette classe.

En revanche, toute définition explicite d'un constructeur dans une classe "désactive" le constructeur par défaut. Dans notre exemple actuel, on ne peut plus instancier un compte bancaire sans lui fournir les trois paramètres que son constructeur attend.

```
// Impossible : le seul constructeur existant attend 3 paramètres
CompteBancaire comptePaul = new CompteBancaire();

// appels de méthodes
comptePierre.Crediter(300);
comptePierre.Debiter(500);
Console.WriteLine(comptePier
```

```
CompteBancaire.CompteBancaire(string leTitulaire, double soldelInitial, string laDevise)
Erreur :
'Chap2_Concepts.CompteBancaire' ne contient pas un constructeur qui accepte des arguments 0
```

REMARQUE : une classe peut disposer de plusieurs constructeurs initialisant différents attributs. Nous étudierons cette possibilité dans un [prochain chapitre](#).

Encapsulation

L'écriture de classes offre d'autres avantages que le simple regroupement de données et de traitements. Parmi ceux-ci figure la possibilité de restreindre l'accès à certains éléments de la classe. C'est ce que l'on appelle **l'encapsulation**.

Exemple d'utilisation

On souhaite qu'un compte bancaire créé ne puisse pas changer de titulaire ni de devise. Cela est possible en définissant les attributs `titulaire` et `devise` comme étant **privés**.

```
public class CompteBancaire
{
    private string titulaire;    // attribut privé
    public double solde;
    private string devise;      // attribut privé

    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    // ...
}
```

A présent, la seule manière de définir des valeurs pour `titulaire` et `devise` est d'utiliser le constructeur. Toute tentative d'accès externe aux propriétés privées générera une erreur lors de la compilation.

```
CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");

comptePierre.titulaire = "Pierre"; // Erreur : titulaire est un attribut privé
comptePierre.solde = 500;          // OK : solde est un attribut public
comptePierre.devise = "euros";     // Erreur : devise est un attribut privé
```

Définition

Les mots-clés `public` et `private` permettent de modifier le **niveau d'encapsulation** (on parle aussi de **visibilité** ou **d'accessibilité**) des éléments de la classe (attributs et méthodes) :

- un élément **public** est librement utilisable depuis le reste du programme.
- un élément **privé** est uniquement utilisable depuis les méthodes de la classe elle-même.

REMARQUE : dans de nombreux langages dont le C#, il existe un niveau d'encapsulation intermédiaire (`protected`) qui sera étudié plus loin.

DEFINITION : l'**encapsulation** est l'un des principes fondamentaux de la POO. Il consiste à restreindre l'accès à certains éléments d'une classe (le plus souvent ses attributs). L'objectif de l'encapsulation est de ne laisser accessible que le strict nécessaire pour que la classe soit utilisable.

CONSEIL : sauf cas particulier, on donne le niveau de visibilité `private` à tous les attributs d'une classe afin d'assurer leur encapsulation par défaut.

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    // ...
}
```

Avantages

L'encapsulation offre de nombreux avantages :

- diminution des risques de mauvaise manipulation d'une classe.
- création de classes "boîtes noires" par masquage des détails internes.
- possibilité de modifier les détails internes d'une classe (la manière dont elle fonctionne) sans changer son comportement extérieur (ce qu'elle permet de faire).

Accesseurs

L'encapsulation des attributs a permis d'interdire toute modification (accidentelle ou volontaire) des données d'un compte bancaire. Cependant, il est maintenant impossible de consulter le solde, le titulaire ou la devise d'un compte créé, ce qui est gênant. On aimerait pouvoir accéder aux données de la classe, tout en maintenant un certain niveau de contrôle. Cela est possible en ajoutant des **accesseurs** à la classe.

Définition

DEFINITION : un **accesseur** est une méthode le plus souvent *publique* qui permet d'accéder à un attribut *privé*.

- un accesseur en lecture (*getter*) permet de **lire** la valeur d'un attribut.
- un accesseur en écriture (mutateur ou *setter*) permet de **modifier** la valeur d'un attribut.

Spécificités du langage C#

En C#, les accesseurs prennent la forme de **propriétés**. Une propriété se manipule comme un champ, mais il s'agit en réalité d'un couple d'accesseurs *get* et *set*. Dans la plupart des autres langages, les accesseurs sont des méthodes de la forme `getXXX` et `setXXX`.

Voici la classe `CompteBancaire` modifiée pour intégrer des accesseurs vers ses attributs, ainsi que son nouveau diagramme de classe.

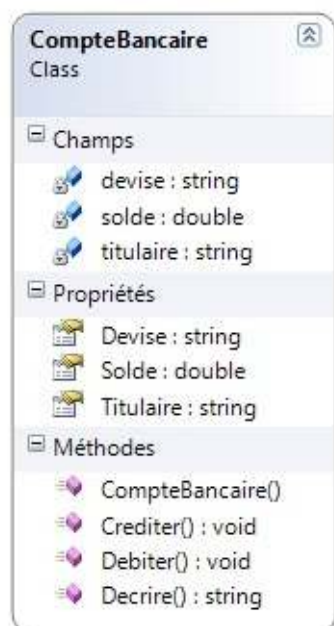
```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    public string Titulaire
    {
        get { return titulaire; }
        set { titulaire = value; }
    }

    public double Solde
    {
        get { return solde; }
        set { solde = value; }
    }

    public string Devise
    {
        get { return devise; }
        set { devise = value; }
    }

    // ...
}
```



Depuis la version 3 du langage, on peut définir les propriétés de manière automatique. Dans ce cas, l'attribut disparaît de la définition de la classe : sa présence est implicite. Voici l'exemple précédent réécrit en utilisant des propriétés automatiques.

```
public class CompteBancaire
{
    public string Titulaire { get; set; }

    public double Solde { get; set; }

    public string Devise { get; set; }

    // ...
}
```

Cette syntaxe permet de gagner en concision si, comme le plus souvent, les accesseurs se contentent de donner accès à l'attribut sous-jacent sans rien faire de particulier.

DANGER ! Ne pas confondre un champ ou attribut (qui, par convention, commence par une lettre minuscule) avec une propriété au sens du C# (qui commence toujours par une lettre majuscule). **Une propriété C# est un accesseur vers un champ.**

Avantages

Voici un exemple d'utilisation des accesseurs définis précédemment.

```
CompteBancaire comptePierre = new CompteBancaire("Pierre", 1000, "euros");

// utilisation des propriétés (accesseurs)
string titulaireCompte = comptePierre.Titulaire; // en lecture (getter)
comptePierre.Solde = 500;                        // en écriture (setter)
comptePierre.Devise = "dollars";                 // en écriture (setter)
```

En remplaçant l'accès direct à un attribut par l'utilisation d'une méthode, un accesseur permet d'effectuer des contrôles supplémentaires : respect d'une plage de valeurs, accès en lecture uniquement, etc.

Attributs en lecture seule

On observe dans l'exemple ci-dessus qu'il est à nouveau possible de modifier directement les données d'un compte. Pour interdire ces modifications tout en permettant de lire les informations d'un compte, il suffit de supprimer les accesseurs en écriture (*setters*).

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    public string Titulaire
    {
        get { return titulaire; }
    }

    public double Solde
    {
        get { return solde; }
    }

    public string Devise
    {
        get { return devise; }
    }

    // ...
}
```

Avec des propriétés automatiques, on peut seulement jouer sur le niveau de visibilité des accesseurs. La version 6 du langage C#, sortie en 2015, permet d'avoir réellement des propriétés automatiques en lecture seule ([Plus de détails](#)).

```
public class CompteBancaire
{
    public string Titulaire { get; private set; }
    // public string Titulaire { get; } // Incorrect avant C# 6 (.NET framework 4.6)

    public double Solde { get; private set; }

    public string Devise { get; private set; }

    // ...
}
```

A présent, l'accès aux champs est toujours possible, mais toute tentative de modification des données sera refusée.

```
CompteBancaire comptePierre = new CompteBancaire("Pierre", 1000, "euros");

string titulaireCompte = comptePierre.Titulaire; // OK : le getter existe
comptePierre.Solde = 500;                       // Erreur : pas de setter
comptePierre.Devise = "dollars";                 // Erreur : pas de setter
```

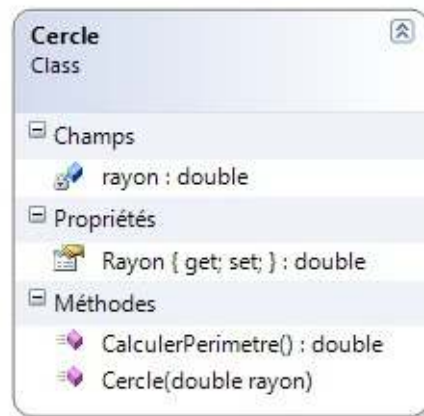

Gestion des objets en mémoire

L'objectif de ce chapitre est de découvrir les spécificités des variables de type objet, ainsi que la notion de **référence**.

Les exemples de code associés sont [disponibles en ligne](#).

Préambule

Ce chapitre utilise une classe `Cercle` définie de la manière suivante.



```
// Modélise un cercle
public class Cercle
{
    private double rayon; // rayon du cercle

    public double Rayon
    {
        get { return rayon; }
        set { rayon = value; }
    }

    // constructeur
    public Cercle(double rayon)
    {
        this.rayon = rayon;
    }

    // calcule le périmètre du cercle
    public double CalculerPerimetre()
    {
        double perimetre = 2 * Math.PI * rayon;
        return perimetre;
    }
}
```

Création d'un nouvel objet

Nous savons déjà que la création d'un nouvel objet se fait en deux étapes :

1. **déclaration** de l'objet
2. **instanciation** de l'objet

Le code ci-dessous crée un nouvel objet `monCercle` de la classe `Cercle`.

```
Cercle monCercle; // déclaration
monCercle = new Cercle(7.5); // instanciation
```

On peut rassembler ces deux étapes sur une seule ligne. Cependant, il est important de bien les distinguer conceptuellement.

```
Cercle monCercle = new Cercle(7.5); // déclaration et instanciation
```

Avant son instanciation, la “valeur” d'un objet en mémoire, observable au débogueur, est `null`. Elle correspond à un objet non instancié. L'instanciation permet de réserver une zone mémoire spécifique pour y stocker les données de l'objet.

Affectation d'objets

La spécificité des objets

Nous allons découvrir une différence fondamentale entre le fonctionnement d'une variable de type prédéfini (types C# de base : `int` , `double` , `bool` , etc) et le fonctionnement d'un objet.

Le code suivant utilise deux variables entières.

```
int nombre1;  
nombre1 = 5;  
int nombre2 = 3;  
  
nombre2 = nombre1;  
nombre1 = 10;  
  
Console.WriteLine("nombre1 = " + nombre1);  
Console.WriteLine("nombre2 = " + nombre2);
```

A la fin de son exécution, la variable `nombre1` vaut 10 et la variable `nombre2` vaut 5. Pas de surprise ici.

Ecrivons un code similaire, mais qui utilise cette fois deux variables objets.

```
Cercle cercle1;           // déclaration  
cercle1 = new Cercle(5);  // instantiation  
Cercle cercle2 = new Cercle(3); // déclaration et instantiation  
  
cercle2 = cercle1;  
cercle1.Rayon = 10;  
  
Console.WriteLine("cercle1.Rayon = " + cercle1.Rayon); // 10  
Console.WriteLine("cercle2.Rayon = " + cercle2.Rayon); // 10 (???)
```



A la fin de l'exécution, le rayon de `cercle1` vaut 10 et celui de `cercle2` vaut... 10 .



Le résultat précédent devrait vous surprendre... On peut le compléter en observant ce qui se passe au débogueur. Voici le contenu des variables avant le changement du rayon de `cercle1` .

```
cercle2 = cercle1;
cercle1.Rayon = 10;
Console.WriteLine("cercle1.Rayon = " + cercle1.Rayon);
Console.WriteLine("cercle2.Rayon = " + cercle2.Rayon);
```

Variables locales

Nom	Valeur	Type
args	{string[0]}	string[]
cercle1	{Exemple_2.Cercle}	Exemple_2.Cercle
Rayon	5.0	double
rayon	5.0	double
cercle2	{Exemple_2.Cercle}	Exemple_2.Cercle
Rayon	5.0	double
rayon	5.0	double

Et voici le contenu des variables *après* le changement du rayon de `cercle1` .

```
cercle2 = cercle1;
cercle1.Rayon = 10;
Console.WriteLine("cercle1.Rayon = " + cercle1.Rayon);
Console.WriteLine("cercle2.Rayon = " + cercle2.Rayon);
```

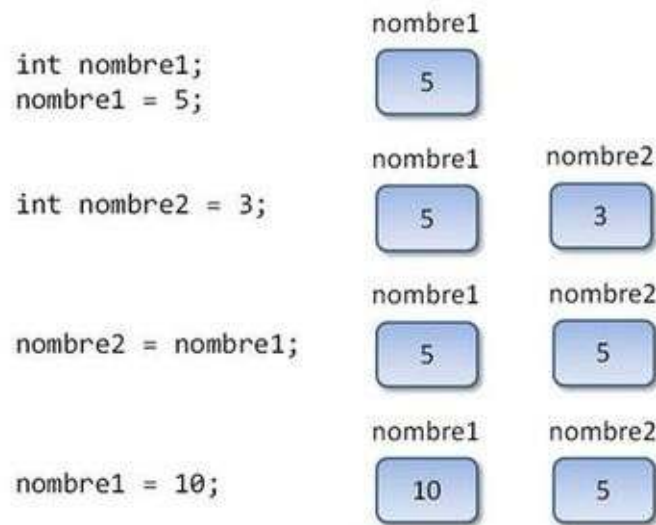
Variables locales

Nom	Valeur	Type
args	{string[0]}	string[]
cercle1	{Exemple_2.Cercle}	Exemple_2.Cercle
Rayon	10.0	double
rayon	10.0	double
cercle2	{Exemple_2.Cercle}	Exemple_2.Cercle
Rayon	10.0	double
rayon	10.0	double

L'instruction `cercle1.Rayon = 10` a simultanément modifié le rayon de `cercle2` . Comment expliquer ce mystère ?

La notion de référence

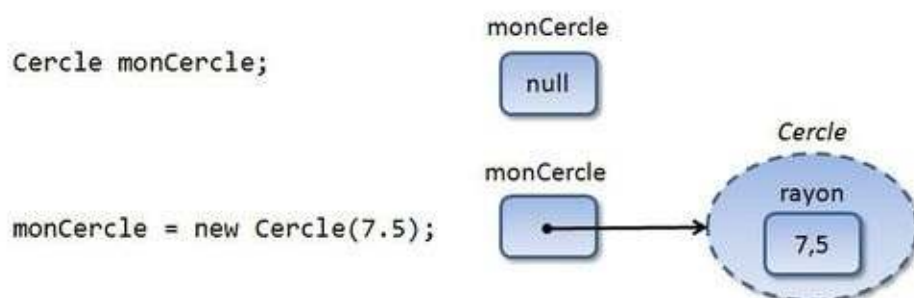
Revenons à notre exemple initial. Nous savons que la création d'une variable déclenche la réservation en mémoire d'une zone dédiée au stockage de sa valeur.



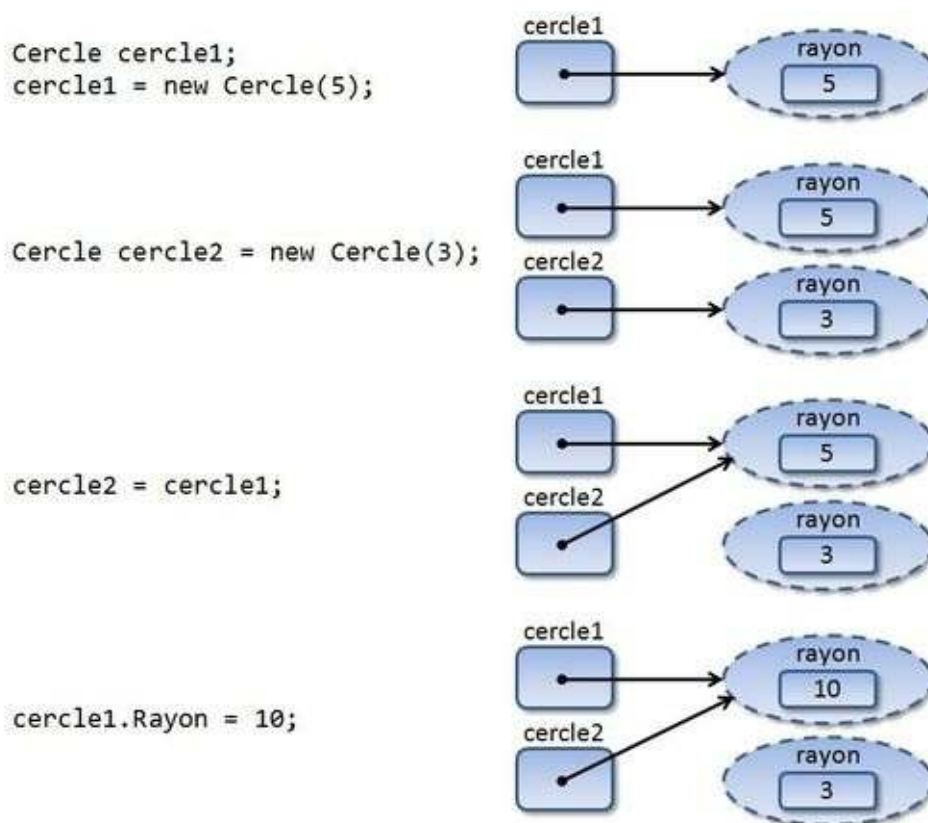
Les variables `nombre1` et `nombre2` correspondent à des zones mémoire distinctes. Toute modification de l'une (exemple : `nombre1 = 10`) n'a aucun impact sur l'autre.

