

# Un langage très orienté objet



- 
- **Classes, objets et méthodes**
  - **Polymorphisme d'objets**
  - **Polymorphisme de méthodes**
  - **Polymorphisme d'interfaces**
  - **Classe de délégation**
  - **Traitement d'exceptions**
  - **Processus et multi-threading**

# Classes, objets et méthodes



## Plan général:

### 1. Les classes C# : des nouveaux types

- 1.1 Déclaration d'une classe
- 1.2 Une classe est un type en C#
- 1.3 Toutes les classes ont le même ancêtre - héritage
- 1.4 Encapsulation des classes
- 1.5 Exemple de classe imbriquée dans une autre classe
- 1.6 Exemple de classe incluse dans un même espace de noms
- 1.7 Méthodes abstraites
- 1.8 Classe abstraite, Interface

### 2. Les objets : des références ou des valeurs

- 2.1 Modèle de la référence
- 2.2 Les constructeurs d'objets référence ou valeurs
- 2.3 Utilisation du constructeur d'objet par défaut
- 2.4 Utilisation d'un constructeur d'objet personnalisé
- 2.5 Le mot clef this- cas de la référence seulement

### 3. Variables et méthodes

- 3.1 Variables dans une classe en général
- 3.2 Variables et méthodes d'instance
- 3.3 Variables et méthodes de classe - static
- 3.4 Bilan et exemple d'utilisation

## Introduction

Nous proposons des comparaisons entre les syntaxes de C# et Delphi et/ou Java, lorsque les définitions sont semblables.

### Tableau des limitations des niveaux de visibilité fourni par microsoft :

Contexte	Notes
Classes	La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même.
Interfaces	Les interfaces de base explicites d'un type d'interface doivent être au moins aussi accessibles que le type d'interface lui-même.
Délégués	Le type de retour et les types de paramètres d'un type délégué doivent être au moins aussi accessibles que le type délégué lui-même.
Constantes	Le type d'une constante doit être au moins aussi accessible que la constante elle-même.
Champs	Le type d'un champ doit être au moins aussi accessible que le champ lui-même.
Méthodes	Le type de retour et les types de paramètres d'une méthode doivent être au moins aussi accessibles que la méthode elle-même.
Propriétés	Le type d'une propriété doit être au moins aussi accessible que la propriété elle-même.
Événements	Le type d'un événement doit être au moins aussi accessible que l'événement lui-même.
Indexeurs	Le type et les types de paramètres d'un indexeur doivent être au moins aussi accessibles que l'indexeur lui-même.
Opérateurs	Le type de retour et les types de paramètres d'un opérateur doivent être au moins aussi accessibles que l'opérateur lui-même.
Constructeurs	Les types de paramètres d'un constructeur doivent être au moins aussi accessibles que le constructeur lui-même.

### Modification de visibilité

Rappelons les classiques modificateurs de visibilité des **variables** et des **méthodes** dans les langages orientés objets, dont C# dispose :

#### Les mots clef (modularité public-privé)

<b>par défaut</b> (aucun mot clef)	Les variables et les méthodes d'une classe non précédées d'un mot clef sont <b>private</b> et ne sont visibles que dans la classe seulement.
<b>public</b>	Les variables et les méthodes d'une classe précédées du mot clef <b>public</b> sont visibles par toutes les classes de tous les modules.
<b>private</b>	Les variables et les méthodes d'une classe précédées du mot clef <b>private</b> ne sont visibles que dans la classe seulement.
<b>protected</b>	Les variables et les méthodes d'une classe précédées du mot clef <b>protected</b> sont visibles par toutes les classes dérivées de cette classe.
<b>internal</b>	Les variables et les méthodes d'une classe précédées du mot clef <b>internal</b> sont visibles par toutes les classes

	inclus dans le même assembly.
--	-------------------------------

Les attributs d'accessibilité **public**, **private**, **protected** sont identiques à ceux de Delphi et Java, pour les classes nous donnons ci-dessous des informations sur leur utilisation.

L'attribut **internal** joue à peu près le rôle (au niveau de l'assembly) des classes Java déclarées sans mot clef dans le même package, ou des classes Delphi déclarées dans la même unit (classes amies). Toutefois pour des raisons de sécurité C# ne possède pas la notion de classe amie.

## 1. Les classes : des nouveaux types

Rappelons un point fondamental déjà indiqué : tout programme C# contient une ou plusieurs classes précédées ou non d'une déclaration d'utilisation d'autres classes contenues dans des bibliothèques (clause **using**) ou dans un package complet composé de nombreuses classes.

La notion de module en C# est représentée par l'espace de noms (clause **namespace**) semblable au package Java, en C# vous pouvez omettre de spécifier un namespace, par défaut les classes déclarées le sont automatiquement dans un espace 'sans nom' (généralement qualifié de global) et tout identificateur de classe déclaré dans cet espace global sans nom est disponible pour être utilisé dans un espace de noms nommé.

Contrairement à Java, en C# les classes non qualifiées par un modificateur de visibilité (déclarées sans rien devant) sont **internal**.

Delphi	Java	C#
<b>Unit</b> Biblio; <b>interface</b>  <i>// les déclarations des classes</i>  <b>implementation</b>  <i>// les implémentations des classes</i>  <b>end.</b>	<b>package</b> Biblio;  <i>// les déclarations et implémentation des classes</i>  si pas de nom de package alors automatiquement dans : <b>package</b> java.lang;	<b>namespace</b> Biblio {  <i>// les déclarations et implémentation des classes</i>  } si pas de nom d'espace de noms alors automatiquement dans l'espace global.

### 1.1 Déclaration d'une classe

En C#, nous n'avons pas comme en Delphi, une partie déclaration de la classe et une partie implémentation séparées l'une de l'autre. La classe avec ses attributs et ses méthodes sont déclarés et implémentés à un seul endroit comme en Java.

Delphi	Java	C#
<pre> <b>interface</b> <b>uses</b> biblio; <b>type</b> Exemple = <b>class</b>   x : real;   y : integer;   <b>function</b> F1(a,b:integer): real;   <b>procedure</b> P2; <b>end</b>;  <b>implementation</b> <b>function</b> F1(a,b:integer): real; <b>begin</b>   .....code de F1 <b>end</b>; <b>procedure</b> P2; <b>begin</b>   .....code de P2 <b>end</b>; <b>end.</b> </pre>	<pre> <b>import</b> biblio;  <b>class</b> Exemple {   <b>float</b> x;   <b>int</b> y;   <b>float</b> F1(<b>int</b> a, <b>int</b> b)   {     .....code de F1   }    <b>void</b> P2()   {     .....code de P2   } } </pre>	<pre> <b>using</b> biblio; <b>namespace</b> Machin {   <b>class</b> Exemple   {     <b>float</b> x;     <b>int</b> y;     <b>float</b> F1(<b>int</b> a, <b>int</b> b)     {       .....code de F1     }      <b>void</b> P2()     {       .....code de P2     }   } } </pre>

### 1.2 Une classe est un type en C#

Comme en Delphi et Java, une classe C# peut être considérée comme un nouveau type dans le programme et donc des variables d'objets peuvent être déclarées selon ce nouveau "type".

*Une déclaration de programme comprenant 3 classes :*

Delphi	Java	C#
<pre> <b>interface</b> <b>type</b> <b>Un</b> = <b>class</b>   ... <b>end</b>; <b>Deux</b> = <b>class</b>   ... <b>end</b>; Appli3Classes = <b>class</b>   x : <b>Un</b>;   y : <b>Deux</b>;   <b>public</b>   <b>procedure</b> main; <b>end</b>;  <b>implementation</b> <b>procedure</b> Appli3Classes.main; <b>var</b>   x : <b>Un</b>;   y : <b>Deux</b>; <b>begin</b>   ... <b>end</b>; <b>end.</b> </pre>	<pre> <b>class</b> Appli3Classes {   <b>Un</b> x;   <b>Deux</b> y;   <b>public static void</b> main(String [ ] arg)   {     <b>Un</b> x;     <b>Deux</b> y;     ...   } } <b>class</b> <b>Un</b> { ... } <b>class</b> <b>Deux</b> { ... } </pre>	<pre> <b>class</b> Appli3Classes {   <b>Un</b> x;   <b>Deux</b> y;   <b>static void</b> Main(String [ ] arg)   {     <b>Un</b> x;     <b>Deux</b> y;     ...   } } <b>class</b> <b>Un</b> { ... } <b>class</b> <b>Deux</b> { ... } </pre>

### 1.3 Toutes les classes ont le même ancêtre - héritage

Comme en Delphi et en Java, toutes les classes C# dérivent automatiquement d'une seule et même classe ancêtre : la classe **Object**. En C# le mot-clef pour indiquer la dérivation (héritage) à partir d'une autre classe est le symbole deux points ':', lorsqu'il est omis c'est donc que la classe hérite automatiquement de la classe **Object** :

*Les deux déclarations de classe ci-dessous sont équivalentes :*

Delphi	Java	C#
<pre> <b>type</b> Exemple = <b>class</b> ( TObject ) ..... <b>end;</b> </pre>	<pre> <b>class</b> Exemple <b>extends</b> Object { ..... } </pre>	<pre> <b>class</b> Exemple : Object { ..... } </pre>
<pre> <b>type</b> Exemple = <b>class</b> ..... <b>end;</b> </pre>	<pre> <b>class</b> Exemple { ..... } </pre>	<pre> <b>class</b> Exemple { ..... } </pre>

L'héritage en C# est tout à fait classiquement de l'**héritage simple** comme en Delphi et en Java. Une classe fille qui dérive d'une seule classe mère, hérite de sa classe mère toutes ses méthodes et tous ses champs. En C# la syntaxe de l'héritage fait intervenir le symbole clef ':', comme dans "**class** Exemple : Object".

Une déclaration du type :

```

class ClasseFille : ClasseMere {
}

```

signifie que la classe ClasseFille dispose de tous les attributs et de toutes les méthodes de la classe ClasseMere.

*Comparaison héritage :*

Delphi	Java	C#
<pre> <b>type</b> ClasseMere = <b>class</b> // champs de ClasseMere // méthodes de ClasseMere <b>end;</b>  ClasseFille = <b>class</b> ( ClasseMere ) // hérite des champs de ClasseMere // hérite des méthodes de ClasseMere <b>end;</b> </pre>	<pre> <b>class</b> ClasseMere { // champs de ClasseMere // méthodes de ClasseMere }  <b>class</b> ClasseFille <b>extends</b> ClasseMere { // hérite des champs de ClasseMere // hérite des méthodes de ClasseMere } </pre>	<pre> <b>class</b> ClasseMere { // champs de ClasseMere // méthodes de ClasseMere }  <b>class</b> ClasseFille : ClasseMere { // hérite des champs de ClasseMere // hérite des méthodes de ClasseMere } </pre>

Bien entendu une classe fille peut définir de nouveaux champs et de nouvelles méthodes qui lui sont propres.

## 1.4 Encapsulation des classes

La visibilité et la protection des classes en Delphi est apportée par le module **Unit** où toutes les classes sont visibles dans le module en entier et dès que la unit est utilisée les classes sont visibles partout. Il n'y a pas de possibilité d'imbriquer une classe dans une autre.

En C#, nous avons la possibilité d'*imbriquer* des classes dans d'autres classes (classes internes), par conséquent la *visibilité de bloc s'applique aussi aux classes*.

### Remarque

La notion de classe interne de C# (qui n'existe pas en Delphi) est sensiblement différente de celle de Java (pas de classe locale et pas de classe anonyme mais des méthodes anonymes), elle correspond à la notion de **classe membre** de Java.

### Mots clefs pour la protection des classes et leur visibilité :

- Une classe C# peut se voir attribuer un modificateur de comportement sous la forme d'un mot clef devant la déclaration de classe. Par défaut si aucun mot clef n'est indiqué la classe est visible dans tout le namespace dans lequel elle est définie. Il y a 4 qualificatifs possibles pour modifier le comportement de visibilité d'une classe selon sa position (imbriquée ou non) : **public**, **private**, **protected**, **internal** (dénommés modificateurs d'accès) et **abstract** (qualificateur d'abstraction pouvant être associé à l'un des 3 autres modificateurs d'accès). On rappelle que sans qualificatif **public**, **private**, **internal** ou **protected**, une classe C# est automatiquement **internal**.
- Le nom du fichier source dans lequel plusieurs classes C# sont stockées n'a aucun rapport avec le nom d'une des classes déclarées dans le texte source, il est laissé au libre choix du développeur et peut éventuellement être celui d'une classe du namespace etc...

### Tableau des possibilités fourni par microsoft :

Membres de	Accessibilité des membres par défaut	Accessibilité déclarée autorisée du membre
<b>enum</b>	<b>public</b>	Aucune
<b>class</b>	<b>private</b>	<b>public</b> <b>protected</b> <b>internal</b> <b>private</b> <b>protected internal</b>
<b>interface</b>	<b>public</b>	Aucune
<b>struct</b>	<b>private</b>	<b>public</b> <b>internal</b> <b>private</b>

### Attention

Par défaut dans une classe tous les membres sans qualificateur de visibilité (classes internes incluses) sont **private**.

C#	Explication
mot clef <b>abstract</b> : <b>abstract class</b> ApplicationClasse1 { ... }	classe abstraite <b>non instanciable</b> . Aucun objet ne peut être créé.
mot clef <b>public</b> : <b>public class</b> ApplicationClasse2 { ... }	classe visible <b>par n'importe quel</b> programme d'un autre namespace
mot clef <b>protected</b> : <b>protected class</b> ApplicationClasse3 { ... } }	classe visible <b>seulement</b> par toutes les autres classes héritant de la classe conteneur de cette classe.
mot clef <b>internal</b> : <b>internal class</b> ApplicationClasse4 { ... }	classe visible <b>seulement</b> par toutes les autres classes du même assembly.
mot clef <b>private</b> : <b>private class</b> ApplicationClasse5 { ... }	classe visible <b>seulement</b> par toutes les autres classes du même namespace où elle est définie.
pas de mot clef : <b>class</b> ApplicationClasse6 { ... } - sauf si c'est une classe interne	qualifiée <b>internal</b> -si c'est une classe interne elle est alors qualifiée <b>private</b> .

Nous remarquons donc qu'une classe dès qu'elle est déclarée dans l'espace de noms est toujours visible dans tout l'assembly, que le mot clef **internal** soit présent ou non. Les mots clefs **abstract** et **protected** n'ont de l'influence que pour l'héritage.

### Remarque

La notion de classe **sealed** en C# correspond strictement à la notion de classe **final** de Java : ce sont des **classes non héritables**.

Nous étudions ci-après la visibilité des classes précédentes dans deux contextes différents.

### 1.5 Exemple de classes imbriquées dans une autre classe

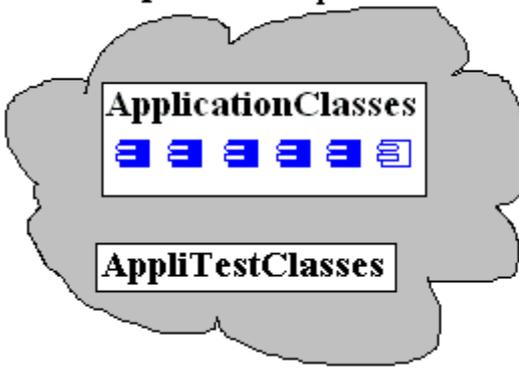
Dans le premier contexte, ces six classes sont utilisées en étant **intégrées** (imbriquées) à une classe publique.

*La classe* ApplicationClasses :

C#	Explication
----	-------------

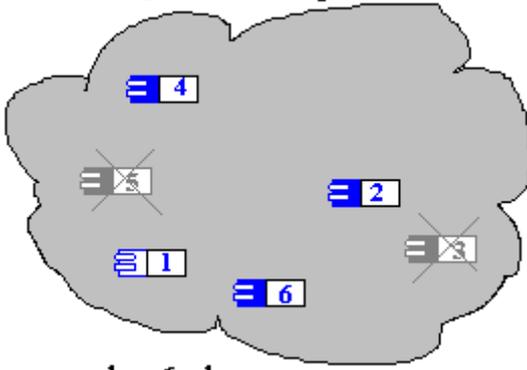
<p><b>namespace Exemple</b></p>  <p><b>namespace Exemple</b>  {  <b>public class</b> ApplicationClasses {  <b>abstract class</b> ApplicationClasse1 { ... }  <b>public class</b> ApplicationClasse2 { ... }  <b>protected class</b> ApplicationClasse3 { ... }  <b>internal class</b> ApplicationClasse4 { ... }  <b>private class</b> ApplicationClasse5 { ... }  <b>class</b> ApplicationClasse6 { ... }  }  }</p>	<p>Ces 6 "sous-classes" sont visibles ou non, à partir de l'accès à la classe englobante "ApplicationClasses", elles peuvent donc être utilisées dans tout programme qui utilise la classe "ApplicationClasses".</p> <p>Par défaut la classe ApplicationClasse6 est ici <b>private</b>.</p>
---	---

*Un programme utilisant la classe ApplicationClasses :*

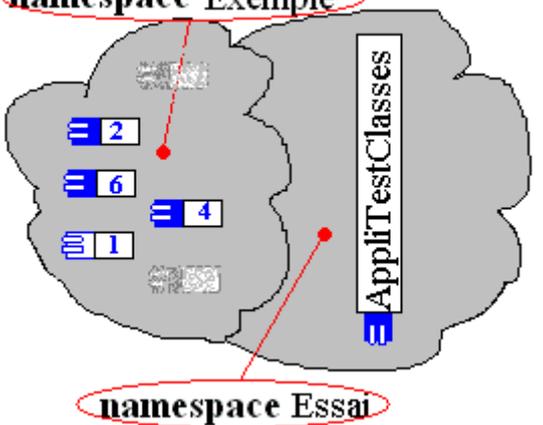
C#	Explication
<p><b>namespace Exemple</b></p>  <p><b>namespace Exemple</b>  {  <b>class</b> AppliTestClasses  {  ApplicationClasses.ApplicationClasse2 a2;  }  }</p>	<p>Le programme de gauche "class AppliTestClasses" <i>utilise</i> la classe précédente ApplicationClasses et ses sous-classes. La notation uniforme de chemin de classe est standard.</p> <p>Seule la classe interne ApplicationClasse2 est visible et permet d'instancier un objet.</p>

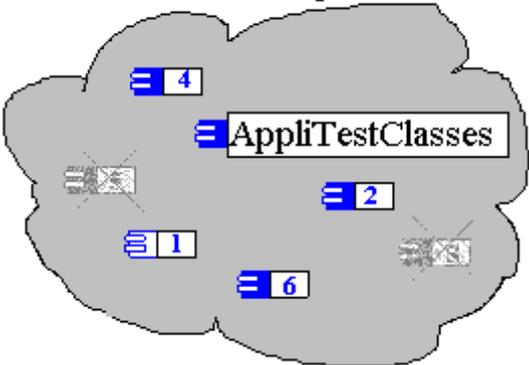
**1.6 Même exemple de classes non imbriquées situées dans le même espace de noms**

Dans ce second exemple, ces mêmes 6 classes sont utilisées en étant **includées** dans le même namespace.

C#	Explication
<p><b>namespace Exemple</b></p>  <p>les 6 classes</p> <pre> namespace Exemple {     abstract class ApplicationClasse1 { ... }     public class ApplicationClasse2 { ... }     protected class ApplicationClasse3 { ... }     internal class ApplicationClasse4 { ... }     private class ApplicationClasse5 { ... }     class ApplicationClasse6 { ... } } </pre>	<p>Une classe dans un espace de nom ne peut pas être qualifiée <b>protected</b> ou <b>private</b> (si elle est imbriquée comme ci-haut cela est possible):</p> <p><i>Les classes ApplicationClasse3 et ApplicationClasse5 ne peuvent donc pas faire partie du même namespace Exemple.</i></p> <p>La classe ApplicationClasse1 est ici abstraite et <b>public</b>, donc visible.</p> <p>La classe ApplicationClasse2 est <b>public</b>, donc visible.</p> <p>La classe ApplicationClasse4 n'est visible que dans le même assembly.</p> <p>Par défaut la classe ApplicationClasse6 est ici <b>public</b>, donc visible.</p>

Un programme AppliTestClasses utilisant ces 4 classes :

C# dans deux namespace différents	Explication
<p><b>namespace Exemple</b></p>  <p><b>namespace Essai</b></p> <pre> using Exemple;  namespace Essai {     class AppliTestClasses{         ApplicationClasse2 a2;         ApplicationClasse6 a6;     } } </pre>	<p>Le programme de gauche "class AppliTestClasses" utilise les classes qui composent le <b>namespace</b> Exemple.</p> <p>Cette classe AppliTestClasses est définie dans un autre <b>namespace</b> dénommé Essai qui est supposé ne pas faire partie du même assembly que le <b>namespace</b> Exemple. Elle ne voit donc pas la classe ApplicationClasse4 (visible dans le même assembly seulement).</p> <p>Si l'on veut instancier des objets, seules les classes ApplicationClasse2 et ApplicationClasse6 sont de bonnes candidates, car la classe ApplicationClasse1 bien qu'elle soit visible est abstraite.</p>

C# dans le même namespace	Explication
<p><b>namespace Exemple</b></p>  <pre data-bbox="188 750 454 1025"> namespace Exemple { class AppliTestClasses{ ApplicationClasse1 a1; ApplicationClasse2 a2; ApplicationClasse4 a4; ApplicationClasse6 a6; } } </pre>	<p>Le programme de gauche "class AppliTestClasses" utilise les classes qui composent le namespace Exemple.</p> <p>La classe AppliTestClasses est définie dans le même namespace Exemple que les 4 autres classes.</p> <p>Toutes les classes du même namespace sont visibles entre elles.</p> <p>Ici toutes les 4 classes sont visibles pour la classe AppliTestClasses.</p>

**Remarque pratique :**

Selon sa situation imbriquée ou non imbriquée, une classe peut ou ne peut pas être qualifiée par les divers modificateurs de visibilité. En cas de doute le compilateur fournit un diagnostic clair, comme ci-dessous :

[C# Erreur] Class.cs(nn): Les éléments namespace ne peuvent pas être déclarés explicitement comme private, protected ou protected internal.

**1.7 Méthodes abstraites**

Le mot clef **abstract** est utilisé pour représenter **une classe ou une méthode abstraite**. Quel est l'intérêt de cette notion ? Avoir des modèles génériques permettant de définir ultérieurement des actions spécifiques.

**Une méthode déclarée en abstract dans une classe mère :**

- N'a pas de corps de méthode.
- N'est pas exécutable.

- Doit obligatoirement être redéfinie dans une classe fille.

Une méthode **abstraite** n'est qu'une **signature** de méthode sans implémentation dans la classe.

**Exemple de méthode abstraite :**

```
class Etre_Vivant { }
```

La classe Etre\_Vivant est une classe mère générale pour les êtres vivants sur la planète, chaque catégorie d'être vivant peut être représentée par une classe dérivée (classe fille de cette classe) :

```
class Serpent : Etre_Vivant { }
class Oiseau : Etre_Vivant { }
class Homme : Etre_Vivant { }
```

Tous ces êtres se déplacent d'une manière générale, donc une méthode SeDeplacer est commune à toutes les classes dérivées, toutefois chaque espèce exécute cette action d'une manière différente et donc on ne peut pas dire que se déplacer est une notion concrète mais une notion abstraite que chaque sous-classe précisera concrètement.

En C#, les méthodes abstraites sont automatiquement virtuelles, elles ne peuvent être déclarées que **public** ou **protected**, enfin elles doivent être redéfinies avec le qualificateur **override**. Ci-dessous deux déclarations possibles pour le déplacement des êtres vivants :

<pre>abstract class Etre_Vivant {     public abstract void SeDeplacer(); }  class Serpent : Etre_Vivant {     public override void SeDeplacer() {         //.....en rampant     } }  class Oiseau : Etre_Vivant {     public override void SeDeplacer() {         //.....en volant     } }  class Homme : Etre_Vivant {     public override void SeDeplacer() {         //.....en marchant     } }</pre>	<pre>abstract class Etre_Vivant {     protected abstract void SeDeplacer(); }  class Serpent : Etre_Vivant {     protected override void SeDeplacer() {         //.....en rampant     } }  class Oiseau : Etre_Vivant {     protected override void SeDeplacer() {         //.....en volant     } }  class Homme : Etre_Vivant {     protected override void SeDeplacer() {         //.....en marchant     } }</pre>
--	--

**Comparaison de déclaration d'abstraction de méthode en Delphi et C# :**

Delphi	C#
<pre>type     Etre_Vivant = class</pre>	<pre>abstract class Etre_Vivant {     public abstract void SeDeplacer();</pre>

<pre> <b>procedure</b> SeDeplacer;<b>virtual; abstract</b> ; <b>end</b>;  Serpent = <b>class</b> ( Etre_Vivant ) <b>procedure</b> SeDeplacer; <b>override</b>; <b>end</b>;  Oiseau = <b>class</b> ( Etre_Vivant ) <b>procedure</b> SeDeplacer; <b>override</b>; <b>end</b>;  Homme = <b>class</b> ( Etre_Vivant ) <b>procedure</b> SeDeplacer; <b>override</b>; <b>end</b>; </pre>	<pre> }  <b>class</b> Serpent : Etre_Vivant { <b>public override void</b> SeDeplacer() { //.....en rampant } }  <b>class</b> Oiseau : Etre_Vivant { <b>public override void</b> SeDeplacer() { //.....en volant } }  <b>class</b> Homme : Etre_Vivant { <b>public override void</b> SeDeplacer() { //.....en marchant } } </pre>
--	--

En C# une méthode **abstraite** est une méthode **virtuelle** n'ayant pas d'implémentation dans la classe où elle est déclarée. Son implémentation est déléguée à une classe dérivée. Les méthodes abstraites doivent être déclarées en spécifiant la directive **abstract** .

## 1.8 Classe abstraite, Interface

### Classe abstraite

Comme nous venons de le voir dans l'exemple précédent, une classe C# peut être précédée du mot clef **abstract**, ce qui signifie alors que cette classe est abstraite, nous avons les contraintes de définition suivantes pour une classe abstraite en C# :

Si une classe contient au moins une méthode **abstract**, elle doit impérativement être déclarée en classe **abstract** elle-même. C'est ce que nous avons écrit au paragraphe précédent pour la classe Etre\_Vivant que nous avons déclarée **abstract** parce qu'elle contenait la méthode abstraite SeDeplacer.

Une classe **abstract** ne peut pas être instanciée directement, seule une classe dérivée (sous-classe) qui redéfinit obligatoirement toutes les méthodes **abstract** de la classe mère peut être instanciée.

Conséquence du paragraphe précédent, une classe dérivée qui redéfinit toutes les méthodes **abstract** de la classe mère sauf une (ou plus d'une) ne peut pas être instanciée et subit la même règle que la classe mère : elle contient au moins une méthode abstraite donc elle est aussi une classe abstraite et doit donc être déclarée en **abstract**.

Une classe **abstract** peut contenir des méthodes non abstraites et donc implantées dans la classe. Une classe **abstract** peut même ne pas contenir du tout de méthodes abstraites, dans ce cas une classe fille n'a pas la nécessité de redéfinir les méthodes de la classe mère pour être instanciée.

*Delphi contrairement à C# et Java, ne possède pas à ce jour le modèle de la classe abstraite, seule la version Delphi2005 pour le Net Framework possède les mêmes caractéristiques que C#.*

### Interface

Lorsqu'une classe est déclarée en **abstract** et que toutes ses méthodes sont déclarées en **abstract**, on appelle en C# une telle classe une **Interface**.

## Rappel classes abstraites-interfaces

- Les interfaces ressemblent aux classes abstraites sur un seul point : elles contiennent des membres **expliquant certains comportements sans les implémenter**.
- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.

### Vocabulaire et concepts :

- Une **interface** est un contrat, elle peut contenir des **propriétés**, des **méthodes** et des **événements** mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une **interface** **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une **interface** doit contenir des méthodes **non** implémentées.
- Une **interface** est héritable.
- On peut construire une hiérarchie d'interfaces.
- Pour pouvoir construire un objet à partir d'une **interface**, il faut définir une classe non abstraite implémentant **toutes** les méthodes de l'**interface**.

Une classe **peut implémenter plusieurs interfaces**. Dans ce cas nous avons une excellente alternative à l'**héritage multiple**.

Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui **ne devraient plus être modifiés**. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **référéncés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

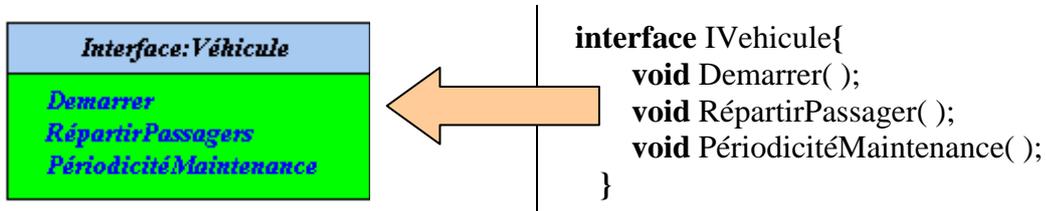
Si vous voulez utiliser la notion d'interface pour fournir un polymorphisme à une famille de classes, elles doivent toutes implémenter cette interface, comme dans l'exemple ci-dessous.

### Exemple :

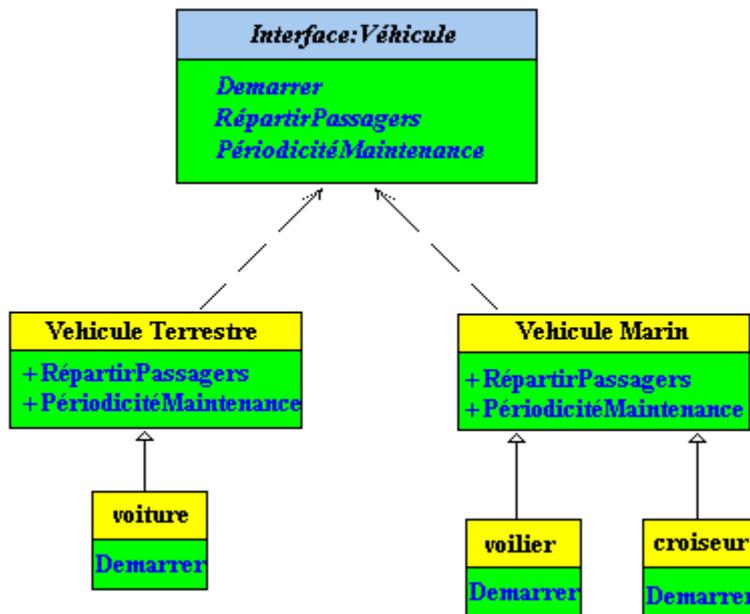
L'interface **Véhicule** définissant 3 méthodes (abstraites) **Demarrer**, **RépartirPassagers** de

répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), et **PériodicitéMaintenance** renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)

Soit l'interface *Véhicule* définissant ces 3 méthodes :



Soient les deux classes **Véhicule terrestre** et **Véhicule marin**, qui implémentent partiellement chacune l'interface *Véhicule* , ainsi que trois classes **voiture**, **voilier** et **croiseur** héritant de ces deux classes :



- Les trois méthodes de l'interface *Véhicule* sont abstraites et publiques par définition.
- Les classes **Véhicule terrestre** et **Véhicule marin** sont abstraites, car la méthode abstraite **Demarrer** de l'interface *Véhicule* n'est pas implémentée elle reste comme "modèle" aux futures classes. C'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

Dans cette vision de la hiérarchie on a supposé que les classes abstraites **Véhicule terrestre** et **Véhicule marin** savent comment répartir leur éventuels passagers et quand effectuer une maintenance du véhicule.

Les classes **voiture**, **voilier** et **croiseur** , n'ont plus qu'à implémenter chacune son propre comportement de démarrage.

Une interface C# peut être qualifiée par un des 4 modificateur **public**, **protected**, **internal**, **private**.

Contrairement à Java, une classe abstraite C# qui implémente une interface doit **obligatoirement** déclarer **toutes** les méthodes de l'interface, celles qui ne sont pas implémentées dans la classe abstraite doivent être déclarées **abstract**. C'est le cas dans l'exemple ci-dessous pour la méthode abstraite **Demarrer**, nous proposons deux écritures possibles pour cette hiérarchie de classe :

C# méthode <b>abstraite</b> sans corps	C# méthode <b>virtuelle</b> à corps vide
<pre> <b>interface</b> IVehicule{     <b>void</b> Demarrer();     <b>void</b> RépartirPassager();     <b>void</b> PériodicitéMaintenance(); }  <b>abstract class</b> Terrestre : IVehicule {     <b>public abstract void</b> Demarrer();     <b>public virtual void</b> RépartirPassager(){...}     <b>public virtual void</b> PériodicitéMaintenance(){...} }  <b>class</b> Voiture : Terrestre {     <b>override public void</b> Demarrer(){...} }  <b>abstract class</b> Marin : IVehicule {     <b>public abstract void</b> Demarrer();     <b>public virtual void</b> RépartirPassager(){...}     <b>public virtual void</b> PériodicitéMaintenance(){...} }  <b>class</b> Voilier : Marin {     <b>override public void</b> Demarrer(){...} } <b>class</b> Croiseur : Marin {     <b>override public void</b> Demarrer(){...} } </pre>	<pre> <b>interface</b> IVehicule{     <b>void</b> Demarrer();     <b>void</b> RépartirPassager();     <b>void</b> PériodicitéMaintenance(); }  <b>abstract class</b> Terrestre : IVehicule {     <b>public virtual void</b> Demarrer(){}     <b>public virtual void</b> RépartirPassager(){...}     <b>public virtual void</b> PériodicitéMaintenance(){...} }  <b>class</b> Voiture : Terrestre {     <b>override public void</b> Demarrer(){...} }  <b>abstract class</b> Marin : IVehicule {     <b>public virtual void</b> Demarrer() {}     <b>public virtual void</b> RépartirPassager(){...}     <b>public virtual void</b> PériodicitéMaintenance(){...} }  <b>class</b> Voilier : Marin {     <b>override public void</b> Demarrer(){...} } <b>class</b> Croiseur : Marin {     <b>override public void</b> Demarrer(){...} } </pre>

Les méthodes RépartirPassagers, PériodicitéMaintenance et Demarrer sont implantées en **virtual**, soit comme des méthodes à liaison dynamique, afin de laisser la possibilité pour des classes enfants de redéfinir ces méthodes.

**Remarque :**

Attention le qualificateur **new** ne peut masquer que des membres non abstract. Un membre abstract doit impérativement être redéfini (implémenté) par le qualificateur **override**, car ce genre de membre est implicitement **virtual** en C#.

Soit à titre de comparaison, les deux mêmes écritures en Java de ces classes :

```
Java , méthode abstraite sans corps

interface Ivehicule{
    void Demarrer( );
    void RépartirPassager( );
    void PériodicitéMaintenance( );
}

abstract class Terrestre implements Ivehicule {
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voiture extends Terrestre {
    public void Demarrer( ){};
}

abstract class Marin implements Ivehicule {
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voilier extends Marin {
    public void Demarrer( ){};
}

class Croiseur extends Marin {
    public void Demarrer( ){};
}

Java méthode virtuelle à corps vide

interface Ivehicule{
    void Demarrer( );
    void RépartirPassager( );
    void PériodicitéMaintenance( );
}

abstract class Terrestre implements Ivehicule {
    public void Demarrer( ){};
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voiture extends Terrestre {
    public void Demarrer( ){};
}

abstract class Marin implements Ivehicule {
    public void Demarrer( ){};
    public void RépartirPassager( ){};
    public void PériodicitéMaintenance( ){};
}

class Voilier extends Marin {
    public void Demarrer( ){};
}

class Croiseur extends Marin {
    public void Demarrer( ){};
}
```

## 2. Les objets : des références ou des valeurs

Les classes sont des descripteurs d'objets, les objets sont les agents effectifs et "vivants" implantant les actions d'un programme. Les objets dans un programme ont une vie propre :

- Ils naissent (ils sont créés ou alloués).
- Ils agissent (ils s'envoient des messages grâce à leurs méthodes).
- Ils meurent (ils sont désalloués, automatiquement en C#).