Les objets fonctionnent de manière différente. Une variable objet ne stocke pas directement les données de l'objet, mais une **référence** vers l'emplacement mémoire où se trouvent ces données. De manière simpliste, on peut considérer une référence comme une adresse mémoire.

Une instantiation d'objet comme `monCercle = new Cercle(7.5)` provoque la réservation d'une zone mémoire pour stocker les données de l'objet, et affecte à la variable `monCercle` la référence vers cette zone mémoire.



Les observations précédentes ont maintenant une explication : l'affectation `cercle2 = cercle1` provoque la copie de la **référence** de `cercle1` dans `cercle2` . Après cette assignation, les deux variables "pointent" vers la même zone mémoire contenant les données du premier cercle.



DEFINITION : l'affectation d'un objet à un autre ne déclenche pas la copie du *contenu* des objets. La **référence** (adresse mémoire) du premier objet est affectée au second, et les deux objets "pointent" vers la même zone mémoire.

Pour dupliquer l'objet lui-même et non sa référence, il faut utiliser d'autres techniques que vous découvrirez ultérieurement.

Types valeurs et types références

DEFINITION : les types de données C# se répartissent en deux catégories :

- les types **valeurs**, où la valeur est directement stockée dans la variable.
- les types **références**, où la variable stocke l'emplacement mémoire de la valeur.

Les types prédéfini (`int` , `double` , etc) sont des types valeurs. Les classes et les tableaux sont des types référence.

Pourquoi avoir introduit la notion de référence dans le langage ? Essentiellement pour des raisons de performances. Contrairement aux types valeur, un objet (ou une liste d'objets) peut occuper une taille non négligeable en mémoire. De trop nombreuses copies d'objets auraient donc pu ralentir l'exécution d'un programme. Grâce aux références, une affectation entre objets est quasi-instantanée : la seule information copiée est une adresse mémoire, et non l'objet lui-même.

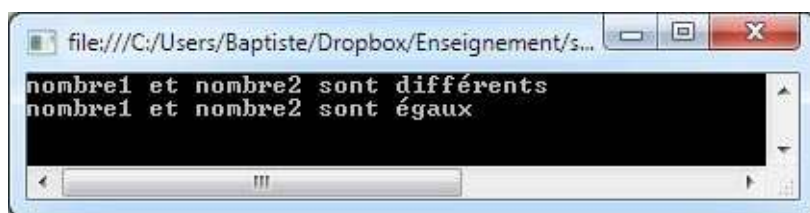
Le langage Java fait aussi la distinction entre types valeurs et références ([Parameter passing in Java](#)). Les objets en Python stockent également des références. Le langage C++ va encore plus loin : il distingue références et *pointeurs*.

Il est très important de savoir si une variable est de type valeur ou de type référence car cela a un impact sur la comparaison d'objets et le passage d'un objet en paramètre.

Comparaison d'objets

Etudions ce qui se produit lorsque l'on compare deux variables de type valeur, comme des entiers.

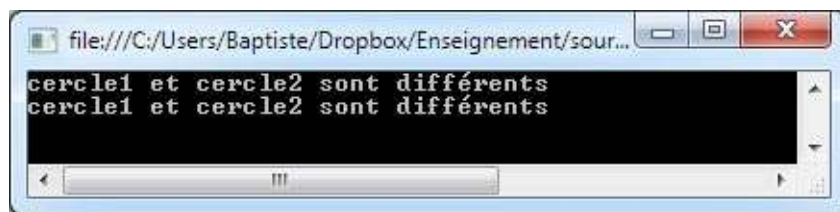
```
int nombre1;  
nombre1 = 5;  
  
int nombre2 = 3;  
  
if (nombre1 == nombre2)  
    Console.WriteLine("nombre1 et nombre2 sont égaux");  
else  
    Console.WriteLine("nombre1 et nombre2 sont différents");  
  
nombre2 = 5;  
  
if (nombre1 == nombre2)  
    Console.WriteLine("nombre1 et nombre2 sont égaux");  
else  
    Console.WriteLine("nombre1 et nombre2 sont différents");
```



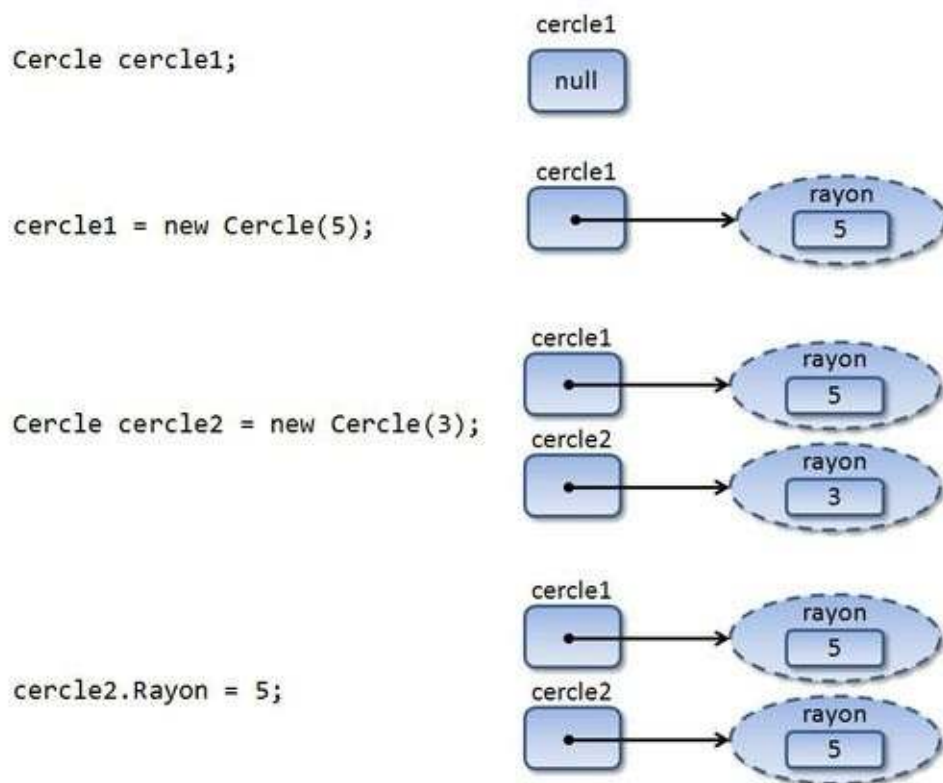
L'exécution de ce code indique d'abord que les variables sont différentes, puis qu'elles sont égales. Ceci est attendu puisque nous avons donné à `nombre2` la même valeur que `nombre1`.

Ecrivons un code similaire, mais qui utilise des objets.

```
Cercle cercle1;  
cercle1 = new Cercle(5);  
  
Cercle cercle2 = new Cercle(3);  
  
if (cercle1 == cercle2)  
    Console.WriteLine("cercle1 et cercle2 sont égaux");  
else  
    Console.WriteLine("cercle1 et cercle2 sont différents");  
  
cercle2.Rayon = 5;  
  
if (cercle1 == cercle2)  
    Console.WriteLine("cercle1 et cercle2 sont égaux");  
else  
    Console.WriteLine("cercle1 et cercle2 sont différents");
```



Son exécution indique que les objets sont différents, puis qu'ils sont différents. Et pourtant, leurs états (valeur du rayon) sont identiques... Pour comprendre, il est utile de représenter ce qui se passe en mémoire pendant l'exécution.



Le contenu des deux objets est bien identique, mais ils se trouvent à des emplacements mémoire distincts. Les variables `cercle1` et `cercle2` stockent des références vers deux objets distincts, qui dans ce cas particulier ont le même état.

L'expression `cercle1 == cercle2` compare les valeurs des références et non les états des objets eux-mêmes : elle renvoie toujours `false` dans notre exemple.

La notion "d'égalité d'objets" est donc plus complexe qu'elle n'y paraît. Ici encore, il existe d'autres techniques pour comparer les objets et non leurs références.

Passage d'un objet en paramètre

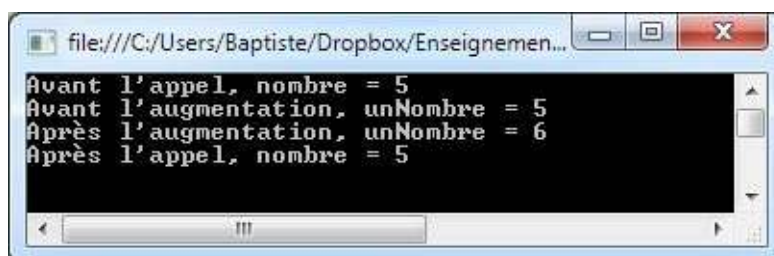
Il est important de comprendre la manière dont un objet est passé en paramètre à un sous-programme ou à une méthode. Pour cela, nous allons revenir en détail sur le fonctionnement du passage de paramètre.

Le code ci-dessous permet d'illustrer le passage d'un paramètre entier.

```
static void Main(string[] args)
{
    int nombre = 5;

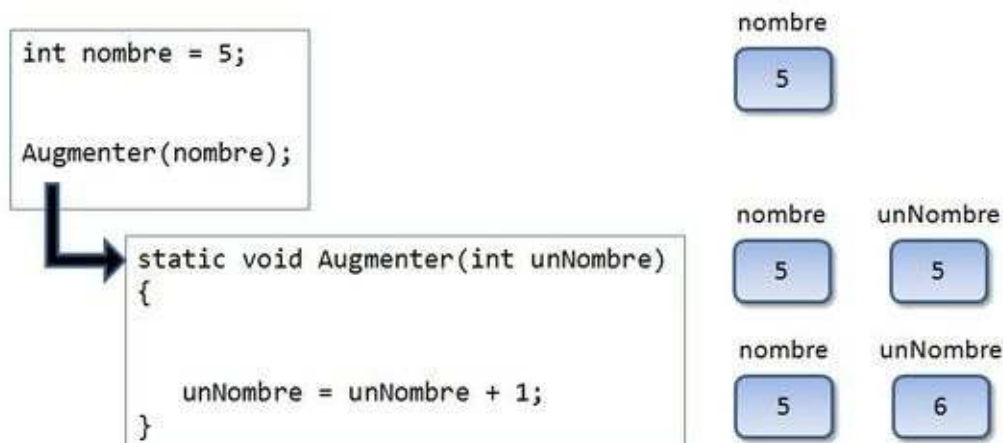
    Console.WriteLine("Avant l'appel, nombre = " + nombre);
    Augmenter(nombre);
    Console.WriteLine("Après l'appel, nombre = " + nombre);
}

static void Augmenter(int unNombre)
{
    Console.WriteLine("Avant l'augmentation, unNombre = " + unNombre);
    unNombre = unNombre + 1;
    Console.WriteLine("Après l'augmentation, unNombre = " + unNombre);
}
```



Nous savons expliquer ce résultat : il est dû au fait que les paramètres sont passés par **valeur**. Au moment de l'appel à `Augmenter`, la valeur de l'argument `nombre` est copiée dans l'espace mémoire du paramètre `unNombre`. La même valeur 5 est donc stockée à deux

emplacements mémoire distincts. Toute modification de `unNombre` dans le sous-programme n'aura aucun impact sur la valeur de `nombre` dans le programme principal.

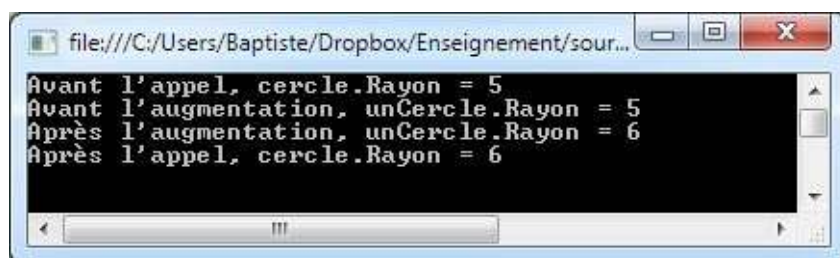


Voyons à présent ce qui se produit dans le cas d'objets.

```
static void Main(string[] args)
{
    Cercle cercle = new Cercle(5);

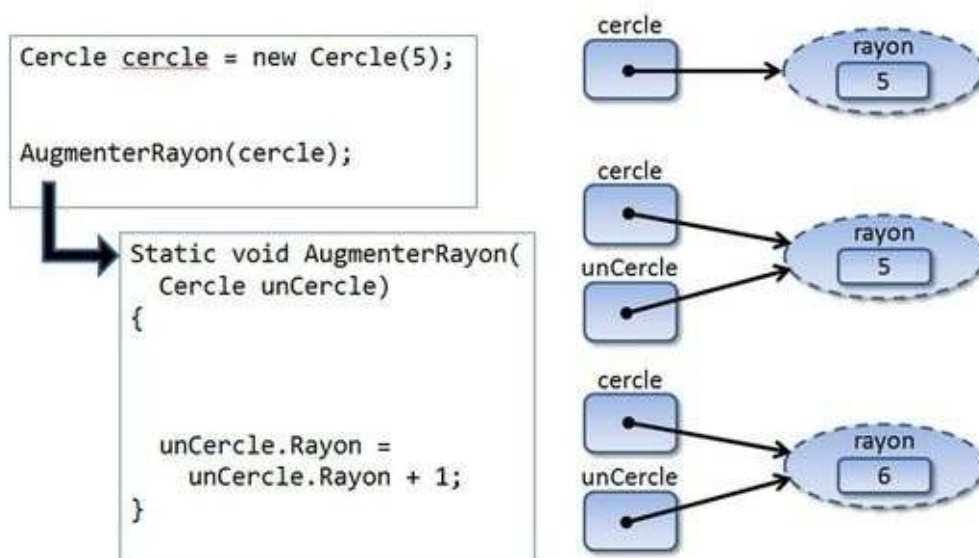
    Console.WriteLine("Avant l'appel, cercle.Rayon = " + cercle.Rayon);
    AugmenterRayon(cercle);
    Console.WriteLine("Après l'appel, cercle.Rayon = " + cercle.Rayon);
}

static void AugmenterRayon(Cercle unCercle)
{
    Console.WriteLine("Avant l'augmentation, unCercle.Rayon = " + unCercle.Rayon);
    unCercle.Rayon = unCercle.Rayon + 1;
    Console.WriteLine("Après l'augmentation, unCercle.Rayon = " + unCercle.Rayon);
}
```



L'appel au sous-programme a modifié la valeur de l'objet ! Ceci semble contradictoire avec le mode de passage des paramètres par valeur, où l'argument est copié dans le paramètre.

Tentons de comprendre pourquoi avec une représentation mémoire de l'exécution.



Au moment de l'appel du sous-programme, la valeur de l'argument `cercle`, autrement dit la référence vers l'objet associé, est copiée dans le paramètre `unCercle`. Les variables `cercle` et `unCercle` contiennent la même référence et "pointent" donc vers le même emplacement mémoire. Cela explique que la modification faite dans le sous-programme ait un impact au niveau du programme principal : l'objet référencé est le même.

DEFINITION : par défaut, **tous les paramètres sont passés par valeur en C#**.

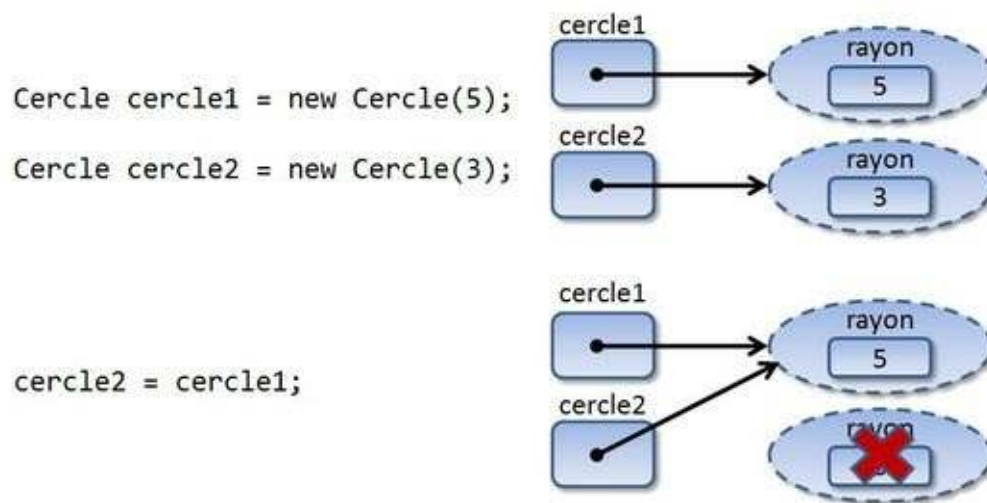
- Dans le cas d'un type **valeur**, la valeur est copiée de l'argument vers le paramètre. Paramètre et argument concernent des zones mémoires différentes. Les modifications du paramètre n'ont pas d'impact sur l'argument.
- Dans le cas d'un type **référence**, la référence est copiée de l'argument vers le paramètre. Paramètre et argument concernent la même zone mémoire. Les modifications du paramètre modifient aussi l'argument.