C'est dans le segment de mémoire du CLR de .NetFramework que s'effectuent l'allocation et la désallocation d'objets.

### Objets type valeur

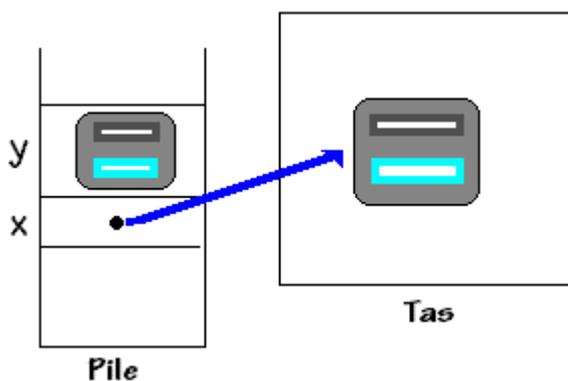
Les classes encapsulant les types élémentaires dans **.NET Framework** sont des classes de **type valeur** du genre **structures**. Dans le CLS une classe de **type valeur** est telle que les allocations d'objets de cette classe se font directement dans la pile et non dans le tas, il n'y a donc pas de référence pour un objet de **type valeur** et lorsqu'un objet de type valeur est passé comme paramètre il est **passé par valeur**.

Dans **.NET Framework** les classes-structures de **type valeur** sont déclarées comme structures et ne sont pas dérivables

### Objets type référence

Le principe d'allocation et de représentation des objets type référence en C# est identique à celui de Delphi il s'agit de la référence, qui est une encapsulation de la notion de pointeur. Dans **.NET Framework** les classes de type référence sont déclarées comme des classes classiques et sont dérivables.

Afin d'éclairer le lecteur prenons par exemple un objet **x** instancié à partir d'une classe de **type référence** et un objet **y** instancié à partir d'un classe de **type valeur** contenant les mêmes membres que la classe par référence. Ci-dessous le schéma d'allocation de chacun des deux catégories d'objets :



Pour les types valeurs, la gestion mémoire des objets est **classiquement celle de la pile dynamique**, un tel objet se comporte comme une variable locale de la méthode dans laquelle il est instancié et ne nécessite pas de gestion supplémentaire. Seuls les objets type référence instanciés sur le tas, nécessitent une gestion mémoire spéciale que nous détaillons ci-après (dans un

programme C# les types références du développeur représentent près de 99% des objets du programme).

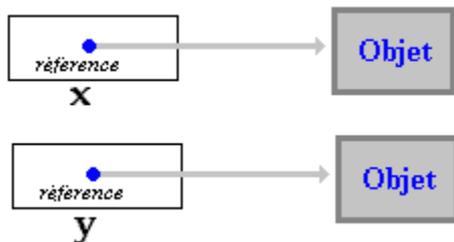
### 2.1 Modèle de la référence en C#

Rappelons que dans le modèle de la référence chaque objet (représenté par un identificateur de variable) est caractérisé par un couple (référence, bloc de données). Comme en Delphi, C# décompose l'**instanciation** (allocation) d'un objet en deux étapes :

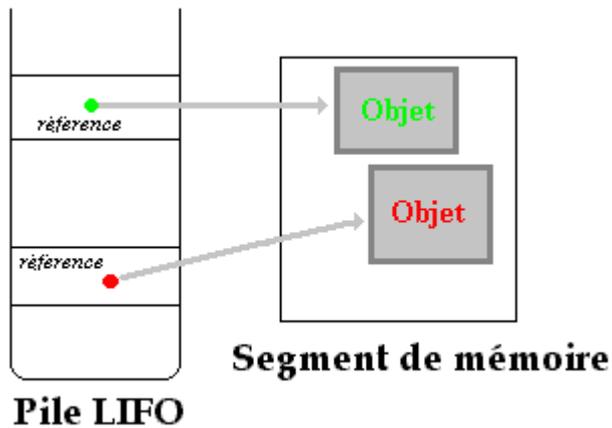
- La déclaration d'identificateur de variable typée qui contiendra la référence,
- la création de la structure de données elle-même (bloc objet de données) avec **new**.

Delphi	C#
<pre> type   Un = class     .....   end;  // la déclaration : var   x , y : Un; ....  // la création : x := Un.create ; y := Un.create ; </pre>	<pre> class Un { ... }  // la déclaration : Un x , y ;  ....  // la création : x = new Un(); y = new Un(); </pre>

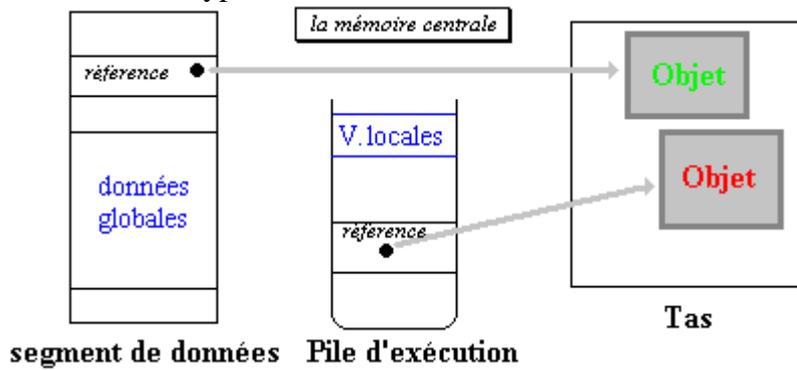
Après exécution du pseudo-programme précédent, les variables x et y contiennent chacune une référence (adresse mémoire) vers un bloc objet différent:



Un programme C# est fait pour être exécuté par l'environnement CLR de .NetFramework. Deux objets C# seront instanciés dans le CLR de la manière suivante :

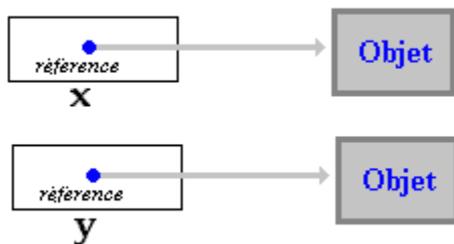


Attitude à rapprocher pour comparaison, à celle dont **Delphi** gère les objets dans une pile d'exécution de type LIFO et un tas :

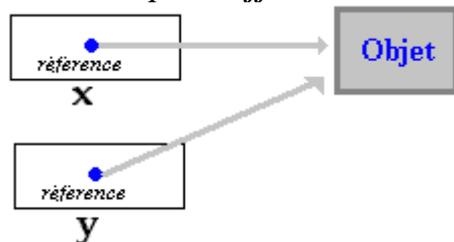


Attention à l'utilisation de l'affectation entre variables d'objets dans le modèle de représentation par référence. L'affectation  $x = y$  ne recopie pas le bloc objet de données de  $y$  dans celui de  $x$ , mais seulement la référence (l'adresse) de  $y$  dans la référence de  $x$ . Visualisons cette remarque importante :

*Situation au départ, avant affectation*



*Situation après l'affectation " x = y "*



En C#, la désallocation étant automatique, le bloc de données objet qui était référencé par  $y$  avant

l'affectation, n'est pas perdu, car le garbage collector se charge de restituer la mémoire libérée au **segment de mémoire** du CLR

## 2.2 Les constructeurs d'objets références ou valeurs

Un constructeur est une **méthode spéciale** d'une classe dont la seule fonction est d'**instancier** un objet (créer le bloc de données). Comme en Delphi une **classe C# peut posséder plusieurs constructeurs**, il est possible de pratiquer des initialisations d'attributs dans un constructeur. Comme toutes les méthodes, un constructeur peut avoir ou ne pas avoir de paramètres formels.

- Si vous ne déclarez pas de constructeur spécifique pour une classe, **par défaut** C# attribue automatiquement un constructeur sans paramètres formels, portant le même nom que la classe. A la différence de Delphi où le nom du constructeur est quelconque, en C# le( ou les) **constructeur doit obligatoirement porter le même nom que la classe** (majuscules et minuscules comprises).
- Un constructeur d'objet d'une classe n'a d'intérêt que s'il est visible par tous les programmes qui veulent instancier des objets de cette classe, c'est pourquoi l'on mettra toujours le mot clef **public** devant la déclaration du constructeur.
- Un constructeur est une méthode spéciale dont la fonction est de créer des objets, dans son en-tête il n'a pas de type de retour et le mot clef **void** n'est pas non plus utilisé !

Soit une classe dénommée Un dans laquelle, comme nous l'avons fait jusqu'à présent nous n'indiquons aucun constructeur spécifique :

```
class Un {  
    int a;  
}
```

Automatiquement C# attribue un constructeur public à cette classe **public Un ()**. C'est comme si C# avait introduit dans votre classe à votre insu , une nouvelle méthode dénommée **Un**. Cette méthode "cachée" n'a aucun paramètre et aucune instruction dans son corps. Ci-dessous un exemple de programme C# correct illustrant ce qui se passe :

```
class Un {  
    public Un () { }  
    int a;  
}
```

- Vous pouvez **programmer** et **personnaliser** vos propres constructeurs.
- Une classe C# peut contenir **plusieurs constructeurs** dont les en-têtes différent uniquement par la liste des paramètres formels.

*Exemple de constructeur avec instructions :*

C#	Explication
<pre>class Un { public Un ( )   { a = 100   }   int a; }</pre>	Le constructeur <b>public Un</b> sert ici à initialiser à 100 la valeur de l'attribut "int a" de chaque objet qui sera instancié.

*Exemple de constructeur avec paramètre :*

C#	Explication
<pre>class Un { public Un (int b )   { a = b;   }   int a; }</pre>	Le constructeur <b>public Un</b> sert ici à initialiser la valeur de l'attribut "int a" de chaque objet qui sera instancié. Le paramètre <b>int b</b> contient cette valeur.

*Exemple avec plusieurs constructeurs :*

C#	Explication
<pre>class Un { public Un (int b )   { a = b;   }   public Un ( )   { a = 100;   }   public Un (float b )   { a = (int)b;   }   int a; }</pre>	La classe Un possède 3 constructeurs servant à initialiser chacun d'une manière différente le seul attribut <b>int a</b> .

Il est possible de rappeler un constructeur de la classe dans un autre constructeur, pour cela C# utilise comme Java le mot clef **this**, avec une syntaxe différente :

*Exemple avec un appel à un constructeur de la même classe:*

C#	Explication
<pre>class Un {   int a;   public Un (int b )   { a = b;   }   public Un ( )</pre>	La classe Un possède 3 constructeurs servant à initialiser chacun d'une manière différente le seul attribut <b>int a</b> .  <b>Soit le dernier constructeur :</b> <b>public Un (int x , float y ) : this(y)</b>

<pre> { a = 100; } public Un (float b ) { a = (int)b; } public Un (int x , float y ) : this(y) { a += 100; } } </pre>	<pre> { a += 100; } </pre> <p>Ce constructeur appelle tout d'abord le constructeur <b>Un (float y )</b> par l'intermédiaire de <b>this(y)</b>, puis il exécute le corps de méthode soit : <b>a += 100;</b></p> <p>Ce qui revient à calculer : a = (int)y + 100 ;</p>
---	--

### Comparaison Delphi - C# pour la déclaration de constructeurs

Delphi	C#
<pre> Un = class   a : integer; public   constructor creer; overload;   constructor creer (b:integer); overload;   constructor creer (b:real); overload; end;  <u>implementation</u>    constructor Un.creer; begin     a := 100   end;   constructor Un.creer(b:integer); begin     a := b   end;   constructor Un.creer(b:real); begin     a := trunc(b)   end;    constructor Un.creer(x:integer; y:real);   begin     self.creer(y);     a := a+100;   end; </pre>	<pre> class Un {   int a;    public Un ( )   { a = 100;   }    public Un (int b )   { a = b;   }    public Un (float b )   { a = (int)b;   }    public Un (int x , float y ) :   this(y)   { a += 100;   } } </pre>

En Delphi un constructeur a un nom quelconque, tous les constructeurs peuvent avoir des noms différents ou le même nom comme en C#.

### 2.3 Utilisation du constructeur d'objet automatique (par défaut)

Le constructeur d'objet par défaut de toute classe C# qu'elle soit de type valeur ou de type référence, comme nous l'avons signalé plus haut est une méthode spéciale sans paramètre, l'appel à cette méthode spéciale afin de construire un nouvel objet répond à une syntaxe spécifique par

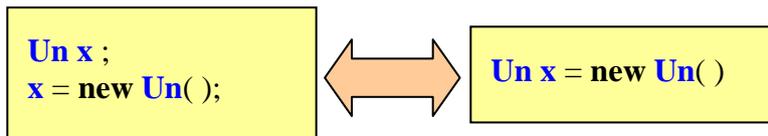
utilisation du mot clef **new**.

### Syntaxe

Pour un constructeur sans paramètres formels, l'instruction d'**instanciation d'un nouvel objet** à partir d'un identificateur de variable déclarée selon un type de classe, s'écrit syntaxiquement ainsi :



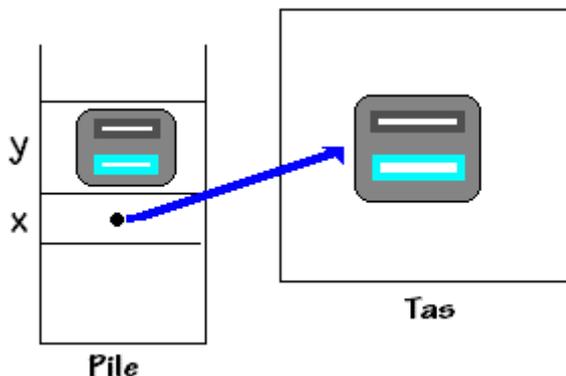
**Exemple** : (deux façons équivalentes de créer un objet **x** de classe **Un**)



Cette instruction crée dans le segment de mémoire, un nouvel objet de classe **Un** dont la référence (l'adresse) est mise dans la variable **x**, si **x** est de type référence, ou bien l'objet est directement créé dans la pile et mis dans la variable **x**, si **x** est de type valeur.

Soit **Un** une classe de type référence et **Deux** une autre classe de type valeur, ci-dessous une image des résultats de l'instanciation d'un objet de chacune de ces deux classes :

```
Un x = new Un() ;  
Deux y = new Deux () ;
```



Dans l'exemple ci-dessous, nous utilisons le constructeur par défaut de la classe **Un** , pour créer deux objets dans une autre classe :

```
class Un  
{ ...  
}
```

```
class UnAutre  
{
```

```

// la déclaration :
Un x, y ;
....
// la création :
x = new Un( );
y = new Un( );
}

```

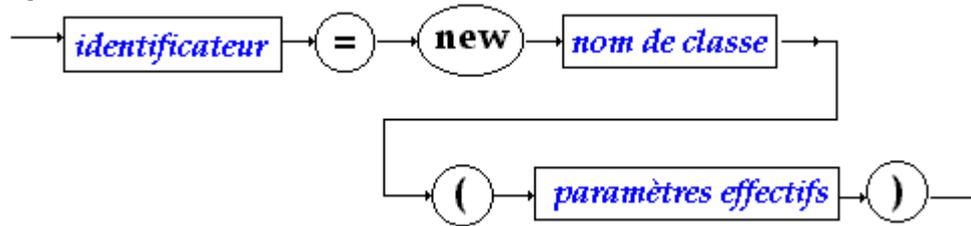
*Un programme de 2 classes, illustrant l'affectation de références :*

C#	Explication
<pre> class AppliClassesReferences { public static void Main(String [ ] arg) { Un x,y ; x = new Un( ); y = new Un( ); System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a); y = x; x.a =12; System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a); } } class Un { int a=10; } </pre>	<p>Ce programme C# contient deux classes :</p> <p><b>class</b> AppliClassesReferences et <b>class</b> Un</p> <p>La classe AppliClassesReferences est une classe exécutable car elle contient la méthode main. C'est donc cette méthode qui agira dès l'exécution du programme.</p>
Détaillons les instructions	Que se passe-t-il à l'exécution ?
<pre> Un x,y ; x = new Un( ); y = new Un( ); </pre>	<p>Instanciation de 2 objets différents x et y de type <b>Un</b>.</p>
<pre> System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a); </pre>	<p><i>Affichage de :</i> x.a = 10 y.a = 10</p>
<pre> y = x; </pre>	<p>La référence de y est remplacée par celle de x dans la variable y (y pointe donc vers le même bloc que x).</p>
<pre> x.a =12; System.Console.WriteLine("x.a="+x.a); System.Console.WriteLine("y.a="+y.a); </pre>	<p><i>On change la valeur de l'attribut a de x, et l'on demande d'afficher les attributs de x et de y :</i> x.a = 12 y.a = 12 <b>Comme y pointe vers x, y et x sont maintenant le même objet sous deux noms différents !</b></p>

## 2.4 Utilisation d'un constructeur d'objet personnalisé

L'utilisation d'un constructeur personnalisé d'une classe est semblable à celle du constructeur par défaut de la classe. La seule différence se trouve lors de l'instanciation : il faut fournir des paramètres effectifs lors de l'appel au constructeur.

## Syntaxe



*Exemple avec plusieurs constructeurs :*

une classe C#	Des objets créés
<pre> class Un {   int a ;   public Un (int b ) {     a = b ; }    public Un ( ) {     a = 100 ; }    public Un (float b ) {     a = (int)b ; }    public Un (int x , float y ) : this(y)   { a += 100;   } } </pre>	<pre> Un obj1 = new Un( ); Un obj2 = new Un( 15 );  int k = 14; Un obj3 = new Un( k );  Un obj4 = new Un( 3.25f );  float r = -5.6; Un obj5 = new Un( r );  int x = 20; float y = -0.02; Un obj6 = new Un( x , y ); </pre>

## 2.5 Le mot clef **this** - cas de la référence seulement

Il est possible de dénommer dans les instructions d'une méthode de classe, un futur objet qui sera instancié plus tard. Le paramètre ou (mot clef) **this** est implicitement présent dans chaque objet instancié et il contient la référence à l'objet actuel. Il joue exactement le même rôle que le mot clef **self** en Delphi. Nous avons déjà vu une de ses utilisations dans le cas des constructeurs.

C#	C# équivalent
<pre> class Un { public Un ( )   { a = 100;   }   int a; } </pre>	<pre> class Un { public Un ( )   { this.a = 100;   }   int a; } </pre>

Dans le programme de droite le mot clef **this** fait référence à l'objet lui-même, ce qui dans ce cas est superflu puisque la variable **int a** est un champ de l'objet.

Montrons deux exemples d'utilisation pratique de **this**.

*Cas où l'objet est passé comme un paramètre dans une de ses méthodes :*

C#	Explications
<pre> class Un { public Un ()   { a = 100;   }   public void methode1(Un x)   {     System.Console.WriteLine("champ a = "                               + x.a);   }   public void methode2( int b )   { a += b;     methode1(this);   }   int a; } </pre>	<p>La methode1(Un x) reçoit un objet de type Un en paramètre et imprime son champ <b>int</b> a.</p> <p>La methode2( <b>int</b> b ) reçoit un entier <b>int</b> b qu'elle additionne au champ <b>int</b> a de l'objet, puis elle appelle la methode1 avec comme paramètre l'objet lui-même.</p>

*Comparaison Delphi - C# sur cet exemple (similitude complète)*

Delphi	C#
<pre> Un = class   a : integer;   public   constructor creer;   procedure methode1( x:Un );   procedure methode2 ( b:integer ); end;  implementation constructor Un.creer; begin   a := 100 end; procedure Un.methode1( x:Un ); begin   showmessage( 'champ a =' + inttostr(x.a) ) end; procedure Un.methode2 ( b:integer ); begin   a := a+b;   methode1(self) end; </pre>	<pre> class Un { public Un ()   { a = 100;   }   public void methode1(Un x)   {     System.Console.WriteLine("champ a =" +x.a);   }   public void methode2( int b )   { a += b;     methode1(this);   }   int a; } </pre>

*Cas où le this sert à outrepasser le masquage de visibilité :*

C#	Explications
<pre> class Un {   int a;   public void methode1(float a)   { a = this.a + 7 ;   } } </pre>	<p>La methode1(float a) possède un paramètre <b>float</b> a dont le nom masque le nom du champ <b>int</b> a.</p> <p>Si nous voulons malgré tout accéder au champ de l'objet, l'objet étant référencé par <b>this</b>, "<b>this.a</b>" est donc le champ <b>int</b> a de l'objet lui-même.</p>

*Comparaison Delphi - C# sur ce second exemple (similitude complète aussi)*

Delphi	C#
<pre> <b>Un = class</b>   a : integer; <b>public</b>   procedure methode( a:real ); <b>end;</b>  <b>implementation</b>    procedure <b>Un.methode</b>( a:real );<b>begin</b>     a = <b>self.a</b> + 7 ;   <b>end;</b> </pre>	<pre> <b>class Un</b> {   <b>int a;</b>   <b>public void methode(float a)</b>   {     a = <b>this.a</b> + 7 ;   } } </pre>

### 3. Variables et méthodes

Nous examinons dans ce paragraphe comment C# utilise les variables et les méthodes à l'intérieur d'une classe. Il est possible de modifier des variables et des méthodes d'une classe ceci sera examiné plus loin.

En C#, les champs et les méthodes sont classés en deux catégories :

- Variables et méthodes de classe
- Variables et méthodes d'instance

#### 3.1 Variables dans une classe en général

Rappelons qu'en C#, nous pouvons déclarer dans un bloc (for, try,...) de nouvelles variables à la condition qu'elles n'existent pas déjà dans le corps de la méthode où elles sont déclarées. Nous les dénommerons : **variables locales de méthode**.

*Exemple de variables locales de méthode :*

<pre> <b>class</b> Exemple {   <b>void</b> calcul ( <b>int</b> x, <b>int</b> y )   {<b>int</b> a = 100;     <b>for</b> ( <b>int</b> i = 1; i&lt;10; i++)     {<b>char</b> carlu;       System.Console.Write("Entrez un caractère : ");       carlu = (<b>char</b>)System.Console.Read();       <b>int</b> b =15;       a =....       .....     }   } } </pre>	<p>La définition <b>int a = 100;</b> est locale à la méthode en général</p> <p>La définition <b>int i = 1;</b> est locale à la boucle <b>for</b>.</p> <p>Les définitions <b>char carlu</b> et <b>int b</b> sont locales au corps de la boucle <b>for</b>.</p>
---	---

}	
---	--

C# ne connaît pas la notion de variable globale au sens habituel donné à cette dénomination, dans la mesure où toute variable ne peut être définie qu'à l'intérieur d'une classe, ou d'une méthode incluse dans une classe. Donc à part les **variables locales de méthode** définies dans une méthode, C# reconnaît une autre catégorie de variables, *les variables définies dans une classe mais pas à l'intérieur d'une méthode spécifique*. Nous les dénommerons : **attributs de classes** parce que ces variables peuvent être de deux catégories.

*Exemple de attributs de classe :*

<pre> class AppliVariableClasse {     float r ;      void calcul ( int x, int y )     {         ....     }     int x =100;     int valeur ( char x )     {         ....     }     long y; } </pre>	<p>Les variables <b>float r</b> , <b>long y</b> et <b>int x</b> sont des attributs de classe (ici en fait plus précisément, des variables d'instance).</p> <p>La position de la déclaration de ces variables n'a aucune importance. Elles sont <b>visibles dans tout le bloc classe</b> (c'est à dire visibles par toutes les méthodes de la classe).</p> <p><b>Conseil</b> : regroupez les variables de classe au début de la classe afin de mieux les gérer.</p>
--	--

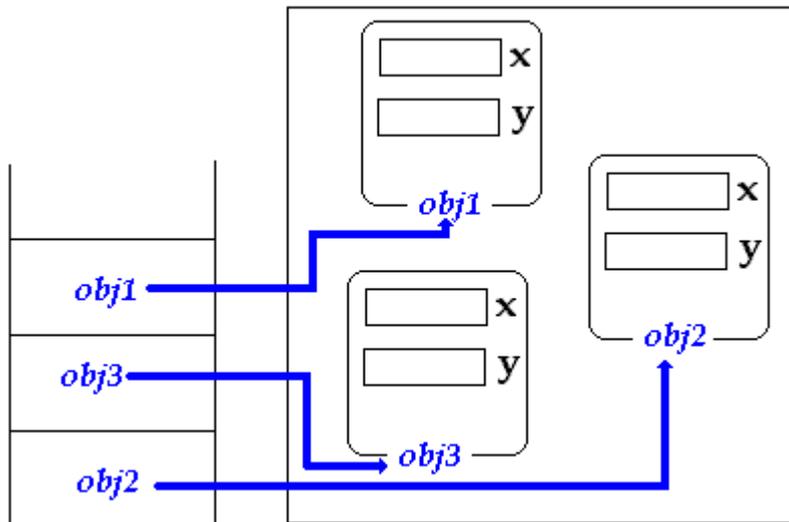
Les attributs de classe peuvent être soit de la catégorie des **variables de classe**, soit de la catégorie des **variables d'instance**.

### 3.2 Variables et méthodes d'instance

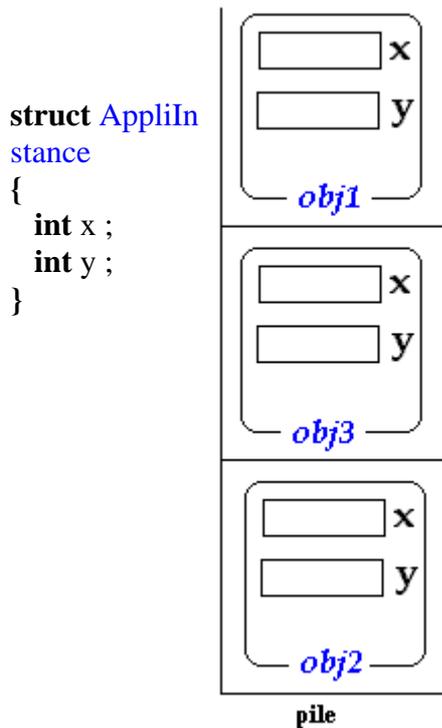
C# se comporte comme un langage orienté objet classique vis à vis de ses variables et de ses méthodes. A chaque instantiation d'un nouvel objet d'une classe donnée, la machine CLR enregistre le p-code des méthodes de la classe dans la **zone de stockage** des méthodes, elle alloue dans le **segment de mémoire autant d'emplacements mémoire pour les variables que d'objet créés**. C# dénomme cette catégorie **les variables et les méthodes d'instance**.

une classe C#	Instantiation de 3 objets
<pre> class AppliInstance {     int x ;     int y ; } </pre>	<pre> AppliInstance obj1 = new AppliInstance(); AppliInstance obj2 = new AppliInstance(); AppliInstance obj3 = new AppliInstance(); </pre>

*Segment de mémoire associé à ces 3 objets si la classe **AppliInstance** est de type référence :*



Segment de mémoire associé à ces 3 objets si la classe `AppliInstance` était de type valeur (pour mémoire):



Un programme C# à 2 classes illustrant l'exemple précédent (classe référence):

Programme C# exécutable
<pre> <b>class</b> AppliInstance { <b>public int</b> x = -58 ;   <b>public int</b> y = 20 ; } <b>class</b> Utilise { <b>public static void</b> Main() {   AppliInstance obj1 = <b>new</b> AppliInstance(); </pre>

```

AppliInstance obj2 = new AppliInstance();
AppliInstance obj3 = new AppliInstance();
System.Console.WriteLine( "obj1.x = " + obj1.x );
}
}

```

### 3.3 Variables et méthodes de classe - static

#### Variable de classe

On identifie une variable ou une méthode de classe en précédant sa déclaration du mot clef **static**. Nous avons déjà pris la majorité de nos exemples simples avec de tels composants.

Voici deux déclarations de variables de classe :

```

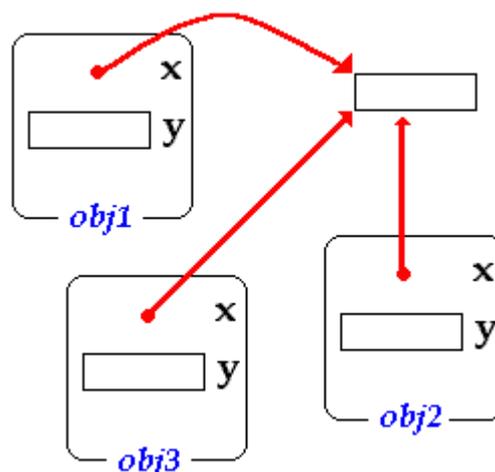
static int x ;
static int a = 5;

```

Une variable de classe est accessible comme une variable d'instance (selon sa visibilité), mais aussi **sans avoir à instancier un objet de la classe**, uniquement en référant la variable par le nom de la classe dans la notation de chemin uniforme d'objet.

une classe C#	Instanciation de 3 objets
<pre> class AppliInstance {     static int x ;     int y ; } </pre>	<pre> AppliInstance obj1 = new AppliInstance(); AppliInstance obj2 = new AppliInstance(); AppliInstance obj3 = new AppliInstance(); </pre>

Voici une image du segment de mémoire associé à ces 3 objets :



Exemple de variables de classe :

<pre> class ApplistaticVar {     public static int x =15 ; } </pre>	<p>La définition "<b>static int x =15 ;</b>" crée une variable de la classe <b>ApplistaticVar</b>, nommée x.</p>
---	--

<pre> class UtiliseApplisticVar {     int a ;     void f()     {         a = ApplisticVar.x ;         .....     } } </pre>	<p>L'instruction "a = <code>ApplisticVar.x</code> ;" utilise la variable x comme variable de classe <code>ApplisticVar</code> sans avoir instancié un objet de cette classe.</p>
--	--

Nous pouvons utiliser la classe Math ( **public sealed class** Math ) qui contient des constantes et des fonctions mathématiques courantes :

```

public static const double E; // la constante e représente la base du logarithme népérien.
public static const double PI; // la constante pi représente le rapport de la circonférence d'un
cercle à son diamètre.

```

### Méthode de classe

Une méthode de classe est une méthode dont l'implémentation est la même pour tous les objets de la classe, en fait la différence avec une méthode d'instance a lieu sur la catégorie des variables sur lesquelles ces méthodes agissent.

**De par leur définition les méthodes de classe ne peuvent travailler qu'avec des variables de classe, alors que les méthodes d'instances peuvent utiliser les deux catégories de variables.**

*Un programme correct illustrant le discours :*

C#	Explications
<pre> class Exemple {     public static int x ;     int y ;     public void f1(int a)     {         x = a;         y = a;     }     public static void g1(int a)     {         x = a;     } } class Utilise {     public static void Main() {         Exemple obj = new Exemple();         obj.f1(10);         System.Console.WriteLine("&lt;f1(10)&gt;obj.x="+ Exemple.x);         Exemple.g1(50);         System.Console.WriteLine("&lt;g1(50)&gt;obj.x="+ Exemple.x);     } } </pre>	<pre> <b>public void</b> f1(int a) {     x = a; //accès à la variable de classe     y = a ; //accès à la variable d'instance }  <b>public static void</b> g1(int a) {     x = a; //accès à la variable de classe     y = a ; //engendrerait un erreur de compilation : accès à une variable non static interdit ! } </pre> <p>La méthode f1 accède à toutes les variables de la classe Exemple, la méthode g1 n'accède qu'aux variables de classe (<b>static</b> et <b>public</b>).</p> <p><i>Après exécution on obtient :</i></p> <pre> &lt;f1(10)&gt;obj.x = 10 &lt;g1(50)&gt;obj.x = 50 </pre>

## Résumé pratique sur les membres de classe en C#

1) - Les méthodes et les variables de classe sont **précédées obligatoirement** du mot clef **static**. Elles jouent un rôle **semblable** à celui qui est attribué aux variables et aux sous-routines globales dans un langage impératif classique.

C#	Explications
<pre> class Exemple1 {     int a = 5;     static int b = 19;     void m1(){...}     static void m2() {...} } </pre>	<p>La variable a dans <b>int a = 5</b>; est une variable d'instance.</p> <p>La variable b dans <b>static int b = 19</b>; est une variable de classe.</p> <p>La méthode m2 dans <b>static void m2() {...}</b> est une méthode de classe.</p>

2) - Pour utiliser une variable **x1** ou une méthode **meth1** de la classe **Classe1**, il suffit de d'écrire **Classe1.x1** ou bien **Classe1.meth1**.

C#	Explications
<pre> class Exemple2 {     public static int b = 19;     public static void m2() {...} } class UtiliseExemple {     Exemple2 b = 53;     Exemple2.m2();     ... } </pre>	<p>Dans la classe Exemple2, b est une variable de classe, m2 une méthode de classe.</p> <p>La classe UtiliseExemple fait appel à la méthode m2 directement avec le nom de la classe, il en est de même avec le champ b de la classe Exemple2</p>

3) - Une variable de classe (précédée du mot clef **static**) est **partagée par tous les objets** de la même classe.

C#	Explications
----	--------------

<pre> class AppliStatic { public static int x = -58 ;   public int y = 20 ;   ... } class Utilise { public static void Main(String [ ] arg) {   AppliStatic obj1 = new AppliStatic( );   AppliStatic obj2 = new AppliStatic( );   AppliStatic obj3 = new AppliStatic( );   obj1.y = 100;   AppliStatic.x = 101;   System.Console.WriteLine("obj1.x="+ AppliStatic.x);   System.Console.WriteLine("obj1.y="+obj1.y);   System.Console.WriteLine("obj2.x="+ AppliStatic.x);   System.Console.WriteLine("obj2.y="+obj2.y);   System.Console.WriteLine("obj3.x="+ AppliStatic.x);   System.Console.WriteLine("obj3.y="+obj3.y);   AppliStatic.x = 99;   System.Console.WriteLine("AppliStatic.x="                           + AppliStatic.x); } } </pre>	<p>Dans la classe AppliStatic <b>x</b> est une variable de classe, et <b>y</b> une variable d'instance.</p> <p>La classe Utilise crée 3 objets (obj1,obj2,obj3) de classe AppliStatic.</p> <p>L'instruction <b>obj1.y = 100;</b> est un accès au champ <b>y</b> de l'instance obj1. Ce n'est que le champ <b>y</b> de cet objet qui est modifié,les champs <b>y</b> des objets obj2 et obj3 restent inchangés</p> <p>Il n'y a qu'une seule manière d'accéder à la variable static <b>x</b>, comme pour toute variable de classe, en utilisant le nom de la classe :</p> <p><b>AppliStatic.x = 101;</b></p> <p>Dans ce cas cette variable <b>x</b> est modifiée globalement et donc tous les champs <b>x</b> de tous les objets obj1, obj2 et obj3 prennent la nouvelle valeur.</p>
--	--

Au début lors de la création des 3 objets, chacun des champs **x** vaut -58 et chacun des champs **y** vaut 20, l'affichage par `System.Console.WriteLine(...)` donne les résultats suivants qui démontrent le partage de la variable **x** par tous les objets.

*Après exécution :*

```

obj1.x = 101
obj1.y = 100
obj2.x = 101
obj2.y = 20
obj3.x = 101
obj3.y = 20
AppliStatic.x = 99

```

4) - Une méthode de classe (précédée du mot clef **static**) **ne peut utiliser que des variables de classe** (précédées du mot clef **static**) et jamais des variables d'instance. Une méthode d'instance peut accéder aux deux catégories de variables

5) - Une méthode de classe (précédée du mot clef **static**) **ne peut appeler** (invoquer) **que des méthodes de classe** (précédées du mot clef **static**).