Destruction d'un objet

L'instanciation d'un objet provoque la réservation d'espace mémoire. Sans libération de cet espace une fois que l'objet sera devenu inutile, le risque existe d'une saturation de la mémoire de l'application, voire du système.

Dans certains langages comme le C++, la libération de la mémoire est de la responsabilité du programmeur, ce qui est source de problèmes complexes. Heureusement pour vous, le C# a suivi l'exemple de Java et dispose d'une fonctionnalité de libération mémoire automatique appelée **ramasse-miettes** (*garbage collector*).

Cela fonctionne de manière simple : tout objet, une fois qu'il n'est plus pointé par aucune référence, devient éligible pour la destruction (c'est-à-dire la libération de l'espace mémoire associé).



Dans l'exemple ci-dessus, l'objet initialement pointé par `cercle2` n'est plus pointé par aucune référence et sera prochainement détruit automatiquement par le ramasse-miettes.

La relation d'association

L'objectif de ce chapitre est de découvrir pourquoi et comment mettre des objets en relation les uns avec les autres.

Les exemples de code associés sont [disponibles en ligne](#).

Introduction aux relations entre objets

La nécessité de relations

Comme nous l'avons déjà vu, la programmation orientée objet consiste à concevoir une application sous la forme de "briques" logicielles appelées des objets. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues.

Nous savons qu'un **objet** est une entité qui représente (modélise) un élément du domaine étudié : une voiture, un compte bancaire, un nombre complexe, une facture, etc. Un objet est toujours créé d'après un modèle, qui est appelé sa **classe**.

Sauf dans les cas les plus simples, on ne pourra pas modéliser fidèlement le domaine étudié en se contentant de concevoir une seule classe. Il faudra définir plusieurs classes et donc instancier des objets de classes différentes. Cependant, ces objets doivent être mis en relation afin de pouvoir communiquer.

Contexte d'exemple

Reprenons notre classe modélisant un compte bancaire, issue d'un précédent chapitre.

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    public string Titulaire
    {
        get { return titulaire; }
    }

    public double Solde
    {
        get { return solde; }
    }

    public string Devise
    {
        get { return devise; }
    }

    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    public void Crediter(double montant)
    {
        solde += montant;
    }

    public void Debiter(double montant)
    {
        solde -= montant;
    }

    public string Decrire()
    {
        return "Le solde du compte de " + titulaire + " est de " + solde + " " + devis
e;
    }
}
```

Evolution des besoins

On souhaite à présent enrichir notre modélisation orientée objet en incluant des informations détaillées sur le titulaire d'un compte : son nom, son prénom et son numéro de client.

Une première idée serait d'ajouter dans notre classe les attributs correspondants. C'est une mauvaise idée : les données d'un client seraient dupliquées dans chacun de ses comptes. Et le rôle de la classe `CompteBancaire` est de modéliser un compte, pas un client.

La bonne solution est de créer une nouvelle classe représentant un client.

Modélisation du client

On crée la classe `Client` dont le rôle sera de représenter les clients titulaires des comptes.

```
// Modélise un client
public class Client
{
    private int numero;    // Numéro de compte
    private string nom;    // Nom
    private string prenom; // Prénom

    public int Numero
    {
        get { return numero; }
    }

    public string Nom
    {
        get { return nom; }
    }

    public string Prenom
    {
        get { return prenom; }
    }

    public Client(int leNumero, string leNom, string lePrenom)
    {
        numero = leNumero;
        nom = leNom;
        prenom = lePrenom;
    }
}
```



On retrouve dans cette classe, sous forme d'attributs, les données caractérisant un client : numéro, nom et prénom. On pourra ensuite instancier des objets de cette classe pour représenter les clients de la banque.

```
Client pierre = new Client(123456, "Kiroul", "Pierre");  
Client paul = new Client(987654, "Ochon", "Paul");
```

A ce stade, nous avons d'un côté des comptes, de l'autre des clients, mais pas encore de relations entre eux. Plus précisément, il faudrait pouvoir modéliser l'information "ce compte a pour titulaire ce client". Pour cela, on modifie le type de l'attribut `titulaire` dans `CompteBancaire` : on remplace le type `string` par le type `Client`. Il faut également modifier l'accessor et le constructeur pour faire le même changement.

```
class CompteBancaire  
{  
    private Client titulaire; // type string => type Client  
  
    public Client Titulaire  
    {  
        get { return titulaire; }  
    }  
  
    public CompteBancaire(Client leTitulaire, double soldeInitial, string laDevise)  
    {  
        titulaire = leTitulaire;  
        solde = soldeInitial;  
        devise = laDevise;  
    }  
  
    // ...  
}
```

La création d'un nouveau compte nécessite de passer en paramètre le client titulaire.

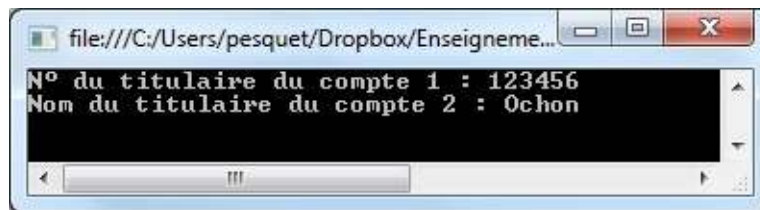
```

Client pierre = new Client(123456, "Kiroul", "Pierre");
Client paul = new Client(987654, "Ochon", "Paul");

CompteBancaire compte1 = new CompteBancaire(pierre, 500, "euros");
CompteBancaire compte2 = new CompteBancaire(paul, 1000, "euros");

Console.WriteLine("N° du titulaire du compte 1 : " + compte1.Titulaire.Numero);
Console.WriteLine("Nom du titulaire du compte 2 : " + compte2.Titulaire.Nom);

```



La propriété C# `Titulaire` de la classe `CompteBancaire` renvoie une instance de la classe `Client`. On peut ensuite utiliser ses propriétés et ses méthodes, comme pour n'importe quel autre objet.

On souhaite à présent afficher la description de chaque compte.

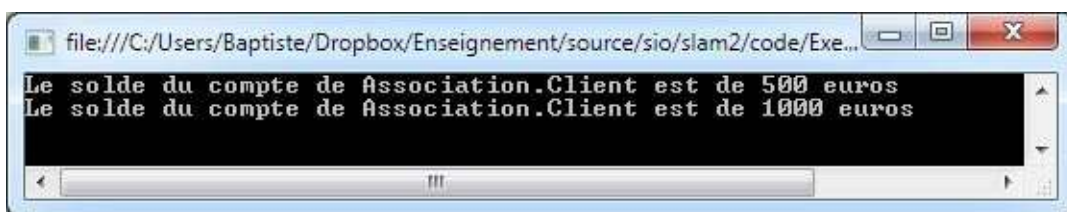
```

Client pierre = new Client(123456, "Kiroul", "Pierre");
Client paul = new Client(987654, "Ochon", "Paul");

CompteBancaire compte1 = new CompteBancaire(pierre, 500, "euros");
CompteBancaire compte2 = new CompteBancaire(paul, 1000, "euros");

Console.WriteLine(compte1.Decrire());
Console.WriteLine(compte2.Decrire());

```



Le résultat d'exécution ci-dessus est étrange mais logique. Pour le comprendre, revenons au code source de la méthode `Decrire`.

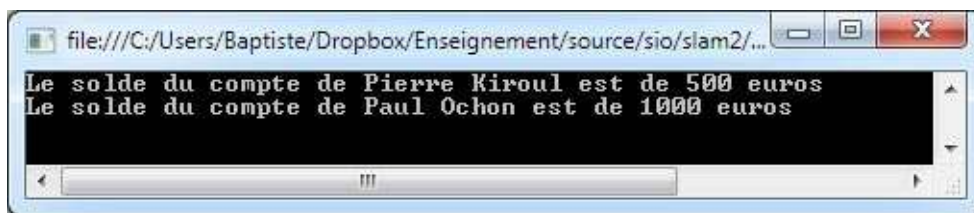
```

public string Decrire()
{
    string description = "Le solde du compte de " + titulaire + " est de " + solde + "
" + devise;
    return description;
}

```

Ici on concatène l'attribut `titulaire`, instance de la classe `Client`, à une chaîne de caractères pour créer la description du compte. Le langage C# ne sait pas comment représenter un client sous forme textuelle et il renvoie le nom complet de la classe `Client` (ici, `Association` correspond à l'espace de noms dont lequel est définie la classe `Client`). Il est donc nécessaire de modifier la méthode `Decrire` afin d'afficher toutes ses propriétés.

```
public string Decrire()  
{  
    string descriptionTitulaire = titulaire.Prenom + " " + titulaire.Nom;  
    string description = "Le solde du compte de " + descriptionTitulaire + " est de "  
+ solde + " " + devise;  
    return description;  
}
```



REMARQUE : il existe une meilleure solution pour obtenir une représentation textuelle d'un objet. Elle sera étudiée dans un [prochain chapitre](#).

Dans cet exemple, on a mis en relation un objet de la classe `CompteBancaire` avec un objet de la classe `Client`. En terminologie objet, on dit qu'on a créé une **association** entre les deux classes.

Associations entre classes

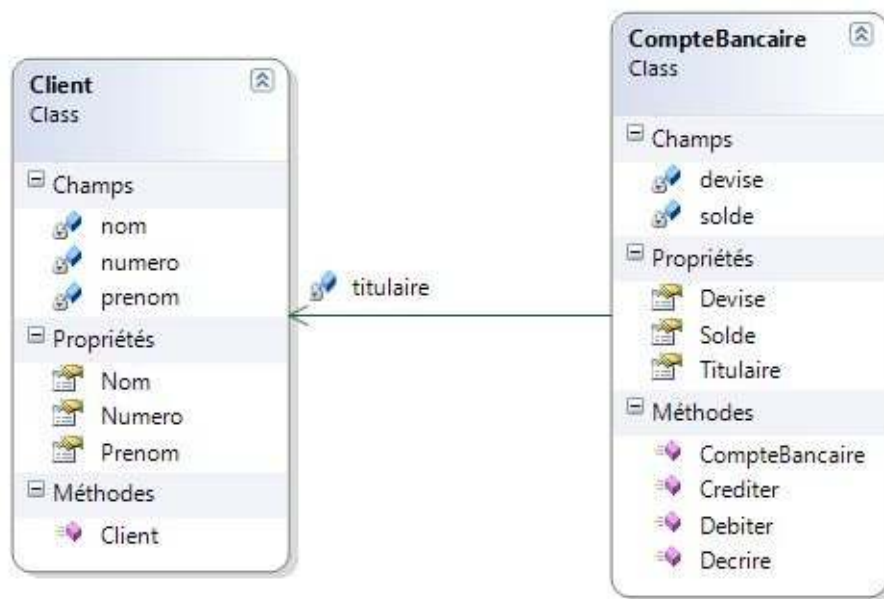
Définition

L'exemple précédent a permis de mettre en relation un compte bancaire et son client titulaire. Grâce à cette association, nous avons modélisé la relation "un compte bancaire a un titulaire".

DEFINITION : une **association** modélise une relation entre classes de type "**a un**" ou "**a plusieurs**". Un compte bancaire *a un* titulaire, un livre *a plusieurs* pages, etc.

AVERTISSEMENT : ne pas confondre avec une relation de type "**est un**", modélisée par l'**héritage** (voir chapitre suivant).

On la représente graphiquement en UML par un trait continu entre les deux classes concernées.



On observe que l'attribut `titulaire` de `CompteBancaire` n'est plus listé dans cette classe, mais représenté au-dessus du trait d'association, côté classe `Client`. Il s'agit du **rôle** que jouent les instances de `Client` dans l'association : un client est le titulaire d'un compte bancaire.

La flèche qui pointe vers `Client` indique le sens de **navigation** de l'association.

Une relation d'association implique un lien entre les classes concernées. Elles ne peuvent plus fonctionner indépendamment l'une de l'autre. On parle également de **couplage** entre les classes.

Multiplicités d'une association

Notre définition de la classe `CompteBancaire` implique qu'un objet de cette classe a un et un seul titulaire. En revanche, rien n'empêche que plusieurs comptes bancaires aient le même client comme titulaire.

Autrement dit, notre modélisation conduit aux règles ci-dessous :

- une instance de `CompteBancaire` est associée à une seule instance de `Client`.
- une instance de `Client` peut être associée à un nombre quelconque d'instances de `CompteBancaire`.

Cela se représente graphiquement par l'ajout de **multiplicités** à chaque extrémité de l'association.



DEFINITION : la **multiplicité** située à une extrémité d'une association UML indique à **combien d'instances de la classe une instance de l'autre classe peut être liée**.

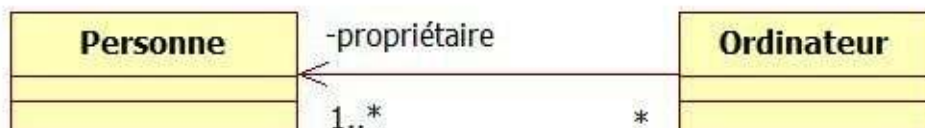
ATTENTION : les multiplicités UML se lisent dans le sens **inverse** des cardinalités Merise.

Le tableau ci-dessous rassemble les principales multiplicités utilisées en UML.

Multiplicité	Signification
0..1	Zéro ou un
1	Un
*	De zéro à plusieurs
1..*	De un à plusieurs

DEFINITION : la multiplicité par défaut (en l'absence de mention explicite à l'extrémité d'une association) est **1**.

En étudiant leurs multiplicités, on peut trouver la sémantique des associations ci-dessous.



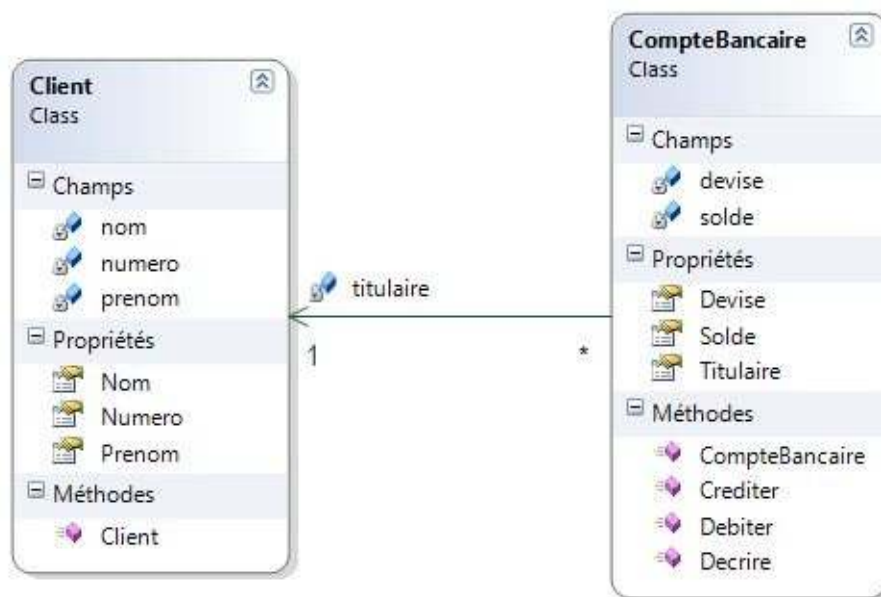
- Une personne possède **zéro ou plusieurs** ordinateurs.
- Un ordinateur appartient à **une ou plusieurs** personnes.



- Une personne possède **zéro ou un** ordinateur.
- Un ordinateur appartient à **une** personne.

Navigabilité d'une association

Revenons à notre exemple des comptes bancaires associés à leur titulaire.



A partir d'une instance de la classe `CompteBancaire`, il est possible d'accéder à son titulaire, instance de la classe `Client`. Par contre, à partir d'une instance de `Client`, rien ne permet de retrouver les comptes dont il est titulaire. L'association entre `CompteBancaire` et `Client` ne peut être *naviguée* que dans un seul sens, de `CompteBancaire` vers `Client`.

DEFINITION : une association navigable dans un seul sens est une association **unidirectionnelle**.

Dans le diagramme UML, le sens de navigation est exprimé par une **flèche** qui pointe vers la classe vers laquelle on peut naviguer (ici la classe `Client`).

AVERTISSEMENT : ne pas confondre la flèche de la navigabilité unidirectionnelle avec la flèche pleine de l'héritage.

On pourrait souhaiter que l'association soit également navigable mais dans le sens inverse, c'est-à-dire qu'on puisse retrouver les comptes dont un client est titulaire. Pour cela, il faut modifier la classe `Client`. Il est nécessaire d'y ajouter un attribut qui stockera les comptes

bancaires dont un client est titulaire. Comme un client peut être titulaire de plusieurs comptes, cet attribut sera une liste d'instances de la classe `CompteBancaire`.