C#	Explications
<pre> class AppliStatic {   public static int x = -58 ;   public int y = 20 ;    public void f1(int a)   { AppliStatic.x = a; </pre>	<p>Nous reprenons l'exemple précédent en ajoutant à la classe AppliStatic une méthode interne f1 :</p> <pre>   public void f1(int a) {     AppliStatic.x = a;     y = 6 ;   } </pre> <p>Cette méthode accède à la variable de classe</p>

<pre>         y = 6 ;     } }  class Utilise {     static void f2(int a)     { AppliStatic.x = a;     }      public static void Main() {         AppliStatic obj1 = new AppliStatic();         AppliStatic obj2 = new AppliStatic();         AppliStatic obj3 = new AppliStatic();         obj1.y = 100;          obj1.x = 101;         AppliStatic.x = 99;         f2(101);         obj1.f1(102);     } } </pre>	<p>comme un champ d'objet.</p> <p>Nous rajoutons à la classe Utilise, un méthode static (méthode de classe) notée f2:</p> <pre> <b>static void</b> f2(int a) { AppliStatic.x = a; } </pre> <p>Cette méthode accède elle aussi à la variable de classe parce qu c'est une méthode static.</p> <p>Nous avons donc <b>trois</b> manières d'accéder à la variable static <b>x</b> :</p> <p>soit directement comme une variable de classe proprement dite : <b>AppliStatic.x = 99;</b></p> <p>soit indirectement par une méthode d'instance sur son champ : <b>obj1.f1(102);</b></p> <p>soit indirectement par une méthode <b>static</b> (de classe) : <b>f2(101);</b></p>
---	---

Comme la méthode Main est **static**, elle peut invoquer la méthode f2 qui est aussi **static**.

Au paragraphe précédent, nous avons indiqué que C# ne connaissait pas la notion de variable globale stricto sensu, mais en fait une variable **static peut jouer le rôle d'un variable globale pour un ensemble d'objets** instanciés à partir de la même classe.

### Attention :

Contrairement à Java qui autorise l'accès à une variable **static** à partir d'un nom d'objet à la place du nom de la classe de définition, le compilateur C# impose une cohérence dans les déclarations, en refusant cette possibilité.

Depuis la version 2.0, une classe peut être **static**, dans ce cas elle correspond à un module de bibliothèque de membres (méthodes et variables) obligatoirement tous **static**.

```

static class AppliStatic
{
    public static int x = -58 ;
    public static int y = 20 ;

    public static void f1(int a)
    { x = a;
      y = 6 ;
    }
}

```



# Polymorphisme d'objet en C#.net

---

Plan général: 

## Rappel des notions de base

## Polymorphisme d'objet en C# : définitions

- 1.1 Instanciation et utilisation dans le même type
- 1.2 Polymorphisme d'objet implicite
- 1.3 Polymorphisme d'objet explicite par transtypage
- 1.4 Utilisation pratique du polymorphisme d'objet
- 1.5 Instanciation dans un type ascendant impossible

## Le polymorphisme en C#

### Rappel utile sur les notions de bases

Il existe un concept essentiel en POO désignant la capacité d'une hiérarchie de classes à fournir différentes implémentations de méthodes portant le même nom et par corollaire la capacité qu'ont des objets enfants de modifier les comportements hérités de leur parents. Ce concept d'adaptation à différentes "situations" se dénomme le **polymorphisme** qui peut être implémenté de différentes manières.

#### Polymorphisme d'objet

C'est une interchangeabilité entre variables d'objets de classes de la même hiérarchie sous certaines conditions, que dénommons le polymorphisme d'objet.

#### Polymorphisme par héritage de méthode

Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires nouvelles peuvent être implémentées dans la classe enfant mais aussi des méthodes des parents peuvent être substituées pour obtenir des implémentations différentes.

#### Polymorphisme par héritage de classes abstraites

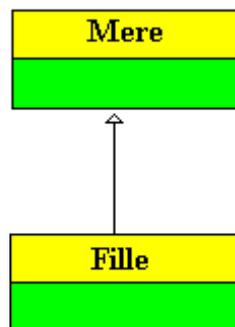
Une classe abstraite est une classe qui ne peut pas s'instancier elle-même ; elle doit être héritée. Certains membres de la classe peuvent ne pas être implémentés, et c'est à la classe qui hérite de fournir cette implémentation.

#### Polymorphisme par implémentation d'interfaces

Une interface décrit la signature complète des membres qu'une classe doit implémenter, mais elle laisse l'implémentation de tous ces membres à la charge de la classe d'implémentation de l'interface.

### Polymorphisme d'objet en C#

Soit une classe **Mere** et une classe **Fille** héritant de la classe **Mere** :



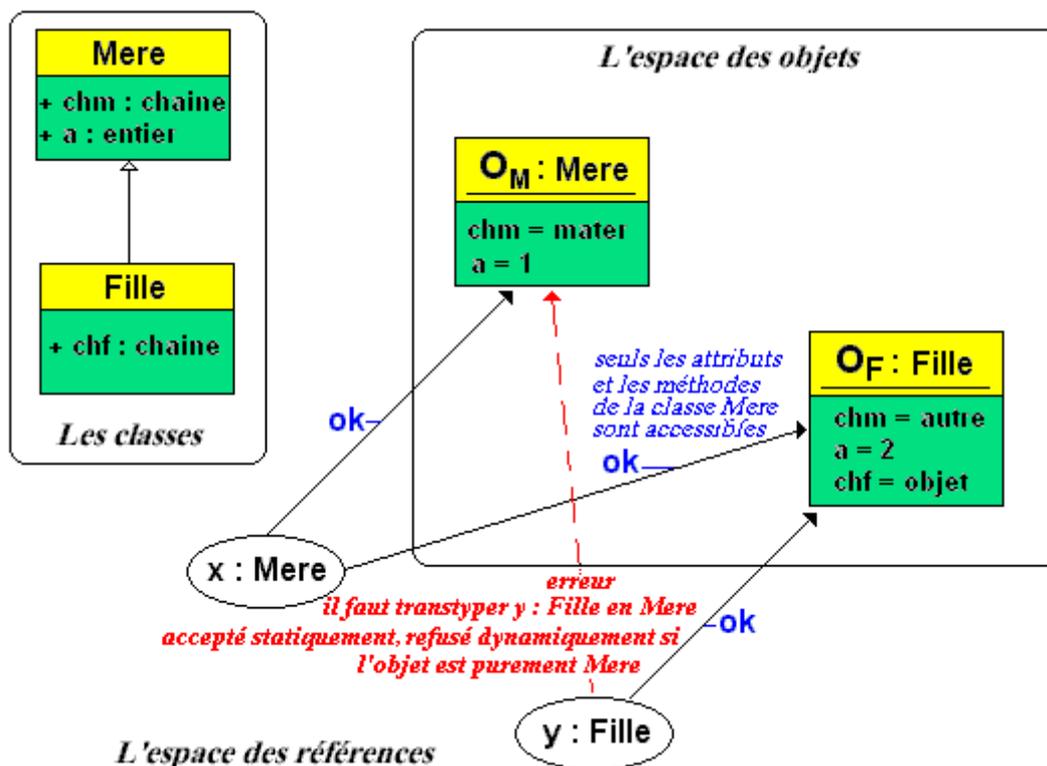
Les objets peuvent avoir des comportements polymorphes (s'adapter et se comporter différemment selon leur utilisation) licites et des comportements polymorphes dangereux selon les langages.

Dans un langage dont le modèle objet est la référence (un objet est un couple : référence, bloc mémoire) comme **C#**, il y a découplage entre les actions statiques du compilateur et les actions dynamiques du système d'exécution, **le compilateur protège statiquement des actions dynamiques sur les objets** une fois créés. C'est la déclaration et l'utilisation des variables de références qui autorise ou non les actions licites grâce à la compilation.

Supposons que nous ayons déclaré deux variables de référence, l'une de classe **Mere**, l'autre de classe **Fille**, une question qui se pose est la suivante : au cours du programme quel genre d'affectation et d'instanciation est-on autorisé à effectuer sur chacune de ces variables dans un programme C#.

<pre> En C# : public class Mere { ..... } public class Fille : Mere { ..... } </pre>	<p>L'héritage permet une variabilité entre variables d'objets de classes de la même hiérarchie, c'est cette variabilité que dénommons le <b>polymorphisme d'objet</b>.</p>
--	--

Nous envisageons toutes les situations possibles et les évaluons, les exemples explicatifs sont écrits en C# (lorsqu'il y a discordance avec java ou Delphi autres langages, celle-ci est mentionnée explicitement), il existe 3 possibilités différentes illustrées par le schéma ci-dessous.



*L'instanciation et l'utilisation de références dans le même type*

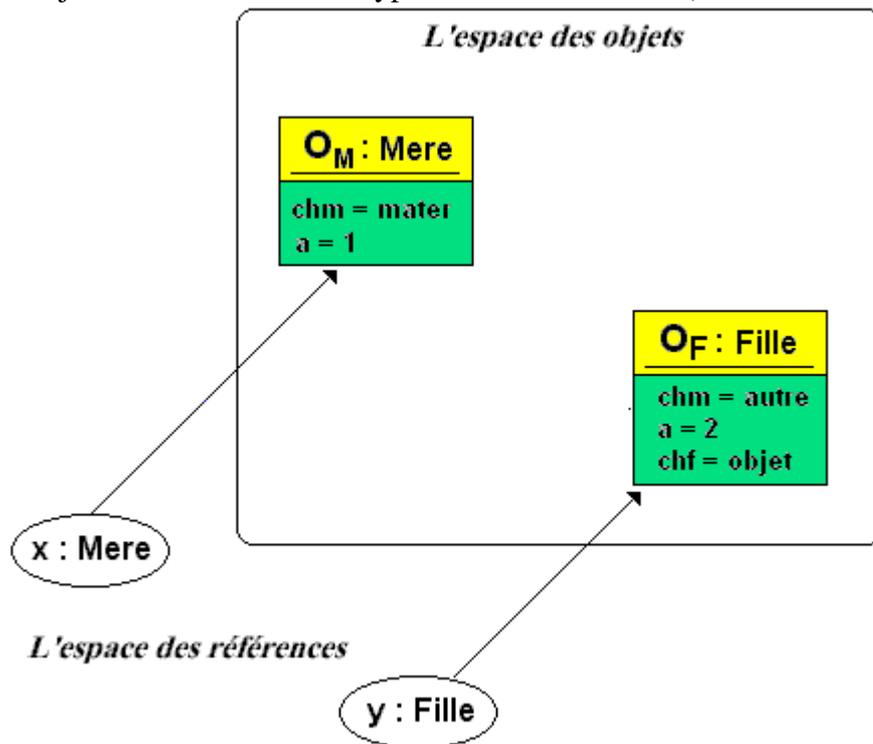
*L'affectation de références : polymorphisme implicite*

*L'affectation de références : polymorphisme par transtypage d'objet*

La dernière de ces possibilités pose un problème d'exécution lorsqu'elle mal employée !

## 1.1 instanciation dans le type initial et utilisation dans le même type

Il s'agit ici d'une utilisation la plus classique qui soit, dans laquelle une variable de référence d'objet est utilisée dans son type de définition initial (valable dans tous les LOO)



En C# :

```
Mere x , u ;
```

```
Fille y , w ;
```

```
.....
```

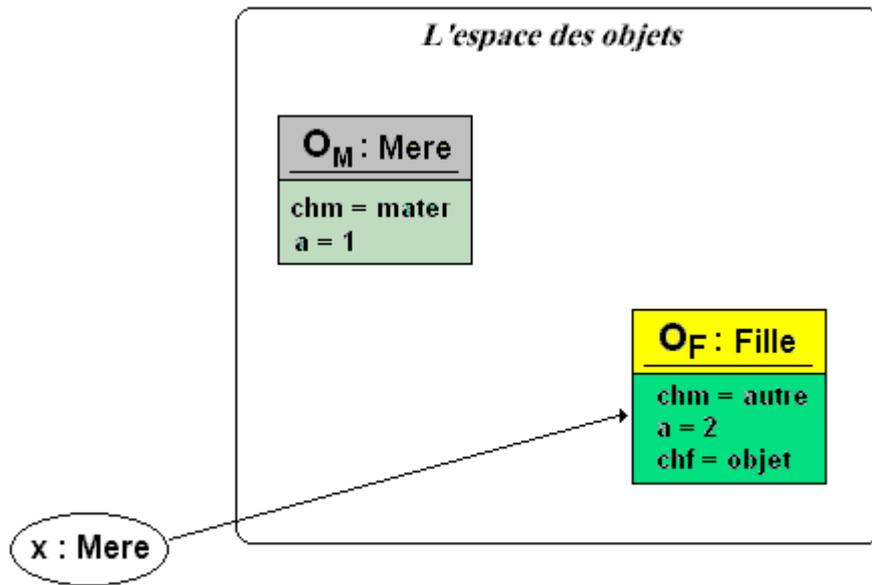
```
x = new Mere( ) ; // instanciation dans le type initial
```

```
u = x ; // affectation de références du même type
```

```
y = new Fille( ) ; // instanciation dans le type initial
```

```
w = y ; // affectation de références du même type
```

## 1.2 Polymorphisme d'objet implicite

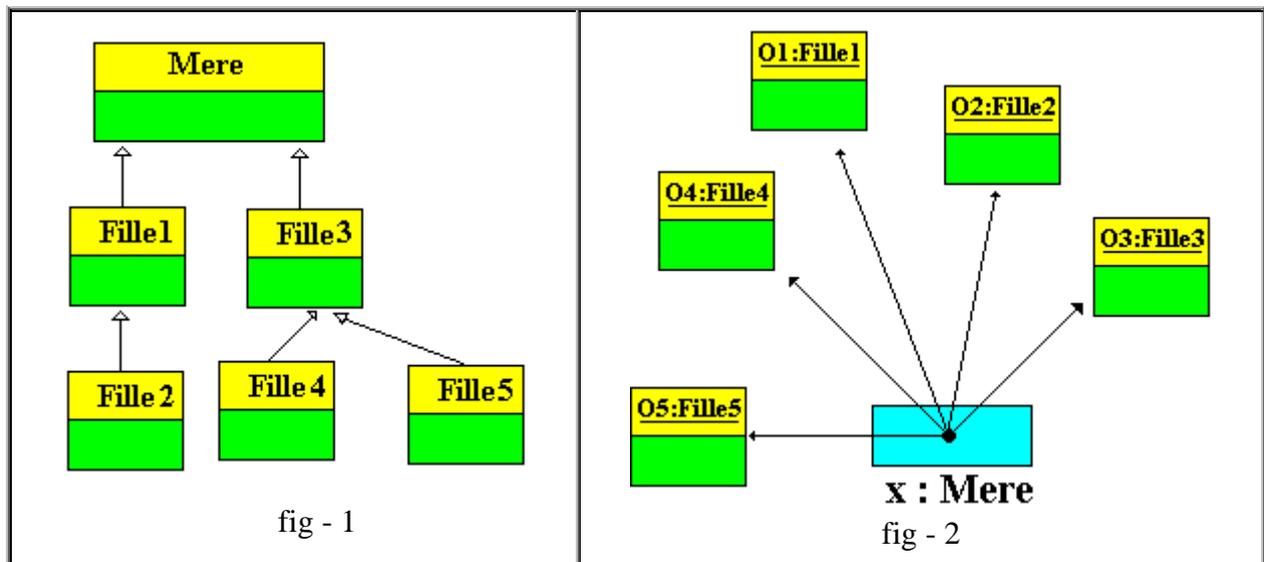


### *L'espace des références*

En C# :

```
Mere x ;
Fille ObjF = new Fille() ;
x = ObjF; // affectation de références du type descendant implicite
```

Nous pouvons en effet dire que **x** peut se référer implicitement à tout objet de classe **Mere** ou de **toute classe héritant** de la classe **Mere**.



Dans la figure fig-1 ci-dessus, une hiérarchie de classes descendant toutes de la classe **Mere**, dans fig-2 ci-contre le schéma montre une référence de type **Mere** qui peut '**pointer**' vers n'importe quel objet de classe descendante (polymorphisme d'objet).

D'une façon générale vous pourrez toujours écrire des affectations entre deux références d'objets :

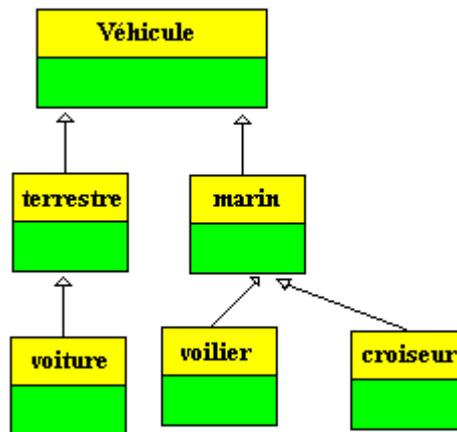
```

En C# :

Classe1 x ;
Classe2 y ;
.....
x = y ;
si et seulement si Classe2 est une classe descendante de Classe1.
    
```

**Exemple pratique tiré du schéma précédent**

1°) Le polymorphisme d'objet est typiquement fait pour représenter des situations pratiques figurées ci-dessous :



Une hiérarchie de classe de véhicules descendant toutes de la classe mère Vehicule, on peut énoncer le fait suivant :

**Un véhicule peut être de plusieurs sortes : soit un croiseur, soit une voiture, soit un véhicule terrestre etc...**

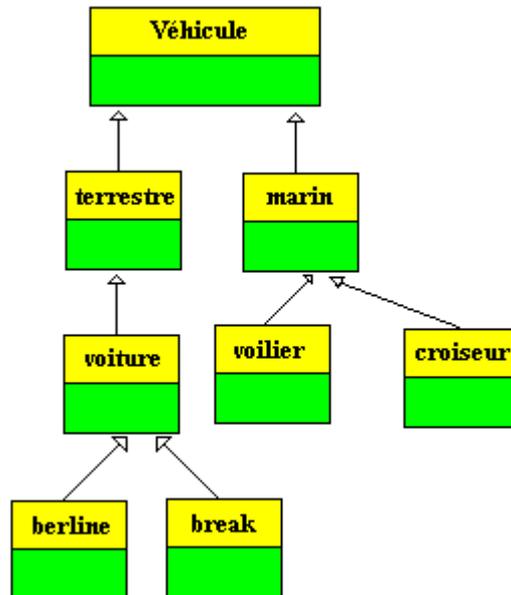
Traduit en termes informatiques, si l'on déclare une référence de type véhicule (**vehicule** x) elle pourra pointer vers n'importe quel objet d'une des classe filles de la classe vehicule.

<pre> En C# :  <b>public class</b> Vehicule { ..... } <b>public class</b> terrestre : Vehicule{ ..... } <b>public class</b> voiture : terrestre { ..... }                 </pre>	<pre> <b>public class</b> marin : Vehicule { ..... }  <b>public class</b> voilier : marin { ..... }  <b>public class</b> croiseur : marin { ..... }                 </pre>
--	--

Mettons en oeuvre la définition du polymorphisme implicite :

*Polymorphisme implicite = création d'objet de classe descendante référencé par une variable parent*

Ajoutons 2 classes à la hiérarchie des véhicules :



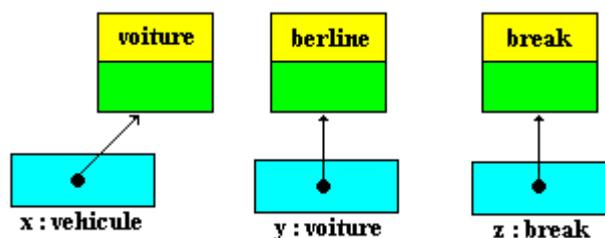
Partons de la situation pratique suivante :

- on crée un **véhicule** du type **voiture**,
- on crée une **voiture** de type **berline**,
- enfin on crée un **break** de type **break**

Traduit en termes informatiques : nous déclarons 3 références **x**, **y** et **z** de type **vehicule**, **voiture** et **break** et nous créons 3 objets de classe **voiture**, **berline** et **break**.

Comme il est possible de créer directement un objet de classe descendante à partir d'une référence de classe mère, nous proposons les instanciations suivantes :

- on crée une **voiture** référencée par la variable de classe **vehicule**,
- on crée une **berline** référencée par la variable de classe **voiture**,
- enfin on crée un **break** référencé par la variable de classe **break**.



<p>En C# :</p> <pre> <b>public class</b> berline : voiture {     ..... }  <b>public class</b> break : voiture {     ..... } </pre>	<pre> <b>public class</b> Fabriquer {     Vehicule x = <b>new</b> voiture ();     voiture y = <b>new</b> berline ();     break z = <b>new</b> break ();     ..... } </pre>
--	--

### 1.3 Polymorphisme d'objet explicite par transtypage

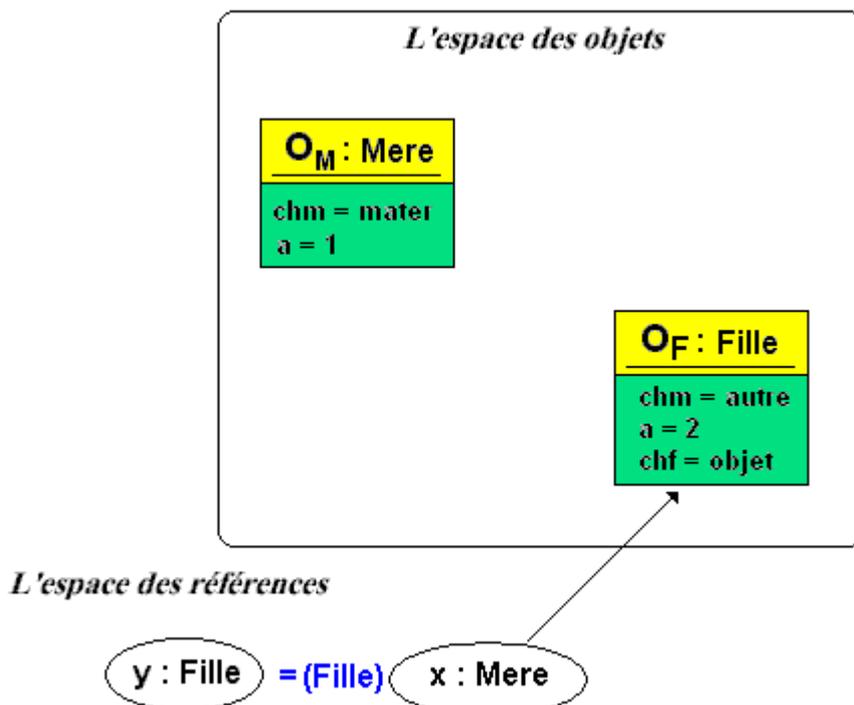
La situation informatique est la suivante :

- on déclare une variable **x** de type Mere,
- on déclare une variable **y** de type Fille héritant de Mere,
- on instancie la variable **x** dans le type descendant Fille (polymorphisme implicite).

Il est alors possible de faire "pointer" la variable **y** (de type Fille) vers l'objet (de type Fille) auquel se réfère **x** en effectuant une affectation de références :

**y = x** ne sera pas acceptée directement car statiquement les variables **x** et **y** ne sont pas du même type, il faut indiquer au compilateur que l'on souhaite temporairement changer le type de la variable **x** afin de pouvoir effectuer l'affectation.

Cette opération de changement temporaire, se dénomme le **transtypage** (notée en C# : **y = (Fille)x**) :

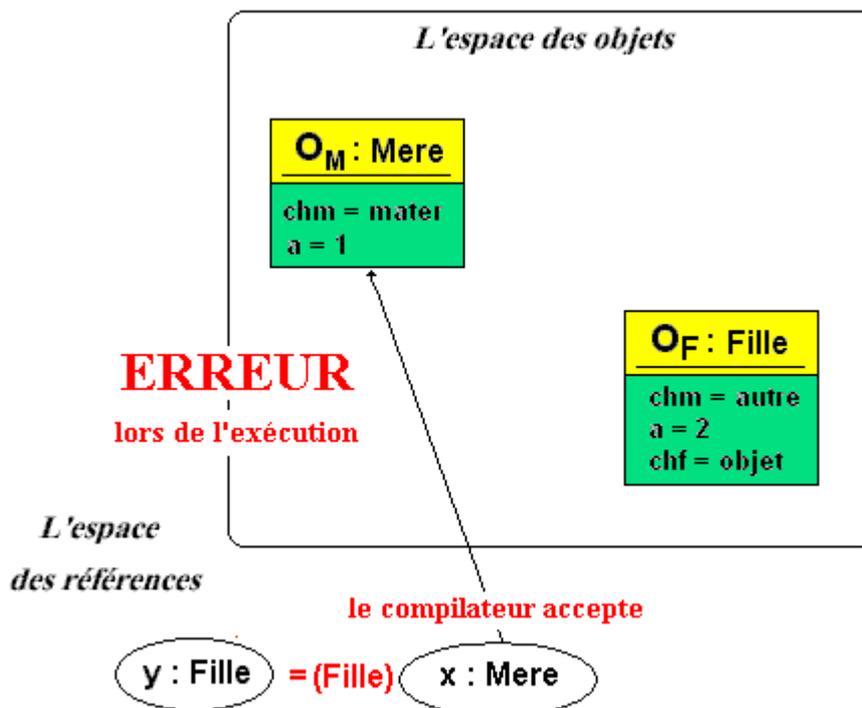


En C# :

```
Mere x ;  
Fille y ;  
Fille ObjF = new Fille() ;  
x = ObjF ; // x pointe vers un objet de type Fille  
y = (Fille) x ; // transtypage et affectation de références du type ascendant explicite compatible dynamiquement.
```

## Attention

- La validité du transtypage n'est pas vérifiée statiquement par le compilateur, donc si votre variable de référence pointe vers un objet qui n'a pas la même nature que l'opérateur de transtypage, c'est lors de l'exécution qu'il y aura production d'un message d'erreur indiquant le transtypage impossible.
- Il est donc impératif de tester l'appartenance à la bonne classe de l'objet à transtyper, les langages C#, Delphi et Java disposent d'un opérateur permettant de tester cette appartenance ou plutôt l'appartenance à une hiérarchie de classes (opérateur **is** en C#).
- L'opérateur **as** est un opérateur de transtypage de référence d'objet semblable à l'opérateur **( )**. L'opérateur **as** fournit la valeur **null** en cas d'échec de conversion alors que l'opérateur **( )** lève une exception.

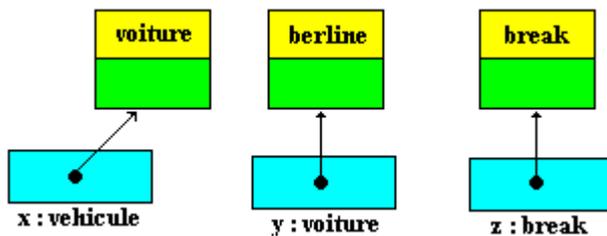


```

En C# :
Mere x ;
Fille y ;
x = new Mere() ; // instantiation dans le type initial
{ affectation de références du type ascendant explicite mais dangereuse si x est uniquement Mere : }
y = (Fille)x ; <--- erreur lors de l'exécution ici
{affectation acceptée statiquement mais refusée dynamiquement, car x pointe vers un objet de type Mere }

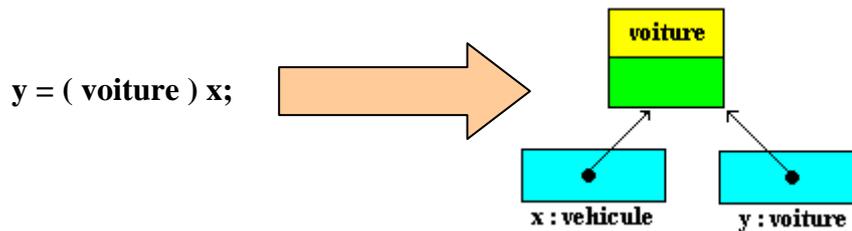
```

En reprenant l'exemple pratique précédant de la hiérarchie des véhicules :



Puisque x pointe vers un objet de type voiture toute variable de référence acceptera de pointer vers cet objet, en particulier la variable voiture après transtypage de la référence de x.

En C# l'affectation s'écrirait par application de l'opérateur de transtypage :



Pour pallier à cet inconvénient de programmation pouvant lever des exceptions lors de l'exécution, C# offre au développeur la possibilité de tester l'appartenance d'un objet référencé par une variable quelconque à une classe ou plutôt une hiérarchie de classe ; en C# cet opérateur se dénote **is** :

**L'opérateur "is" de C# est identique à celui de Delphi :**

L'opérateur **is**, qui effectue une vérification de type dynamique, est utilisé pour vérifier quelle est effectivement la classe d'un objet à l'exécution.

L'expression : objet **is** classeT

renvoie True si objet est une instance de la classe désignée par **classeT** ou de l'un de ses **descendants**, et False sinon. Si objet a la valeur nil, le résultat est False.

```

En C# :
Mere x ;
Fille y ;
x = new Mere( ) ; // instanciation dans le type initial
if ( x is Fille) // test d'appartenance de l'objet référencé par x à la bonne classe
    y = (Fille)x ;

```

## 1.4 Utilisation pratique du polymorphisme d'objet

**Le polymorphisme d'objet associé au transtypage est très utile dans les paramètres des méthodes.**

Lorsque vous déclarez une méthode meth avec un paramètre formel x de type ClasseT :

```

void meth ( ClasseT x )
{
    .....
}

```

Vous pouvez utiliser lors de l'appel de la méthode meth n'importe quel paramètre effectif de ClasseT ou bien d'une quelconque classe descendant de ClasseT et ensuite à l'intérieur de la procédure vous transtypez le paramètre. Cet aspect est utilisé en particulier en C# lors de la création de gestionnaires d'événements communs à plusieurs composants :

```

private void meth1(object sender, System.EventArgs e) {
    if (sender is System.Windows.Forms.TextBox)
        (sender as TextBox).Text="Fin";
    else if (sender is System.Windows.Forms.Label)
        (sender as Label).Text="ok";

    // ou encore :

    if (sender is System.Windows.Forms.TextBox)
        ((TextBox)sender).Text="Fin";

    else if (sender is System.Windows.Forms.Label)
        ((Label)sender).Text="ok";
}

```

Autre exemple avec une méthode personnelle nommée meth2 sur la hiérarchie précédente des véhicules :

```

private void meth2 ( vehicule Sender );
{
    if (Sender is voiture)
        ((voiture)Sender). ..... ;
    else if (Sender is voilier)
        ((voilier)Sender). ..... ;
    .....
}

```

## instanciation dans un type ascendant (*impossible en C#*)

- Il s'agit ici d'une utilisation non licite qui n'est pas commune à tous les langages LOO.
  - Le compilateur **C#** comme le compilateur **Java**, **refuse** ce type de création d'objet, les compilateurs **C++** et **Delphi** **acceptent** ce genre d'instanciation en laissant au programmeur le soin de se débrouiller avec les problèmes de cohérence lorsqu'ils apparaîtront.

# Polymorphisme de méthode en C#.net

---

Plan général: 

## 1. Le polymorphisme de méthodes en C#

*Rappel* des notions de base

- 1.1 Surcharge et redéfinition en C#
- 1.2 Liaison statique et masquage en C#
- 1.3 Liaison dynamique en C#
- 1.4 Comment opère le compilateur

Résumé pratique

## 2. Accès à la super classe en C#

- 2.1 Le mot clef base
- 2.2 Initialiseur de constructeur this et base
- 2.3 Comparaison C#, Delphi et Java sur un exemple
- 2.4 Traitement d'un exercice complet
- 2.5 Destructeur-finaliseur

## 1. Le polymorphisme de méthode en C#

Nous avons vu au chapitre précédent le polymorphisme d'objet, les méthodes peuvent être elles aussi polymorphes. Nous voyons ici comment C# hérite pour une bonne part de Delphi pour ses comportements sur le polymorphisme d'objet et de méthode, et de la souplesse de Java pour l'écriture.

### Rappel de base

#### Polymorphisme par héritage de méthode

Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires **nouvelles** peuvent être implémentées dans la classe enfant, mais aussi des méthodes des **parents substituées** pour obtenir des implémentations différentes.

L'objectif visé en terme de qualité du logiciel est la **réutilisabilité** en particulier lorsque l'on réalise une même opération sur des éléments différents :

opération = **ouvrir** ( )  
**ouvrir** une fenêtre de texte, **ouvrir** un fichier, **ouvrir** une image etc ...

#### Surcharge et redéfinition avec C#

En informatique ce vocable s'applique aux méthodes selon leur degré d'adaptabilité, nous distinguons alors deux dénominations :

- le polymorphisme statique ou la **surcharge** de méthode
- le polymorphisme dynamique ou la **redéfinition** de méthode ou encore la surcharge héritée.

### 1.1 Surcharge

La surcharge de méthode (*polymorphisme statique de méthode*) est une fonctionnalité classique des langages très évolués et en particulier des langages orientés objet dont C# fait partie; elle consiste dans le fait qu'une classe peut disposer de **plusieurs méthodes ayant le même nom**, mais avec des paramètres formels différents ou éventuellement un type de retour différent. On dit alors que ces méthodes n'ont pas la même signature

On rappelle que la **signature** d'une méthode est formée par l'**en-tête** de la méthode avec ses **paramètres formels** et leur **type**.

Nous avons déjà utilisé cette fonctionnalité précédemment dans le paragraphe sur les constructeurs, où la classe **Un** disposait de quatre constructeurs surchargés (quatre **signatures** différentes du constructeur) :

```
class Un
{
    int a;
    public Un ( )
    { a = 100; }

    public Un (int b )
```

```

{ a = b; }

public Un (float b)
{ a = (int)b; }

public Un (int x , float y) : this(y)
{ a += 100; }
}

```

Mais cette surcharge est possible aussi pour n'importe quelle méthode de la classe autre que le constructeur.



Ci-contre deux exemplaires surchargés de la méthode ouvrir dans la classe **PortesEtFenêtres** :

Le compilateur n'éprouve aucune difficulté lorsqu'il rencontre un appel à l'une des versions surchargée d'une méthode, il cherche dans la déclaration de toutes les surcharges celle dont la **signature** (la déclaration des paramètres formels) coïncide avec les paramètres effectifs de l'appel.

### Remarque :

Le polymorphisme statique (ou **surcharge**) de C# est syntaxiquement semblable à celui de Java.

Programme C# exécutable	Explications
<pre> class Un { public int a;   public Un (int b)   { a = b; }   public void f ( )   { a *=10; }   public void f ( int x )   { a +=10*x; }   public int f ( int x, char y )   { a = x+(int)y;     return a; } } class AppliSurcharge {   public static void Main(String [ ] arg) {     Un obj = new Un(15);     System.Console.WriteLine("&lt;création&gt; a="+obj.a);     obj.f();     System.Console.WriteLine("&lt;obj.f()&gt; a="+obj.a);     obj.f(2);     System.Console.WriteLine("&lt;obj.f()&gt; a="+obj.a);     obj.f(50,'a');     System.Console.WriteLine("&lt;obj.f()&gt; a="+obj.a);   } } </pre>	<p>La méthode <b>f</b> de la classe <b>Un</b> est surchargée trois fois :</p> <pre> public void f ( ) { a *=10; }  public void f ( int x ) { a +=10*x; }  public int f ( int x, char y ) { a = x+(int)y;   return a; } </pre> <p>La méthode <b>f</b> de la classe <b>Un</b> peut donc être appelée par un objet instancié de cette classe sous l'une quelconque des trois formes :</p> <ul style="list-style-type: none"> <li>obj.f( ); pas de paramètre =&gt; choix : <b>void f ( )</b></li> <li>obj.f(2); paramètre <b>int</b> =&gt; choix : <b>void f ( int x )</b></li> <li>obj.f(50,'a'); deux paramètres, un int un char =&gt; choix : <b>int f ( int x, char y )</b></li> </ul>

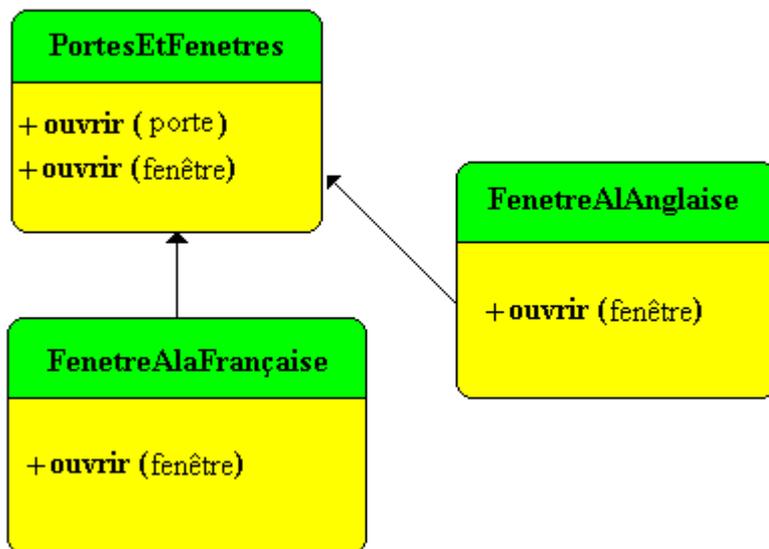
### Comparaison Delphi - C# sur la surcharge :

Delphi	C#
<pre> <b>Un = class</b>   a : integer; <b>public</b>   <b>constructor</b> methode( b : integer );   <b>procedure</b> f;<b>overload</b>;   <b>procedure</b> f(x:integer);<b>overload</b>;   <b>function</b> f(x:integer;y:char):integer;<b>overload</b>; <b>end</b>;  <b>implementation</b>  <b>constructor</b> Un.methode( b : integer ); <b>begin</b>   a:=b <b>end</b>; <b>procedure</b> Un.f; <b>begin</b>   a:=a*10; <b>end</b>; <b>procedure</b> Un.f(x:integer); <b>begin</b>   a:=a+10*x <b>end</b>; <b>function</b> Un.f(x:integer;y:char):integer; <b>begin</b>   a:=x+ord(y);   result:= a <b>end</b>;  <b>procedure</b> Main; <b>var</b> obj:Un; <b>begin</b>   obj:=Un.methode(15);   obj.f;   Memo1.Lines.Add('obj.f='+inttostr(obj.a));   obj.f(2);   Memo1.Lines.Add('obj.f(2)='+inttostr(obj.a));   obj.f(50,'a');   Memo1.Lines.Add('obj.f(50,"a")='+inttostr(obj.a)); <b>end</b>; </pre>	<pre> <b>class</b> Un {   <b>public</b> int a;   <b>public</b> Un (int b )   { a = b;   }    <b>public void</b> f ( )   { a *=10;   }    <b>public void</b> f ( int x )   { a +=10*x;   }    <b>public int</b> f ( int x, char y )   { a = x+(int)y;     <b>return</b> a;   } }  <b>class</b> AppliSurcharge {   <b>public static void</b> Main(String [ ] arg) {     Un obj = <b>new</b> Un(15);     System.Console.WriteLine("&lt;création&gt; a =" +obj.a);     obj.f();     System.Console.WriteLine("&lt;obj.f()&gt; a =" +obj.a);     obj.f(2);     System.Console.WriteLine("&lt;obj.f()&gt; a =" +obj.a);     obj.f(50,'a');     System.Console.WriteLine("&lt;obj.f()&gt; a =" +obj.a);   } } </pre>

## Redéfinition

La redéfinition de méthode (ou polymorphisme dynamique) est spécifique aux langages orientés objet. Elle est mise en oeuvre lors de l'héritage d'une classe mère vers une classe fille dans le cas d'une méthode ayant la même signature dans les deux classes. Dans ce cas les actions dûes à l'appel de la méthode, dépendent du code inhérent à chaque version de la méthode (celle de la classe mère, ou bien celle de la classe fille).

Dans l'exemple ci-dessous, nous supposons que dans la classe **PortesEtFenêtres** la méthode ouvrir(fenetre) explique le mode opératoire général d'ouverture d'une fenêtre, il est clair que dans les deux classes descendantes l'on doit "redéfinir" le mode opératoire selon que l'on est en présence d'une fenêtre à la française, ou d'une fenêtre à l'anglaise :



### Redéfinition et répartition des méthodes en C#

La redéfinition de méthode peut être selon les langages :

- **précoce**  
et/ou
- **tardive**

Ces deux actions sont différentes selon que le compilateur du langage met en place la liaison du code de la méthode immédiatement lors de la compilation (**liaison statique** ou **précoce**) ou bien lorsque le code est lié lors de l'exécution (**liaison dynamique** ou **tardive**). Ce phénomène se dénomme la répartition des méthodes.

Le terme de répartition fait référence à la façon dont un programme **détermine où il doit rechercher** une méthode lorsqu'il rencontre un appel à cette méthode.

Le code qui appelle une méthode ressemble à un appel classique de méthode. Mais les classes ont des façons différentes de répartir les méthodes.

Le langage **C#** supporte d'une manière **identique à Delphi**, ces deux modes de liaison du code, la **liaison statique** étant comme en Delphi le mode **par défaut**.

Le développeur Java sera plus décontenancé sur ce sujet, car la liaison statique en Java n'existe que pour les méthodes de classe **static** ou bien qualifiée de **final**, de plus **la liaison du code par défaut est dynamique en Java**.

Donc en C# comme en Delphi, des mots clefs comme **virtual** et **override** sont nécessaires pour la redéfinition de méthode, ils sont utilisés strictement de la même manière qu'en Delphi.

## 1.2 Liaison statique et masquage en C#

Toute méthode C# qui n'est précédée d'aucun des deux qualificateurs **virtual** ou **override** est à liaison **statique**.

Le compilateur détermine l'adresse exacte de la méthode et lie la méthode au moment de la compilation.

L'avantage principal des méthodes statiques est que leur répartition est très rapide. Comme le compilateur peut déterminer l'adresse exacte de la méthode, il la lie directement (les méthodes virtuelles, au contraire, utilisent un moyen indirect pour récupérer l'adresse des méthodes à l'exécution, moyen qui nécessite plus de temps).

Une méthode statique ne change pas lorsqu'elle est transmise en héritage à une autre classe. Si vous déclarez une classe qui inclut une méthode statique, puis en dérivez une nouvelle classe, la classe dérivée partage exactement la même méthode située à la même adresse. Cela signifie qu'il est **impossible de redéfinir** les méthodes statiques; une méthode statique fait toujours exactement la même chose, quelque soit la classe dans laquelle elle est appelée.

Si vous déclarez dans une classe dérivée une méthode ayant le même nom qu'une méthode statique de la classe ancêtre, la nouvelle méthode **remplace simplement** (on dit aussi **masque**) la méthode héritée dans la classe dérivée.

### Comparaison masquage en Delphi et C# :

Delphi	C#
<pre>type ClasseMere = class   x : integer;   procedure f (a:integer); end;  ClasseFille = class ( ClasseMere )   y : integer;   procedure f (a:integer);//masquage end;  implementation  procedure ClasseMere.f (a:integer); begin... end;  procedure ClasseFille.f (a:integer); begin... end;</pre>	<pre>public class ClasseMere {   public int x = 10;    public void f ( int a)   { x +=a; } }  public class ClasseFille : ClasseMere {   int y = 20;   public void f ( int a) //masquage   { x +=a*10+y; } }</pre>

### Remarque importante :

L'expérience montre que les étudiants comprennent immédiatement le masquage lorsque le polymorphisme d'objet n'est pas présent. Ci-dessous un exemple de classe UtiliseMereFille qui instancie et utilise dans le même type un objet de classe ClasseMere et un objet de classe ClasseFille :

```

public class ClasseMere
{
    public int x = 1;
    public void meth ( int a)
    {
        x += a ;
    }
}
class ClasseFille : ClasseMere
{
    public void meth ( int a) //masquage
    {
        x += a*100 ;
    }
}

public class UtiliseMereFille
{
    public static void Main (string [ ] args)
    {
        ClasseMere M ;
        ClasseFille F ;
        M = new ClasseMere ( ) ;
        F = new ClasseFille ( ) ;
        M.meth (10) ;
        F.meth (10) ;
    }
}

```

- Lors de la compilation de l'instruction **M.meth(10)**, c'est le code de la méthode **meth** de la classe **ClasseMere** qui est lié avec comme paramètre par valeur 10; ce qui donnera la valeur 11 au champ x de l'objet M.
- Lors de la compilation de l'instruction **F.meth(10)**, c'est le code de la méthode **meth** de la classe **ClasseFille** qui masque celui de la classe parent et qui est donc lié avec comme paramètre par valeur 10; ce qui donnera la valeur 101 au champ x de l'objet F.

Pour bien comprendre toute la portée du masquage statique et les risques de mauvaises interprétations, il faut étudier le même exemple légèrement modifié en incluant le cas du polymorphisme d'objet, plus précisément le polymorphisme d'objet implicite.

Dans l'exemple précédent nousinstancions la variable **ClasseMere** M en un objet de classe **ClasseFille** (*polymorphisme implicite d'objet*) soient les instructions

```

ClasseMere M ;
M = new ClasseFille ( ) ;

```

Une **erreur courante** est de croire que dans ces conditions, dans l'instruction **M.meth(10)** c'est la méthode **meth(int a)** de la classe **ClasseFille** (en particulier si l'on ne connaît que Java qui ne pratique pas le masquage) :

```

public class ClasseMere
{
    int x = 1;
    public void meth ( int a)
    {
        x += a ;
    }
}
class ClasseFille : ClasseMere
{
    public void meth ( int a) //masquage
    {
        x += a*100 ;
    }
}

public class UtiliseMereFille
{
    public static void Main (string [ ] args)
    {
        ClasseMere M ;
        ClasseFille F ;
        M = new ClasseFille ( ) ;
        F = new ClasseFille ( ) ;
        M.meth (10) ;
        F.meth (10) ;
    }
}

```

**ERREUR**

Que fait alors le compilateur C# dans ce cas ? : il réalise une liaison statique :

- Lors de la compilation de l'instruction **M.meth(10)**, c'est le code de la méthode **meth(int a)** de la classe ClasseMere qui est lié, car la référence M a été déclarée de type ClasseMere et peu importe dans quelle classe elle a été instanciée ( avec comme paramètre par valeur 10; ce qui donnera la valeur 11 au champ x de l'objet M).
- Lors de la compilation de l'instruction **F.meth(10)**, c'est le code de la méthode **meth** de la classe ClasseFille comme dans l'exemple précédent (avec comme paramètre par valeur 10; ce qui donnera la valeur 101 au champ x de l'objet F).

Voici la bonne configuration de liaison effectuée lors de la compilation :

```

public class ClasseMere
{
    int x = 1;
    public void meth ( int a)
    {
        x += a ;
    }
}
class ClasseFille : ClasseMere
{
    public void meth ( int a) //masquage
    {
        x += a*100 ;
    }
}

public class UtiliseMereFille
{
    public static void Main (string [ ] args)
    {
        ClasseMere M ;
        ClasseFille F ;
        M = new ClasseFille ( ) ;
        F = new ClasseFille ( ) ;
        M.meth (10) ;
        F.meth (10) ;
    }
}

```

Afin que le programmeur soit bien conscient d'un effet de masquage d'une méthode héritée par une méthode locale, le compilateur C# envoie, comme le compilateur Delphi, un message d'avertissement indiquant une possibilité de **manque de cohérence sémantique** ou un **masquage**.

S'il s'agit d'un masquage voulu, le petit plus apporté par le langage C# est la proposition que vous fait le compilateur de l'utilisation optionnelle du mot clef **new** qualifiant la nouvelle méthode masquant la méthode parent. Cette écriture améliore la lisibilité du programme et permet de se rendre compte que l'on travaille avec une liaison statique. Ci-dessous deux écritures équivalentes du masquage de la méthode meth de la classe ClasseMere :

masquage C#	masquage C# avec new
<pre> public class ClasseMere {     public int x = 10;     public void meth ( int a) //liaison statique     { x +=a; } }  public class ClasseFille : ClasseMere {     public void meth ( int a) //masquage     { x +=a*10+y; } } </pre>	<pre> public class ClasseMere {     public int x = 10;     public void meth ( int a) //liaison statique     { x +=a; } }  public class ClasseFille : ClasseMere {     public new void meth ( int a) //masquage     { x +=a*10+y; } } </pre>

L'exemple ci-dessous récapitule les notions de **masquage** et de **surcharge** en C# :

```

public class ClasseMere {
public int x = 1;
public void meth1 ( int a ) { x += a ; }
public void meth1 ( int a , int b ) { x += a*b ; }
}
public class ClasseFille : ClasseMere {
public new void meth1 ( int a ) { x += a*100 ; }
public void meth1 ( int a , int b , int c ) { x += a*b*c ; }
}

public class UtiliseMereFille {
public static void Main (string [ ] args) {
ClasseMere M ;
ClasseFille F ;
M = new ClasseFille ( ) ;
F = new ClasseFille ( ) ;
M.meth1 (10) ; <--- meth1(int a) de ClasseMere
M.meth1 (10,5) ; <--- meth1(int a, int b) de ClasseMere
M.meth1 (10,5,2) ; <--- erreur! n'existe pas dans ClasseMere .
F.meth1 (10) ; <--- meth1(int a) de ClasseFille
F.meth1 (10,5) ; <--- meth1(int a, int b) de ClasseFille
F.meth1 (10,5,2) ; <--- meth1(int a, int b, int c) de ClasseFille
}
}

```

### 1.3 Liaison dynamique (ou redéfinition) en C#

Toute méthode C# qui est précédée par l'un des deux qualificatifs **virtual** ou **override** est à liaison **dynamique**, on dit aussi que c'est une méthode virtuelle.

Le compilateur ne détermine pas l'adresse exacte de la méthode et lie la méthode au moment de l'exécution. (le CLR utilise un moyen indirect comme une table pour récupérer l'adresse des méthodes lors de l'exécution).

Une méthode "f" de classe fille qui **redéfinit** la méthode virtuelle "f" de classe mère, **doit avoir la même signature** que la méthode "f" de classe mère qu'elle redéfinit.

Dans l'exemple ci-dessous la classe ClasseFille qui hérite de la classe ClasseMere, redéfinit la méthode virtuelle f de sa classe mère :

#### Comparaison redéfinition Delphi et C# :

Delphi	C#
<pre> type ClasseMere = class x : integer; <b>procedure</b> f (a:integer);<b>virtual</b>;//déclaration <b>procedure</b> g(a,b:integer); end;  ClasseFille = class ( ClasseMere ) y : integer; </pre>	<pre> class ClasseMere { <b>public int</b> x = 10;  <b>public virtual void</b> f ( int a ) //déclaration { x +=a; } <b>void</b> g ( int a, int b) { x +=a*b; } } </pre>

<pre> <b>procedure</b> f (a:integer);<b>override</b>;<i>//redéfinition</i> <b>procedure</b> g1(a,b:integer); <b>end</b>;  <b>implementation</b>  <b>procedure</b> ClasseMere.f (a:integer); <b>begin...</b> <b>end</b>; <b>procedure</b> ClasseMere.g(a,b:integer); <b>begin...</b> <b>end</b>; <b>procedure</b> ClasseFille.f (a:integer); <b>begin...</b> <b>end</b>; <b>procedure</b> ClasseFille.g1(a,b:integer); <b>begin...</b> <b>end</b>; </pre>	<pre> <b>class</b> ClasseFille : ClasseMere {   <b>int</b> y = 20;   <b>public override void</b> f ( <b>int</b> a) <i>//redéfinition</i>   { x +=a; }   <b>void</b> g1 (<b>int</b> a, <b>int</b> b) <i>//nouvelle méthode</i>   { ..... } } </pre>
--	--

Comme delphi, C# peut combiner la surcharge et la redéfinition sur une même méthode, c'est pourquoi nous pouvons parler de surcharge héritée :

C#
<pre> <b>class</b> ClasseMere {   <b>public int</b> x = 10;    <b>public virtual void</b> f ( <b>int</b> a) <i>// déclaration de f et signature initiale</i>   { x +=a; } }  <b>class</b> ClasseFille : ClasseMere {   <b>int</b> y = 20;   <b>public override void</b> f ( <b>int</b> a) <i>//redéfinition de f et même signature</i>   { x +=a; }   <b>public virtual void</b> f (<b>char</b> b) <i>//nouvelle déclaration par surcharge de f, signature différente</i>   { x +=b*y; } } </pre>

#### 1.4 Comment opère le compilateur C# en liaison dynamique

C'est le compilateur C# qui fait tout le travail de recherche de la bonne méthode. Prenons un objet obj de classe Classe1, lorsque le compilateur C# trouve une instruction du genre "obj.**method1**(paramètres effectifs);", sa démarche d'analyse est semblable à celle du compilateur Delphi, il cherche dans l'ordre suivant :

- Y-a-t-il dans Classe1, une méthode qui se nomme **method1** ayant une signature identique aux paramètres effectifs ?
- si oui c'est la méthode ayant cette signature qui est appelée,
- si non le compilateur remonte dans la hiérarchie des classes mères de Classe1 en posant la même question récursivement jusqu'à ce qu'il termine sur la classe Object.
- Si aucune méthode ayant cette signature n'est trouvée il signale une erreur.

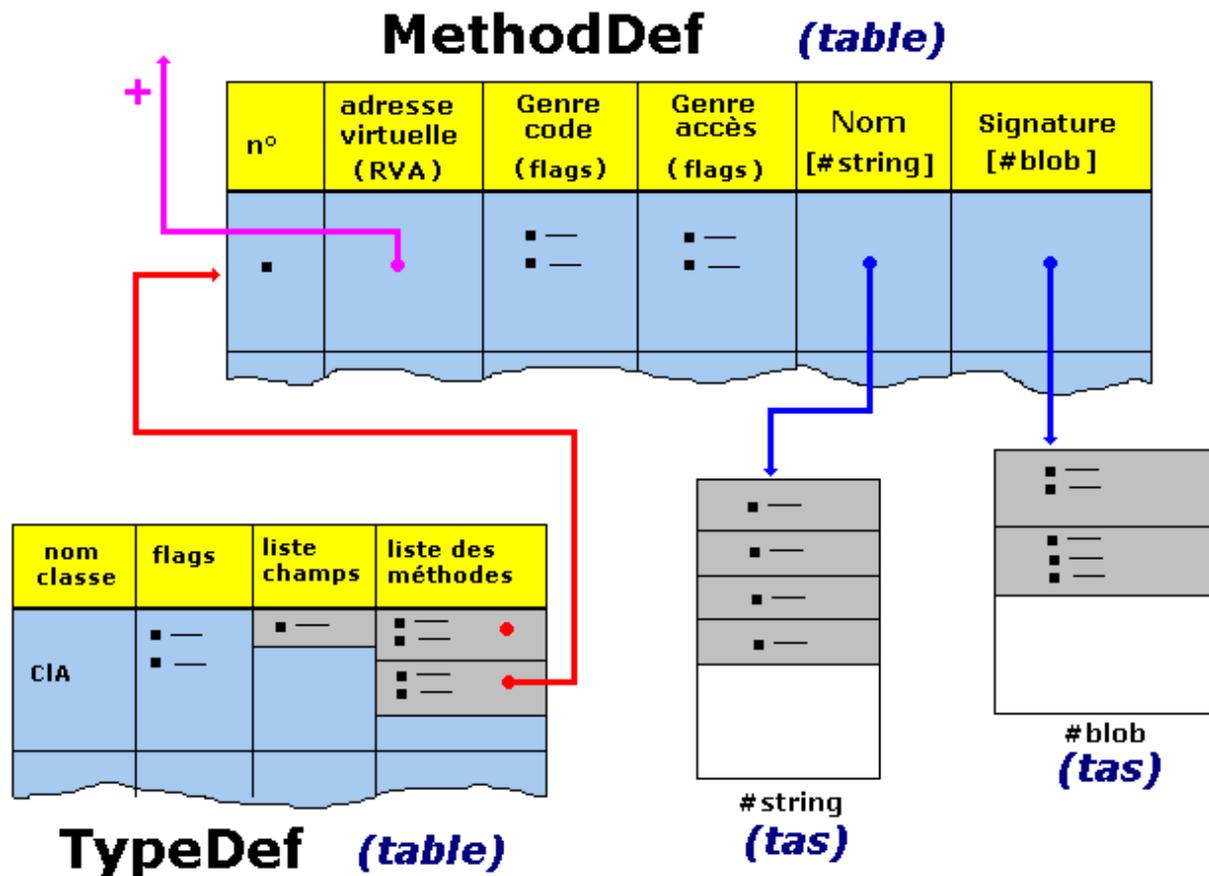
Soit à partir de l'exemple précédent les instructions suivantes :  
 ClasseFille obj = **new** ClasseFille( );  
 obj.g(-3,8);  
 obj.g('h');

Le compilateur C# applique la démarche d'analyse décrite, à l'instruction "obj.g(-3,8);". Ne trouvant pas dans ClasseFille de méthode ayant la bonne signature (signature = deux entiers), le compilateur remonte dans la classe mère ClasseMere et trouve une méthode " **void** g ( **int** a, **int** b )" de la classe ClasseMere ayant la bonne signature (signature = deux entiers), lors de l'exécution, la machine virtuelle CLR procède alors à l'appel de cette méthode sur les paramètres effectifs (-3,8).

Dans le cas de l'instruction obj.g('h');, le compilateur trouve immédiatement dans ClasseFille la méthode " **void** g (**char** b )" ayant la bonne signature, c'est donc elle qui est appelée sur le paramètre effectif 'h'.

Le compilateur et le CLR consultent les méta-données (informations de description) de l'assemblage en cours ( applicationXXX.exe ), plus particulièrement les métadonnées de type qui sont stockées au fur et à mesure dans de nombreuses tables.

Nous figurons ci-dessous deux tables de définition importantes relativement au polymorphisme de méthode **MethodDef** et **TypeDef** utilisées par le compilateur.



## Résumé pratique sur le polymorphisme en C#

La **surcharge** (polymorphisme statique) consiste à proposer différentes signatures de la même méthode.

La **redéfinition** (polymorphisme dynamique) ne se produit que dans l'héritage d'une classe, par redéfinition (**liaison dynamique**) de la méthode mère avec une méthode fille (ayant ou n'ayant pas la même signature).

Le **masquage** ne se produit que dans l'héritage d'une classe, par redéfinition (**liaison statique**) de la méthode mère par une méthode fille (ayant la même signature).

Toute méthode est considérée à **liaison statique** sauf si vous la déclarez autrement.

## 2. Accès à la super classe en C#

### 2.1 Le mot clef ' base '

Nous venons de voir que le compilateur s'arrête dès qu'il trouve une méthode ayant la bonne signature dans la hiérarchie des classes, il est des cas où nous voudrions accéder à une méthode de la classe mère alors que celle-ci est redéfinie dans la classe fille. C'est un problème analogue à l'utilisation du **this** lors du masquage d'un attribut.

classe mère	classe fille
<pre>class ClasseA {     public int attrA ;     private int attrXA ;      public void meth01 () {         attrA = 57 ;     } }</pre>	<pre>class ClasseB : ClasseA {     public new void meth01 () {         attrA = 1000 ;     }     public void meth02 () {         meth01 () ;     } }</pre>

La méthode `meth02 ()` invoque la méthode `meth01 ()` de la classe `ClasseB`. Il est impossible de faire directement appel à la méthode `meth01 ()` de la classe mère `ClasseA` car celle-ci est masquée dans la classe fille.

```

class ClasseB : ClasseA
{
    public new void meth01 () {
        attrA = 1000 ;
    }
    public void meth02 () {
        meth01 ();
    }
}

```

Il existe en C# un mécanisme déclenché par un mot clef qui permet d'accéder à la classe mère (classe immédiatement au dessus): ce mot est **base**.

Le mot clef **base** fonctionne comme une référence d'objet et il est utilisé pour accéder à **tous les membres visibles de la classe mère** à partir d'une classe fille dérivée directement de cette classe mère (la super-classe en Java).

Ce mot clef **base** est très semblable au mot clef **inherited** de Delphi qui joue le même rôle sur les méthodes et les propriétés (il est en fait plus proche du mot clef **super** de Java car il ne remonte qu'à la classe mère), il permet l'appel d'une méthode de la classe de base qui a été substituée (masquée ou redéfinie) par une autre méthode dans la classe fille.

*Exemple :*

```

class ClasseA
{
    public int attrA ;
    private int attrXA ;

    public void meth01 () {
        attrA = 57 ;
    }
}

class ClasseB : ClasseA
{
    public new void meth01 () {
        attrA = 1000 ;
    }
    public void meth02 () {
        base.meth01 ();
        meth01 ();
    }
}

```

**Remarques :**

- Le fait d'utiliser le mot clef **base** à partir d'une méthode statique constitue une erreur.
- **base** est utile pour spécifier un constructeur de classe mère lors de la création d'instances de la classe fille.

Nous développons ci-dessous l'utilisation du mot clef base afin d'initialiser un constructeur.

## 2.2 Initialiseur de constructeur *this* et *base*

Semblablement à Delphi et à Java, tous les constructeurs d'instance C# autorisent l'appel d'un autre constructeur d'instance immédiatement avant le corps du constructeur, cet appel est dénommé l'initialiseur du constructeur, en Delphi cet appel doit être explicite, en C# et en Java cet appel peut être implicite.

Rappelons que comme en Java où dans toute classe ne contenant aucun constructeur, en C# **un constructeur sans paramètres par défaut** est implicitement défini :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :
<pre>class ClasseA {     public int attrA ;     public string attrStrA ; }</pre>	<pre>class ClasseA {     public int attrA ;     public string attrStrA ;     public ClasseA () {     } }</pre>

### Remarque :

Lors de l'héritage d'une classe fille, différemment à Delphi et à Java, si un constructeur d'instance C# de la classe fille **ne fait pas figurer explicitement** d'initialiseur de constructeur, c'est qu'en fait un initialiseur de constructeur ayant la forme **base()** lui a été fourni **implicitement**.

Soit par exemple une classe ClasseA possédant 2 constructeurs :

```
class ClasseA {  
    public int attrA ;  
    public string attrStrA ;  
  
    public ClasseA () { /* premier constructeur */  
        attrA = 57 ;  
    }  
    public ClasseA ( string s ) { /* second constructeur */  
        attrStrA = s + "...1..." ;  
    }  
}
```

Soit par suite une classe fille ClasseB dérivant de ClasseA possédant elle aussi 2 constructeurs, les deux déclarations ci-dessous sont équivalentes :

Initialiseur implicite	Initialiseur explicite équivalent
------------------------	-----------------------------------

<pre> class ClasseB : ClasseA {      /* premier constructeur */     public ClasseB () {         attrStrA = "...";     }      /* second constructeur */     public ClasseB ( string s ) {         attrStrA = s ;     } } </pre>	<pre> class ClasseB : ClasseA {      /* premier constructeur */     public ClasseB () : base() {         attrStrA = "...";     }      /* second constructeur */     public ClasseB ( string s ) : base() {         attrStrA = s ;     } } </pre>
--	--

Dans les deux cas le corps du constructeur de la classe fille est initialisé par un premier appel au constructeur de la classe mère (), en l'occurrence << **public ClasseA () ... /\* premier constructeur \*/>>**

### Initialiseur base() avant le corps

L'appel explicite au constructeur sans paramètre de classe mère doit obligatoirement être mis à la suite de l'en-tête du constructeur **de classe fille** (on doit d'abord construire l'objet comme étant de classe mère et ensuite on continue sa construction comme un objet de classe fille).

### Remarque :

Dans une classe fille, en C# comme en Java, **tout constructeur de la classe fille appelle implicitement** et automatiquement le constructeur par défaut (celui qui est sans paramètres) de la classe mère.

### Exemple :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :
<pre> class ClasseA {     public int attrA ;     public string attrStrA ; }  class ClasseB : ClasseA {  } </pre> <p>Le constructeur de ClasseA sans paramètres est implicitement déclaré par le compilateur.</p>	<pre> class ClasseA {     public int attrA ;     public string attrStrA ;      public ClasseA () {     } }  class ClasseB : ClasseA {     public ClasseB () : base() {     } } </pre>

Si la **classe mère ne possède pas de constructeur par défaut**, le compilateur engendre un message d'erreur :

vous écrivez votre code comme ceci :	il est complété implicitement ainsi :

<pre> class ClasseA {     public int attrA ;     public string attrStrA ;     public ClasseA ( int a ) {     } } class ClasseB : ClasseA {     public ClasseB () {         //.....     } } </pre> <p>La classe de base ClasseA ne comporte qu'un seul constructeur explicite à un paramètre. Le constructeur sans paramètres n'existe que si vous le déclarez explicitement, ou bien si la classe ne possède pas de constructeur explicitement déclaré.</p>	<pre> class ClasseA {     public int attrA ;     public string attrStrA ;     public ClasseA ( int a ) {     } } class ClasseB : ClasseA {     public ClasseB () : base() {         // ....     } } </pre> <p>L'initialiseur implicite <b>base()</b> renvoie le compilateur chercher dans la classe de base un constructeur sans paramètres. Or il n'existe pas dans la classe de base (ClasseA) de constructeur par défaut sans paramètres. Donc la tentative échoue !</p>
---	---

Le message d'erreur sur la ligne " **public ClasseB () {** ", est le suivant :  
*[C# Erreur] Class.cs(54): Aucune surcharge pour la méthode 'ClasseA' ne prend d'arguments '0'*

### Remarque :

Donc sans initialiseur explicite, **tout objet de classe fille** ClasseB est à minima et **par défaut, instancié** comme un **objet de classe de base** ClasseA.

## Initialiseur **this()** avant le corps

Lorsque l'on veut **invoker dans un constructeur** d'une classe donnée **un autre constructeur de cette même classe** étant donné que tous les constructeurs ont le même nom, il faut utiliser le mot clef **this** comme nom d'appel. La syntaxe d'utilisation du **this()** est semblable à celle du **base()** : à la suite de l'en-tête du constructeur **de classe fille** .

### Exemple :

Reprenons la même classe ClasseA possédant 2 constructeurs et la classe ClasseB dérivant de ClasseA, nous marquons les actions des constructeurs par une chaîne indiquant le numéro du constructeur invoqué ainsi que sa classe :

```

class ClasseA {
    public int attrA ;
    public string attrStrA ;

    public ClasseA () { /* premier constructeur */
        attrA = 57 ;
    }
}

```

```

public ClasseA ( string s ) { /* second constructeur */
    attrStrA = s + "...classeA1..." ;
}
}

```

Ci-dessous la ClasseB écrite de deux façons équivalentes :

avec initialiseurs implicites-explicites	avec initialiseurs explicites équivalents
<pre> class ClasseB : ClasseA {      /* premier constructeur */     public ClasseB () {         attrA = 100+attrA ;     }      /* second constructeur */     public ClasseB ( string s ) {         attrStrA = attrStrA +s+"...classeB2..." ;     }      /* troisième constructeur */     public ClasseB ( int x , string ch ) : this( ch ) {         attrStrA = attrStrA+"...classeB3..." ;     }      /* quatrième constructeur */     public ClasseB ( char x , string ch ) : base( ch ) {         attrStrA = attrStrA+"...classeB4..." ;     } } </pre>	<pre> class ClasseB : ClasseA {      /* premier constructeur */     public ClasseB () : base() {         attrA = 100+attrA ;     }      /* second constructeur */     public ClasseB ( string s ) : base() {         attrStrA = attrStrA +s+"...classeB2..." ;     }      /* troisième constructeur */     public ClasseB ( int x , string ch ) : this( ch ) {         attrStrA = attrStrA+"...classeB3..." ;     }      /* quatrième constructeur */     public ClasseB ( char x , string ch ) : base( ch )     {         attrStrA = attrStrA+"...classeB4..." ;     } } </pre>

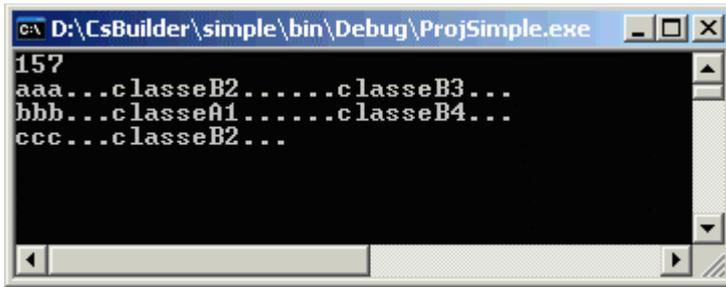
Créons quatre objets de ClasseB, chacun avec l'un des 4 constructeurs de la ClasseB :

```

class MaClass {
    static void Main(string[] args) {
        int x=68;
        ClasseB ObjetB= new ClasseB ();
        System.Console.WriteLine(ObjetB.attrA);
        ObjetB= new ClasseB(x,"aaa");
        System.Console.WriteLine(ObjetB.attrStrA);
        ObjetB= new ClasseB((char)x,"bbb");
        System.Console.WriteLine(ObjetB.attrStrA);
        ObjetB= new ClasseB("ccc");
        System.Console.WriteLine(ObjetB.attrStrA);
        System.Console.ReadLine();
    }
}

```

Voici le résultat console de l'exécution de ce programme :



Explications du code des classes précédentes :

<pre> ClasseB ObjetB= new ClasseB(); System.Console.WriteLine(ObjetB.attrA);  <i>constructeur mis en œuvre :</i>  public ClasseB () {     attrA = 100+attrA ; } </pre>	<p>C# sélectionne la signature du premier constructeur de la ClasseB (le constructeur sans paramètres).</p> <p>C# appelle d'abord implicitement le constructeur sans paramètre de la classe mère ( : base() )</p> <pre> public ClasseA () {     attrA = 57 ; } </pre> <p>Le champ attrA vaut 57,</p> <p>puis C# exécute le corps du constructeur :</p> <pre> attrA = 100+attrA ; </pre> <p>attrA vaut 100+57 = 157</p>
<pre> ObjetB= new ClasseB(x,"aaa"); System.Console.WriteLine(ObjetB.attrStrA);  <i>constructeurs mis en œuvre :</i>  public ClasseB ( string s ) {     attrStrA = attrStrA +s+"...classeB2..." ; }  public ClasseB ( int x , string ch ) : this( ch ) {     attrStrA = attrStrA+"...classeB3..." ; } </pre>	<p>C# sélectionne la signature du troisième constructeur de la ClasseB (le constructeur avec paramètres : int x , string ch).</p> <p>C# appelle d'abord explicitement le constructeur local de la classeB avec un paramètre de type string ( le second constructeur de la ClasseB )</p> <pre> s = "aaa" ; public ClasseB ( string s ) {     attrStrA = attrStrA +s+"...classeB2..." ; } </pre> <p>Le champ attrStrA vaut "aaa...classeB2...",</p> <p>puis C# exécute le corps du constructeur :</p> <pre> attrStrA = attrStrA+"...classeB3..." ; </pre> <p>attrStrA vaut "aaa...classeB2.....classeB3..."</p>

<pre>ObjetB= new ClasseB((char)x,"bbb"); System.Console.WriteLine(ObjetB.attrStrA);</pre> <p><i>constructeur mis en œuvre :</i></p> <pre>public ClasseB ( char x , string ch ) : base( ch ) {     attrStrA = attrStrA+"...classeB4..." ; }</pre>	<p>C# sélectionne la signature du quatrième constructeur de la ClasseB (le constructeur avec paramètres : <b>char</b> x , <b>string</b> ch).</p> <p>C# appelle d'abord explicitement le constructeur de la classe mère (de base) classeA avec un paramètre de type string ( ici le second constructeur de la ClasseA )</p> <pre>s = "bbb" ; public ClasseA ( string s ) {     attrStrA = s +"...classeA1..." ; }</pre> <p>Le champ attrStrA vaut "<b>bbb...classeA1...</b>"</p> <p>puis C# exécute le corps du constructeur :</p> <pre>attrStrA = attrStrA+"...classeB4..." ;</pre> <p>attrStrA vaut "<b>bbb...classeA1.....classeB4...</b>"</p>
--	--

La dernière instantiation : `ObjetB= new ClasseB("ccc");` est strictement identique à la première mais avec appel au second constructeur.