```
public class Client
{
    // ...
    private List<CompteBancaire> comptes;

    public Client(int numero, string nom, string prenom)
    {
        comptes = new List<CompteBancaire>();
        // ...
    }

    public List<CompteBancaire> Comptes
    {
        get { return comptes; }
    }

    // ...
}
```

L'utilisation des classes reflète les changements effectués : il faut maintenant ajouter à un client les comptes dont il est titulaire.

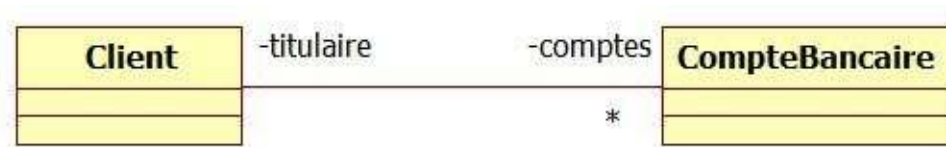
```
Client pierre = new Client(123456, "Kiroul", "Pierre");
Client paul = new Client(987654, "Ochon", "Paul");

// association entre pierre et compte1
CompteBancaire compte1 = new CompteBancaire(pierre, 500, "euros");
pierre.Comptes.Add(compte1);

// association entre paul et compte2
CompteBancaire compte2 = new CompteBancaire(paul, 1000, "euros");
paul.Comptes.Add(compte2);
```

DEFINITION : une association navigable dans les deux sens est une association **bidirectionnelle**.

Dans un diagramme de classes UML, l'absence de flèche sur le lien d'association indique qu'elle est bidirectionnelle et donc navigable dans les deux sens.



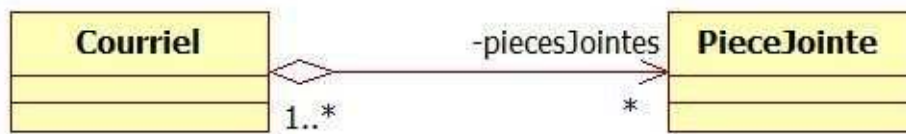
CONSEIL : en pratique, une association bidirectionnelle est plus complexe à coder : il faut penser à mettre à jour les attributs de chaque classe pour qu'elle reste cohérente. On privilégiera le plus souvent les associations unidirectionnelles en choisissant le sens de navigation le plus utile.

Types particuliers d'associations

L'agrégation

Prenons l'exemple d'un courrier électronique (courriel). Il peut éventuellement contenir une ou plusieurs pièces jointes. Les pièces jointes font partie du courriel. Un courriel et ses pièces jointes sont liés par une relation de type "se compose de", "composant/composé" ou encore "tout/partie".

Dans le cadre d'une modélisation objet, on représente cette association particulière par un lien dont une extrémité (celle du composé) comporte un losange vide. On la nomme **agrégation**.



DEFINITION : une **agrégation** est une association qui modélise une relation "se compose de".

REMARQUES

- Un composant (ici, une pièce jointe) peut être partagé par plusieurs composés (ici, des courriels).
- La traduction en code source d'une agrégation est identique à celle d'une association.

La composition

Un livre se compose d'un certain nombre de pages. On pourrait modéliser le lien livre-page par une agrégation. Cependant, les cycles de vie des objets associés sont liés : si un livre disparaît, toutes les pages qui le composent disparaissent également. Ce n'est pas le cas pour l'exemple précédent : une pièce jointe n'est pas détruite si le courriel auquel elle est jointe disparaît.

Pour traduire une relation composant/composé dont les cycles de vie sont liés, on représente l'association correspondante avec un losange plein du côté du composé. On la nomme **composition**.



DEFINITION : une **composition** est une agrégation entre objets dont les cycles de vie sont liés.

REMARQUES

- Un composant (ici, une page) ne peut pas être partagé par plusieurs composés (ici, des livres). Par conséquent, la multiplicité côté composé est toujours 1.
- La traduction en code source d'une composition ressemble à celle d'une agrégation. Comme les cycles de vie des objets sont liés, les composants sont parfois instanciés par le constructeur du composé.

Traduction en code d'une association

Une association entre deux classes se traduit par l'ajout d'attributs. Les classes associées possèdent en attribut une ou plusieurs références vers l'autre classe.

La présence ou l'absence de références dans une classe dépend de la **navigabilité** de l'association. Si des références sont présentes, leur nombre dépend des **multiplicités** de l'association.

Soit le diagramme UML ci-dessous.



L'association est navigable de `Ordinateur` vers `Personne`. Il y aura donc une ou plusieurs références à `Personne` dans la classe `Ordinateur`, mais pas de référence à `Ordinateur` dans la classe `Personne`.

La multiplicité au niveau de `Personne` est la multiplicité par défaut, 1. Un ordinateur appartient à une et une seule personne. Il y a donc une référence à `Personne` dans la classe `ordinateur`. Afin de respecter le rôle défini pour la classe `Personne` dans l'association, l'attribut correspondant est nommé `proprietaire`.

```

class Ordinateur
{
    private Personne proprietaire;
    // ...
}
  
```

DEFINITION : une multiplicité 1 ou 0..1 se traduit par l'ajout d'une seule référence comme attribut.

Soit le diagramme UML ci-dessous.



L'association est navigable de `Livres` vers `Pages`. Il y aura donc une ou plusieurs références à `Pages` dans la classe `Livres`, mais pas de référence à `Livres` dans la classe `Pages`.

La multiplicité au niveau de `Pages` est `1..*`. Un livre est composé de une ou plusieurs pages. L'attribut ajouté dans `Livres` doit permettre de stocker un nombre quelconque d'instances de `Pages`. On utilise donc une liste d'objets de type `Pages`. Afin de respecter le rôle défini pour la classe `Pages` dans l'association, l'attribut correspondant est nommé `pages`.

```
class Livres
{
    private List<Pages> pages;
    // ...
}
```

DEFINITION : une multiplicité * ou 1..* se traduit par l'ajout d'une liste de références comme attribut.

La relation d'héritage

L'héritage est l'un des mécanismes fondamentaux de la POO. Il permet de créer des classes à partir de classes existantes. L'objectif de ce chapitre est de découvrir son fonctionnement.

Les exemples de code associés sont [disponibles en ligne](#).

Premiers pas

Exemple d'utilisation

Reprenons la classe `CompteBancaire` utilisée dans un précédent chapitre.

```
// Définit un compte bancaire
public class CompteBancaire
{
    private string titulaire; // Titulaire du compte
    private double solde;     // Solde du compte
    private string devise;    // Devise du compte

    public string Titulaire
    {
        get { return titulaire; }
    }

    public double Solde
    {
        get { return solde; }
    }

    public string Devise
    {
        get { return devise; }
    }

    // Constructeur
    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    // Ajoute un montant au compte
    public void Crediter(double montant)
    {
        solde = solde + montant;
    }

    // Retire un montant au compte
    public void Debiter(double montant)
    {
        solde = solde - montant;
    }

    // Renvoie la description du compte
    public string Decrire()
    {
        string description = "Le solde du compte de " + titulaire + " est de " + solde
+ " " + devise;
        return description;
    }
}
```


Supposons maintenant que nous ayons à gérer un nouveau type de compte : le compte épargne. Comme un compte classique, un compte épargne possède un titulaire, un solde et une devise. Sa spécificité est qu'il permet d'appliquer des intérêts à l'argent déposé sur le compte.

Bien sûr, il serait possible de concevoir une classe `CompteEpargne` totalement distincte de la classe `CompteBancaire`. Cependant, on constate qu'un compte épargne possède toutes les caractéristiques d'un compte bancaire plus des caractéristiques spécifiques. Nous allons donc définir un compte épargne par **héritage** de la définition d'un compte bancaire.

```
public class CompteEpargne : CompteBancaire
{
    private double tauxInteret;

    public CompteEpargne(string leTitulaire, double soldeInitial, string laDevise, double leTauxInteret)
        : base(leTitulaire, soldeInitial, laDevise)
    {
        // appel du constructeur de la classe CompteBancaire
        // le mot-clé "base" permet d'accéder à la classe parente
        tauxInteret = leTauxInteret;
    }

    // Calcule et ajoute les intérêts au solde du compte
    public void AjouterInterets()
    {
        // ... (détaillé plus bas)
    }
}
```

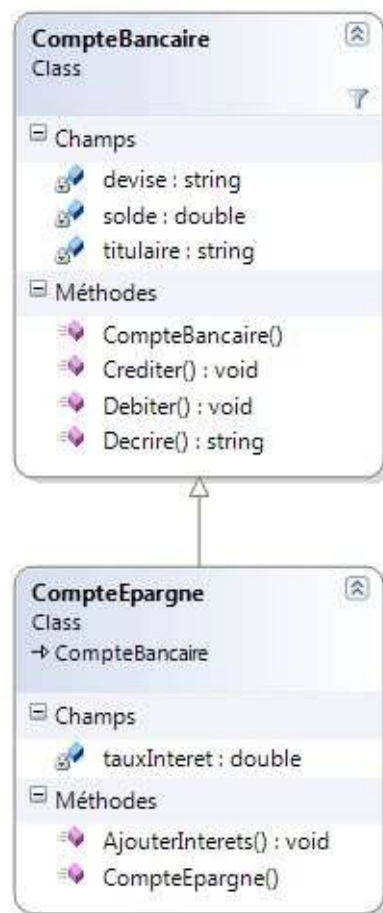
Dans la déclaration `class CompteEpargne : CompteBancaire`, les deux-points spécifient que la classe `CompteEpargne` hérite de la classe `CompteBancaire`.

REMARQUE : d'autres langages comme Java ou PHP utilisent le mot-clé `extends` plutôt que le symbole `:` pour indiquer une relation d'héritage entre deux classes.

On observe que le constructeur de `CompteEpargne` fait appel au constructeur de `CompteBancaire` pour en initialiser les attributs. Le mot-clé `base` désigne la classe parente. Le constructeur de `CompteEpargne` initialise également l'attribut `tauxInteret` qui lui est propre.

Représentation graphique

Le formalisme graphique UML décrit la relation d'héritage entre deux classes par une **flèche pleine** allant de la classe dérivée à la classe de base (les propriétés C# de `CompteBancaire` ont été masquées afin d'alléger le diagramme).



Définition

DEFINITION : l'**héritage** est un mécanisme objet qui consiste à définir une classe à partir d'une classe existante. Une classe héritant d'une autre classe possède les caractéristiques de la classe initiale et peut définir ses propres éléments.

La nouvelle classe (ou classe **dérivée**) correspond à une **spécialisation** de la classe de base (appelée classe **parente** ou **superclasse**). On dit que l'héritage crée une relation de type **est un** entre les classes. Dans notre exemple, un compte épargne *est un* type particulier de compte bancaire.

ATTENTION : le constructeur d'une classe dérivée doit obligatoirement faire explicitement appel au constructeur de la classe parente lorsque celui-ci prend des paramètres. C'est le cas dans notre exemple.

Avantages

Grâce à la relation d'héritage, un objet de la classe `CompteEpargne` peut utiliser les fonctionnalités de la classe `CompteBancaire` sans avoir à les redéfinir. On peut donc débiter ou créditer un compte épargne exactement comme un compte bancaire.

```
double tauxInteret = 0.05; // taux d'intérêt : 5%
CompteEpargne comptePaul = new CompteEpargne("paul", 100, "dollars", tauxInteret);

// appel des méthodes de CompteBancaire sur le compte épargne
comptePaul.Debiter(1000);
comptePaul.Crediter(1500);
Console.WriteLine(comptePaul.Decrire()); // Affiche 600 $
```

Par contre, le calcul des intérêts (méthode `AjouterInterets`) ne peut se faire que sur un objet de la classe `CompteEpargne`. L'héritage est une relation *unidirectionnelle*.

```
// OK : comptePaul est un compte épargne
comptePaul.AjouterInterets();
Console.WriteLine(comptePaul.Decrire());

CompteBancaire comptePierre = new CompteBancaire("pierre", 100, "dollars");
// Erreur : comptePierre est un compte bancaire, pas un compte épargne
comptePierre.AjouterInterets();
```

Grâce à l'héritage, il est possible de réutiliser les fonctionnalités d'une classe existante en la spécialisant. Il est également possible de spécialiser une classe dérivée.

On voit bien tous les avantages que l'héritage peut apporter : gain de temps de développement, amélioration de la qualité du code, création de hiérarchies de classes reflétant précisément le domaine d'étude, etc.

Héritage et encapsulation

Nous avons volontairement laissé de côté un point délicat. La méthode `AjouterInterets` de la classe `CompteEpargne`, qui doit ajouter les intérêts au solde, n'est pas encore définie. En voici une première version.

```
public class CompteEpargne : CompteBancaire
{
    // ...

    public void AjouterInterets()
    {
        // calcul des intérêts sur le solde
        double interets = solde * tauxInteret;
        // ajout des intérêts au solde
        solde += interets;
    }
}
```

Cependant, le compilateur nous signale l'erreur suivante.

```
public void AjouterInterets()
{
    // calcul des intérêts sur le solde
    double interets = solde * tauxInteret;
    // ajout des intérêts
    solde += interets;
}
```

Exemple_8.CompteBancaire.solde' est inaccessible en raison de son niveau de protection

Dans cet exemple, la classe dérivée `CompteEpargne` tente d'accéder à l'attribut `solde` qui appartient à la classe de base `CompteBancaire`. Cependant, cet attribut est défini avec le niveau de visibilité `private` ! Cela signifie qu'il n'est utilisable que dans la classe où il est défini, et non dans les classes dérivées.

Pour interdire l'accès à un membre d'une classe (attribut, propriété C# ou méthode) depuis l'extérieur tout en permettant son utilisation par une classe dérivée, il faut associer à ce membre un niveau de visibilité intermédiaire : `protected`.

```
public class CompteBancaire
{
    private string titulaire;
    protected double solde; // attribut protégé
    private string devise;

    // ...
}
```

Une autre solution à ce problème consiste à laisser le champ `solde` privé et à définir un accesseur *protégé* pour modifier le solde depuis les méthodes de la classe `CompteEpargne`.

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    // ...

    public double Solde
    {
        get { return solde; } // accesseur public pour la lecture
        protected set { solde = value; } // mutateur protégé pour la modification
    }
    // public string Solde { get; protected set; } // Equivalent avec une propriété automatique

    // ...
}
```

Bien entendu, il faut alors utiliser le mutateur `Solde` et non plus l'attribut `solde` pour accéder au solde depuis la classe dérivée.

```
public class CompteEpargne : CompteBancaire
{
    // ...

    public void AjouterInterets()
    {
        // utilisation du mutateur Solde pour accéder au solde du compte
        double interets = Solde * tauxInteret;
        Solde += interets;
    }
}
```

Le tableau ci-dessous rassemble les trois niveaux de visibilité utilisables.

Visibilité	Classe	Classes dérivées	Extérieur
public	X	X	X
protected	X	X	
private	X		

Polymorphisme

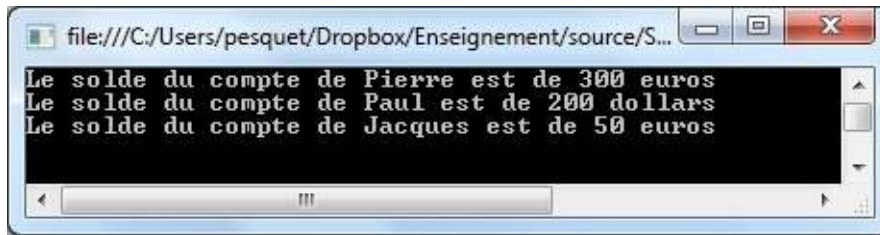
Afin de gagner en genericité, on peut appliquer le même code à des objets de types différents, lorsque les classes de ces objets sont liées par héritage. C'est le principe du **polymorphisme**.

Prenons l'exemple d'une liste de comptes bancaires dont l'un est un compte épargne.

```
CompteBancaire compte1 = new CompteBancaire("Pierre", 300, "euros");
CompteEpargne compte2 = new CompteEpargne("Paul", 200, "dollars", 0.05);
CompteBancaire compte3 = new CompteBancaire("Jacques", 50, "euros");

List<CompteBancaire> listeComptes = new List<CompteBancaire>();
listeComptes.Add(compte1);
listeComptes.Add(compte2); // Est-ce bien un compte bancaire ?
listeComptes.Add(compte3);

foreach (CompteBancaire compte in listeComptes)
    Console.WriteLine(compte.Decrire());
```



```
file:///C:/Users/pesquet/Dropbox/Enseignement/source/S...
Le solde du compte de Pierre est de 300 euros
Le solde du compte de Paul est de 200 dollars
Le solde du compte de Jacques est de 50 euros
```

Ce code fonctionne ! Un compte épargne "est un" compte bancaire. On peut donc stocker un compte épargne dans une liste de comptes bancaires. On peut même appeler la méthode `Decrire` sur chacun des éléments de la liste de comptes bancaires. C'est un exemple très simple de ce qu'on appelle le **polymorphisme**.

DEFINITION : utiliser le **polymorphisme** consiste à écrire un code générique qui pourra s'appliquer à des objets appartenant à des classes différentes.

Le polymorphisme rend le code plus concis, plus élégant et plus sûr. C'est un mécanisme à utiliser dès que l'occasion se présente. Il peut être enrichi grâce aux mécanismes que nous allons découvrir maintenant.

Classes et méthodes abstraites

Evolution des besoins

Nous obtenons les précisions suivantes sur notre domaine d'étude :

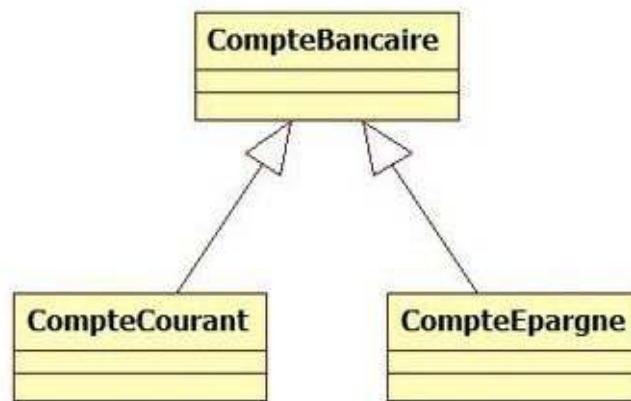
- un compte bancaire est soit un compte courant, soit un compte épargne.
- un compte courant se caractérise par le numéro de la carte bancaire qui lui est associée, ainsi que par un découvert maximal autorisé. Tout retrait qui ferait passer le nouveau solde en dessous du découvert maximal est interdit et non effectué.
- on ne peut retirer en une seule fois plus de la moitié du solde d'un compte épargne.

Création d'une classe abstraite

Notre modélisation objet du domaine doit refléter ces évolutions. Jusqu'à présent, un compte épargne hérite de toutes les caractéristiques d'un compte bancaire, et en ajoute d'autres (taux d'intérêt). Nous voyons apparaître des éléments spécifiques à un compte courant : numéro de CB, découvert maximal. Il serait maladroit d'ajouter ces attributs à la classe `CompteBancaire`, puisqu'ils seraient hérités par la classe `CompteEpargne` alors qu'ils ne la concernent pas.

La bonne solution est de placer dans la classe `CompteBancaire` les éléments communs à tous les types de comptes. Deux autres classes, `CompteCourant` et `CompteEpargne`, héritent de `CompteBancaire` afin d'intégrer ces éléments communs. Chaque classe dérivée contient

ce qui est spécifique à chaque type de compte. Le diagramme ci-dessous reflète cette modélisation.



Une instance de `CompteCourant` ou de `CompteEpargne` représente respectivement un compte courant ou un compte épargne. Ce sont des concepts concrets du domaine d'étude.

En revanche, que représenterait une instance de `CompteBancaire` ? Un compte bancaire est soit un compte courant, soit un compte épargne. Un compte bancaire en général n'a pas d'existence concrète. La classe `CompteBancaire` est une **abstraction** destinée à factoriser ce qui est commun à tous les comptes, mais pas à être instanciée.

Pour refléter cela, on définit la classe `CompteBancaire` comme étant abstraite.

```
public abstract class CompteBancaire
{
    // ...
}
```

En C# (ainsi qu'en Java et en C++), le mot-clé `abstract` permet de préciser qu'une classe est abstraite. Dans un diagramme de classe UML, le nom d'une classe abstraite est écrit en *italiques*.



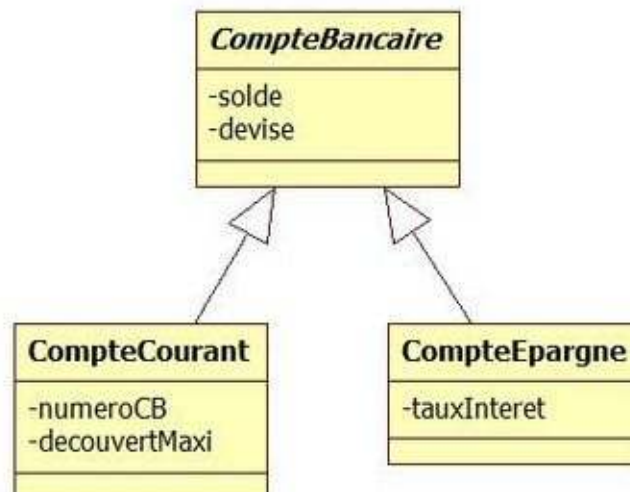
DEFINITION : une **classe abstraite** définit un concept abstrait, incomplet ou théorique. Elle rassemble des éléments **communs** à plusieurs classes dérivées. **Elle n'est pas destinée à être instanciée.**

Par opposition aux classes abstraites, les classes instanciables sont parfois appelées classes **concrètes**.

Modélisation des caractéristiques

Il est maintenant nécessaire de compléter nos classes en y ajoutant les attributs reflétant les caractéristiques (données) des éléments du domaine.

Tout compte possède un titulaire, un solde et une devise. Un compte courant se caractérise par un numéro de carte bancaire et un découvert maximal autorisé. Quant à un compte épargne, il est défini par son taux d'intérêt. On peut donc imaginer la modélisation ci-dessous pour les attributs des classes.



Modélisation du comportement

Les constructeurs de chaque classe sont simples à définir : ils doivent initialiser leurs attributs. Les constructeurs de `CompteCourant` et `CompteEpargne` feront appel à celui de `CompteBancaire` afin d'initialiser les attributs communs.

Les opérations qu'on souhaite appliquer aux comptes sont :

- le dépôt d'argent (crédit).
- le retrait d'argent (débit).
- la description du compte.

Le dépôt d'argent fonctionne de la même manière pour tous les types de comptes : le solde est simplement augmenté du montant. Ce n'est pas le cas du débit. Chaque type de compte peut être débité, mais de manière très différente :

- un compte courant autorise un découvert maximal.
- un compte épargne limite le montant du retrait par rapport au solde.

Déclaration d'une méthode abstraite

Pour traiter la problématique du débit d'argent, il faudrait pouvoir déclarer une opération de débit dans la superclasse `CompteBancaire` et laisser les classes dérivées définir comment cette opération est effectuée.

Il existe une technique pour obtenir ce résultat : la définition d'une **méthode abstraite**.

On modifie la classe `CompteBancaire` pour rendre la méthode `Debiter` abstraite, en la faisant précéder du mot-clé `abstract` .

```
public abstract class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;

    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    public double Solde
    {
        get { return solde; }
        protected set { solde = value; }
    }

    public string Devise
    {
        get { return devise; }
    }

    public string Titulaire
    {
        get { return titulaire; }
    }

    public void Crediter(double montant)
    {
        solde += montant;
    }

    // La méthode Debiter est maintenant abstraite
    public abstract void Debiter(double montant);

    public string Decrire()
    {
        return "Le solde du compte de " + titulaire + " est de " + solde + " " + devis
e;
    }
}
```

Remarquez l'absence d'accolades ouvrantes et fermantes après le nom `Debiter`, remplacées par un simple `;`. La méthode est *déclarée* mais pas *définie*. Ce sera aux classes qui héritent de `CompteBancaire` de fournir une implémentation respectant la signature de la méthode.

DEFINITIONS

- Le terme **implémenter** signifie : rendre concret, traduire en code.
- La **signature** d'une méthode est constituée de son nom et de la liste de ses paramètres.

ATTENTION : il faut bien faire la distinction entre une déclaration et une définition.

- **Déclarer** une méthode, c'est dire ce que doit faire cette méthode, sans dire comment.
- **Définir** une méthode, c'est dire comment cette méthode fait ce qu'elle doit faire, autrement dit l'implémenter.

On constate que notre classe `CompteCourant` actuelle ne compile plus : il est obligatoire d'y redéfinir la méthode `Debiter`.

```
class CompteCourant : CompteBancaire
{
    private double decouvertMaxi;
    // ...
}
```

'Exemple_5.CompteCourant' n'implémente pas le membre abstrait hérité 'Exemple_5.CompteBancaire.Debiter(double)'

Redéfinition d'une méthode abstraite

On ajoute dans la classe `CompteCourant` la méthode `Debiter` requise, avec la même signature que celle de la superclasse.

```
public class CompteCourant : CompteBancaire
{
    private string numeroCB;
    private double decouvertMaxi;

    // Constructeur
    public CompteCourant(string leTitulaire, double soldeInitial, string laDevise, string numeroCB, double decouvertMaxi)
        : base(leTitulaire, soldeInitial, laDevise) // appel au constructeur de CompteBancaire
    {
        this.numeroCB = numeroCB;
        this.decouvertMaxi = decouvertMaxi;
    }

    // Redéfinition de la méthode Debiter
    public override void Debiter(double montant)
    {
        // on n'effectue le débit que si le solde final reste supérieur au découvert
        if (Solde - montant >= decouvertMaxi)
            Solde -= montant;
    }
}
```

En C#, la redéfinition d'une méthode abstraite doit être précédée du mot-clé `override`.

Suivant le même principe, on complète la définition de la classe `CompteEpargne` pour préciser de quelle manière un compte épargne est débité.

```
public class CompteEpargne : CompteBancaire
{
    // ...

    // Redéfinition de la méthode Debiter
    public override void Debiter(double montant)
    {
        // Le montant maximal d'un retrait est la moitié du solde actuel
        if (montant <= Solde / 2)
            Solde -= montant;
    }
}
```

On peut maintenant instancier nos classes dérivées et tester leurs fonctionnalités.

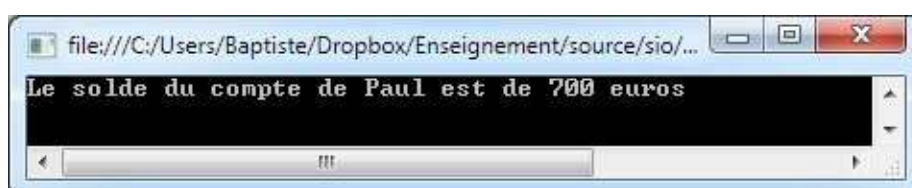
```
CompteCourant compteCourant = new CompteCourant("Pierre", 250, "dollars", "1234 5678 9
123 4567", -500);
compteCourant.Debiter(300);
compteCourant.Debiter(500);
Console.WriteLine(compteCourant.Decrire());
```



On constate que le second retrait de 500 dollars n'a pas eu lieu, puisqu'il aurait fait passer le solde en dessous du découvert maximal autorisé.

REMARQUE : le programme principal n'est pas informé de l'échec du second retrait, ce qui peut laisser croire que ce retrait a réussi. Nous découvrirons prochainement le mécanisme de remontée d'erreur qu'on utilise dans ces cas de figure.

```
CompteEpargne compteEpargne = new CompteEpargne("Paul", 1000, "euros", 0.04);
compteEpargne.Debiter(300);
compteEpargne.Debiter(500);
Console.WriteLine(compteEpargne.Decrire());
```



Ici encore, le second retrait n'a pas eu lieu : son montant est supérieur à la moitié du solde (700 euros au moment de l'appel).

Bilan

DEFINITION : une **méthode abstraite** (mot-clé `abstract`) déclare un comportement sans le définir. Elle doit être redéfinie (mot-clé `override`) dans toutes les classes dérivées.

Une classe comportant au moins une méthode abstraite est nécessairement une **classe abstraite**.

Déclarer une méthode abstraite dans une superclasse permet d'imposer à toutes les classes dérivées de fournir une implémentation de cette méthode. Ainsi, on demande à ces classes de fournir un certain comportement tout en les laissant choisir comment elles procèdent.

Méthodes virtuelles

Evolution du contexte

Intéressons-nous à la description d'un compte. Elle devrait renvoyer les données communes (solde, devise) et les données spécifiques au type (numéro de CB, découvert maximal ou taux d'intérêt), ce qui n'est pas le cas actuellement : seuls les attributs de la classe abstraite `CompteBancaire` sont affichés.

Dans ce cas de figure, on voudrait pouvoir utiliser le comportement commun (celui de `CompteBancaire`) et le compléter par un comportement particulier à chaque sous-classe. Pour cela, nous pouvons rendre la méthode `Decrire` virtuelle.

Mise en oeuvre

Modifiez la définition de `Decrire` dans `CompteBancaire` pour ajouter le mot `virtual` . Le reste de sa définition ne change pas.

```
public abstract class CompteBancaire
{
    // ...

    public virtual string Decrire()
    { // ... }
}
```

En faisant cela, on indique au compilateur que cette méthode est **virtuelle**, autrement dit susceptible d'être *redéfinie* dans une classe dérivée. C'est justement ce que nous allons faire dans `CompteCourant` et `CompteEpargne` pour intégrer à la description les données spécifiques à chaque type de compte.

```
public class CompteCourant : CompteBancaire
{
    // ...

    // Redéfinition de la méthode Decrire
    public override string Decrire()
    {
        return base.Decrire() + ". Son numéro CB est " + numeroCB +
            " et son découvert maxi est de " + decouvertMaxi + " " + Devise + ".";
    }
}

public class CompteEpargne : CompteBancaire
{
    // ...

    // Redéfinition de la méthode Decrire
    public override string Decrire()
    {
        return base.Decrire() + ". Son taux d'intérêt est de " + (tauxInteret * 100) +
            "%.";
    }
}
```

Le mot-clé `base` permet d'accéder aux membres de la classe de base depuis une méthode d'une classe dérivée. Ici, `base.Decrire()` appelle la méthode `Decrire` de `CompteBancaire`.

Nous obtenons le résultat suivant.



```
file://vmware-host/Shared Folders/Documents/Projets/GitHub/bpesquet/poo-csharp-exemples/C...
Le solde du compte de Pierre est de -50 dollars. Son numéro CB est 1234 5678 912
3 4567 et son découvert maxi est de -500 dollars.
Le solde du compte de Paul est de 700 euros. Son taux d'intérêt est de 4%.
```

A présent, chaque type de compte dispose d'une description spécifique, et la description des informations communes à tous les comptes (titulaire, solde et devise) se fait sans aucune duplication de code.

Bilan

DEFINITION : une méthode **virtuelle** (`virtual`) fournit un comportement par défaut dans une classe. Elle peut être redéfinie (`override`) dans une classe dérivée.

Grâce aux méthodes virtuelles, on pousse encore plus loin les possibilités du polymorphisme.

ATTENTION : ne pas confondre méthode virtuelle et méthode abstraite :

- Une méthode virtuelle *définit* un comportement, *éventuellement* redéfini.
- Une méthode abstraite *déclare* un comportement, *obligatoirement* redéfini.

Compléments sur l'écriture de classes

L'objectif de ce chapitre est de découvrir comment enrichir la définition de nos classes en utilisant certaines possibilités des langages à objets.

Les exemples de code associés sont [disponibles en ligne](#).

Contexte d'exemple

Nous allons utiliser la classe `CompteBancaire` ci-dessous, issue d'un précédent chapitre.


```
// Définit un compte bancaire
public class CompteBancaire
{
    private string titulaire; // Titulaire du compte
    private double solde;     // Solde du compte
    private string devise;    // Devise du compte

    public string Titulaire
    {
        get { return titulaire; }
    }

    public double Solde
    {
        get { return solde; }
    }

    public string Devise
    {
        get { return devise; }
    }

    // Constructeur
    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

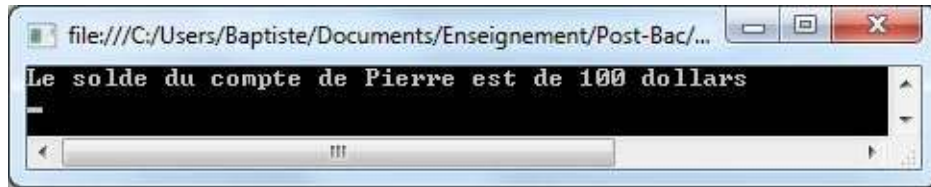
    // Ajoute un montant au compte
    public void Crediter(double montant)
    {
        solde = solde + montant;
    }

    // Retire un montant au compte
    public void Debiter(double montant)
    {
        solde = solde - montant;
    }

    // Renvoie la description du compte
    public string Decrire()
    {
        string description = "Le solde du compte de " + titulaire + " est de " + solde
+ " " + devise;
        return description;
    }
}
```

Le programme de test associé est le suivant.

```
CompteBancaire compte = new CompteBancaire("Pierre", 100, "dollars");  
Console.WriteLine(compte.Decrire());
```



Autoréférence : le mot-clé `this`

DEFINITION : à l'intérieur d'une méthode, le mot-clé `this` permet d'accéder à l'instance (l'objet) sur lequel la méthode est appelée.

L'une des utilisations fréquentes de `this` consiste à lever les **ambiguïtés de nommage** entre attributs et paramètres. Par exemple, on pourrait réécrire le constructeur de `CompteBancaire` en changeant les noms de ses paramètres de la manière suivante.

```
public CompteBancaire(string titulaire, double soldeInitial, string devise)  
{  
    this.titulaire = titulaire;  
    solde = soldeInitial;  
    this.devise = devise;  
}
```

Ici, l'expression `this.titulaire` désigne sans ambiguïté l'attribut de la classe, alors que l'expression `titulaire` désigne le paramètre du constructeur. Par contre, il n'y a pas d'ambiguïté entre l'attribut `solde` et le paramètre `soldeInitial` qui porte un nom différent.