### 2.3 Comparaison de construction C#, Delphi et Java

*Exemple classe mère :*

C#	Java
<pre>class ClasseA {     public int attrA ;     public string attrStrA ;      public ClasseA () {         attrA = 57 ;     }     public ClasseA ( string s ) {         attrStrA = s +"...classeA1..." ;     } }</pre>	<pre>class ClasseA {     public int attrA ;     public String attrStrA = "" ;      public ClasseA () {         attrA = 57 ;     }     public ClasseA ( String s ) {         attrStrA = s +"...classeA1..." ;     } }</pre>

C#	Delphi
----	--------

<pre> class ClasseA { public int attrA ; public string attrStrA ;  public ClasseA () { attrA = 57 ; } public ClasseA ( string s ) { attrStrA = s + "...classeA1..." ; } } </pre>	<pre> ClasseA = class public attrA : integer ; attrStrA: string ; constructor Creer; overload; constructor Creer(s:string); overload; end;  constructor ClasseA.Creer begin attrA := 57 ; end; constructor ClasseA.Creer(s:string); begin attrStrA := s + '...classeA1...'; end; </pre>
--	---

*Exemple classe fille :*

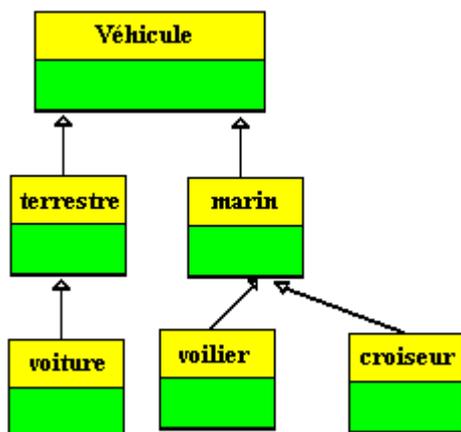
C#	Java
<pre> class ClasseB : ClasseA {  /* premier constructeur */ public ClasseB () { attrA = 100+attrA ; }  /* second constructeur */ public ClasseB ( string s ) { attrStrA = attrStrA +s+"...classeB2..." ; }  /* troisième constructeur */ public ClasseB ( int x , string ch ) : this( ch ) { attrStrA = attrStrA+"...classeB3..." ; }  /* quatrième constructeur */ public ClasseB ( char x , string ch ) : base( ch ) { attrStrA = attrStrA+"...classeB4..." ; } } </pre>	<pre> class ClasseB extends ClasseA {  /* premier constructeur */ public ClasseB () { super() ; attrA = 100+attrA ; }  /* second constructeur */ public ClasseB ( String s ) { super() ; attrStrA = attrStrA +s+"...classeB2..." ; }  /* troisième constructeur */ public ClasseB ( int x , String ch ) { this( ch ) ; attrStrA = attrStrA+"...classeB3..." ; }  /* quatrième constructeur */ public ClasseB ( char x , String ch ) { super( ch ) ; attrStrA = attrStrA+"...classeB4..." ; } } </pre>

C#	Delphi
----	--------

<pre> class ClasseB : ClasseA {      /* premier constructeur */     public ClasseB ()     {         attrA = 100+attrA ;     }      /* second constructeur */     public ClasseB ( string s )     {         attrStrA = attrStrA +s+"...classeB2..." ;     }      /* troisième constructeur */     public ClasseB ( int x , string ch ) : this( ch )     {         attrStrA = attrStrA+"...classeB3..." ;     }      /* quatrième constructeur */     public ClasseB ( char x , string ch ) : base( ch )     {         attrStrA = attrStrA+"...classeB4..." ;     }  } </pre>	<pre> ClasseB = class( ClasseA ) public     constructor Creer; overload;     constructor Creer(s:string); overload;     constructor Creer(x:integer;ch:string); overload;     constructor Creer(x:char;ch:string); overload; end;  /* premier constructeur */ constructor ClasseB.Creer; begin     inherited ;     attrA := 100+attrA ; end;  /* second constructeur */ constructor ClasseB.Creer(s:string); begin     inherited Creer ;     attrStrA := attrStrA +s+'...classeB2...'; end;  /* troisième constructeur */ constructor ClasseB.Creer(x:integer;ch:string); begin     Creer( ch ) ;     attrStrA := attrStrA+'...classeB3...'; end;  /* quatrième constructeur */ constructor ClasseB.Creer(x:integer;ch:string); begin     inherited Creer( ch ) ;     attrStrA := attrStrA+'...classeB4...'; end; </pre>
---	--

## 2.4 Traitement d'un exercice complet

soit une hiérarchie de classe de véhicules :



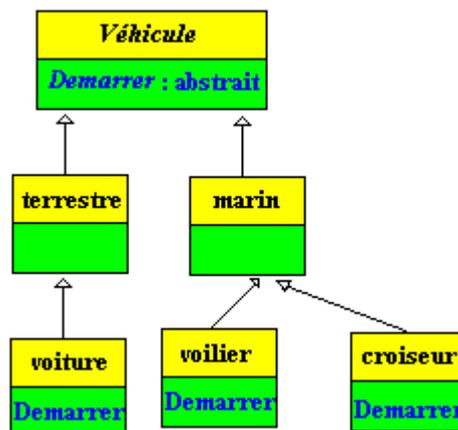
Syntaxe de base :

<pre> class Vehicule { } class Terrestre :Vehicule { } class Marin :Vehicule { } </pre>	<pre> class Voiture : Terrestre { } class Voilier : Marin { } class Croiseur : Marin { } </pre>
---	---

Supposons que la classe Véhicule contienne 3 méthodes, qu'elle n'implémente pas la méthode **Demarrer** qui est alors **abstraite**, qu'elle fournit et implante à vide la méthode "**RépartirPassagers**" de répartition des passagers à bord du véhicule, qu'elle fournit aussi et implante à vide une méthode "**PériodicitéMaintenance**" renvoyant la périodicité de la maintenance obligatoire du véhicule.



La classe Véhicule est abstraite : car la méthode **Demarrer** est abstraite et sert de "modèle" aux futures classes dérivant de Véhicule. Supposons que l'on implémente le comportement précis du genre de démarrage dans les classes **Voiture**, **Voilier** et **Croiseur**.



Dans cette hiérarchie, les classes **Terrestre** et **Marin** héritent de la classe **Vehicule**, mais n'implémentent pas la méthode abstraite **Démarrer**, ce sont donc par construction des classes abstraites elles aussi. Elles implantent chacune la méthode "**RépartirPassagers**" (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...) et la méthode "**PériodicitéMaintenance**" (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)

Les classes **Voiture**, **Voilier** et **Croiseur** savent par héritage direct comment répartir leurs éventuels passagers et quand effectuer une maintenance, chacune d'elle implémente son propre comportement de démarrage.

## Quelques implantations en C#

Une implémentation de la classe Voiture avec des méthodes non virtuelles (*Version-1*) :

<pre>abstract class Vehicule {     public abstract void Demarrer();     public void RépartirPassagers(){}     public void PériodicitéMaintenance(){} }</pre>	<p>La méthode <b>Demarrer</b> de la classe Vehicule est abstraite et donc automatiquement virtuelle (à liaison dynamique).</p> <p>Les méthodes RépartirPassagers et PériodicitéMaintenance sont concrètes mais avec un corps vide.</p> <p>Ces deux méthodes sont non virtuelles (à liaison statique)</p>
<pre>abstract class Terrestre : Vehicule {     public new void RépartirPassagers(){         //...}     public new void PériodicitéMaintenance(){         //...} }</pre>	<p>La classe Terrestre est abstraite car elle n'implémente pas la méthode abstraite <b>Demarrer</b>.</p> <p>Les deux méthodes déclarées dans la classe Terrestre masquent chacune la méthode du même nom de la classe Vehicule (d'où l'utilisation du mot clef <b>new</b>)</p>
<pre>class Voiture : Terrestre {     public override void Demarrer(){         //...} }</pre>	<p>La classe Voiture est la seule à être instanciable car toutes ses méthodes sont concrètes :</p> <p>Elle hérite des 2 méthodes implémentées de la classe Terrestre et elle implante (redéfinition avec <b>override</b>) la méthode abstraite de l'ancêtre.</p>

La même implémentation de la classe Voiture avec des méthodes virtuelles (*Version-2*):

<pre>abstract class Vehicule {     public abstract void Demarrer();     public virtual void RépartirPassagers(){}     public virtual void PériodicitéMaintenance(){} }</pre>	<p>La méthode <b>Demarrer</b> de la classe Vehicule est abstraite et donc automatiquement virtuelle (à liaison dynamique).</p> <p>Les méthodes RépartirPassagers et PériodicitéMaintenance sont concrètes mais avec un corps vide.</p> <p>Ces deux méthodes sont maintenant virtuelles (à liaison dynamique)</p>
<pre>abstract class Terrestre : Vehicule {     public override void RépartirPassagers(){         //...}     public override void PériodicitéMaintenance(){         //...} }</pre>	<p>La classe Terrestre est abstraite car elle n'implémente pas la méthode abstraite <b>Demarrer</b>.</p> <p>Les deux méthodes déclarées dans la classe Terrestre redéfinissent chacune la méthode du même nom de la classe Vehicule (d'où l'utilisation du mot clef <b>override</b>)</p>
<pre>class Voiture : Terrestre {     public override void Demarrer(){         //...} }</pre>	<p>La classe Voiture est la seule à être instanciable car toutes ses méthodes sont concrètes :</p> <p>Elle hérite des 2 méthodes implémentées de la classe Terrestre et elle implante (redéfinition avec <b>override</b>) la méthode abstraite de l'ancêtre.</p>

Supposons que les méthodes non virtuelles RépartirPassagers et PériodicitéMaintenance sont implantées complètement dans la classe Vehicule, puis reprenons la classe Terrestre en masquant ces deux méthodes :

```
abstract class Vehicule {
```

```

    public abstract void Demarrer( );
    public void RépartirPassagers(){
    //....}
    public void PériodicitéMaintenance( ){
    //....}
}

abstract class Terrestre : Vehicule {
    public new void RépartirPassagers( ){
    //...}
    public new void PériodicitéMaintenance( ){
    //...}
}

```

### Question

Nous voulons qu'un véhicule Terrestre répartisse ses passagers ainsi :

- 1°) d'abord comme tous les objets de classe Vehicule,
- 2°) ensuite qu'il rajoute un comportement qui lui est propre

### Réponse

La méthode RépartirPassagers est non virtuelle, elle masque la méthode mère du même nom, si nous voulons accéder au comportement de base d'un véhicule, il nous faut utiliser le mot clef **base** permettant d'accéder aux membres de la classe mère :

```

abstract class Terrestre : Vehicule {
    public new void RépartirPassagers( ){
        base.RépartirPassagers( ); //... 1°; comportement du parent
        //... 2° comportement propre
    }
    public new void PériodicitéMaintenance( ){
    //...}
}

```

Il est conseillé au lecteur de reprendre le même schéma et d'implanter à l'identique les autres classes de la hiérarchie pour la branche des véhicules **Marin**.

## 2.5 Destructeur – Finaliseur

Les classes de C# peuvent posséder un finaliseur (syntaxe semblable au destructeur de C++) mais comme la gestion de la mémoire est automatiquement assurée par le CLR, l'usage d'un finaliseur n'est pas le même que celui qui en est fait avec Delphi ou C++ dans lesquels la récupération de la mémoire est à la charge du développeur.

- Les structs ne peuvent pas posséder de finaliseur (ou destructeur)
- Propriétés et usage des finaliseurs avec C# :
- Une classe C# ne peut posséder qu'un seul finaliseur.
- Un finaliseur ne peut pas être redéfini ni surchargé.
- Un finaliseur n'a pas de paramètre.
- Un destructeur est appelé **automatiquement** par le CLR lors de la destruction effective de

l'objet par le CLR, il ne peut pas être appelé directement.

De ces propriétés il découle qu'un destructeur peut être utilisée par le développeur pour libérer certaines ressources non gérées par le CLR (fermeture de fichier, fermeture de connexion,...) lorsque le programme ne les utilise plus.

Exemple :

```
class classeB : classeA
{
    // constructeur :
    public classeB()
    {
        .....
    }
    // destructeur :
    ~classeB()
    {
        Console.WriteLine("finaliseur de la classeB ");
    }
}
```

A chaque fois qu'un objet de classeA est détruit par le CLR, le message "finaliseur de la classeB" apparaîtra sur la console. Il est entendu qu'il ne faut pas se servir de la programmation des destructeurs (finaliseurs) pour gérer un programme d'une manière synchrone puisque le programme n'a aucun contrôle sur la façon dont les objets sont libérés par le CLR qui utilise son propre algorithme de garbage collection et de restitution de mémoire.

Enfin, un destructeur appelle automatiquement le destructeur de la classe mère et ainsi de suite récursivement jusqu'à la classe object. Dans l'exemple précédent, le destructeur de la classeB, après écriture du message "finaliseur de la classeB", appelle le destructeur de la classeA.

# Polymorphisme et interfaces en C#.net

---

## Plan général: 📑

### Rappels sur la notion d'interface

#### 1. Concepts et vocabulaire d'interface en C#

- les interfaces peuvent constituer des hiérarchies et hériter entre elles
- la construction d'un objet nécessite une classe implémentant l'interface
- les implémentations d'un membre d'interface sont en général public
- les implémentations explicites d'un membre d'interface sont spéciales

##### 1.1 Spécification d'un exemple complet

###### 1.1.A Une classe abstraite

###### 1.1.B Une interface

###### 1.1.C Une simulation d'héritage multiple

###### 1.1.D Encore une classe abstraite, mais plus concrète

###### 1.1.E Une classe concrète

##### 1.2 Implantation en C# de l'exemple

###### 1.2.A La classe abstraite

###### 1.2.B L'interface

###### 1.2.C La simulation d'héritage multiple

###### 1.2.D La nouvelle classe abstraite

###### 1.2.E La classe concrète

#### 2. Analyse du code de liaison de la solution précédente

##### 2.1 Le code de la classe Vehicule

##### 2.2 Le code de l'interface IVehicule

##### 2.3 Le code de la classe UnVehicule

##### 2.4 Le code de la classe Terrestre

##### 2.5 Le code de la classe Voiture

#### 3. Cohérence de C# entre les notions de classe et d'interface

- une classe peut implémenter plusieurs interfaces
- les interfaces et les classes respectent les mêmes règles de polymorphisme
- les conflits de noms dans les interfaces

## Rappels essentiels sur la notion d'interface

- Les interfaces ressemblent aux classes abstraites : elles contiennent des membres **spécifiant certains comportements sans les implémenter**.
- Les classes abstraites et les interfaces se différencient principalement par le fait qu'**une classe peut implémenter un nombre quelconque d'interfaces**, alors qu'une classe abstraite ne peut hériter que d'**une seule classe** abstraite ou non.
- Une **interface** peut servir à représenter des comportements d'héritage multiple.

Quelques conseils généraux prodigués par des développeurs professionnels (microsoft, Borland, Sun) :

- *Les interfaces bien conçues sont plutôt petites et indépendantes les unes des autres.*
- *Un trop grand nombre de fonctions rend l'interface peu maniable.*
- *Si une modification s'avère nécessaire, une nouvelle interface doit être créée.*
- *Si la fonctionnalité que vous créez peut être utile à de nombreux objets différents, faites appel à une interface.*
- *Si vous créez des fonctionnalités sous la forme de petits morceaux concis, faites appel aux interfaces.*
- *L'utilisation d'interfaces permet d'envisager une conception qui sépare la manière d'utiliser une classe de la manière dont elle est implémentée.*
- *Deux classes peuvent partager la même interface sans descendre nécessairement de la même classe de base.*

## 1. Vocabulaire et concepts en C#

- Une **interface** C# est un contrat, elle peut contenir des **propriétés**, des **méthodes**, des **événements** ou des **indexeurs**, mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une **interface** **ne** peut **pas** contenir des méthodes déjà implémentées.
- Une **interface** ne contient que des signatures (**propriétés**, **méthodes**).
- **Tous les membres** d'une interface sont automatiquement **public**.
- Une **interface** est héritable.
- On peut construire une hiérarchie d'**interfaces**.
- Pour pouvoir construire un objet à partir d'une **interface**, il faut définir une classe non abstraite implémentant **tous** les membres de l'**interface**.

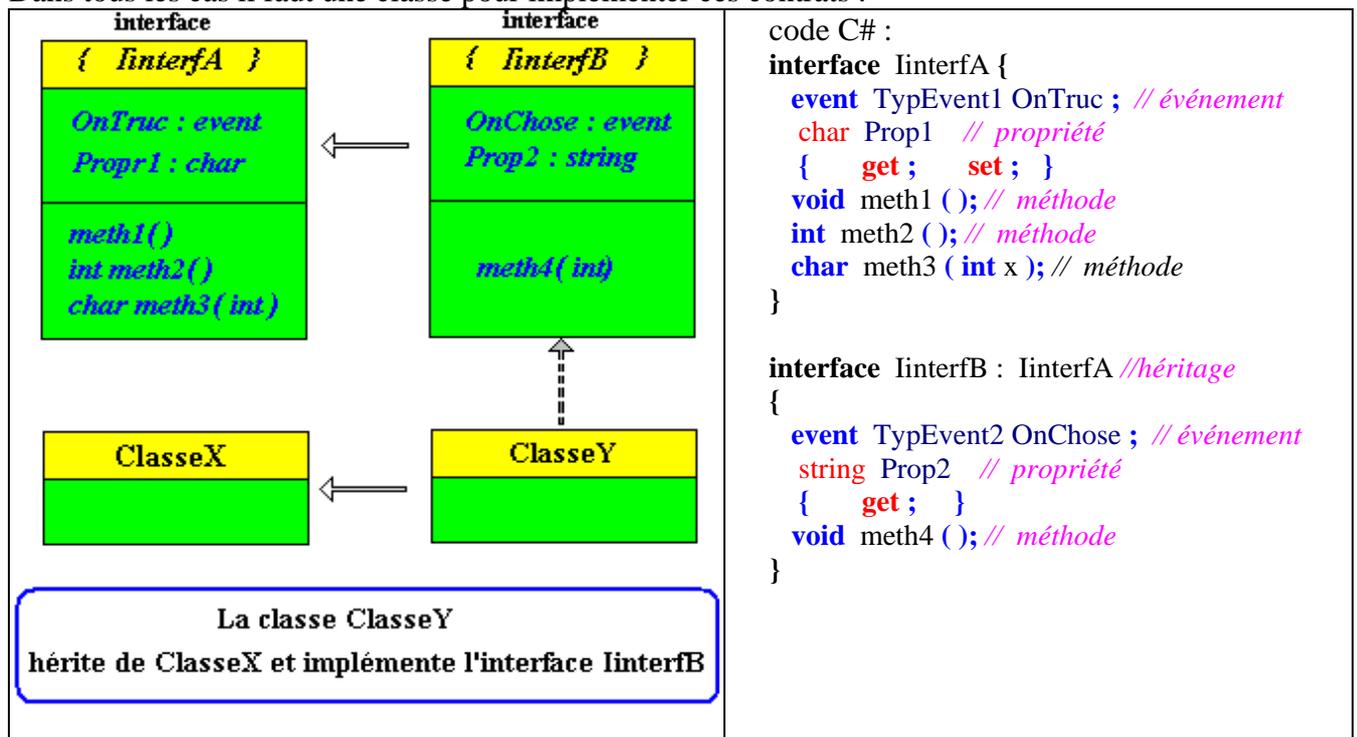
### Les interfaces peuvent constituer des hiérarchies et hériter entre elles

soient l'interface **IinterfA** et l'interface **IinterfB** héritant de **IinterfA**. On pourra employer aussi le vocable d'étendre au sens où l'interface dérivée **IinterfB** "étend" le contrat (augmente le nombre de membres contractuels) de l'interface **IinterfA**.

### Syntaxe de déclaration d'une interface

```
interface < nom de l'interface >  
{  
    < corps des déclarations : méthodes, ... >  
}
```

Dans tous les cas il faut une classe pour implémenter ces contrats :



## La construction d'un objet nécessite une classe implémentant l'interface

La classe ClasseY doit implémenter tous les 8 membres provenant de l'héritage des interfaces : les 2 événements **OnTruc** et **Onchose**, les 2 propriétés **Prop1** et **Prop2**, et enfin les 4 méthodes **meth1**, ... , **meth4** . La classe ClasseY est une classe concrète (instanciable), un objet de cette classe possède en particulier tous les membres de l'interface IinterfB (et donc IinterfA car IinterfB hérite de IinterfA)

```

class ClasseY : ClasseX , IinterfB {
    // ... implémente tous les membres de InterfB
}
// construction et utilisation d'un objet :
ClasseY Obj = new ClasseY() ;
Obj.Prop1 = 'a' ; // propriété héritée de InterfA
string s = Obj.Prop2 ; // propriété héritée de InterfB
Obj.meth1() ; // méthode héritée de InterfA
etc ...
    
```

## Tous les membres doivent être implémentés

Si ClasseY n'implémente par exemple que 7 membres sur les 8 alors C# considère que c'est une erreur et le compilateur le signale, **tous les membres d'une interface doivent être implémentés dans une classe qui en hérite** (contrairement à java qui accepte cette implémentation partielle en classe abstraite).

## Les implémentations des membres d'une interface sont en général public

Par défaut sans déclaration explicite, les membres (indexeurs, propriétés, événements, méthodes) d'une interface ne nécessitent pas de qualificatifs de visibilité car ils sont automatiquement déclarés par C# comme étant de visibilité public, contrairement à une classe ou par défaut les membres sont du niveau assembly.

Ce qui signifie que toute classe qui implémente un membre de l'interface doit obligatoirement le qualifier de **public** sous peine d'avoir un message d'erreur du compilateur, dans cette éventualité le membre devient un membre d'instance. Comme la signature de la méthode n'est qu'un contrat, le mode de liaison du membre n'est pas fixé; la classe qui implémente le membre peut alors choisir de l'implémenter soit **en liaison statique**, soit **en liaison dynamique**.

Soient une interface IinterfA et une classe ClasseX héritant directement de la classe Object et implémentant cette interface, ci-dessous les deux seules implémentations possibles d'une méthode avec un rappel sur les redéfinitions possibles dans des classes descendantes :

```
interface IinterfA {
    void meth1 (); // méthode de l'interface
}
```

Implémentation en liaison précoce	Implémentation en liaison tardive
meth1 devient une méthode d'instance	
<pre>class ClasseX : IinterfA {     public void meth1 (){ ... } }</pre>	<pre>class ClasseX : IinterfA {     public virtual void meth1 (){ ... } }</pre>
Redéfinitions possibles	Redéfinitions possibles
<pre>class ClasseY : ClasseX {     public new void meth1 (){ ... }     <i>masque statiquement celle de ClasseX</i> }  class ClasseZ : ClasseX {     public new virtual void meth1 (){ ... }     <i>masque dynamiquement celle de ClasseX</i> }</pre>	<pre>class ClasseY : ClasseX {     public new void meth1 (){ ... }     <i>masque statiquement celle de ClasseX</i> }  class ClasseZ : ClasseX {     public new virtual void meth1 (){ ... }     <i>masque dynamiquement celle de ClasseX</i> }  class ClasseT : ClasseX {     public override void meth1 (){ ... }     <i>redéfinit dynamiquement celle de ClasseX</i> }</pre>

## Les implémentations explicites des membres d'une interface sont spéciales

Une classe qui implémente une interface peut aussi implémenter de façon **explicite** un membre de cette interface. Lorsqu'un membre est implémenté de façon explicite (le nom du membre est préfixé par le nom de l'interface : **InterfaceXxx.NomDuMembre**), il n'est pas accessible via une référence de classe, il est alors **invisible** à tout objet instancié à partir de la classe où il est défini. Un membre implémenté de façon explicite n'est donc pas un membre d'instance.

Pour utiliser un membre d'interface implémenté de manière explicite, il faut utiliser une référence sur cette interface et non une référence de classe; il devient visible uniquement à travers une référence sur l'interface.

Nous reprenons le même tableau de différentes implémentations de la méthode **void** meth1 () en ajoutant une nouvelle méthode **void** meth2 ( int x ) que nous **implémentons explicitement** dans les classes dérivées :

```
interface IinterfA {
    void meth2 ( int x ); // méthode de l'interface
    void meth1 (); // méthode de l'interface
}
```

Nous implémentons l'interface IinterfA dans la classe ClasseX :

- 1°) nous implémentons **explicitement void** meth2 ( int x ),
- 2°) nous implémentons **void** meth1 () en méthode **virtuelle**.

```
interface IinterfA {
    void meth2 ( int x ); // méthode de l'interface
    void meth1 (); // méthode de l'interface
}
class ClasseX : IinterfA {
    void IinterfA.meth2 ( int x ){ ... }
    public virtual void meth1 () { ... }
}
```

Comprenons bien que la classe ClasseX ne possède pas à cet instant une méthode d'instance qui se nommerait meth2, par exemple si dans la méthode virtuelle meth1 nous utilisons le paramètre implicite **this** qui est une référence à la future instance, l'audit de code de C#Builder nous renvoie 7 méthodes comme visibles (6 provenant de la classe mère Object et une seule provenant de ClasseX), la méthode IinterfA.meth2 n'est pas visible :

```

[ ] interface IinterfA {
    |   void meth2 ( int x ); // méthode de l'interface
    |   void meth1 (); // méthode de l'interface
    | }
[ ] class ClasseX : IinterfA {
[ ]   void IinterfA.meth2 ( int x ){ }
[ ]   public virtual void meth1 () {
    |   this.
    | }
[ ] }
[ ]
[ ]
[ ]
```

Method	meth1
Destructor	Finalize
Method	GetType
Method	Equals
Method	MemberwiseClone
Method	ToString
Method	GetHashCode

Lorsque l'on instancie effectivement un objet de classe ClasseX, cet objet ne voit comme méthode provenant de ClasseX que la méthode meth1 :

```

class Test(
void methTest() {
    ClasseX Obj = new ClasseX();
    Obj.
}
}

```

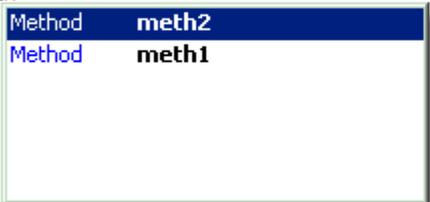


La méthode meth2 implémentée explicitement en IinterfA.meth2 devient visible uniquement si l'on utilise une référence sur l'interface IinterfA, l'exemple ci-dessous montre qu'alors les deux méthodes meth1 et meth2 sont visibles :

```

interface IinterfA {
    void meth2 ( int x ); // méthode de l'interface
    void meth1 ( ); // méthode de l'interface
}
class ClasseX : IinterfA {
    void IinterfA.meth2 ( int x ){ }
    public virtual void meth1 ( ){ }
}
class Test(
void methTest() {
    IinterfA Obj = new ClasseX();
    Obj.
}
}

```

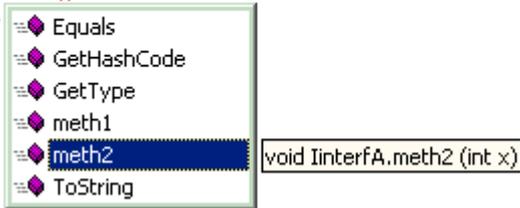


L'audit de code de Visual C# fournit plus de précision directement :

```

interface IinterfA {
    void meth2 ( int x ); // méthode de l'interface
    void meth1 ( ); // méthode de l'interface
}
class ClasseX : IinterfA {
    void IinterfA.meth2 ( int x ){ }
    public virtual void meth1 ( ){ }
}
class Test {
    void methTest() {
        IinterfA Obj = new ClasseX();
        Obj.~
    }
}

```



Nous voyons bien que la méthode est qualifiée avec sa signature dans IinterfA, voyons dans l'exemple ci-dessous que nous pouvons déclarer une méthode d'instance ayant la même signature que la méthode explicite, voir même de surcharger cette méthode d'instance sans que le compilateur C# n'y voit de conflit car la méthode explicite n'est pas rangé dans la table des méthodes d'instances de la classe :

```

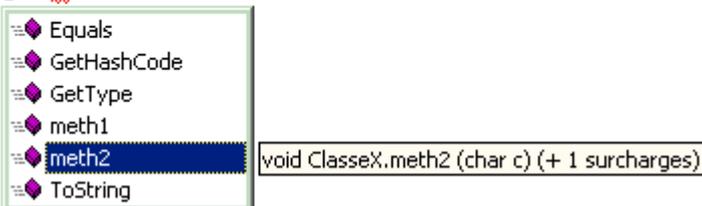
class ClasseX : IinterfA {
    void IinterfA.meth2 ( int x ){ ... } //méthode de IinterfA implémentée explicitement
    public virtual void meth2 ( int x ){ ... } //méthode de la ClasseX surchargée
    public virtual void meth2 ( char c ){ ... } //méthode de la ClasseX surchargée
    public virtual void meth1 ( ) { ... } //méthode de IinterfA implémentée virtuellement
}

```

```

interface IinterfA {
    void meth2 ( int x ); // méthode de l'interface
    void meth1 ( ); // méthode de l'interface
}
class ClasseX : IinterfA {
    void IinterfA.meth2 ( int x ){ }
    public virtual void meth1 ( ){ }
    public virtual void meth2 ( char c ){ }
    public virtual void meth2 ( int x ){ }
}
class Test {
    void methTest() {
        ClasseX Obj1 = new ClasseX();
        IinterfA Obj2 = new ClasseX();
        Obj1.~
    }
}

```



La référence Obj1 peut appeler les deux surcharges de la méthode d'instance meth2 de la classe ClasseX :

```

class Test {
    void methTest() {
        ClasseX Obj1 = new ClasseX();
        IinterfA Obj2 = new ClasseX();
        Obj1.meth2 (|
        ▲ 1 sur 2 ▼ void ClasseX.meth2 (char c)
    }
}

class Test {
    void methTest() {
        ClasseX Obj1 = new ClasseX();
        IinterfA Obj2 = new ClasseX();
        Obj1.meth2 (|
        ▲ 2 sur 2 ▼ void ClasseX.meth2 (int x)
    }
}

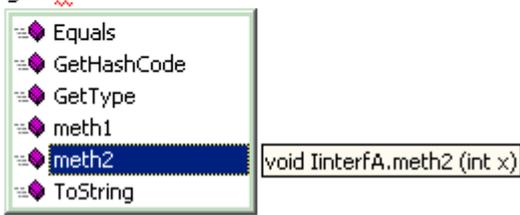
```

La référence Obj2 sur IinterfA fonctionne comme nous l'avons montré plus haut, elle ne peut voir de la méthode meth2 que son implémentation explicite :

```

class Test {
    void methTest () {
        ClasseX Obj1 = new ClasseX();
        IinterfA Obj2 = new ClasseX();
        Obj2.~
    }
}

```



Cette fonctionnalité d'implémentation explicite spécifique à C# peut être utilisée dans au moins deux cas utiles au développeur :

- Lorsque vous voulez qu'un membre (une méthode par exemple) implémenté d'une interface soit privé dans une classe pour toutes les instances de classes qui en dériveront, l'implémentation explicite vous permet de rendre ce membre (cette méthode) inaccessible à tout objet.
- Lors d'un conflit de noms si deux interfaces possèdent un membre ayant la même signature et que votre classe implémente les deux interfaces.

### 1.1 Spécification d'un exemple complet

Utilisons la notion d'interface pour fournir un polymorphisme à une hiérarchie de classe de véhicules fondée sur une **interface** :

Soit au départ une classe abstraite **Vehicule** et une interface **IVehicule**.

#### 1.1.A) Une classe abstraite

La classe abstraite **Vehicule** contient trois méthodes :

<b>classe abstraite</b> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <b><i>Vehicule</i></b> </div> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> + <i>Demarrer()</i>  + <i>RepartirPassagers()</i>  + <i>PeriodiciteMaintenance()</i> </div>	<ul style="list-style-type: none"> <li>• La méthode <b>Démarrer</b> qui est abstraite.</li> <li>• La méthode <b>RepartirPassagers</b> de répartition des passagers à bord du véhicule, implantée avec un corps vide.</li> <li>• La méthode <b>PériodicitéMaintenance</b> renvoyant la périodicité de la maintenance obligatoire du véhicule, implantée avec un corps vide.</li> </ul>
--	---

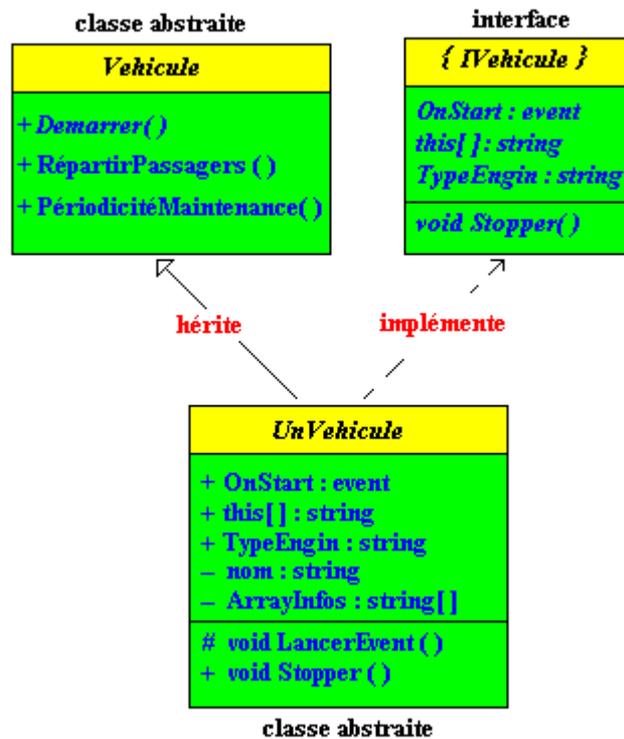
### 1.1.B) Une interface

Afin d'utiliser les possibilités de C#, l'interface **IVehicule** propose un contrat d'implémentation pour un **événement**, un **indexeur**, une **propriété** et une **méthode** :

<b>interface</b> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <b>{ <i>IVehicule</i> }</b> </div> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <i>OnStart</i> : <i>event</i>  <i>this</i> [ ] : <i>string</i>  <i>TypeEngin</i> : <i>string</i> </div> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <i>void Stopper()</i> </div>	<ul style="list-style-type: none"> <li>• L'événement <b>OnStart</b> est de type délégué (on construit un type délégué <code>Starting()</code> spécifique pour lui) et se déclenchera au démarrage du futur véhicule,</li> <li>• L'indexeur <b>this</b> [int ] est de type string et permettra d'accéder à une liste indicée d'informations sur le futur véhicule,</li> <li>• La propriété <b>TypeEngin</b> est en lecture et écriture et concerne le type du futur véhicule dans la marque,</li> <li>• La méthode <b>Stopper()</b> indique comment le futur véhicule s'immobilisera.</li> </ul>
---	---

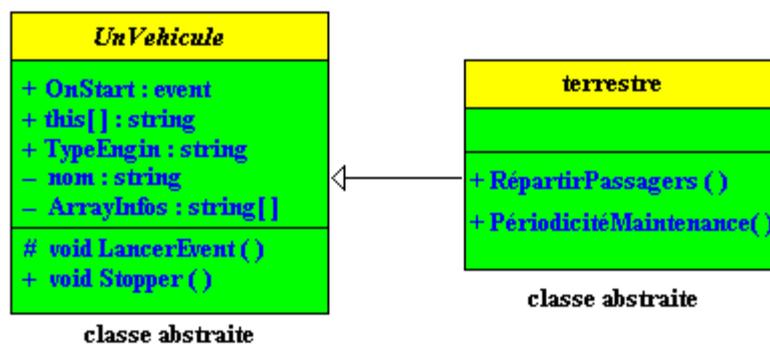
### 1.1.C) Une simulation d'héritage multiple

Nous souhaitons construire une classe abstraite **UnVehicule** qui "hérite" à la fois des fonctionnalités de la classe **Vehicule** et de celles de l'interface **IVehicule**. Il nous suffit en C# de faire hériter la classe **UnVehicule** de la classe **Vehicule**, puis que la classe **UnVehicule** implémente les propositions de contrat de l'interface **IVehicule** :



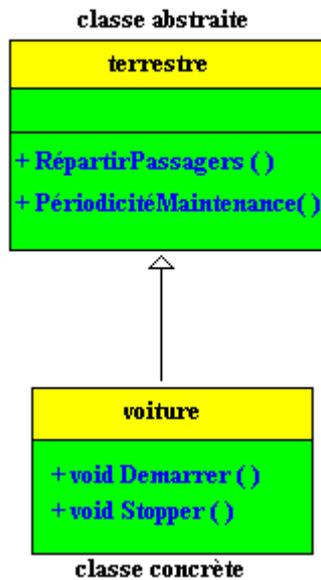
### 1.1.D) Encore une classe abstraite, mais plus "concrète"

Nous voulons maintenant proposer une spécialisation du véhicule en créant une classe abstraite **Terrestre**, base des futurs véhicules terrestres. Cette classe implantera de façon explicite la méthode **RépartirPassagers** de répartition des passagers et la méthode **PériodicitéMaintenance** renvoyant la périodicité de la maintenance. Cette classe **Terrestre** reste abstraite car elle ne fournit pas l'implémentation de la méthode **Demarrer** :

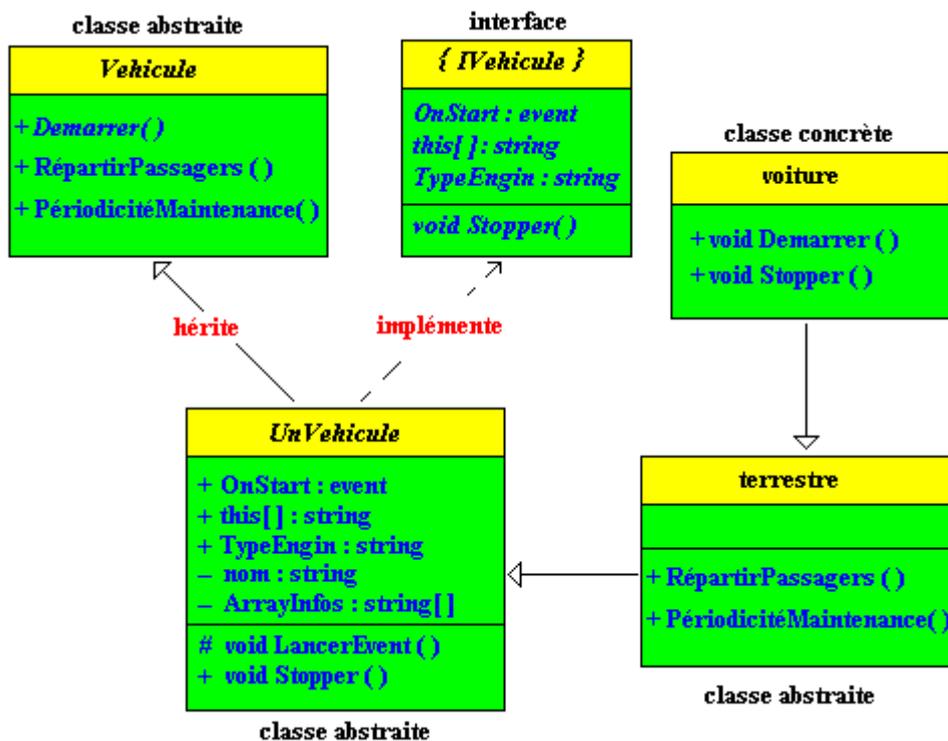


### 1.1.E) Une classe concrète

Nous finissons notre hiérarchie par une classe **Voiture** qui descend de la classe **Terrestre**, qui implante la méthode **Demarrer()** et qui redéfinit la méthode **Stopper()** :



Ce qui nous donne le schéma d'héritage total suivant :

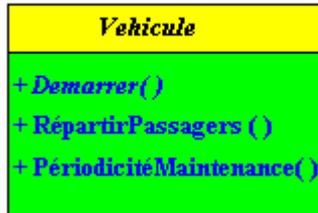


## 1.2 Implantation en C# de l'exemple

Nous proposons ci-dessous pour chaque classe ou interface une implémentation en C#.

### 1.2.A) La classe abstraite Vehicule

classe abstraite



```
abstract class Vehicule // classe abstraite mère
{
    public abstract void Demarrer (); // méthode abstraite
    public void RépartirPassagers () { } // implantation de méthode avec corps vide
    public void PériodicitéMaintenance () { } // implantation de méthode avec corps vide
}
```

### 1.2.B) L'interface IVehicule

interface



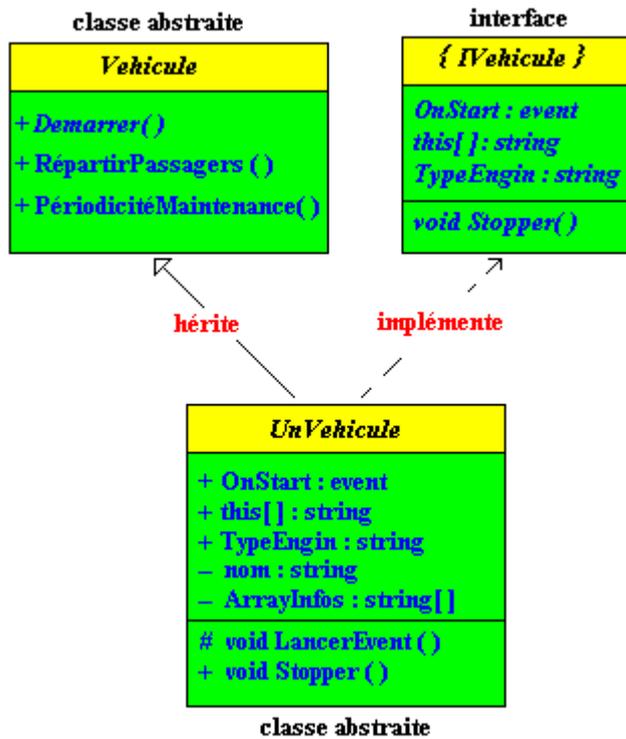
```
public delegate void Starting (); // déclaration de type délégué
interface IVehicule
{
    event Starting OnStart ; // déclaration d'événement du type délégué : Starting
    string this [ int index] // déclaration d'indexeur
    {
        get ;
        set ;
    }
    string TypeEngin // déclaration de propriété
    {
        get ;
        set ;
    }
    void Stopper (); // déclaration de méthode
}
```

### 1.2.C) La classe UnVehicule

Nous souhaitons définir une classe explicite effectivement pour un véhicule comment fonctionne le démarrage OnStart, comment on index les éléments, comment on récupère le type de l'engin, mais qui ne dise pas comment stopper le véhicule.

Nous définissons alors une classe abstraite nommée **UnVehicule** qui hérite de la classe abstraite **Vehicule** qui implémente effectivement les 3 premiers membres de l'interface **IVehicule** (**Event** OnStart, **Property** Item, **Property** TypeEngin) et qui implémente fictivement la méthode " **Sub** Stopper() " (en laissant son corps vide sans aucune instruction).

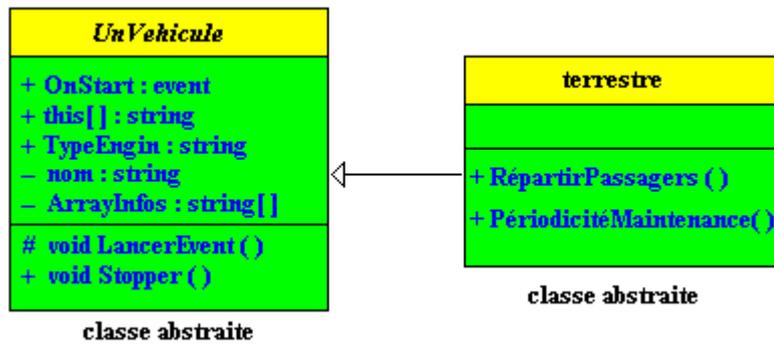
La classe **UnVehicule** reste abstraite car la méthode Demarrer() de la classe mère **Vehicule** n'est pas encore implémentée.



```

abstract class UnVehicule : Vehicule , IVehicule // hérite de la classe mère et implémente l'interface
{
    private string nom = "";
    private string [] ArrayInfos = new string [10];
    public event Starting OnStart ;
    protected void LancerEvent ()
    {
        if( OnStart != null)
            OnStart ();
    }
    public string this [ int index] // implantation Indexeur
    {
        get { return ArrayInfos[index]; }
        set { ArrayInfos[index] = value ; }
    }
    public string TypeEngin // implantation propriété
    {
        get { return nom ; }
        set { nom = value ; }
    }
    public virtual void Stopper () { } // implantation de méthode avec corps vide
}
  
```

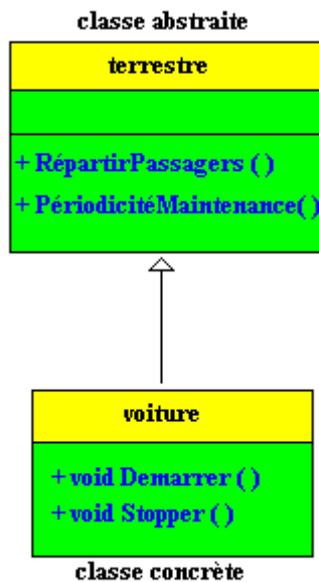
### 1.2.D) La classe Terrestre



```

abstract class Terrestre : UnVehicule
{
  public new void RépartirPassagers () {
    //...implantation de méthode masquant la méthode mère
  }
  public new void PériodicitéMaintenance () {
    //...implantation de méthode masquant la méthode mère
  }
}
  
```

### 1.2.E) La classe Voiture



```

class Voiture : Terrestre
{
  public override void Demarrer () {
    LancerEvent ();
  }
  public override void Stopper () {
    //...
  }
}
  
```

## 2. Analyse du code de liaison de la solution précédente

Nous nous intéressons au mode de liaison des membres du genre :

**méthodes, propriétés, indexeurs et événements.**

Rappelons au lecteur que la liaison statique indique que le compilateur lie le code lors de la compilation, alors que dans le cas d'une liaison dynamique le code n'est choisi et lié que lors de l'exécution.

### 2.1 Le code de la classe Vehicule

```
abstract class Vehicule
{
    public abstract void Demarrer (); // méthode abstraite
    public void RépartirPassagers () { } // implantation de méthode avec corps vide
    public void PériodicitéMaintenance () { } // implantation de méthode avec corps vide
}
```

#### Analyse :

Sans qualification particulière une méthode est à liaison statique :

- La méthode **public void** RépartirPassagers () est donc à liaison **statique**.
- La méthode **public void** PériodicitéMaintenance () est donc à liaison **statique**.

Une méthode qualifiée **abstract** est implicitement virtuelle :

- La méthode **public abstract void** Demarrer () est donc à liaison **dynamique**.

### 2.2 Le code de l'interface IVehicule

```
interface IVehicule
{
    event Starting OnStart ; // déclaration d'événement du type délégué : Starting
    string this [ int index] // déclaration d'indexeur
    { get ; set ; }
    string TypeEngin // déclaration de propriété
    { get ; set ; }
    void Stopper () ; // déclaration de méthode
}
```

## Analyse :

Une interface n'est qu'un contrat, les membres déclarés comme signatures dans l'interface n'étant pas implémentés, la question de leur liaison ne se pose pas au niveau de l'interface, mais lors de l'implémentation dans une classe ultérieure :

- La méthode **void** Stopper (); pourra donc être plus tard soit statique, soit dynamique.
- L'événement **event** Starting OnStart ; pourra donc être plus tard soit statique, soit dynamique.
- La propriété **string** TypeEngin , pourra donc être plus tard soit statique, soit dynamique.
- L'indexeur **string this [ int index]** , pourra donc être plus tard soit statique, soit dynamique.

## 2.3 Le code de la classe UnVehicule

```
abstract class UnVehicule : Vehicule , IVehicule
{
    private string nom = "";
    private string [] ArrayInfos = new string [10];
    public event Starting OnStart ;
    protected void LancerEvent () {
        if( OnStart != null) OnStart ();
    }
    public string this [ int index] //implantation Indexeur
    {
        get { return ArrayInfos[index]; }
        set { ArrayInfos[index] = value ; }
    }
    public string TypeEngin //implantation propriété
    {
        get { return nom ; }
        set { nom = value ; }
    }
    public virtual void Stopper () { } //implantation de méthode avec corps vide
}
```

## Analyse :

Le qualificateur **virtual** indique que l'élément qualifié est virtuel, donc à liaison dynamique; sans autre qualification un élément est par défaut à liaison statique :

- La méthode **public virtual void** Stopper () est à liaison **dynamique**.
- L'événement **public event** Starting OnStart est à liaison **statique**.
- La propriété **public string** TypeEngin est à liaison **statique**.

- L'indexeur **public string this [ int index]** est à liaison **statique**.
- La méthode **protected void LancerEvent ()** est à liaison **statique**.

## 2.4 Le code de la classe Terrestre

```
abstract class Terrestre : UnVehicule
{
    public new void RépartirPassagers () {
        //...implantation de méthode
    }
    public new void PériodicitéMaintenance () {
        //...implantation de méthode
    }
}
```

### Analyse :

Dans la classe mère Vehicule les deux méthodes **RépartirPassagers** et **PériodicitéMaintenance** sont à liaison statique, dans la classe Terrestre :

- La méthode **public new void RépartirPassagers ()** est à liaison **statique** et masque la méthode mère.
- La méthode **public new void PériodicitéMaintenance ()** est à liaison **statique** et masque la méthode mère.

## 2.5 Le code de la classe Voiture

```
class Voiture : Terrestre
{
    public override void Demarrer () {
        LancerEvent ();
    }
    public override void Stopper () {
        //...
    }
}
```

## Analyse :

La méthode Demarrer() est héritée de la classe mère Vehicule, la méthode Stopper() est héritée de la classe UnVehicule :

- La méthode **public override void** Demarrer () est à liaison **dynamique** et redéfinit la méthode abstraite mère.
- La méthode **public override void** Stopper () est à liaison **dynamique** et redéfinit la méthode virtuelle à corps vide de la classe UnVehicule.

## 3. Cohérence entre les notions de classe et d'interface dans C#

Une classe **peut implémenter plusieurs interfaces**. Dans ce cas nous avons une excellente alternative à l'**héritage multiple**.

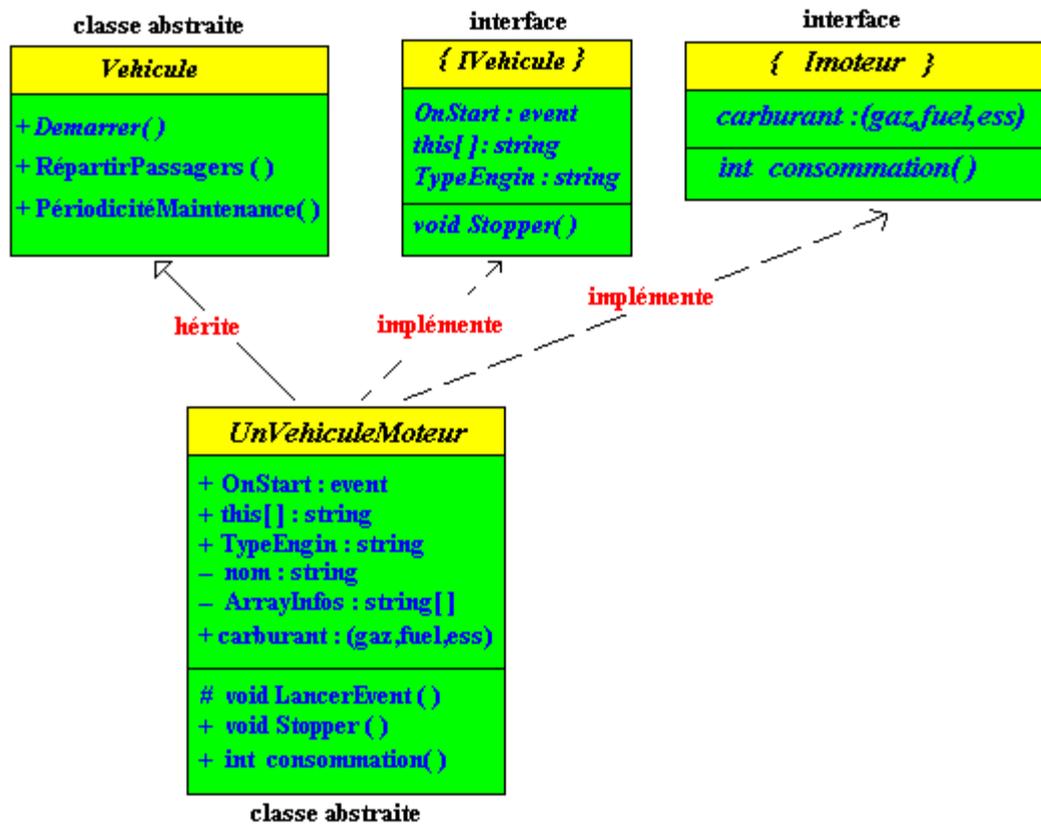
Lorsque l'on crée une interface, on fournit un ensemble de définitions et de comportements qui **ne devraient plus être modifiés**. Cette attitude de constance dans les définitions, protège les applications écrites pour utiliser cette interface.

Les variables de types interface respectent les mêmes règles de **transtypage** que les variables de types classe.

Les **objets** de type classe **clA** peuvent être transtypés et **référéncés** par des variables d'interface **IntfA** dans la mesure où la classe **clA implémente l'interface IntfA**. (cf. polymorphisme d'objet)

### Une classe peut implémenter plusieurs interfaces

Soit la définition suivante où la classe UnVehiculeMoteur hérite de la classe abstraite Vehicule et implémente l'interface IVehicule de l'exemple précédent, et supposons qu'en plus cette classe implémente l'interface IMoteur. L'interface IMoteur explique que lorsqu'un véhicule est à moteur, il faut se préoccuper de son type de carburant et de sa consommation :



Ci-dessous le code C# correspondant à cette définition, plus une classe concrète **Voiture** instanciable dérivant de la classe abstraite UnVehiculeMoteur :

```

abstract class Vehicule {
    public abstract void Demarrer (); // méthode abstraite
    public void RépartirPassagers () { } // implantation de méthode avec corps vide
    public void PériodicitéMaintenance () { } // implantation de méthode avec corps vide
}
interface IVehicule {
    event Starting OnStart ; // déclaration d'événement du type délégué : Starting
    string this [ int index] // déclaration d'indexeur
    { get ; set ; }
    string TypeEngin // déclaration de propriété
    { get ; set ; }
    void Stopper (); // déclaration de méthode
}
enum Energie { gaz , fuel , ess } // type énuméré pour le carburant

interface IMoteur {
    Energie carburant // déclaration de propriété
    { get ; }
    int consommation (); // déclaration de méthode
}

abstract class UnVehiculeMoteur : Vehicule , IVehicule , IMoteur
{
    private string nom = "";
    private Energie typeEnerg = Energie.fuel ;
    private string [] ArrayInfos = new string [10] ;
    public event Starting OnStart ;
}
  
```

```

protected void LancerEvent () {
    if( OnStart != null) OnStart ();
}
public string this [ int index] // implantation Indexeur de IVehicule
{
    get { return ArrayInfos[index] ; }
    set { ArrayInfos[index] = value ; }
}
public string TypeEngin // implantation propriété de IVehicule
{
    get { return nom ; }
    set { nom = value ; }
}
    public Energie carburant // implantation propriété de IMoteur
    {
        get { return typeEnerg ; }
    }
    public virtual void Stopper () { } // implantation vide de méthode de IVehicule
    public virtual int consommation () { return .... } // implantation de méthode de IMoteur
}

```

```

class Voiture : UnVehiculeMoteur {
    public override void Demarrer () {
        LancerEvent ();
    }
    public override void Stopper () {
        //...implantation de méthode
    }
    public new void RépartirPassagers () {
        //...implantation de méthode
    }
    public new void PériodicitéMaintenance () {
        //...implantation de méthode
    }
}

```

## Les interfaces et les classes respectent les mêmes règles de polymorphisme

Il est tout à fait possible d'utiliser des variables de référence sur des interfaces et de les transtyper d'une manière identique à des variables de référence de classe. En particulier le polymorphisme de référence s'applique aux références d'interfaces.

**Le polymorphisme de référence sur les classes de l'exemple précédent**

```

abstract class Vehicule { ... }
interface IVehicule { ... }
enum Energie { gaz , fuel , ess }
interface IMoteur { ... }

abstract class UnVehiculeMoteur : Vehicule ,
IVehicule , IMoteur { ... }
class Voiture : UnVehiculeMoteur { ... }

class UseVoiture1 {
    public string use ( IVehicule x ){
        return x.TypeEngin ;
    }
}
class UseVoiture2 {
    public string use ( UnVehiculeMoteur x ){
        return x.TypeEngin ;
    }
}
class UseVoiture3 {
    public string use ( Voiture x ){
        return x.TypeEngin ;
    }
}

```

```

class MaClass {
    static void Main(string [ ] args) {
        string s = "le client utilise une ";
        string ch ;
        IVehicule a1;
        UnVehiculeMoteur b1;
        Voiture c1;
        a1 = new Voiture( );
        a1.TypeEngin = "Renault";
        b1= new Voiture( );
        b1.TypeEngin = "Citroën";
        c1 = new Voiture( );
        c1.TypeEngin = "Peugeot";
        UseVoiture1 client = new UseVoiture1( );
        ch = s+client.use(a1);
        System.Console.WriteLine(ch);
        ch = s+client.use(b1);
        System.Console.WriteLine(ch);
        ch = s+client.use(c1);
        System.Console.WriteLine(ch);
    }
}

```

### code d'exécution avec 3 objets différents

```

static void Main(string [ ] args) {
    .... idem
    UseVoiture1 client = new UseVoiture1( );
    ch = s+client.use(a1);
    System.Console.WriteLine(ch);
    ch = s+client.use(b1);
    System.Console.WriteLine(ch);
    ch = s+client.use(c1);
    System.Console.WriteLine(ch);
}

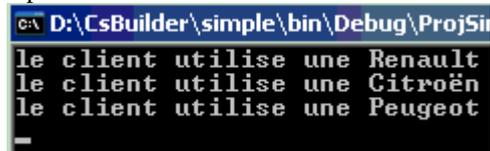
```

```

IVehicule
    |__ UnVehiculeMoteur
        |__ Voiture

```

Après exécution :



```

C:\D:\CsBuilder\simple\bin\Debug\ProjSi
le client utilise une Renault
le client utilise une Citroën
le client utilise une Peugeot
_

```

**a1** est une référence sur **IVehicule**, **b1** est une référence sur une interface fille de **IVehicule**, **c1** est de classe **Voiture** implémentant **IVehicule**.

Donc chacun de ces trois genres de paramètre peut être passé par polymorphisme de référence d'objet à la méthode **public string** use ( **IVehicule x** )

```

static void Main(string [ ] args) {
    .... idem
    UseVoiture2 client = new UseVoiture2( );
    ch = s+client.use(UnVehiculeMoteura1);
    System.Console.WriteLine(ch);
    ch = s+client.use(b1);
    System.Console.WriteLine(ch);
    ch = s+client.use(c1);
    System.Console.WriteLine(ch);
}

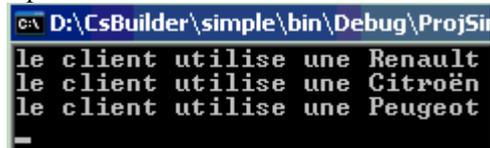
```

```

IVehicule
    |__ UnVehiculeMoteur
        |__ Voiture

```

Après exécution :



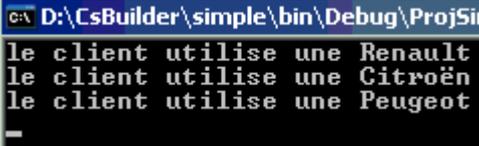
```

C:\D:\CsBuilder\simple\bin\Debug\ProjSi
le client utilise une Renault
le client utilise une Citroën
le client utilise une Peugeot
_

```

Le polymorphisme de référence d'objet appliqué à la méthode **public string** use ( **UnVehiculeMoteur x** ) indique que les paramètres passés doivent être de type **UnVehiculeMoteur** ou de type descendant.

La variable **a1** est une référence sur **IVehicule** qui ne descend pas de **UnVehiculeMoteur**, il faut donc transtyper la référence **a1** soit : **(UnVehiculeMoteur)a1**.

<pre> <b>static void</b> Main(string [ ] args) { .... idem UseVoiture3 client = <b>new</b> UseVoiture3( ); ch = s+client.use( <b>(Voiture)</b>a1 ); System.Console.WriteLine(ch); ch = s+client.use( <b>(Voiture)</b>b1 ); System.Console.WriteLine(ch); ch = s+client.use(c1); System.Console.WriteLine(ch); }  <b>IVehicule</b>  __UnVehiculeMoteur  __Voiture </pre>	<p>Après exécution :</p>  <p>Le polymorphisme de référence d'objet appliqué à la méthode <b>public string</b> use ( <b>(Voiture</b> x ) indique que les paramètres passé doivent être de type Voiture ou de type descendant.</p> <p>La variable <b>a1</b> est une référence sur IVehicule qui ne descend pas de Voiture, il faut donc transtyper la référence <b>a1</b> soit : <b>(Voiture)</b>a1</p> <p>La variable <b>b1</b> est une référence sur UnVehiculeMoteur qui ne descend pas de Voiture, il faut donc transtyper la référence <b>b1</b> soit : <b>(Voiture)</b>b1</p>
---	---

Les opérateurs **is** et **as** sont utilisables avec des références d'interfaces en C#. Reprenons l'exemple précédent :

```

abstract class Vehicule { ... }
interface IVehicule { ... }
enum Energie { gaz , fuel , ess }
interface IMoteur { ... }
abstract class UnVehiculeMoteur : Vehicule , IVehicule , IMoteur { ... }
class Voiture : UnVehiculeMoteur { ... }

class UseVoiture1 {
  public string use ( (IVehicule x ){
    if (x is UnVehiculeMoteur) {
      int consom = (x as UnVehiculeMoteur).consommation( );
      return " consommation="+consom.ToString( );
    }
    else
      return x.TypeEngin ;
  }
}

```

## Les conflits de noms dans les interfaces

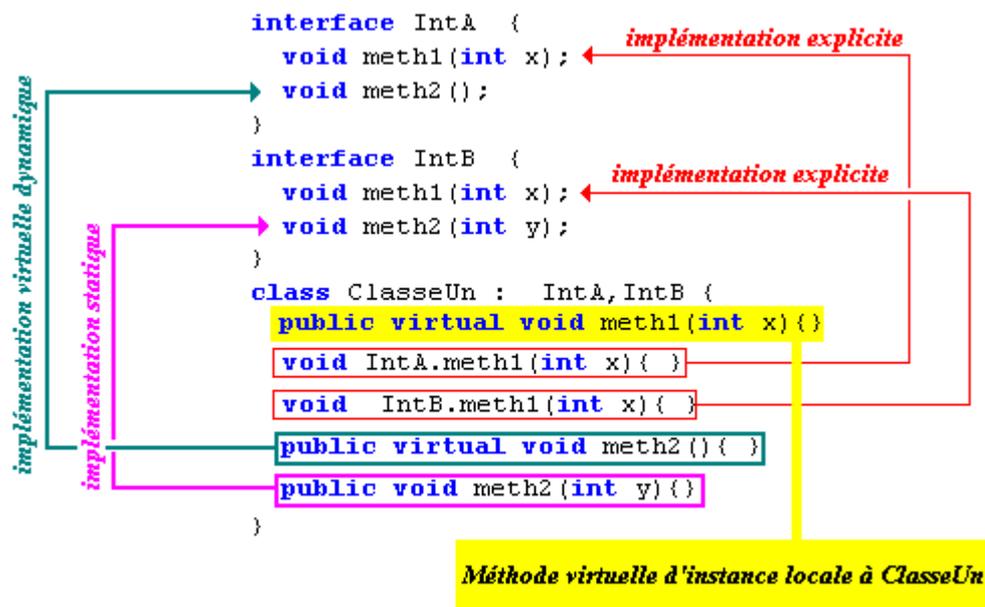
Il est possible que deux interfaces différentes possèdent des membres ayant la même signature. Une classe qui implémente ces deux interfaces se trouvera confrontée à un conflit de nom (ambiguïté). Le compilateur C# exige dès lors que l'ambiguïté soit levée avec le préfixage du nom du membre par celui de l'interface correspondante (implémentation explicite).

L'exemple ci-dessous est figuré avec deux interfaces IntA et IntB contenant chacune deux méthodes portant les mêmes noms et plus particulièrement la méthode meth1 possède la même signature dans chaque interface. Soit ClasseUn une classe implémentant ces deux interfaces. Voici comment fait C# pour choisir les appels de méthodes implantées.

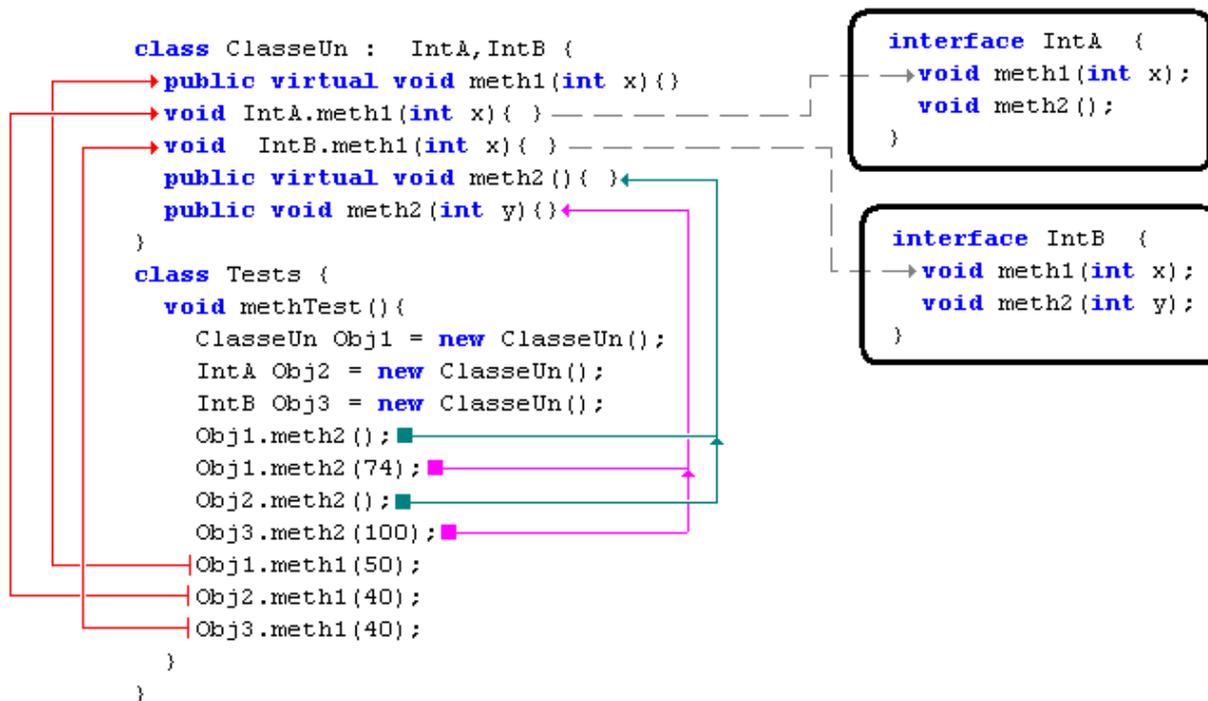
Le code source de ClasseUn implémentant les deux interfaces IntA et IntB :

<pre> <b>interface</b> IntA{   <b>void</b> meth1(<b>int</b> x);   <b>void</b> meth2(); } <b>interface</b> IntB{   <b>void</b> meth1(<b>int</b> x);   <b>void</b> meth2(<b>int</b> y); } </pre>	<pre> <b>class</b> ClasseUn : IntA , IntB {   <b>public virtual void</b> meth1(<b>int</b> x){ }   <b>void</b> IntA.meth1(<b>int</b> x){ }   <b>void</b> IntB.meth1(<b>int</b> x){ }   <b>public virtual void</b> meth2(){ }   <b>public void</b> meth2(<b>int</b> y){ } } </pre>
--	--

Schéma expliquant ce que C# analyse dans le code source précédent :



Lorsque l'on instancie effectivement 3 objets de classe **ClasseUn** précédente, et que l'on déclare chaque objet avec un type de référence différent : classe ou interface, le schéma ci-dessous indique quels sont les différents appels de méthodes corrects possibles :



Il est aussi possible de transtyper une référence d'objet de classe ClasseUn en une référence d'interface dont elle hérite (les appels sont identiques à ceux du schéma précédent) :

```

class Tests {
    void methTest() {
        ClasseUn Obj1 = new ClasseUn();
        IntA Obj2 = (IntA)Obj1;
        IntB Obj3 = (IntB)Obj1;
        Obj1.meth2();
        Obj1.meth2(74);
        Obj2.meth2();
        Obj3.meth2(100);
        Obj1.meth1(50);
        Obj2.meth1(40);
        Obj3.meth1(40);
    }
}

```

Nous remarquons qu'aucun conflit et aucune ambiguïté de méthode ne sont possibles et que grâce à l'implémentation explicite, toutes les méthodes de même nom sont accessibles.

Enfin, nous avons préféré utiliser le transtypage détaillé dans :

```

IntA Obj2 = (IntA)Obj1;
Obj2.meth1(40);

```

Plutôt que l'écriture équivalente :

```

((IntA)Obj1).meth1(40); //...appel de IntA.meth1(int x)

```

Car l'oubli du parenthésage externe dans l'instruction " (( IntA )Obj1).meth1(40) " peut provoquer des incompréhensions dans la mesure où aucune erreur n'est signalé par le compilateur car ce n'est plus la même méthode qui est appelée.

```
( IntA )Obj1.meth1(40) ; //...appel de public virtual ClasseUn.meth1(int x)
```

# Classe de délégation



---

**Plan général:** 

## Les classes de délégations

- 1.1 Définition classe de délégation - délégué
- 1.2 Délégué et méthodes de classe - définition
- 1.3 Délégué et méthodes de classe - informations pendant l'exécution
- 1.4 Délégué et méthodes d'instance - définition
- 1.5 Plusieurs méthodes pour le même délégué
- 1.6 Exécution des méthodes d'un délégué multicast - Exemple de code

Il existe en Delphi la notion de **pointeur de méthode** utilisée pour implanter la notion de gestionnaire d'événements. C# a repris cette idée en l'encapsulant dans un concept objet plus abstrait : la notion de **classes de délégations**. Dans la machine virtuelle CLR, la gestion des événements est fondée sur les classes de délégations, il est donc essentiel de comprendre le modèle du délégué pour comprendre le fonctionnement des événements en C#.

## 1. Les classes de délégations

*Note de microsoft à l'attention des développeurs :*

*La classe **Delegate** est la classe de base pour les types délégués. Toutefois, seuls le système et les compilateurs peuvent dériver de manière explicite de la classe **Delegate** ou **MulticastDelegate**. En outre, il n'est pas possible de dériver un nouveau type d'un type délégué. La classe **Delegate** n'est pas considérée comme un type délégué. Il s'agit d'une classe utilisée pour dériver des types délégués.*

### 1.1 Définition classe de délégation - délégué

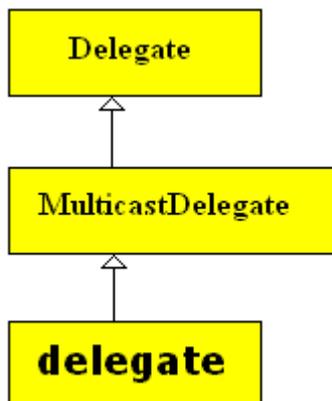
Le langage C# contient un mot clé **delegate**, permettant au compilateur de construire une classe dérivée de la classe **MulticastDelegate** dérivée elle-même de la classe abstraite **Delegate**. Nous ne pouvons pas instancier un objet de classe **Delegate**, car le constructeur est spécifié **protected** et n'est donc pas accessible :

#### Constructeurs protégés

 <a href="#">Delegate, constructeur</a>	Surchargé. Initialise un nouveau délégué.
--	---

Les classes **Delegate** et **MulticastDelegate** sont des classes de base pour construire des délégués, il n'est toutefois pas possible par programme de dériver explicitement (hériter) une classe fille de ces classes, seuls le système et les compilateurs le peuvent. On peut les considérer dans un programme comme des classes **sealed**.