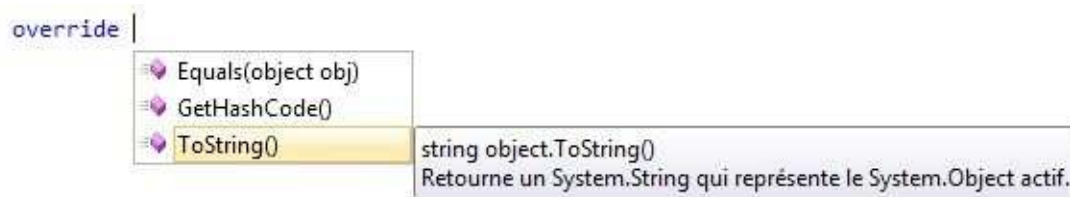
Utilisation des méthodes de base

Nous allons voir que toute classe C# dispose d'un certain nombre de méthodes de base que nous pouvons exploiter.

Découverte de la classe `Object`

Utilisons une fonctionnalité de Visual Studio pour découvrir si des méthodes sont redéfinissables dans `CompteBancaire`. En tapant le mot-clé `override` dans une classe, l'IDE nous propose la liste des méthodes de cette classe qu'il est possible de redéfinir. Etant

donné que `CompteBancaire` n'hérite a priori d'aucune classe, on s'attend à ce que cette liste soit vide.



L'IDE nous propose pourtant trois méthodes à redéfinir : `Equals` , `GetHashCode` et `ToString` . Ceci s'explique par le fait que toute classe C# qui n'hérite *explicitement* d'aucune autre hérite *implicitement* d'une classe de base nommée `object` . Ce design est inspiré de celui du langage Java.

La [documentation Microsoft](#) donne plus de précisions sur cette classe : "Il s'agit de la classe de base fondamentale parmi toutes les classes du .NET Framework. Elle constitue la racine de la hiérarchie des types."

La classe `object` dispose de plusieurs méthodes. Toute classe C# hérite directement ou indirectement de cette classe et peut utiliser ou redéfinir ses méthodes.

Redéfinition de ToString

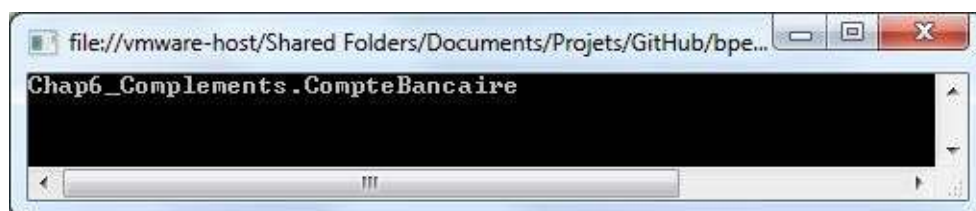
Parmi les méthodes présentes dans la classe `object` , la plus souvent redéfinie dans les classes dérivées est `ToString` .

DEFINITION : la méthode `ToString` permet de décrire un objet sous la forme d'une chaîne de caractères.

REMARQUE :

Examinons tout d'abord le comportement par défaut (sans redéfinition) de cette méthode.

```
CompteBancaire compte = new CompteBancaire("Pierre", 100, "dollars");
Console.WriteLine(compte.ToString());
```



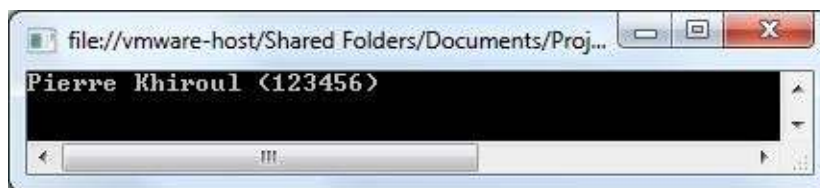
Par défaut, la méthode `ToString` affiche le nom complet de la classe. Ici, `Chap6_Complements` désigne **l'espace de noms** (*namespace*) dans lequel la classe `CompteBancaire` est définie.

```
namespace Chap6_Complements
{
    public class CompteBancaire
    {
        // ...
    }
}
```

A présent, nous allons redéfinir la méthode `ToString` pour qu'elle renvoie des informations plus utiles sur le compte.

```
public class CompteBancaire
{
    // ...

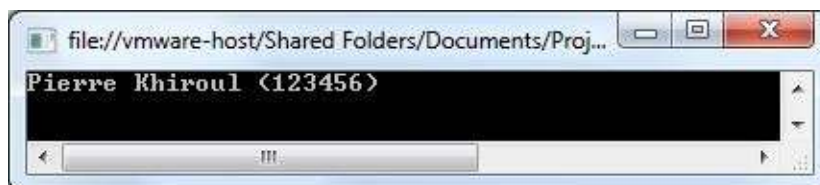
    public override string ToString()
    {
        return "Le solde du compte de " + titulaire + " est de " + solde + " " + devis
    }
}
```



La méthode `ToString` est utilisée implicitement lorsqu'un objet est affiché dans la console ou dans un contrôle graphique WinForms (liste déroulante, etc). On peut donc simplifier le programme principal en passant simplement l'objet à afficher à `Console.WriteLine`.

```
CompteBancaire compte = new CompteBancaire("Pierre", 100, "dollars");
Console.WriteLine(compte);
```

Le résultat est identique au précédent.



REMARQUE : la méthode `Decrire`, qui fait doublon avec `ToString`, peut maintenant être supprimée.

Surcharge de méthodes

Définition de plusieurs constructeurs

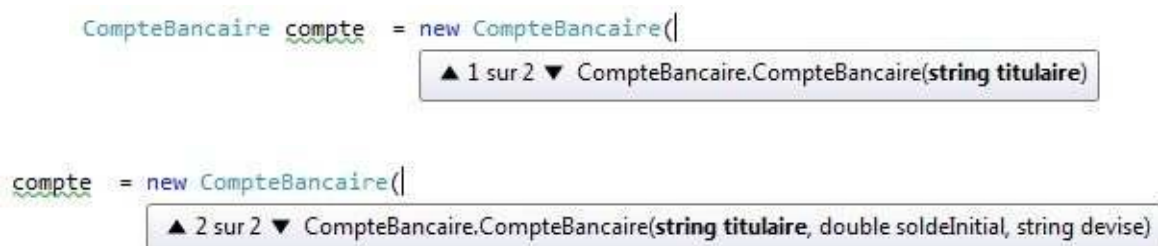
La classe `CompteBancaire` possède un constructeur qui initialise les attributs du compte créé.

```
public CompteBancaire(string titulaire, double soldeInitial, string devise)
{
    this.titulaire = titulaire;
    solde = soldeInitial;
    this.devise = devise;
}
```

Ce constructeur nous oblige à définir systématiquement le titulaire, le solde et la devise de tout nouveau compte. Or, les nouveaux comptes bancaires disposent le plus souvent d'un solde égal à zéro et utilisent l'euro comme devise. Pour refléter cet aspect du domaine et faciliter l'utilisation de notre classe, on va ajouter un autre constructeur. Il prendra uniquement le titulaire en paramètre, et initialisera solde et devise avec leurs valeurs par défaut.

```
public CompteBancaire(string titulaire)
{
    this.titulaire = titulaire;
    solde = 0;
    devise = "euros";
}
```

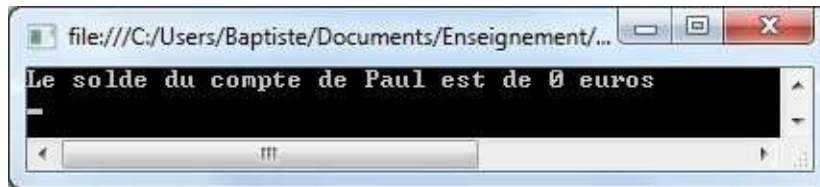
On peut à présent instancier un compte bancaire en utilisant ce constructeur. On remarque que Visual Studio nous propose deux choix d'autocomplétion, visualisables à l'aide de petites flèches noires.



The screenshot shows two instances of the `CompteBancaire` constructor being used in code. In the first instance, the autocomplete dropdown shows "▲ 1 sur 2 ▼ CompteBancaire.CompteBancaire(string titulaire)". In the second instance, the dropdown shows "▲ 2 sur 2 ▼ CompteBancaire.CompteBancaire(string titulaire, double soldeInitial, string devise)".

Cela signifie qu'il existe à présent deux constructeurs, donc deux manières d'instancier un compte. On choisit d'utiliser le nouveau constructeur.

```
CompteBancaire compte = new CompteBancaire("Paul");
Console.WriteLine(compte);
```



On constate que tous les attributs du compte ont été correctement initialisés.

Chaînage de constructeurs

Revenons à notre nouveau constructeur. Il est possible d'améliorer sa définition de la manière suivante.

```
public CompteBancaire(string titulaire) : this(titulaire, 0, "euros")
{
    // Rien à faire !
}
```

Cette variante n'initialise pas directement les attributs, mais fait appel à l'autre constructeur en lui passant en paramètres des valeurs par défaut pour les attributs que lui-même n'a pas reçus en paramètres.

CONSEIL : le chaînage des constructeurs les uns aux autres est une bonne pratique pour éviter la duplication du code d'initialisation des attributs.

Imaginons qu'on souhaite pouvoir instancier un compte bancaire en définissant son titulaire et son solde (la devise par défaut étant toujours l'euro). On va ajouter à notre classe un troisième constructeur en les chaînant les uns aux autres. Voici les trois constructeurs de

CompteBancaire .

```
public CompteBancaire(string titulaire) : this(titulaire, 0)
{}

public CompteBancaire(string titulaire, double soldeInitial)
    : this(titulaire, soldeInitial, "euros")
{}

public CompteBancaire(string titulaire, double soldeInitial, string devise)
{
    this.titulaire = titulaire;
    solde = soldeInitial;
    this.devise = devise;
}
```

En diversifiant la manière dont un objet peut être instancié, la présence de plusieurs constructeurs facilite l'utilisation d'une classe,

Surcharge d'une méthode

Le mécanisme qui consiste à définir plusieurs constructeurs avec des *signatures* différentes s'appelle la **surcharge**.

DEFINITION : la **signature** d'une méthode comprend son nom et la liste de ses paramètres (nombre, types et ordre).

On peut également appliquer ce mécanisme pour définir plusieurs méthodes avec des signatures différentes. Par exemple, on pourrait vouloir créditer (ou débiter) un compte avec un montant exprimé dans une devise différente. La solution logique est de créer une nouvelle méthode `Crediter` (ou `Debiter`) prenant en paramètres montant et devise du montant.

```
// ...

public void Crediter(double montant)
{
    solde += montant;
}

public void Crediter(double montant, string devise)
{
    if (devise == Devise)
        Crediter(montant);
    else
    {
        // TODO : gérer la conversion de devises
    }
}

// ...
```

Les deux méthodes portent le même nom, mais leurs signatures (nom + paramètres) sont différentes, d'où l'absence d'erreur de compilation. Le comportement de l'IDE lors de l'utilisation de la nouvelle méthode reflète la présence de deux signatures pour la méthode `Crediter`.

```
CompteBancaire compte = new CompteBancaire("Paul");
Console.WriteLine(compte.ToString());

compte.Crediter(|
▲ 2 sur 2 ▼ void CompteBancaire.Crediter(double montant, string devise)
```

DEFINITION : la **surcharge** est un mécanisme qui consiste à définir plusieurs variantes d'une même méthode avec des signatures différentes.

Le mécanisme de surcharge permet de proposer plusieurs variantes d'un même comportement. On le trouve fréquemment à l'oeuvre dans les classes du framework .NET. A titre d'exemple, la méthode `Console.WriteLine` totalise ci-dessous 19 signatures différentes.



ATTENTION : ne pas confondre **surcharge** et **redéfinition**.

- surcharger (*to overload*) une méthode, c'est ajouter une méthode de même nom avec une signature différente.
- Redéfinir (*to override*) une méthode, c'est la réécrire (avec la même signature) dans une classe dérivée.

Surcharge et valeur de retour

On pourrait envisager de surcharger la méthode `Crediter` afin de renvoyer le solde du compte après l'opération.

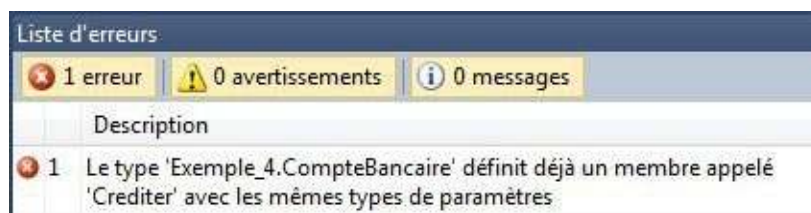
```
// ...

public void Crediter(double montant)
{
    solde += montant;
}

public double Crediter(double montant)
{
    solde += montant;
    return solde;
}

// ...
```

La définition de cette méthode provoque une erreur de compilation.



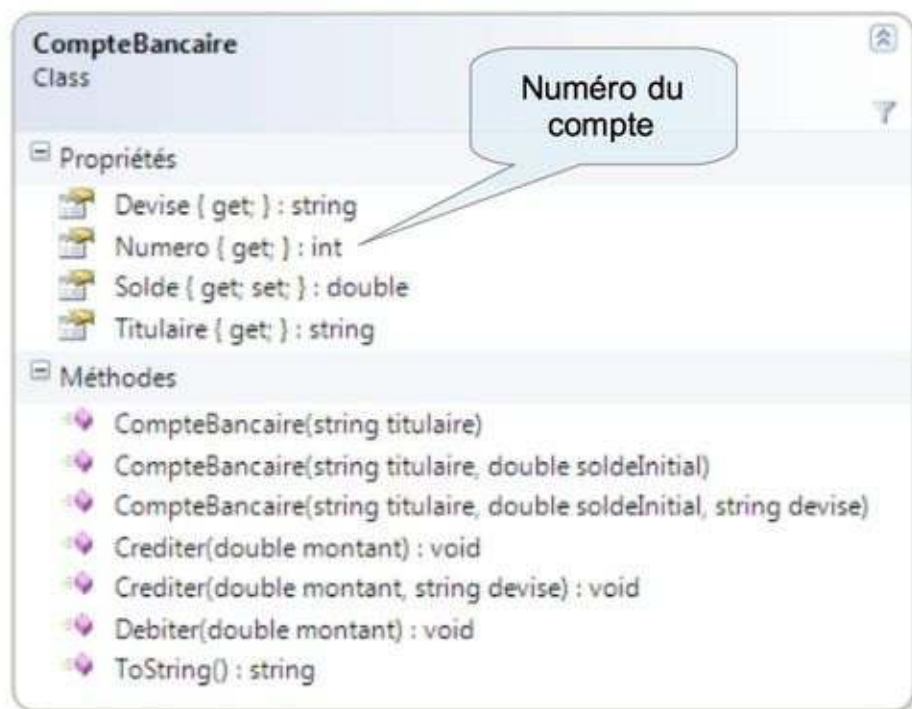
Cette erreur s'explique par le fait que la signature d'une méthode (nom + paramètres) n'inclut pas son type de retour.

ATTENTION : il est impossible de surcharger une méthode en modifiant uniquement son type de retour.

Membres de classe

Evolution des besoins

Imaginons qu'on souhaite identifier les comptes bancaires par un numéro unique. Le premier compte créé porterait le numéro 1, le second le numéro 2, etc.



Une possibilité serait de passer le numéro de compte en paramètre au constructeur. Cependant, elle serait fastidieuse à utiliser (l'utilisateur de la classe doit gérer lui-même l'unicité et l'incrémentation des numéros) et source d'erreurs (aucun contrôle et risque de doublons dans les numéros).

Une meilleure solution consiste à *internaliser* la problématique de définition du numéro à l'intérieur de la classe `CompteBancaire`. Il faudrait stocker dans la classe l'information sur le numéro de prochain compte. Lors de la création du compte suivant (c'est-à-dire lors du prochain appel au constructeur de la classe `CompteBancaire`), le numéro du prochain compte serait attribué au compte en cours de création, puis incrémenté.

Essayons de mettre en oeuvre cette stratégie.

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    private string devise;
    private int numero; // numéro du compte
    private int numeroProchainCompte = 1; // numéro du prochain compte créé

    public CompteBancaire(string titulaire, double soldeInitial, string devise)
    {
        // ... (initialisation des attributs titulaire, solde et devise)

        numero = numeroProchainCompte;
        numeroProchainCompte++;
    }

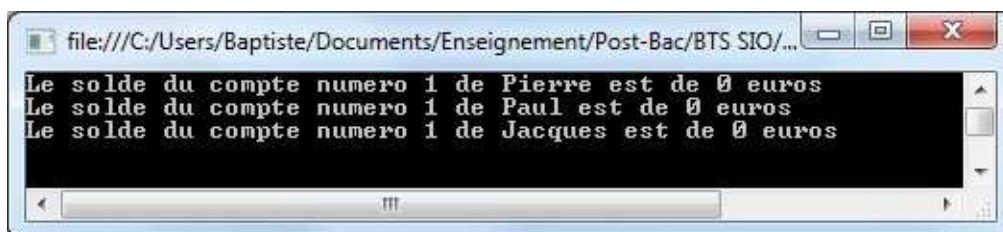
    // ...

    public override string ToString()
    {
        return "Le solde du compte numéro " + numero + " de " + titulaire + " est de "
+ solde + " " + devise;
    }
}
```

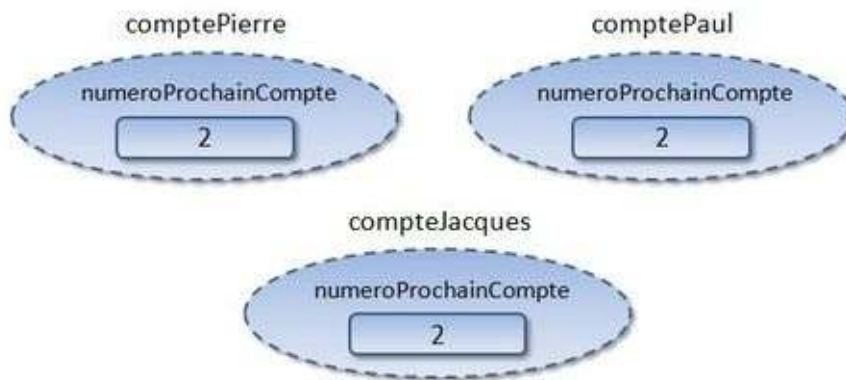
Cependant, on constate un problème lorsque l'on instancie plusieurs objets.

```
CompteBancaire comptePierre = new CompteBancaire("Pierre");
CompteBancaire comptePaul = new CompteBancaire("Paul");
CompteBancaire compteJacques = new CompteBancaire("Jacques");

Console.WriteLine(comptePierre);
Console.WriteLine(comptePaul);
Console.WriteLine(compteJacques);
```



Tous les comptes portent le même numéro ! Cela vient du fait que chaque instance de la classe `CompteBancaire` dispose de ses propres attributs. Chaque objet a donc un attribut `numeroProchainCompte` qui lui est propre, avec une valeur spécifique. L'incrémentation de cet attribut dans le constructeur, lors de l'instanciation d'un compte, n'a aucun effet sur les attributs des instances existantes. Voici les valeurs de ces attributs une fois les objets créés.



Définition d'un attribut de classe

La solution serait de lier l'attribut `numeroProchainCompte` à la classe `CompteBancaire` elle-même, et non à chacune de ses instances. Le même attribut serait partagé entre toutes les instances de la classe, ce qui permettrait d'obtenir des numéros de compte uniques.

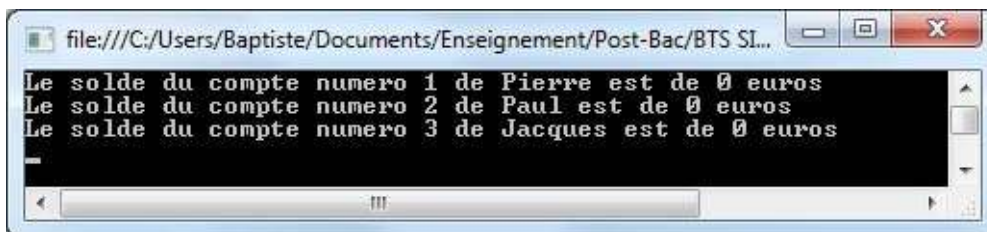
On parle **d'attributs de classe**, par opposition aux attributs appelés **attributs d'instance** quand on veut distinguer les deux types d'attributs.

En C# (ainsi qu'en Java et en C++), la création d'un attribut de classe se fait en précisant sa définition grâce au mot-clé `static`, que vous avez déjà rencontré par ailleurs.

```
public class CompteBancaire
{
    // ...
    private int numero;
    private static int numeroProchainCompte = 1; // Numéro du prochain compte créé

    // ...
}
```

Le même programme principal donne à présent un résultat différent.



Ce résultat illustre le fait que l'attribut `numeroProchainCompte` est maintenant lié à la classe et partagé entre toutes ses instances. A chaque appel du constructeur, sa valeur courante est récupérée puis incrémentée. On obtient le comportement désiré.

Définition d'une méthode de classe

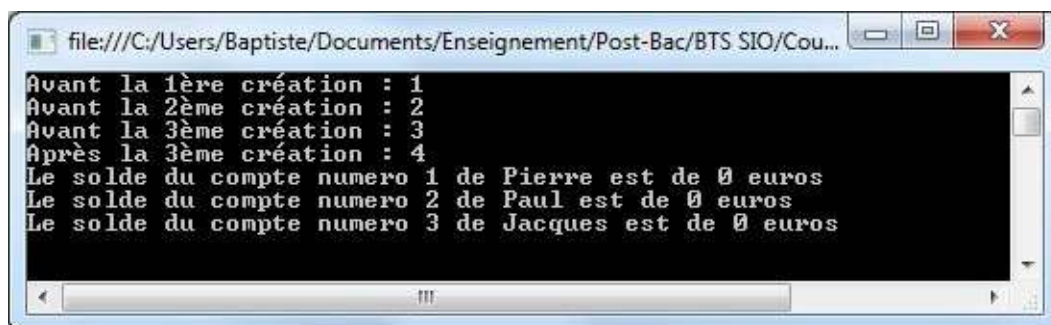
Ce qui vient d'être vu pour les attributs s'applique également aux méthodes. En utilisant le mot-clé `static`, on définit une **méthode de classe**, par opposition aux méthodes habituelles appelées **méthodes** d'instance pour les distinguer.

Imaginons qu'on souhaite pouvoir récupérer le numéro du prochain compte. Il faut ajouter une méthode à la classe `CompteBancaire`.

```
// ...  
  
public static int GetNumeroProchainCompte()  
{  
    return numeroProchainCompte;  
}  
  
// ...
```

On peut modifier le programme principal afin d'exploiter cette nouvelle méthode.

```
Console.WriteLine("Avant la 1ère création : " + CompteBancaire.GetNumeroProchainCompte()  
());  
CompteBancaire comptePierre = new CompteBancaire("Pierre");  
  
Console.WriteLine("Avant la 2ème création : " + CompteBancaire.GetNumeroProchainCompte()  
());  
CompteBancaire comptePaul = new CompteBancaire("Paul");  
  
Console.WriteLine("Avant la 3ème création : " + CompteBancaire.GetNumeroProchainCompte()  
());  
CompteBancaire compteJacques = new CompteBancaire("Jacques");  
  
Console.WriteLine("Après la 3ème création : " + CompteBancaire.GetNumeroProchainCompte()  
());  
  
Console.WriteLine(comptePierre);  
Console.WriteLine(comptePaul);  
Console.WriteLine(compteJacques);
```



Une méthode de classe s'utilise de manière différente d'une méthode d'instance. Elle peut s'utiliser en l'absence de toute instance de la classe. Etant donné qu'elle est liée à la classe et non à ses instances, on préfixe le nom de la méthode par le nom de la classe, par exemple `CompteBancaire.GetNumeroProchainCompte()` .

Une méthode de classe sert à définir un comportement indépendant de toute instance. Vous en avez déjà rencontrées certaines en utilisant les classes du framework .NET, par exemple `Console.WriteLine` OU `Convert.ToDouble` .

REMARQUE : la méthode `Main` , point d'entrée dans une application console en C#, est en réalité une **méthode de classe** de la classe `Program` .

Il existe des contraintes, somme toute logiques, concernant les interactions entre membres de classe et membres d'instance.

- une méthode d'instance **peut utiliser** un membre de classe (attribut ou méthode).
- une méthode de classe **peut utiliser** un membre de classe.
- une méthode de classe **ne peut pas utiliser** un membre d'instance (attribut ou méthode).
- une méthode de classe **ne peut pas utiliser** le mot-clé `this` .

Gestion des exceptions

L'objectif de ce chapitre est de se familiariser avec le mécanisme de gestion des exceptions.

Les exemples de code associés sont [disponibles en ligne](#).

Introduction : qu'est-ce qu'une exception ?

Commençons ce chapitre par une définition.

DEFINITION : une exception est un évènement qui apparaît pendant le déroulement d'un programme et qui empêche la poursuite normale de son exécution.

Autrement dit, une exception représente un problème qui survient dans un certain contexte : base de données inaccessible, mauvaise saisie utilisateur, fichier non trouvé... Comme son nom l'indique, une exception traduit un évènement exceptionnel et a priori imprévisible. Pour essayer de répondre à ce problème, le programmeur doit mettre en place un mécanisme de gestion des exceptions.

La suite de ce chapitre utilise le langage C#, mais presque tous les concepts étudiés se retrouvent dans d'autres langages comme Java.

Fonctionnement général des exceptions

Contexte choisi

Pour illustrer le fonctionnement des exceptions en C#, nous allons prendre exemple sur le plus célèbre des gaffeurs (© Marsu Productions).



Nous allons modéliser Gaston sous la forme d'une classe dont les méthodes reflèteront le comportement.

REMARQUE : afin de vous permettre d'identifier le moment où se produisent des exceptions , les méthodes de cet exemple affichent les messages sur la sortie standard (`Console.WriteLine`). En règle générale, cette pratique est déconseillée.

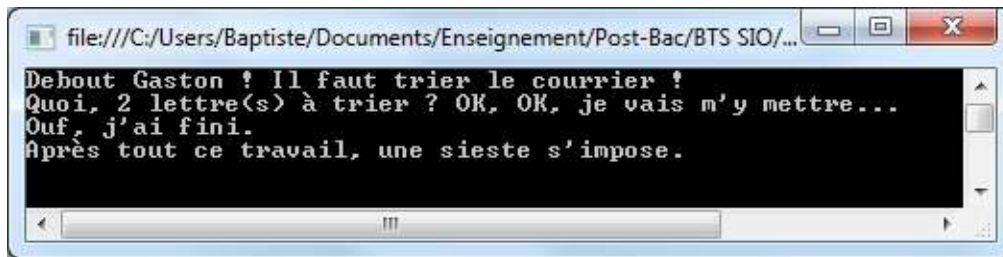
Un premier exemple

Pour commencer, ajoutons à Gaston la capacité (limitée...) de trier le courrier en retard.

```
public class GastonLagaffe
{
    public void TrierCourrierEnRetard(int nbLettres)
    {
        Console.Write("Quoi, " + nbLettres + " lettre(s) à trier ? ");
        try
        {
            Console.WriteLine("OK, OK, je vais m'y mettre...");
            if (nbLettres > 2)
            {
                throw new Exception("Beaucoup trop de lettres...");
            }
            Console.WriteLine("Ouf, j'ai fini.");
        }
        catch (Exception e)
        {
            Console.WriteLine("M'enfin ! " + e.Message);
        }
        Console.WriteLine("Après tout ce travail, une sieste s'impose.");
    }
}
```

Le programme principal demande à Gaston de trier deux lettres. Exécutons-le.

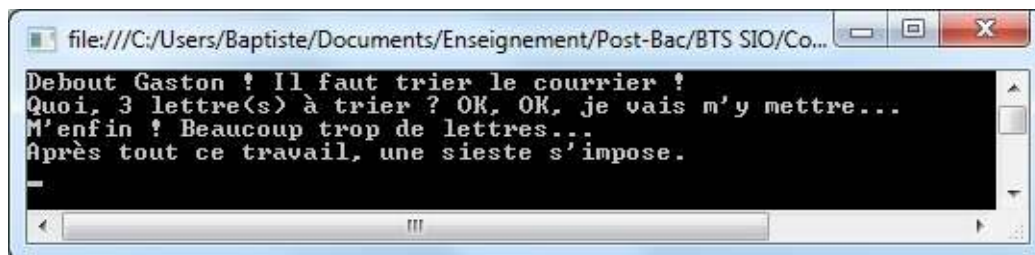
```
GastonLagaffe gaston = new GastonLagaffe();  
Console.WriteLine("Debout Gaston ! Il faut trier le courrier !");  
gaston.TrierCourrierEnRetard(2);
```



Modifions notre programme principal afin de demander à Gaston de trier trois lettres.

```
// ...  
gaston.TrierCourrierEnRetard(3);
```

Nous obtenons cette fois-ci un résultat bien différent.



Analyse de l'exemple

La capacité de travail de Gaston étant limitée, celui-ci est incapable de trier plus de deux lettres en retard. Comme le résultat de l'exécution nous le montre, le début de l'appel de la méthode `TrierCourrierEnRetard` avec trois lettres s'est déroulé normalement.

```
Console.Write("Quoi, " + nbLettres + " lettre(s) à trier ? ");  
try  
{  
    Console.WriteLine("OK, OK, je vais m'y mettre...");  
    // ...
```

Ensuite, le test ci-dessous a déclenché ce qu'on appelle la **levée** d'une exception.


```
// ...  
if (nbLettres > 2)  
{  
    throw new Exception("Beaucoup trop de lettres...");  
}  
// ...
```

On remarque que cette levée d'exception a interrompu le déroulement du reste de la méthode (absence de l'affichage "Ouf, j'ai fini"). En revanche, nous obtenons le message "M'enfin ! Beaucoup trop de lettres", qui provient du bloc de code `catch`.

```
// ...  
catch (Exception e)  
{  
    Console.WriteLine("M'enfin ! " + e.Message);  
}  
// ...
```

Cela signifie que l'exception levée a été **interceptée** dans ce bloc de code, déclenchant l'affichage du message d'erreur. Pour terminer, on observe que la dernière instruction de la méthode a malgré tout été exécutée.

```
// ...  
Console.WriteLine("Après tout ce travail, une sieste s'impose.");  
}
```

Conclusion provisoire

Nous avons découvert de nouveaux mots-clés qui permettent la gestion des exceptions en C# :

- `try` délimite un bloc de code dans lequel des exceptions peuvent se produire.
- `catch` délimite un bloc de code qui intercepte et gère les exceptions levées dans le bloc `try` associé.
- `throw` lève une nouvelle exception.

La syntaxe générale de la gestion des exceptions est la suivante.

```
try
{
    // code susceptible de lever des exceptions
}
catch (Exception e)
{
    // code de gestion de l'exception qui s'est produite dans le bloc try
}
```

Il s'agit d'une première approche très simpliste. Il nous reste de nombreux points à examiner.

Détail du fonctionnement des exceptions

Une exception est un objet

Intéressons-nous à la partie du code précédent qui lève une exception.

```
// ...
throw new Exception("Beaucoup trop de lettres...");
// ...
```

On voit que l'exception est **instanciée** comme un objet classique grâce au mot-clé `new`, puis levée (ou encore "jetée") grâce au mot-clé `throw`. On aurait pu décomposer le code ci-dessus de la manière suivante (rarement utilisée en pratique).

```
// ...
Exception exception = new Exception("Beaucoup trop de lettres...");
throw exception;
// ...
```

Analysons le contenu du bloc `catch`.

```
// ...
catch (Exception e)
{
    Console.WriteLine("M'enfin ! " + e.Message);
}
// ...
```

On observe que la variable `e` utilisée est en réalité un **objet**, instance de la classe `Exception`. On constate que cette classe dispose (entre autres) d'une propriété C# nommée `Message`, qui renvoie le message véhiculé par l'exception.

La classe `Exception` propose également des méthodes comme `ToString`, qui renvoient des informations plus détaillées sur l'erreur.

Une exception remonte la chaîne des appels

Dans l'exemple précédent, nous avons intercepté dans le programme principal une exception levée depuis une méthode. Découvrons ce qui se produit lors d'un appel de méthodes en cascade.

Pour cela, on ajoute plusieurs méthodes à la classe `GastonLagaffe`.