C'est en fait via le mot clé **delegate** que nous allons construire des classes qui ont pour nom : classes de délégations. Ces classes sont des classes du genre référence, elles sont instanciables et un objet de classe délégation est appelé un délégué. Les classes ainsi construites sont automatiquement **sealed** donc non héritables.



**Un objet de classe délégation permet de référencer (pointer vers) une ou plusieurs méthodes.**

Il s'agit donc de l'extension de la notion de pointeur de méthode de Delphi. Selon les auteurs

une classe de délégation peut aussi être nommée classe déléguée, type délégué voir même tout simplement délégué. Il est essentiel dans le texte lu de bien distinguer la classe et l'objet instancié.

Lorsque nous utiliserons le vocable classe délégué ou type délégué nous parlerons de la classe de délégation d'un objet délégué.

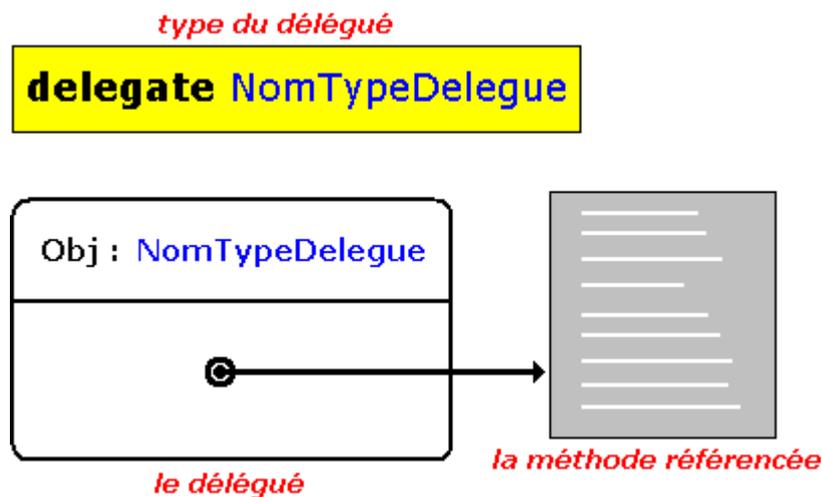
Il est donc possible de créer un nouveau type de délégué (une nouvelle classe) dans un programme, ceci d'une seule manière : en utilisant le qualificateur **delegate**. Ci-dessous nous déclarons un type délégué (une nouvelle classe particulière) nommé **NomTypeDelegue** :

```
delegate string NomTypeDelegue ( int parametre );
```

Un objet délégué peut donc être instancié à partir de cette "classe" comme n'importe quel autre objet :

```
NomTypeDelegue Obj = new NomTypeDelegue ( <paramètre> )
```

Il ne doit y avoir qu'un seul paramètre <paramètre> et c'est obligatoirement un **nom de méthode**. Soit **MethodeXYZ** le nom de la méthode passé en paramètre, nous dirons alors que le délégué (l'objet délégué) référence la méthode **MethodeXYZ**.



Les méthodes référencées par un délégué peuvent être :

des méthodes de classe ( **static** )  
ou  
des méthodes d'instance

Toutes les méthodes référencées par un même délégué ont la même signature partielle :

- même type de retour du résultat,
- même nombre de paramètres,
- même ordre et type des paramètres,
- seul leur nom diffère.

## 1.2 Délégué et méthodes de classe - définition

Un type commençant par le mot clef **delegate** est une classe délégation.

ci-dessous la syntaxe de 2 exemples de déclaration de classe délégation :

- **delegate string** Deleger1( **int** x );
- **delegate void** Deleger2( **string** s );

Un objet instancié à partir de la classe Deleger1 est appelé un délégué de classe Deleger1 :

- **Deleger1** FoncDeleg1 = **new** Deleger1 ( Fonc1 );
- où Fonc1 est une méthode :
- **static string** Fonc1 ( **int** x ) { ... }

Un objet instancié à partir de la classe Deleger2 est appelé un délégué de classe Deleger2 :

- **Deleger2** FoncDeleg2 = **new** Deleger2 ( Fonc2 );
- où Fonc2 est une autre méthode :
- **static void** Fonc2 ( **string** x ) { ... }

Nous avons créé deux types délégations nommés Deleger1 et Deleger2 :

- Le type Deleger1 permet de référencer des méthodes ayant un paramètre de type **int** et renvoyant un **string**.
- Le type Deleger2 permet de référencer des méthodes ayant un paramètre de type **string** et ne renvoyant **rien**.

Les méthodes de classe Fonc1 et Fonc11 répondent à la signature partielle du type Deleger1

- **static string** Fonc1 ( **int** x ) { ... }
- **static string** Fonc11 ( **int** x ) { ... }

On peut créer un objet (un délégué) qui va référencer l'une ou l'autre de ces deux fonctions :

- Deleger1 FoncDeleg1 = **new** Deleger1 ( Fonc1 );  
ou bien
- Deleger1 FoncDeleg1 = **new** Deleger1 ( Fonc11 );

On peut maintenant appeler le délégué FoncDeleg1 dans une instruction avec un paramètre d'entrée de type **int**, selon que le délégué référence Fonc1 ou bien Fonc11 c'est l'une ou l'autre des ces fonctions qui est en fait appelée.

Source d'un exemple C# et exécution :

```
delegate string Deleger1 ( int x );

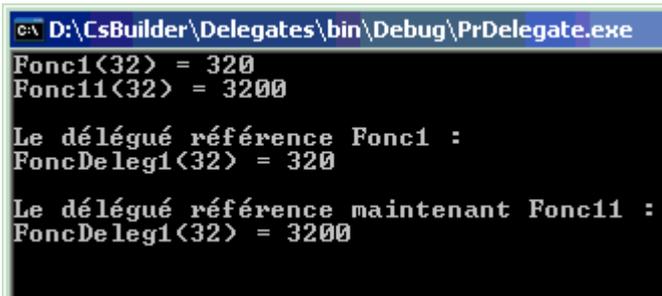
class ClasseA {
    static string Fonc1 ( int x ) {
        return ( x * 10 ).ToString ();
    }
    static string Fonc11 ( int x ) {
        return ( x * 100 ).ToString ();
    }
    static void Main ( string [] args ) {
        string s = Fonc1 ( 32 );
        System.Console.WriteLine ("Fonc1(32) = " + s );
        s = Fonc11 ( 32 ); // appel de fonction classique
        System.Console.WriteLine ("Fonc11(32) = " + s );
        System.Console.WriteLine ("\nLe délégué référence Fonc1 :");
    }
}
```

```

Deleguer1 FoncDeleg1 = new Deleguer1 ( Fonc1 );
s = FoncDeleg1 ( 32 ); // appel au délégué qui appelle la fonction
System .Console.WriteLine ("FoncDeleg1(32) = " + s );
System .Console.WriteLine ("\nLe délégué référence maintenant Fonc11 :");
FoncDeleg1 = new Deleguer1 ( Fonc11 ); // on change d'objet référencé (de fonction)
s = FoncDeleg1 ( 32 ); // appel au délégué qui appelle la fonction
System .Console.WriteLine ("FoncDeleg1(32) = " + s );
System .Console.ReadLine ();
}
}

```

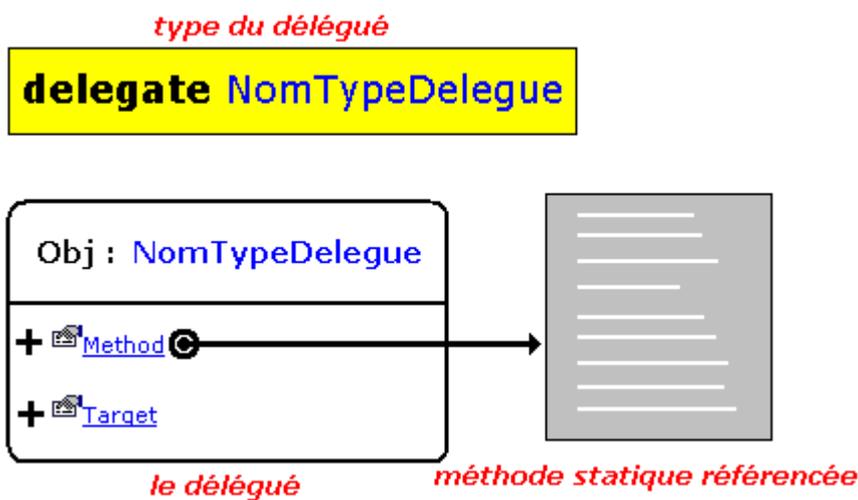
Résultats d'exécution sur la console :



### 1.3 Délégué et méthodes de classe - informations pendant l'exécution

Comme une référence de délégué peut pointer (référencer) vers des méthodes de classes différentes au cours de l'exécution, il est intéressant d'obtenir des informations sur la méthode de classe actuellement référencée par le délégué.

Nous donnons ci-dessous les deux propriétés publiques qui sont utiles lors de cette recherche d'informations, elles proviennent de la classe mère Delegate non héritable par programme :



## Propriétés publiques

<a href="#">Method</a> (hérité de <b>Delegate</b> )	Obtient la méthode static représentée par le délégué.
<a href="#">Target</a> (hérité de <b>Delegate</b> )	Obtient l'instance de classe sur laquelle le délégué en cours appelle la méthode d'instance.

La propriété **Target** sert plus particulièrement lorsque le délégué référence une méthode d'instance, la propriété **Method** est [la seule utilisable](#) lorsque la méthode référencée par le délégué est une méthode de classe (méthode marquée **static**). Lorsque la méthode référencée par le délégué est une méthode de classe le champ **Target** a la valeur **null**.

Ci-dessous nous avons extrait quelques informations concernant la propriété **Method** qui est elle-même de classe MethodInfo :

[Method](#) (hérité de **Delegate**)

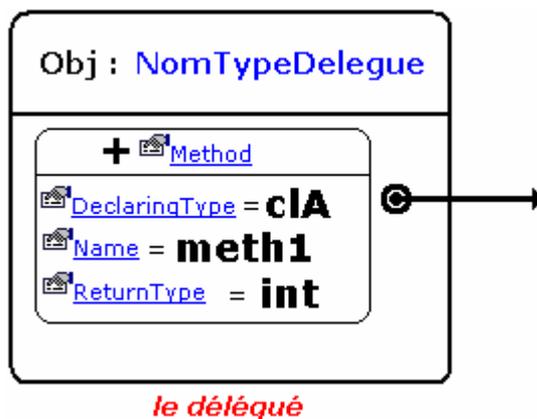
Obtient la méthode static représentée par le délégué.

```
[Serializable]
[ClassInterface(ClassInterfaceType.AutoDual)]
public MethodInfo Method {get;}
```

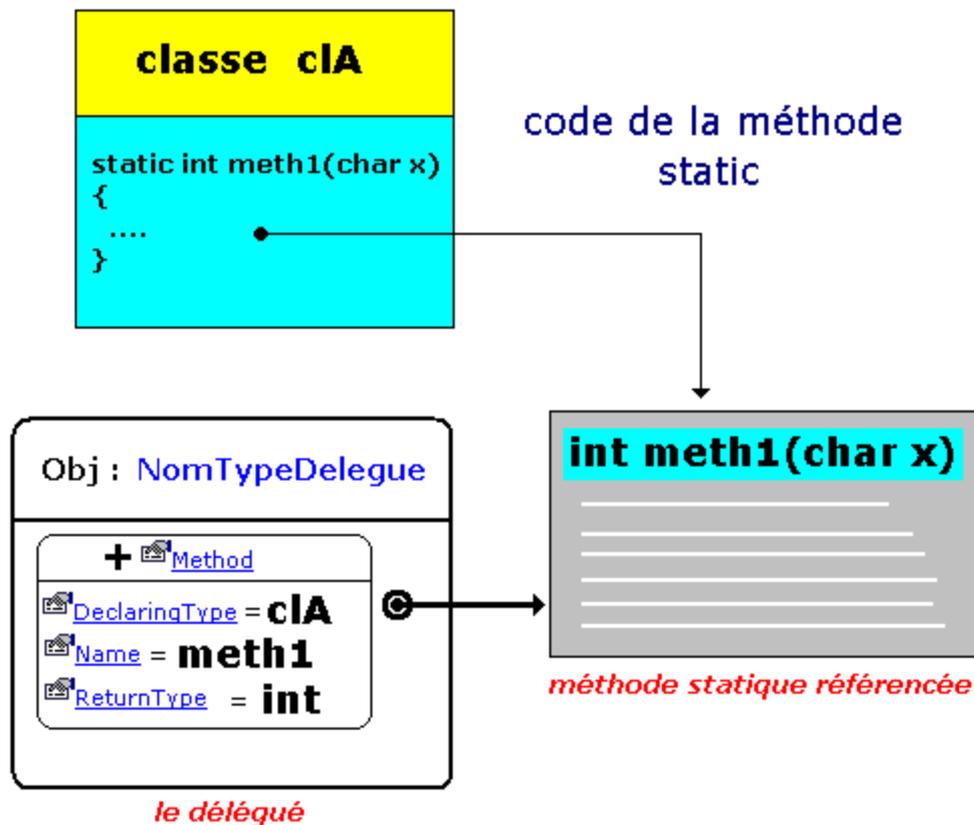
**MethodInfo : Propriétés publiques**

<a href="#">CallingConvention</a> (hérité de <b>MethodBase</b> )	Obtient une valeur indiquant les conventions d'appel de cette méthode.
<a href="#">DeclaringType</a> (hérité de <b>MemberInfo</b> )	Obtient la classe qui déclare ce membre.
<a href="#">Name</a> (hérité de <b>MemberInfo</b> )	Obtient le nom de ce membre.
<a href="#">ReturnType</a>	Obtient le type de retour de cette méthode.

Ces propriétés sont des membres de la propriété **Method** qui est applicable uniquement lorsque le délégué en cours référence une méthode de classe (qualifiée **static**).



Nous illustrons dans la figure ci-après, dans le cas d'une méthode de classe, l'utilisation des propriétés **Name**, **DeclaringType** et **ReturnType** membres de la propriété **Method** :



Nous obtenons ainsi des informations sur le **nom**, le **type** du resultat et la **classe** de la methode **static** pointee par le delégue.

Source complet executable d'un exemple d'information sur la methode de classe referenciee :

```
namespace PrDelegate {
    delegate string Deleger1 ( int x );

    class ClasseA {
        static string Fonc1 ( int x ) {
            return ( x * 10 ).ToString ();
        }

        static void Main ( string [] args ) {
            System.Console.WriteLine ("\nLe delégue reference Fonc1 :");
            Deleger1 FoncDeleg1 = new Deleger1 ( Fonc1 );
            System.Console.WriteLine ( "nom : "+FoncDeleg1.Method.Name );
            System.Console.WriteLine ( "classe : "+FoncDeleg1.Method.DeclaringType.ToString () );
            System.Console.WriteLine ( "retour : "+FoncDeleg1.Method.ReturnType.ToString () );
            System.Console.ReadLine ();
        }
    }
}
```

Résultats d'exécution sur la console :

```

C:\ D:\CsBuilder\Delegates\bin\Debug\PrD
Le délégué référence Fonc1 :
nom : Fonc1
classe : PrDelegate.ClasseA
retour : System.String

```

### 1.4 Délégué et méthodes d'instance - définition

Outre une méthode de classe, un délégué peut pointer aussi vers une méthode d'instance (une méthode d'un objet). Le fonctionnement (déclaration, instanciation, utilisation) est identique à celui du référencement d'une méthode de classe, avec syntaxiquement l'obligation, lors de l'instanciation du délégué, d'indiquer le nom de l'objet ainsi que le nom de la méthode **Obj.Methode** (similitude avec le pointeur de méthode en Delphi).

Ci-dessous la syntaxe d'un exemple de déclaration de classe délégation pour une méthode d'instance, nous devons :

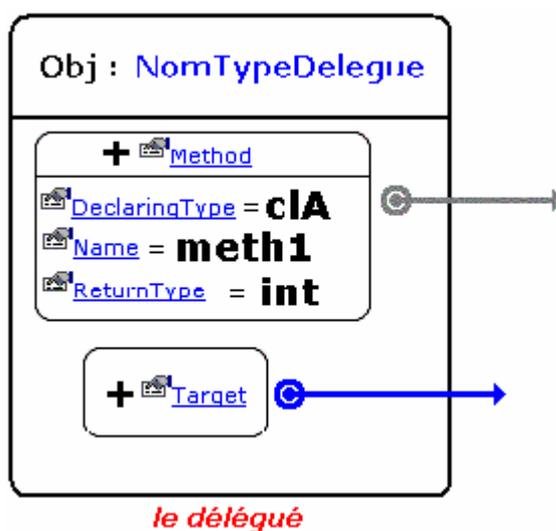
- 1°) Déclarer une classe contenant une méthode public
 

```
class clA {
    public int meth1(char x) { ... }
}
```
- 2°) Déclarer un type délégation
 

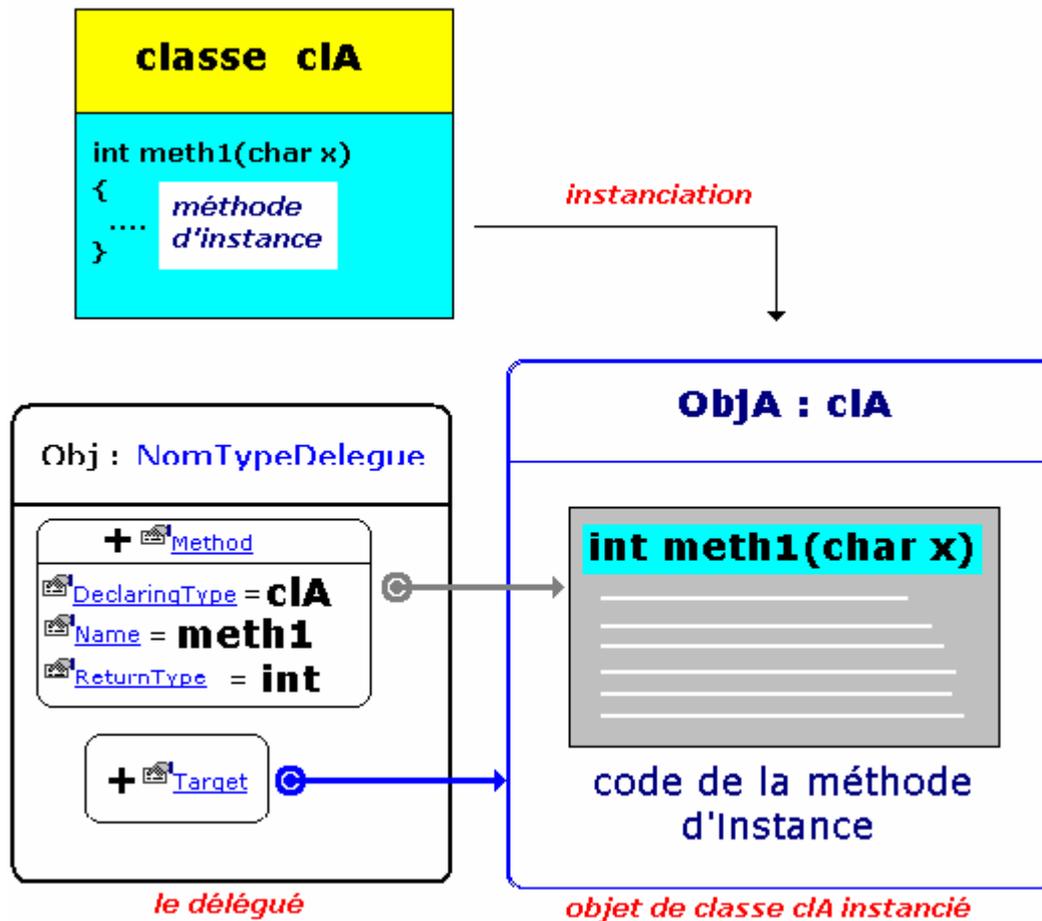
```
delegate int Deleger( char x );
```
- 3°) Instancier un objet de la classe clA
 

```
clA ObjA = new clA ( );
```
- 4°) Instancier à partir de la classe Deleger un délégué
 

```
Deleger FoncDeleg = new Deleger ( ObjA.meth1 );
```



Nous illustrons dans la figure ci-après, dans le cas d'une méthode d'instance, l'utilisation de membres de la propriété **Method** et de la propriété **Target** :



Source complet exécutable d'un exemple d'information sur la méthode d'instance référencée :

```

namespace PrDelegate {
    delegate int Deleger ( char x );

    class ClasseA {
        public int meth1 ( char x ) {
            return x ;
        }
    }

    static void Main ( string [] args ) {
        ClasseA ObjX , ObjA = new ClasseA ( );
        System.Console.WriteLine ("Un délégué référence ObjA.meth1 :");
        Deleger FoncDeleg = new Deleger ( ObjA.meth1 );
        ObjX = (ClasseA)FoncDeleg.Target;
        if (ObjX.Equals(ObjA))
            System.Console.WriteLine ("Target référence bien ObjA");
        else System.Console.WriteLine ("Target ne référence pas ObjA");
        System.Console.WriteLine ( "\nnom : "+FoncDeleg.Method.Name );
        System.Console.WriteLine ("classe : "+ FoncDeleg.Method.DeclaringType.ToString() );
        System.Console.WriteLine ("retour : "+FoncDeleg.Method.ReturnType.ToString() );
        System.Console.ReadLine ();
    }
}

```

Résultats d'exécution sur la console :

```

C:\ D:\CsBuilder\Delegates\bin\Debug\PrDelegate.exe
Un délégué référence ObjA.meth1 :
Target référence bien ObjA

nom : meth1
classe : PrDelegate.ClasseA
retour : System.Int32

```

Dans le programme précédent, les lignes de code suivantes :

```

ObjX = (ClasseA)FoncDeleg.Target ;
if (ObjX.Equals(ObjA))
    System.Console.WriteLine ("Target référence bien ObjA") ;
else System.Console.WriteLine ("Target ne référence pas ObjA") ;

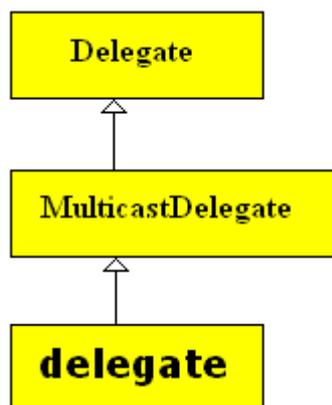
```

servent à faire "pointer" la référence *ObjX* vers l'objet vers lequel pointe *FoncDeleg.Target*. La référence de cet objet est transtypée car *ObjX* est de type *ClasseA*, *FoncDeleg.Target* est de type *Object* et le compilateur n'accepterait pas l'affectation *ObjX = FoncDeleg.Target*. Le test *if (ObjX.Equals(ObjA))...* permet de nous assurer que les deux références *ObjX* et *ObjA* pointent bien vers le même objet.

### 1.5 Plusieurs méthodes pour le même délégué

C# autorise le référencement de plusieurs méthodes par le même délégué, nous utiliserons le vocabulaire de délégué multicast pour bien préciser qu'il référence plusieurs méthodes. Le délégué multicast conserve les référencements dans une liste d'objet. Les méthodes ainsi référencées peuvent chacune être **du genre méthode de classe ou méthode d'instance**, elles **doivent avoir la même signature**.

Rappelons qu'un type délégué multicast est une classe qui hérite intrinsèquement de la classe **MulticastDelegate** :



La documentation de .Net Framework indique que la classe **MulticastDelegate** contient en particulier trois champs privés :

```

Object _target ;
Int32 _methodPtr ;
MulticastDelegate _prev ;

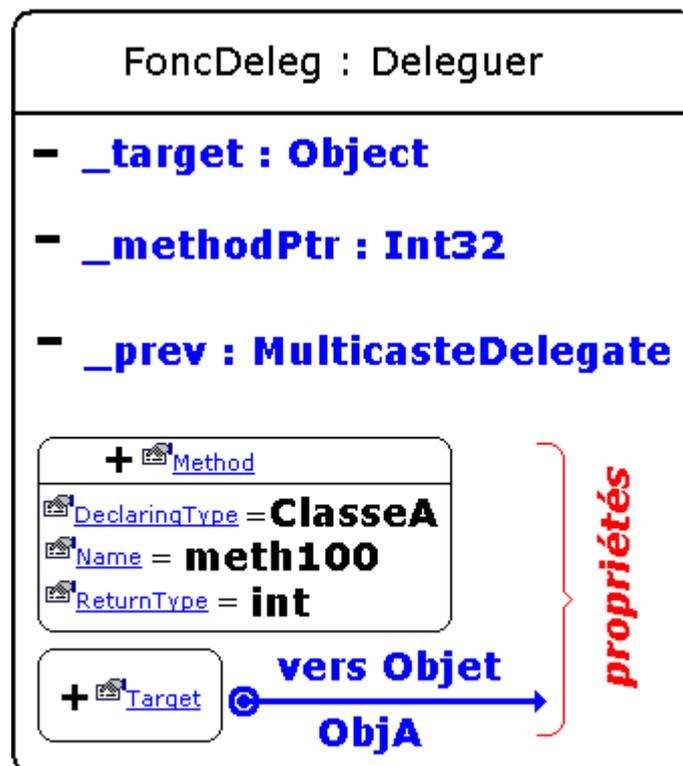
```

Le champ `_prev` est utilisé pour maintenir une liste de `MulticastDelegate`

Lorsque nous déclarons un programme comme celui-ci :

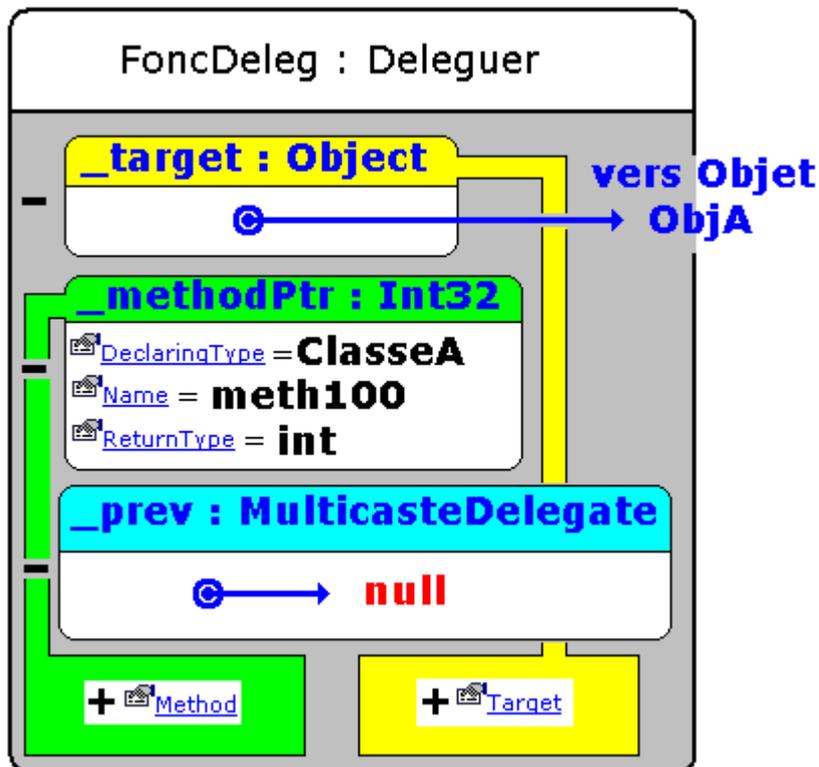
```
delegate int Deleger ( char x );  
  
class ClasseA {  
    public int meth100 ( char x ) {  
        System.Console.WriteLine ("Exécution de meth100("+x+"");  
        return x+100 ;  
    }  
    static void Main ( string [ ] args ) {  
        ClasseA ObjA = new ClasseA();  
        Deleger FoncDeleg = new Deleger ( ObjA.meth100 ) ;  
    }  
}
```

Lors de l'exécution, nous avons vu qu'il y a création d'un `ObjA` de `ClasseA` et création d'un objet délégué `FoncDeleg`, les propriétés **Method** et **Target** sont automatiquement initialisées par le compilateur :



En fait, ce sont les champs privés qui sont initialisés et les propriétés `Method` et `Target` qui sont en lecture seulement, lisent les contenus respectifs de `_methodPtr` et de `_target`; le champ `_prev` est pour l'instant mis à `null`, enfin la méthode `meth100(...)` est actuellement en tête de liste.

Figure virtuelle de l'objet délégué à ce stade :



Il est possible d'ajouter une nouvelle méthode `meth101(...)` au délégué qui va la mettre en tête de liste à la place de la méthode `meth100(...)` qui devient le deuxième élément de la liste. C# utilise l'opérateur d'addition pour implémenter l'ajout d'une nouvelle méthode au délégué. Nous étendons le programme précédent :

```

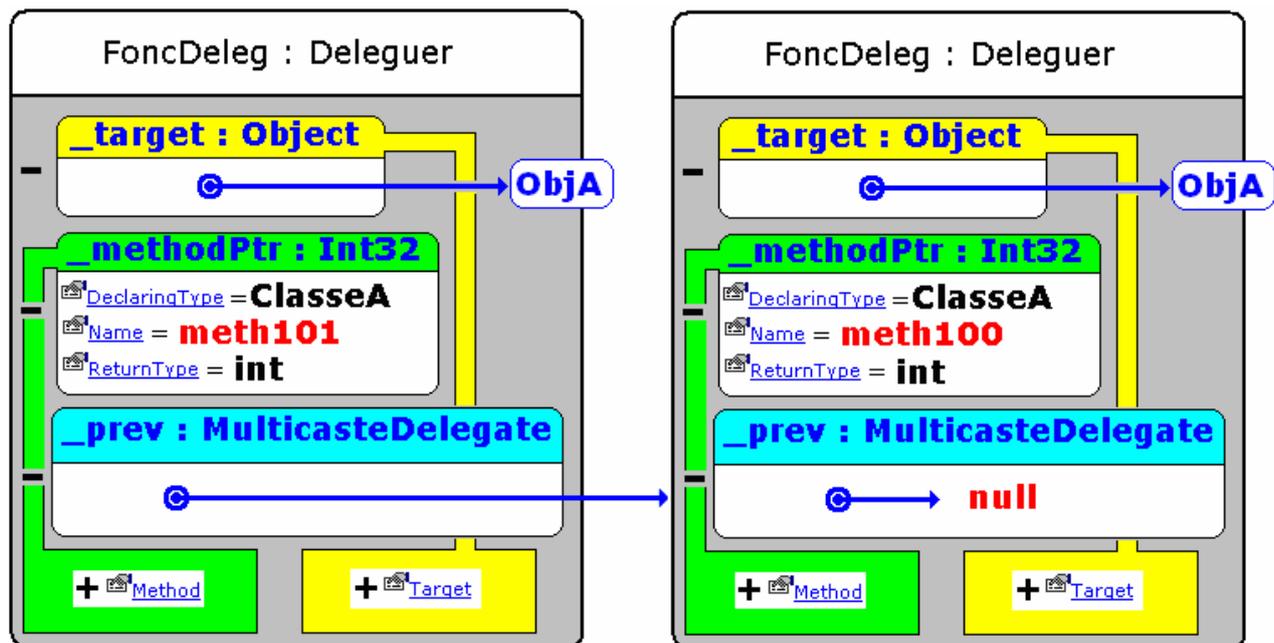
delegate int Deleger ( char x );

class ClasseA {
    public int meth100 ( char x ) {
        System.Console.WriteLine ("Exécution de meth100("+x+"");
        return x+100 ;
    }
    public int meth101 ( char x ) {
        System.Console.WriteLine ("Exécution de meth101("+x+"");
        return x+101 ;
    }
}

static void Main ( string [] args ) {
    ClasseA ObjA = new ClasseA();
    //-- meth100 est en tête de liste :
    Deleger FoncDeleg = new Deleger ( ObjA.meth100 );
    // meth101 est ajoutée en tête de liste devant meth100 :
    FoncDeleg = FoncDeleg + new Deleger ( ObjA.meth101 );
}
}

```

Figure virtuelle de l'objet délégué à cet autre stade :



C# permet de consulter et d'utiliser si nous le souhaitons toutes les références de méthodes en nous renvoyant la liste dans un tableau de référence de type **Delegate** grâce à la méthode **GetInvocationList**. La ligne de code source ci-dessous retourne dans le tableau Liste, la liste d'appel dans l'ordre d'appel, du délégué FoncDeleg :

```
Delegate[] Liste = FoncDeleg.GetInvocationList();
```

### 1.6 Exécution des méthodes d'un délégué multicast - Exemple de code

Lorsque l'on invoque le délégué sur un paramètre effectif, C# appelle et exécute séquentiellement les méthodes contenues dans la liste jusqu'à épuisement. **L'ordre d'appel est celui du stockage** : la première stockée est exécutée en premier, la suivante après, **la dernière méthode ajoutée est exécutée en dernier**, s'il y a un **résultat de retour, c'est celui de la dernière méthode ajoutée qui est renvoyé**, les autres résultats de retour sont ignorés. L'exemple ci-dessous reprend les notions que nous venons d'exposer.