```
public class GastonLagaffe
{
    // ...
    public void FaireSignerContrats()
    {
        try
        {
            Console.WriteLine("Encore ces contrats ? OK, je les imprime...");
            ImprimerContrats();
            Console.WriteLine("A présent une petite signature...");
            AjouterSignature();
            Console.WriteLine("Fantasio, les contrats sont signés !");
        }
        catch (Exception e)
        {
            Console.WriteLine("M'enfin ! " + e.Message);
        }
    }

    public void AjouterSignature()
    {
        Console.WriteLine("Signez ici, M'sieur Demesmaecker.");
    }

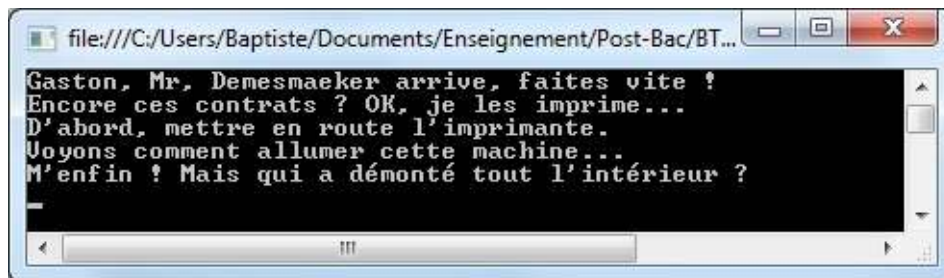
    public void ImprimerContrats()
    {
        Console.WriteLine("D'abord, mettre en route l'imprimante.");
        AllumerImprimante();
        Console.WriteLine("Voilà, c'est fait !");
    }

    public void AllumerImprimante()
    {
        Console.WriteLine("Voyons comment allumer cette machine...");
        throw new Exception("Mais qui a démonté tout l'intérieur ?");
    }
}
```

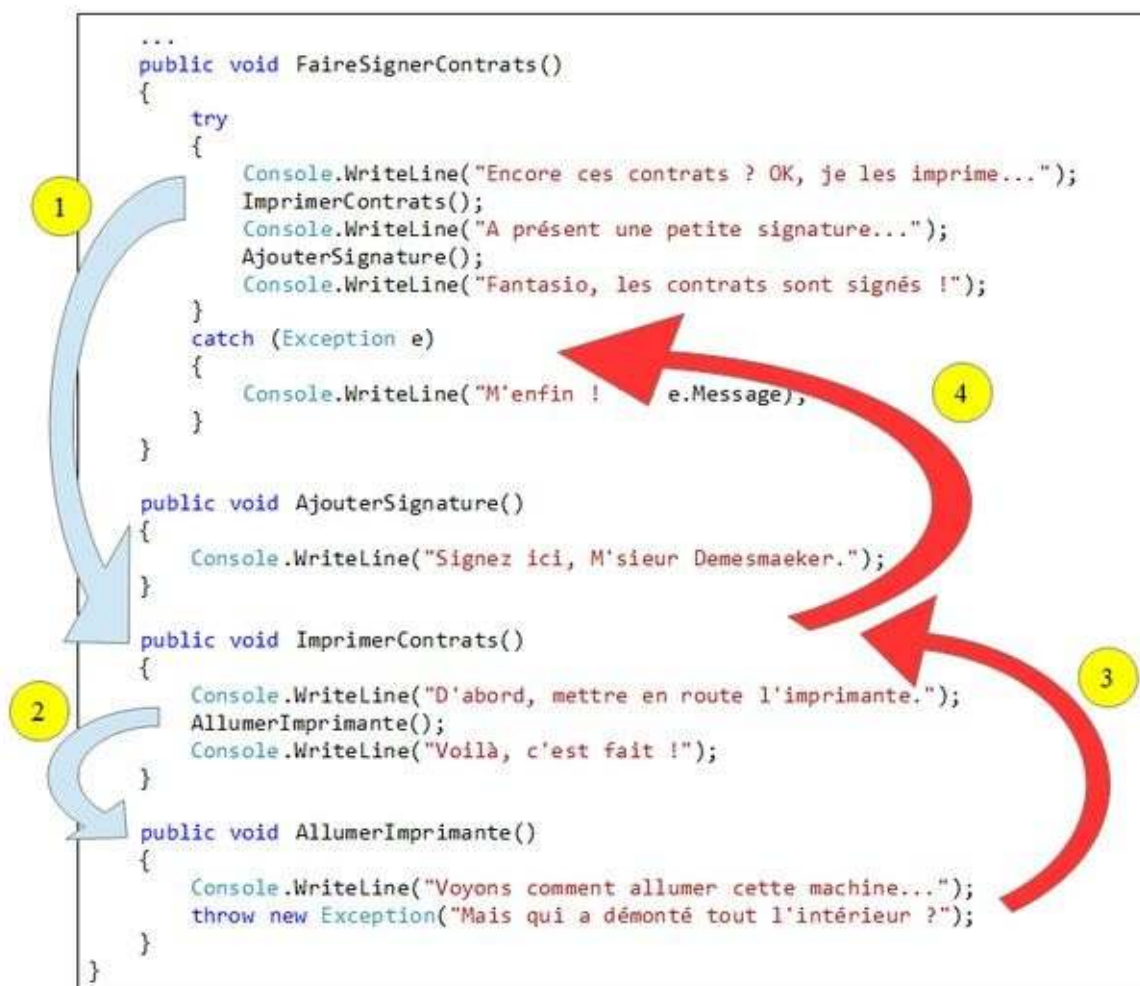
On modifie également le programme principal pour demander à Gaston de faire signer les contrats.

```
GastonLagaffe gaston = new GastonLagaffe();
Console.WriteLine("Gaston, Mr, Demesmaeker arrive, faites vite !");
gaston.FaireSignerContrats();
```

Le résultat de l'exécution est donné ci-après.



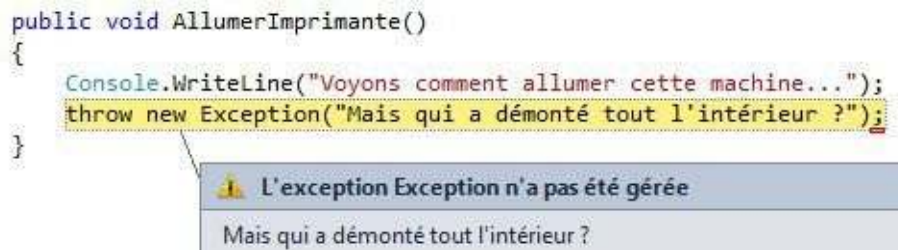
On constate que l'exception levée dans la méthode `AllumerImprimante` a été propagée par la méthode `ImprimerContrats` puis finalement interceptée dans le bloc `catch` de la méthode `FaireSignerContrats`.



Une exception levée remonte la chaîne des appels dans l'ordre inverse, jusqu'à être interceptée dans un bloc `catch`. Dans le cas où aucun gestionnaire d'exception n'est trouvé, l'exécution du programme s'arrête avec un message d'erreur. Pour le constater, on modifie le programme principal pour faire appel à une méthode qui lève une exception.

```
// ...  
gaston.AllumerImprimante();
```

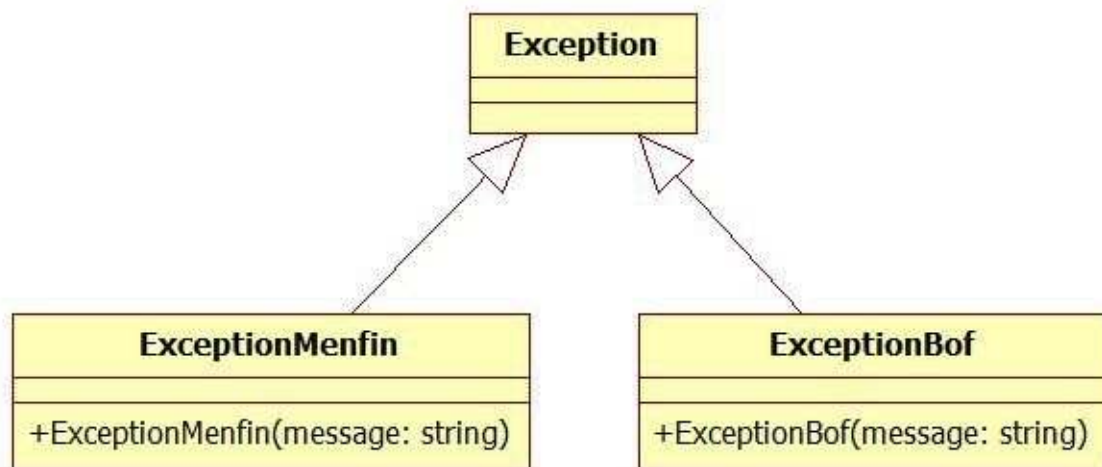
On obtient cette fois-ci une erreur à l'exécution.



DANGER ! Une exception non interceptée provoque un arrêt brutal de l'exécution d'un programme.

Les exceptions forment une hiérarchie

Jusqu'à présent, nous avons utilisé uniquement la classe C# standard `Exception` pour véhiculer nos exceptions. Il est possible de dériver cette classe afin de créer nos propres classes d'exception. Voici le diagramme de classes UML associé.



La valeur ajoutée de ces classes se limite à la modification du message d'erreur de l'exception.

```
public class ExceptionMenfin : Exception
{
    public ExceptionMenfin(string message) : base("M'enfin ! " + message)
    { }
}
```

```
public class ExceptionBof : Exception
{
    public ExceptionBof(string message) : base("Bof ! " + message)
    { }
}
```

On ajoute à notre classe `GastonLagaffe` la capacité de répondre (ou pas...) au téléphone.

```
public class GastonLagaffe
{
    // ...

    public void RepondreAuTelephone(string appellant)
    {
        if (appellant == "Mr. Boulier")
        {
            throw new ExceptionMenfin("Je finis un puzzle.");
        }
        else if (appellant == "Prunelle")
        {
            throw new ExceptionBof("Pas le temps, je suis dé-bor-dé !");
        }
        else
        {
            Console.WriteLine("Allô, ici Gaston, j'écoute...");
        }
    }
}
```

On ajoute au programme principal un sous-programme qui appelle cette méthode et intercepte les éventuelles exceptions.

```
// ...
private static void GererAppel(GastonLagaffe gaston, string appellant)
{
    Console.WriteLine("Gaston, " + appellant + " au téléphone !");
    try
    {
        gaston.RepondreAuTelephone(appellant);
    }
    catch (ExceptionMenfin e)
    {
        Console.WriteLine("Pas de réponse... Et pourquoi ?");
        Console.WriteLine(e.Message);
    }
    catch (ExceptionBof e)
    {
        Console.WriteLine("Ca sonne toujours... vous dormez ou quoi ?");
        Console.WriteLine(e.Message);
    }
    Console.WriteLine();
}
```

Enfin, le programme principal appelle ce sous-programme plusieurs fois.

```
GastonLagaffe gaston = new GastonLagaffe();

GererAppel(gaston, "Mr. Boulrier");
GererAppel(gaston, "Prunelle");
GererAppel(gaston, "Jules-de-chez-Smith");
```

Le résultat de l'exécution est présenté ci-dessous.

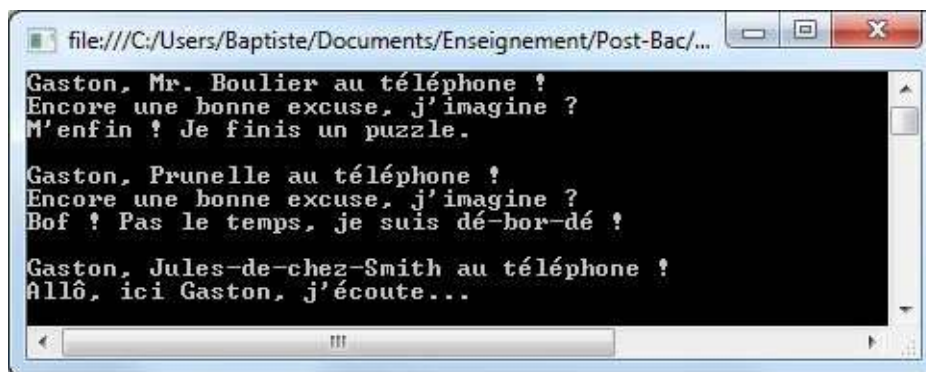


On constate que l'exception de type `ExceptionMenfin` a été interceptée dans le premier bloc `catch`, et l'exception de type `ExceptionBof` a été interceptée dans le second.

Comme nos exceptions héritent toutes deux de la classe `Exception`, il est possible de simplifier l'appel en interceptant uniquement `Exception`.

```
// ...
private static void GererAppel(GastonLagaffe gaston, string appellant)
{
    Console.WriteLine("Gaston, " + appellant + " au téléphone !");
    try
    {
        gaston.RepondreAuTelephone(appellant);
    }
    catch (Exception e) // intercepte toute exception
    {
        Console.WriteLine("Encore une bonne excuse, j'imagine ?");
        Console.WriteLine(e.Message);
    }
    Console.WriteLine();
}
```

Le résultat de l'exécution est maintenant le suivant.



Afin de limiter le nombre de blocs `catch`, on peut donc intercepter une superclasse commune à plusieurs exceptions. C'est particulièrement intéressant dans le cas où le traitement de l'erreur est le même dans tous les `catch`. En revanche, intercepter une superclasse plutôt que les classes dérivées fait perdre l'information sur le type exact de l'exception. Dans l'exemple précédent, on ne sait plus si l'exception interceptée est une `ExceptionMenfin` ou une `ExceptionBof`. Quand le traitement doit être différencié selon le type de l'erreur rencontrée, il faut donc ajouter autant de blocs `catch` que de types d'exception.

Avantages apportés par les exceptions

Il existe d'autres solutions que l'utilisation des exceptions pour gérer les erreurs qui peuvent se produire durant l'exécution d'un programme. Cependant, les exceptions apportent de nombreux avantages. Le principal est qu'elles permettent de regrouper et de différencier le code de gestion des erreurs du code applicatif.

Sans utiliser d'exceptions, on est obligé de tester la réussite de chaque opération au fur et à mesure du déroulement. Par exemple, une méthode d'écriture dans un fichier pourrait s'écrire comme ci-dessous.

```
public void EcrireDansFichier(string texte)
{
    <ouverture du fichier en écriture>
    if <ouverture en écriture ok>
    {
        <écriture du texte dans le fichier>
        if <écriture du texte ok>
        {
            <fermeture du fichier>
            if <fermeture ok>
            {
                <affichage message : succès>
            }
            else
            {
                <affichage message : erreur de fermeture>
            }
        }
        else
        {
            <affichage message : erreur d'écriture>
        }
    }
    else
    {
        <affichage message : impossible d'ouvrir le fichier>
    }
}
```

Réécrit en utilisant des exceptions, ce code pourrait devenir :

```
public void EcrireDansFichier(string texte)
{
    try
    {
        <ouverture du fichier en écriture>
        <écriture du texte dans le fichier>
        <fermeture du fichier>
        <affichage message : succès>
    }
    catch (<exception de type ouverture impossible>)
    {
        <affichage message : impossible d'ouvrir le fichier>
    }
    catch (<exception de type écriture impossible>)
    {
        <affichage message : erreur d'écriture>
    }
    catch (<exception de type fermeture impossible>)
    {
        <affichage message : erreur de fermeture>
    }
}
```

Cette solution sépare le code applicatif du code de gestion des erreurs, regroupé dans les `catch`. Si la gestion des erreurs est la même dans tous les blocs `catch`, on peut même simplifier encore le code précédent.

```
public void EcrireDansFichier(string texte)
{
    try
    {
        <ouverture du fichier en écriture>
        <écriture du texte dans le fichier>
        <fermeture du fichier>
        <affichage message : succès>
    }
    catch (<exception la plus générale>)
    {
        <affichage message véhiculé par l'exception>
    }
}
```

Bonnes pratiques dans la gestion des exceptions

Comme nous l'avons vu, les exceptions constituent une nette amélioration pour gérer les erreurs, à condition de savoir les employer correctement.

Savoir quand lever une exception

La première règle, et probablement la plus importante, est que l'usage des exceptions doit rester réservé aux cas exceptionnels (comme leur nom l'indique).

IMPORTANT : une méthode ne lève une exception qu'en dernier recours pour signaler qu'elle est incapable de continuer à s'exécuter normalement.

Par exemple, on pourrait avoir la mauvaise idée d'employer les exceptions comme ci-dessous, pour sortir d'une boucle.

```
i = 0;
trouve = false;
while (!trouve)
{
    i++;
    if (i == 10)
        throw new Exception("Fin de la boucle");
    else
        // ...
}
```

Ecrire ce genre de code est très maladroit pour plusieurs raisons :

- il ne correspond pas à la philosophie des exceptions, à savoir la gestion des erreurs,
- il rend le code moins lisible et son exécution difficile à prévoir,
- il est beaucoup plus lent : lever une exception est un processus coûteux en temps d'exécution.

Savoir quand intercepter une exception

Une exception signale une erreur rencontrée durant l'exécution et remonte la chaîne des appels de méthode jusqu'à son interception (ou le "plantage" du programme). Il faut bien réfléchir avant de lancer une exception depuis une méthode, et également avant de l'intercepter dans une autre méthode.

IMPORTANT : une méthode n'intercepte une exception que si elle est capable d'effectuer un traitement approprié (message d'erreur, nouvelle tentative, etc).

Dans le cas contraire, il vaut mieux laisser l'exception se propager plus haut dans la chaîne des appels. Il ne sert strictement à rien d'intercepter une exception sans savoir quoi en faire, comme par exemple dans le code ci-dessous.

```
try
{
    // code susceptible de lever des exceptions
}
catch (Exception e) // ce bloc catch est parfaitement inutile !
{
    throw e;
}
```

Cet exemple intercepte les exceptions de la classe `Exception` (et de toutes les classes dérivées), mais se contente de relancer l'exception interceptée sans faire aucun traitement d'erreur. Il est plus lisible et plus efficace de supprimer le `try` ... `catch` autour des instructions du bloc de code.

Autre mauvaise idée : écrire un bloc `catch` vide.

```
try
{
    // code susceptible de lever des exceptions
}
catch (Exception e)
{
    // l'exception est avalée par ce bloc catch !
}
```

Dans ce cas, l'exception est interceptée silencieusement, ou encore **avalée**, par ce bloc. Elle ne remonte plus la chaîne des appels et l'appelant n'aura aucun moyen de détecter l'erreur qui s'est produite. Plutôt que d'écrire un bloc `catch` vide, mieux vaut laisser l'exception se propager aux méthodes appelantes qui sauront mieux gérer l'erreur.

Savoir quand créer ses propres classes d'exception

Dans un paragraphe précédent, nous avons créé deux nouvelles classes d'exception pour illustrer certains concepts : `ExceptionMenfin` et `ExceptionBof`. En pratique, un programmeur doit bien réfléchir avant de créer sa ou ses propre(s) classe(s) d'exception.

En règle générale, on ne crée de nouvelles classes d'exception que dans les cas suivants :

- on souhaite distinguer ses propres exceptions des exceptions standards.
- on a besoin de véhiculer dans les exceptions des données spécifiques au programme.

Dans la plupart des situations courantes et pour des projets ne dépassant pas une certaine complexité, l'utilisation de la classe standard `Exception` suffit.

REMARQUE : dans le cas de la création d'une classe dérivée de `Exception`, il est fortement conseillé d'inclure le mot *Exception* dans le nom de la nouvelle classe.