Source complet exécutable d'un exemple de délégué multicast :

```
namespace PrDelegate {
    delegate int Deleger ( char x );

    class ClasseA {
        public int champ;

        public int meth100 ( char x ) {
            System.Console.WriteLine ("Exécution de meth100("+x+"");
            return x+100 ;
        }
        public int meth101 ( char x ) {
            System.Console.WriteLine ("Exécution de meth101("+x+"");
            return x+101 ;
        }
    }
}
```

```

public int meth102 ( char x ) {
    System.Console.WriteLine ("Exécution de meth102("+x+"");
    return x+102 ;
}
public static int meth103 ( char x ) {
    System.Console.WriteLine ("Exécution de meth103("+x+"");
    return x+103 ;
}

static void Main ( string [] args ) {
    System.Console.WriteLine ("Un délégué référence ObjA.meth1 :");
    ClasseA ObjX , ObjA = new ClasseA();
    /-- instanciation du délégué avec ajout de 4 méthodes :
    Deleger FoncDeleg = new Deleger ( ObjA.meth100 );
    FoncDeleg += new Deleger ( ObjA.meth101 );
    FoncDeleg += new Deleger ( ObjA.meth102 );
    FoncDeleg += new Deleger ( meth103 );

    /--la méthode meth103 est en tête de liste :
    ObjX = (ClasseA)FoncDeleg.Target ;
    if (ObjX == null) System.Console.WriteLine ("Méthode static, Target = null");
    else if (ObjX.Equals(ObjA))System.Console.WriteLine ("Target référence bien ObjA");
    else System.Console.WriteLine ("Target ne référence pas ObjA");
    System.Console.WriteLine ( "\nnom : "+FoncDeleg.Method.Name );
    System.Console.WriteLine ( "classe : "+FoncDeleg.Method.DeclaringType.ToString() );
    System.Console.WriteLine ( "retour : "+FoncDeleg.Method.ReturnType.ToString() );

    /--Appel du délégué sur le paramètre effectif 'a' :
    ObjA.champ = FoncDeleg('a') ; //code ascii 'a' = 97
    System.Console.WriteLine ( "\nvaleur du champ : "+ObjA.champ) ;
    System.Console.WriteLine ( "-----");

    /-- Parcours manuel de la liste des méthodes référencées :
    Delegate[] Liste = FoncDeleg.GetInvocationList() ;
    foreach ( Delegate Elt in Liste )
    {
        ObjX = (ClasseA)Elt.Target ;
        if (ObjX == null) System.Console.WriteLine ("Méthode static, Target = null");
        else if (ObjX.Equals(ObjA))System.Console.WriteLine ("Target référence bien ObjA");
        else System.Console.WriteLine ("Target ne référence pas ObjA");
        System.Console.WriteLine ( "\nnom : "+Elt.Method.Name );
        System.Console.WriteLine ( "classe : "+Elt.Method.DeclaringType.ToString() );
        System.Console.WriteLine ( "retour : "+Elt.Method.ReturnType.ToString() );
        System.Console.WriteLine ( "-----");
    }
    System.Console.ReadLine ( ) ;
}
}
}

```

Résultats d'exécution sur la console :

```

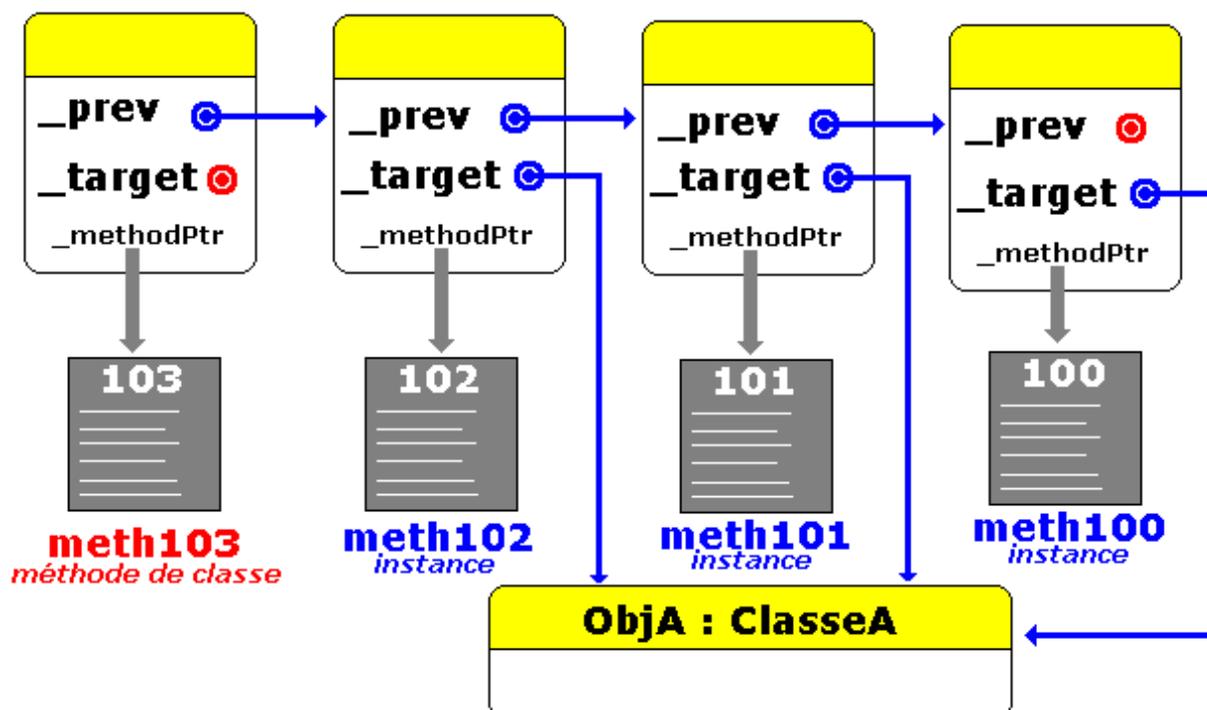
D:\CsBuilder\Delegates\bin\Debug\PrDelegate.exe
Un délégué référence ObjA.meth1 :
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
Exécution de meth100('a')
Exécution de meth101('a')
Exécution de meth102('a')
Exécution de meth103('a')

valeur du champ : 200
-----
Target référence bien ObjA
nom : meth100
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Target référence bien ObjA
nom : meth101
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Target référence bien ObjA
nom : meth102
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Méthode static, Target = null
nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
-----

```

Nous voyons bien que le délégué **FoncDeleg** contient la liste des référencements des méthodes meth100, meth101, meth102 et meth103 ordonné comme figuré ci-dessous :



Remarquons que le premier objet de la liste est une référence sur une méthode de classe, la

propriété Target renvoie la valeur **null** (le champ \_target est à **null**).

La méthode de classe meth103 ajoutée en dernier est bien en tête de liste :

```
Un délégué référence ObjA.meth1 :
Méthode static, Target = null

nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
```

L'invocation du délégué lance l'exécution séquentielle des 4 méthodes :

```
Exécution de meth100('a')
Exécution de meth101('a')
Exécution de meth102('a')
Exécution de meth103('a')
```

et le retour du résultat est celui de meth103('a') :

```
valeur du champ : 200
```

Le parcours manuel de la liste montre bien que ce sont des objets de type **Delegate** qui sont stockés et que l'on peut accéder entre autre possibilités, à leurs propriétés :

```
Target référence bien ObjA
nom : meth100
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Target référence bien ObjA
nom : meth101
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Target référence bien ObjA
nom : meth102
classe : PrDelegate.ClasseA
retour : System.Int32
-----
Méthode static, Target = null
nom : meth103
classe : PrDelegate.ClasseA
retour : System.Int32
-----
```

# Les exceptions en C#.net

---

**Plan général:** 

## 1. Les exceptions : syntaxe, rôle, classes

- 1.1 Comment gérer une exception dans un programme
- 1.2 Principe de fonctionnement de l'interception

## 2. Interception d'exceptions hiérarchisées

- 2.1 Interceptions de plusieurs exceptions
- 2.2 Ordre d'interception d'exceptions hiérarchisées

## 3. Redéclenchement d'une exception mot-clef : throw

- 3.1 Déclenchement manuel d'une exception de classe déjà existante
- 3.2 Déclenchement manuel d'une exception personnalisée

## 4. Clause finally

## 5. Un exemple de traitement d'exceptions sur des fichiers

## 6. Une solution de l'exemple précédent en C#

# 1. Les exceptions : syntaxe, rôle, classes

Rappelons au lecteur que la sécurité de fonctionnement d'une application peut être rendue instable par toute une série de facteurs :

Des problèmes liés au matériel : par exemple la perte subite d'une connexion à un port, un disque défectueux... Des actions imprévues de l'utilisateur, entraînant par exemple une division par zéro... Des débordements de stockage dans les structures de données...

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue, lors d'actions sur les données. Le logiciel doit donc se "**défendre**" contre de tels incidents potentiels, nous nommerons cette démarche la programmation défensive !

## Programmation défensive

La **programmation défensive** est une attitude de pensée consistant à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

En programmation défensive il est possible de protéger directement le code à l'aide de la notion d'exception. L'objectif principal est d'améliorer la qualité de "**robustesse**" (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et efficace de logiciels robustes.

## Rôle d'une exception

Une exception est chargée de signaler un comportement **exceptionnel** (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même. C'est le cas de C# qui intègre les exceptions comme une **classe particulière**: la classe **Exception**. Cette classe contient un nombre important de classes dérivées.

## Comment agit une exception

Dès qu'une erreur se produit comme un manque de mémoire, un calcul impossible, un fichier inexistant, un transtypage non valide,..., un **objet de la classe adéquate dérivée de la classe Exception** est instancié. Nous dirons que le logiciel "**déclenche une exception**".

### 1.1 Comment gérer une exception dans un programme

#### Programme sans gestion de l'exception

Soit un programme C# contenant un incident d'exécution (une division par zéro dans l'instruction  $x = 1/0;$ ) dans la méthode meth() de la classe Action1, cette méthode est appelée dans la classe UseAction1 à travers un objet de classe Action1 :

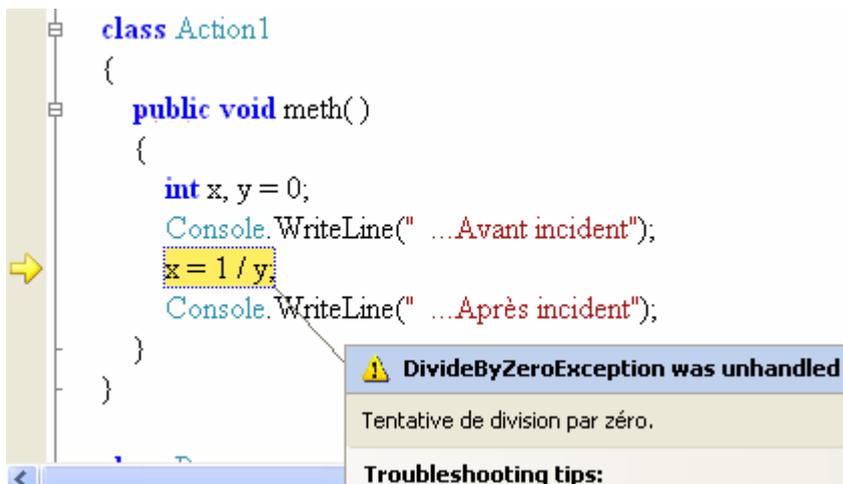
```

class Action1 {
    public void meth()
    {
        int x, y =0;
        Console.WriteLine(" ...Avant incident");
        x = 1 / y;
        Console.WriteLine(" ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1();
        Console.WriteLine("Début du programme.");
        Obj.meth();
        Console.WriteLine("Fin du programme.");
    }
}

```

Lors de l'exécution, après avoir affiché les chaînes "**Début du programme**" et "**...Avant incident**", le programme s'arrête et le CLR signale une erreur. Voici ci-dessous l'affichage obtenu dans Visual C# :



Sur la console l'affichage est le suivant :



*Le programme s'est arrêté à cet endroit et ne peut plus poursuivre son exécution.*

**Que s'est-il passé ?**

La méthode Main :

- a instancié un objet Obj de classe Action1,
- a affiché sur la console la phrase " **Début du programme** ",
- a invoqué la méthode meth() de l'objet Obj,
- a affiché sur la console la phrase " **...Avant incident**",
- a exécuté l'instruction "x = 1/0;"

Dès que l'instruction "x = 1/0;" a été exécutée celle-ci a provoqué un incident. **En fait une exception de la classe `DivideByZeroException` a été "levée" (un objet de cette classe a été instancié) par le CLR**, cette classe hérite de la classe `ArithmeticException` selon la hiérarchie d'héritage suivante de .Net Framework :

```

System.Object
  |__ System.Exception
      |__ System.SystemException
          |__ System.ArithmeticException
              |__ System.DivideByZeroException
  
```

La classe mère de **toutes** les exceptions de .Net Framework est la classe **Exception**.

Le CLR a arrêté le programme immédiatement à cet endroit parce qu'elle n'a pas trouvé de code d'interception de cette exception qu'il a levée automatiquement :

```

class Action1 {
    public void meth()
    {
        int x, y=0;
        Console.WriteLine(" ... Avant incident");
        x = 1 / y;
        Console.WriteLine(" ... Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1();
        Console.WriteLine("Début du programme. ");
        Obj.meth();
        Console.WriteLine("Fin du programme. ");
    }
}
  
```

Sortie du bloc

Sortie du bloc

Nous allons voir comment intercepter (on dit aussi "attraper" - to catch) cette exception afin de faire réagir notre programme pour qu'il ne s'arrête pas brutalement.

### Programme avec gestion de l'exception

C# possède une instruction qui permet d'intercepter des exceptions dérivant de la classe `Exception` :

#### try ... catch

On dénomme cette instruction : un gestionnaire d'exceptions. Syntaxe **minimale** d'un tel gestionnaire `try ... catch` :

```

try
  
```

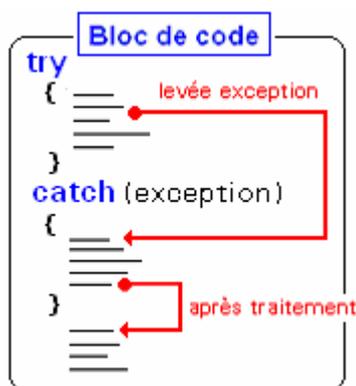
```

{
    <lignes de code à protéger>
}
catch ( UneException )
{
    <lignes de code réagissant à l'exception UneException >
}

```

Le type **UneException** est obligatoirement une classe qui **hérite de la classe Exception**.

Schéma du fonctionnement d'un tel gestionnaire :



Le gestionnaire d'exception "déroute" l'exécution du programme vers le bloc d'interception catch qui traite l'exception (exécute le code contenu dans le bloc catch), puis renvoie et continue l'exécution du programme vers le code situé après le gestionnaire lui-même.

## 1.2 Principe de fonctionnement de l'interception

Dès qu'une **exception est levée** (instanciée), le CLR **stoppe immédiatement** l'exécution normale du programme à la **recherche d'un gestionnaire** d'exception susceptible d'intercepter (saisir) et de traiter cette exception. Cette recherche s'effectue à partir du **bloc englobant** et se poursuit sur les blocs plus englobants si aucun gestionnaire de **cette exception** n'a été trouvé.

Soit le même programme C# que précédemment, contenant un incident d'exécution (une division par zéro dans l'instruction `x = 1/0;` ). Cette fois nous allons gérer l'incident grâce à un gestionnaire d'exception **try..catch** dans le bloc englobant immédiatement supérieur.

### Programme avec traitement de l'incident par try...catch :

```

class Action1 {
    public void meth()
    {
        int x, y =0;
        Console.WriteLine(" ...Avant incident");
        x = 1 / y;
        Console.WriteLine(" ...Après incident");
    }
}

class Program {

```

```

static void Main(string[] args)
{
    Action1 Obj = new Action1();
    Console.WriteLine("Début du programme.");
    try
    {
        Obj.meth();
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Interception exception");
    }
    Console.WriteLine("Fin du programme.");
}
}

```

Figuration du déroulement de l'exécution de ce programme :

```

class Action1 {
    public void meth()
    {
        int x, y=0;
        Console.WriteLine(" ...Avant incident");
        x = 1 / y; ← engendre une exception
        Console.WriteLine(" ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth(); ← levée d'une DivideByZeroException
        }
        catch (DivideByZeroException) ← traitement puis
        {
            Console.WriteLine("Interception exception"); ← poursuite
        }
        Console.WriteLine("Fin du programme."); ← de l'exécution
    }
}

```

Ci-dessous l'affichage obtenu sur la console lors de l'exécution de ce programme :

```

Début du programme.
...Avant incident
Interception d'une OverflowException
Fin du programme.

```

- Nous remarquons que le CLR a donc bien exécuté le code d'interception situé dans le corps du "catch (DivideByZeroException){...}", il a poursuivi l'exécution normale

après le gestionnaire.

- Le gestionnaire d'exception se situe dans la méthode Main (code englobant) qui appelle la méthode meth() qui lève l'exception.

Il est aussi possible d'atteindre l'objet d'exception qui a été instancié (ou levé) en déclarant un identificateur local au bloc catch du gestionnaire d'exception try ... catch, cet objet est disponible dans tout le corps de la clause catch. Dans l'exemple qui suit, on déclare un objet Except de classe `DivideByZeroException` et l'on lit le contenu de deux de ses propriétés `Message` et `Source` :

```
try{
    Obj.meth();
}
catch ( DivideByZeroException Except ) {
    // accès aux membres publiques de l'objet Except :
    Console.WriteLine("Interception exception message : " + Except.Message);
    Console.WriteLine("Interception exception source : " + Except.Source);
    ....
}
```

## 2. Interception d'exceptions hiérarchisées

### 2.1 Interceptions de plusieurs exceptions

Dans un gestionnaire try...catch, il est en fait possible d'intercepter plusieurs types d'exceptions différentes et de les traiter.

Ci-après nous montrons la syntaxe d'un tel gestionnaire qui fonctionne comme un sélecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.

Dès qu'une exception intervient dans le < bloc de code à protéger>, le CLR scrute séquentiellement toutes les clauses **catch** de la première jusqu'à la nième. Si l'exception actuellement levée est d'un des types présents dans la liste des clauses le traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

```
try {
    < bloc de code à protéger >
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

Où TypeException1, TypeException2, ... , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes**.

Seule une seule clause **catch ( TypeException E ) { ... }** est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

## Exemple théorique :

Soit `SystemException` la classe des exceptions prédéfinies dans le nom d'espace `System` de `.Net Framework`, `InvalidCastException`, `IndexOutOfRangeException`, `NullReferenceException` et `ArithmeticException` sont des classes dérivant directement de la classe `SystemException`.

Supposons que la méthode `meth()` de la classe `Action2` puisse lever quatre types différents d'exceptions: `InvalidCastException`, `IndexOutOfRangeException`, `NullReferenceException` et `ArithmeticException`.

Notre gestionnaire d'exceptions est programmé pour intercepter l'une de ces 4 catégories. Nous figurons ci-dessous les trois schémas d'exécution correspondant chacun à la levée (l'instanciation d'un objet) d'une exception de l'un des trois types et son interception :

```
class Action2 {
    public void meth()
    {
        // une exception est levée .....
    }
}

class Program {
    static void Main(string[] args)
    {
        Action2 Obj = new Action2();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch ( InvalidCastException E )
        {
            Console.WriteLine("Interception InvalidCastException");
        }
        catch ( IndexOutOfRangeException E )
        {
            Console.WriteLine("Interception IndexOutOfRangeException ");
        }
        catch ( NullReferenceException E )
        {
            Console.WriteLine("Interception NullReferenceException");
        }
        catch ( ArithmeticException E )
        {
            Console.WriteLine("Interception ArithmeticException");
        }
        Console.WriteLine("Fin du programme.");
    }
}
```

Nous figurons ci-après deux schémas d'interception sur l'ensemble des quatre possibles d'une éventuelle exception qui serait levée dans la méthode `meth()` de l'objet `Obj`.

### ❑ Schéma d'interception d'une `IndexOutOfRangeException` :

```

static void Main(string[] args)
{
    Action2 Obj = new Action2();
    Console.WriteLine("Début du programme.");
    try
    {
        Obj.meth();
    }
    catch ( InvalidCastException E )
    {
        Console.WriteLine("Interception InvalidCastException");
    }
    catch ( IndexOutOfRangeException E )
    {
        Console.WriteLine("Interception IndexOutOfRangeException ");
    }
    catch ( NullReferenceException E )
    {
        Console.WriteLine("Interception NullReferenceException");
    }
    catch ( ArithmeticException E )
    {
        Console.WriteLine("Interception ArithmeticException");
    }
    Console.WriteLine("Fin du programme.");
}

```

Diagram illustrating the execution flow of the code above:

- A red box highlights `Obj.meth();` with the label "levée d'une IndexOutOfRangeException".
- A red arrow points from this box to the `catch ( IndexOutOfRangeException E )` block, labeled "Traitement".
- Another red arrow points from the end of the `try` block to the `Console.WriteLine("Fin du programme.");` line, labeled "poursuite de l'exécution après le bloc du gestionnaire".

□ Schéma d'interception d'une ArithmeticException :

```

static void Main(string[] args)
{
    Action2 Obj = new Action2();
    Console.WriteLine("Début du programme.");
    try
    {
        Obj.meth();
    }
    catch ( InvalidCastException E )
    {
        Console.WriteLine("Interception InvalidCastException");
    }
    catch ( IndexOutOfRangeException E )
    {
        Console.WriteLine("Interception IndexOutOfRangeException ");
    }
    catch ( NullReferenceException E )
    {
        Console.WriteLine("Interception NullReferenceException");
    }
    catch ( ArithmeticException E )
    {
        Console.WriteLine("Interception ArithmeticException");
    }
    Console.WriteLine("Fin du programme.");
}

```

Diagram illustrating the execution flow of the code above:

- A red box highlights `Obj.meth();` with the label "levée d'une ArithmeticException".
- A red arrow points from this box to the `catch ( ArithmeticException E )` block, labeled "Traitement".
- Another red arrow points from the end of the `try` block to the `Console.WriteLine("Fin du programme.");` line, labeled "poursuite de l'exécution après le bloc du gestionnaire".

## 2.2 Ordre d'interception d'exceptions hiérarchisées

Dans un gestionnaire `try...catch` comprenant plusieurs clauses, la recherche de la clause `catch` contenant le traitement de la classe d'exception appropriée, s'effectue séquentiellement dans l'ordre d'écriture des lignes de code.

Soit le pseudo-code C# suivant :

```

try {
    < bloc de code à protéger générant un objet exception >
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
.....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }

```

La recherche va s'effectuer comme si le programme contenait des **if...else if...** imbriqués :

```

if (<Objet exception> is TypeException1) { <Traitement TypeException1 > }
else if (<Objet exception> is TypeException2) { <Traitement TypeException2 > }
...
else if (<Objet exception> is TypeExceptionk) { <Traitement TypeExceptionk > }

```

Les tests sont effectués sur l'appartenance de l'objet d'exception à une classe à l'aide de l'opérateur **is**.

Signalons que l'opérateur **is** agit sur une classe et ses classes filles (sur une hiérarchie de classes), c'est à dire que tout objet de classe `TypeExceptionX` est aussi considéré comme un objet de classe parent au sens du test d'appartenance en particulier cet objet de classe `TypeExceptionX` est aussi considéré objet de classe `Exception` qui est la classe mère de toutes les exceptions `C#`.

Le test d'appartenance de classe dans la recherche d'une clause **catch** fonctionne d'une façon identique à l'opérateur **is** dans les `if...else`

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

**Exemple** : Soit une hiérarchie d'exceptions de `C#`

```

System.Exception
|__ System.SystemException
|   |__ System.InvalidCastException
|   |__ System.IndexOutOfRangeException
|   |__ System.NullReferenceException
|   |__ System.ArithmeticException

```

Soit le modèle de gestionnaire d'interception déjà fourni plus haut :

```

try { < bloc de code à protéger générant un objet exception > }
catch ( InvalidCastException E ) { <Traitement InvalidCastException > }
catch ( IndexOutOfRangeException E ) { <Traitement IndexOutOfRangeException > }
catch ( NullReferenceException E ) { <Traitement NullReferenceException > }
catch ( ArithmeticException E ) { <Traitement ArithmeticException > }

```

Supposons que nous souhaitions intercepter une cinquième classe d'exception, par exemple

une `DivideByZeroException`, nous devons rajouter une clause :

```
catch (DivideByZeroException E ) { <Traitement DivideByZeroException > }
```

Insérons cette clause en dernier dans la liste des clauses d'interception :

```
try
{
    Obj.meth();
}
catch ( InvalidCastException E )
{
    Console.WriteLine("Interception InvalidCastException");
}
catch ( IndexOutOfRangeException E )
{
    Console.WriteLine("Interception IndexOutOfRangeException ");
}
catch ( NullReferenceException E )
{
    Console.WriteLine("Interception NullReferenceException");
}
catch ( ArithmeticException E )
{
    Console.WriteLine("Interception ArithmeticException");
}
catch (DivideByZeroException E )
{
    Console.WriteLine("Interception DivideByZeroException ");
}
Console.WriteLine("Fin du programme.");
}
```

Nous lançons ensuite la compilation de cette classe et nous obtenons un message d'erreur :

**Error 1 :** *Une clause catch précédente intercepte déjà toutes les expressions de this ou d'un super type ('System.ArithmeticException')*

Le compilateur proteste à partir de la clause `catch (DivideByZeroException E )` en nous indiquant que l'exception est déjà interceptée.

## Que s'est-il passé ?

Le fait de placer en premier la clause `catch (ArithmeticException E )` chargée d'intercepter les exceptions de classe `ArithmeticException` implique que n'importe quelle exception héritant de `ArithmeticException` comme par exemple `DivideByZeroException`, est considérée comme une `ArithmeticException`. Dans un tel cas cette `DivideByZeroException` est interceptée par la clause `catch (ArithmeticException E )` mais elle **n'est jamais interceptée** par la clause `catch (DivideByZeroException E )`.

Le seul endroit où le compilateur C# acceptera l'écriture de la clause `catch (DivideByZeroException E )` se situe dans cet exemple **avant la clause catch (ArithmeticException E )**. Ci-dessous l'écriture d'un programme correct :

```
try
{
    Obj.meth();
}
catch ( InvalidCastException E )
```

```

    {
        Console.WriteLine("Interception InvalidCastException");
    }
    catch ( IndexOutOfRangeException E )
    {
        Console.WriteLine("Interception IndexOutOfRangeException ");
    }
    catch ( NullReferenceException E )
    {
        Console.WriteLine("Interception NullReferenceException");
    }
    catch (DivideByZeroException E )
    {
        Console.WriteLine("Interception DivideByZeroException ");
    }
    catch ( ArithmeticException E )
    {
        Console.WriteLine("Interception ArithmeticException");
    }
    Console.WriteLine("Fin du programme.");
}

```

Dans ce cas la recherche séquentielle dans les clauses permettra le filtrage correct des classes filles puis ensuite le filtrage des classes mères.

**On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs clauses dans l'ordre inverse de la hiérarchie.**

### 3. Redéclenchement d'une exception mot-clef : throw

Il est possible de déclencher soi-même des exceptions en utilisant l'instruction **throw**, voir même de déclencher des exceptions personnalisées ou non. Une exception personnalisée est une classe héritant de la classe `System.Exception` définie par le développeur lui-même.

#### 3.1 Déclenchement manuel d'une exception de classe déjà existante

Le CLR peut déclencher une exception automatiquement comme dans l'exemple de la levée d'une `DivideByZeroException` lors de l'exécution de l'instruction "x = 1/y ;".

Le CLR peut aussi lever (déclencher) une exception à votre demande suite à la rencontre d'une instruction **throw**. Le programme qui suit lance une `ArithmeticException` (ie: instancie un objet de type `ArithmeticException`) avec le message "Mauvais calcul !" dans la méthode `meth()` et intercepte cette exception dans le bloc englobant **Main**. Le traitement de cette exception consiste à afficher le contenu du champ message de l'exception grâce à la propriété **Message** de l'exception :

```

class Action3
{
    public void meth()
    {
        int x = 0;

```

```

    Console.WriteLine(" ...Avant incident");
    if (x == 0)
        throw new ArithmeticException("Mauvais calcul !");
    Console.WriteLine(" ...Après incident");
}
}

class Program {
    static void Main(string[] args)
    {
        Action3 Obj = new Action3();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch (ArithmeticException E)
        {
            Console.WriteLine("Interception ArithmeticException : "+E.Message);
        }
        Console.WriteLine("Fin du programme.");
    }
}

```

*Résultats de l'exécution du programme précédent :*

```

Début du programme.
...Avant incident
Interception ArithmeticException : Mauvais calcul ?
Fin du programme.

```

### 3.2 Déclenchement manuel d'une exception personnalisée

Pour une exception personnalisée, le mode d'action est strictement identique, il vous faut seulement auparavant créer une nouvelle classe **héritant obligatoirement de la classe Exception** ou de n'importe laquelle de ses sous-classes.

Reprenons le programme précédent et créons une classe d'exception que nous nommerons **ArithmeticExceptionPerso** héritant de la classe des **ArithmeticException** puis exécutons ce programme :

```

class ArithmeticExceptionPerso : ArithmeticException
{
    public ArithmeticExceptionPerso(String s) : base(s)
    {
    }
}

class Action3
{
    public void meth()
    {
        int x = 0;
        Console.WriteLine(" ...Avant incident");
        if (x == 0)
            throw new ArithmeticExceptionPerso ("Mauvais calcul !");
        Console.WriteLine(" ...Après incident");
    }
}

class Program {

```

```

static void Main(string[] args)
{
    Action3 Obj = new Action3();
    Console.WriteLine("Début du programme.");
    try
    {
        Obj.meth();
    }
    catch (ArithmeticExceptionPerso E)
    {
        Console.WriteLine("Interception ArithmeticExceptionPerso: "+E.Message);
    }
    Console.WriteLine("Fin du programme.");
}
}

```

*Résultats de l'exécution du programme précédent :*

**Début du programme.**

**...Avant incident**

**Interception ArithmeticExceptionPerso : Mauvais calcul !**

**Fin du programme.**

L'exécution de ce programme est identique à celle du programme précédent, notre exception personnalisée fonctionne bien comme les exceptions prédéfinies de C#.

## 4. Clause finally

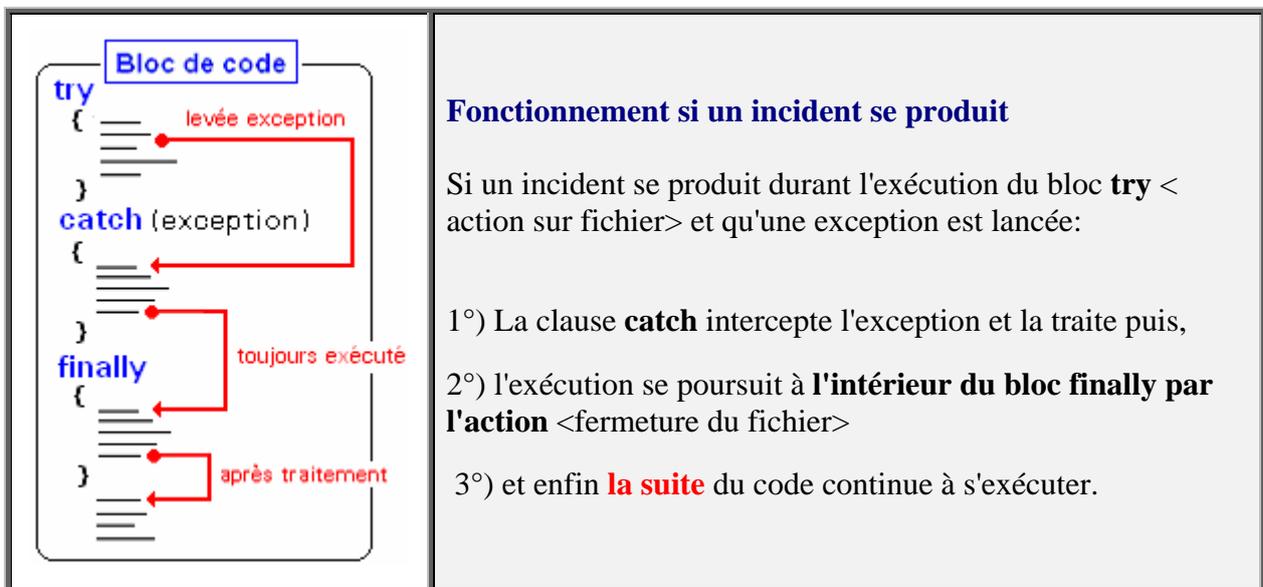
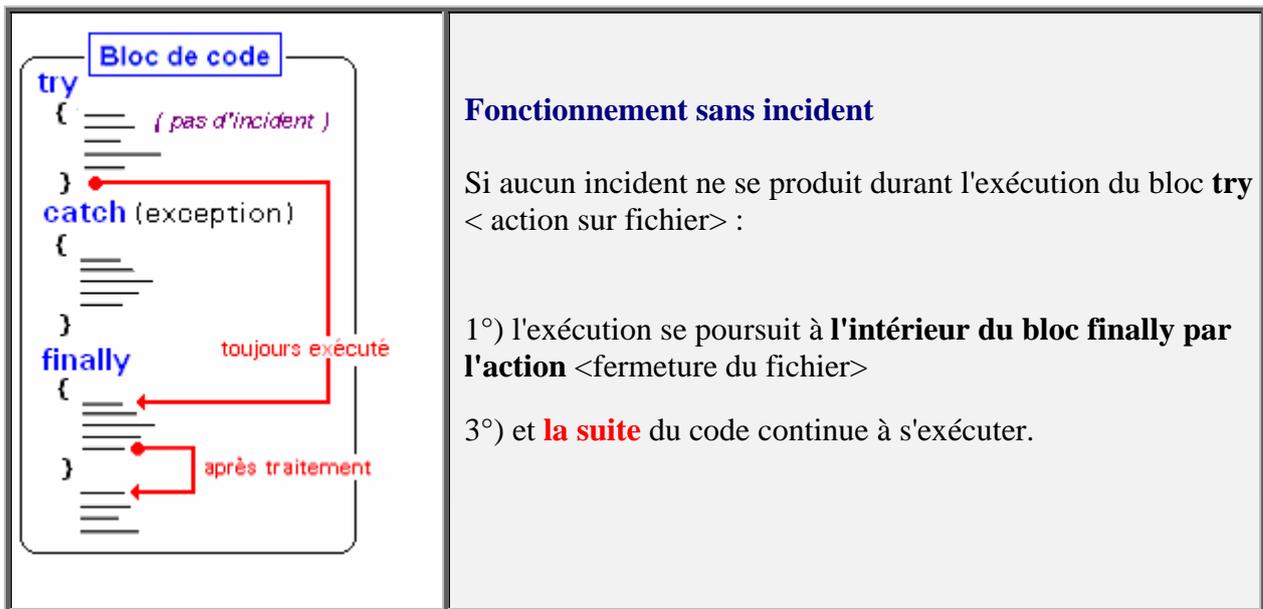
Supposons que nous soyons en présence d'un code contenant une éventuelle levée d'exception, mais supposons que quoiqu'il se passe nous désirions qu'un certain type d'action ait toujours lieu (comme par exemple fermer un fichier qui a été ouvert auparavant). Il existe en C# une **clause spécifique optionnelle** dans la syntaxe des gestionnaires d'exception permettant ce type de réaction du programme, c'est la clause **finally**. Voici en pseudo C# une syntaxe de cette clause :

```

<Ouverture du fichier>
try {
    < action sur fichier >
}

catch( ...) { < traitement si exception >}
finally {
    <fermeture du fichier>
}
.... suite

```



La syntaxe C# autorise l'écriture d'une clause **finally** associée à plusieurs clauses **catch** :

```

try {
  <code à protéger>
}
catch (exception1 e) { <traitement de l'exception1> }
catch (exception2 e) { <traitement de l'exception2> }
...
finally { <action toujours effectuée> }

```

**Remarque :**

Si le code du bloc à protéger dans try...finally contient une instruction de rupture de séquence comme break, return ou continue, le code de la clause **finally**{...} est malgré tout exécuté avant la rupture de séquence.

Nous avons vu lors des définitions des itérations **while**, **for** et de l'instruction **continue**, que l'équivalence suivante entre un **for** et un **while** valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue** (instruction forçant l'arrêt d'un tours de boucle et relançant l'itération suivante) :

*Equivalence incorrecte si Instr contient un **continue** :*

```

Expr1 ;
while ( Expr2 )
for ( Expr1 ; Expr2 ; Expr3 ) Instr { Instr ;
    Expr3
}
    
```

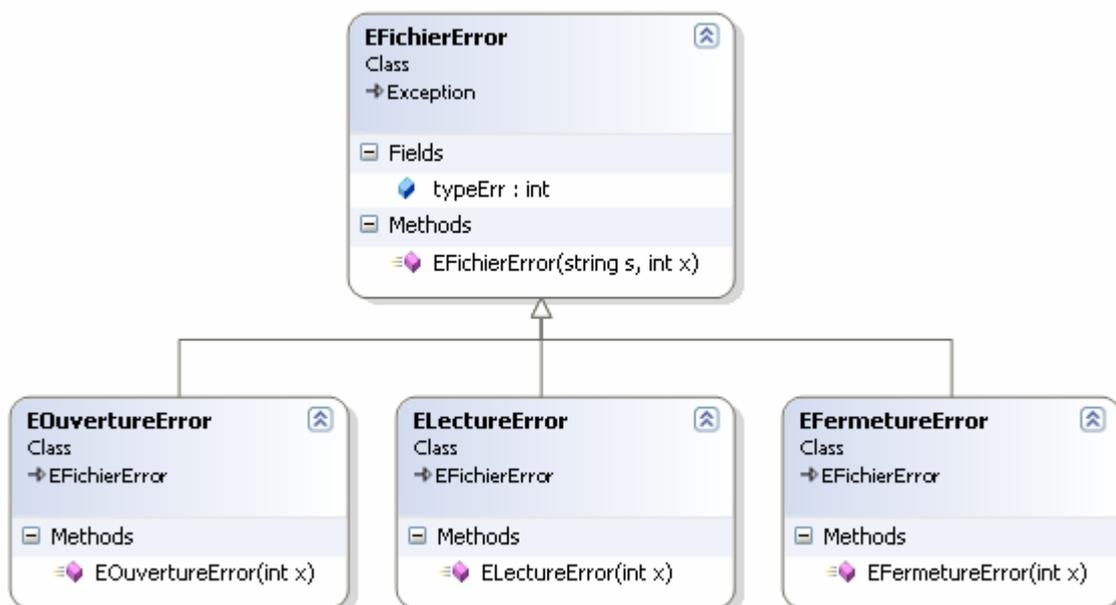
*Equivalence correcte même si Instr contient un **continue** :*

```

Expr1 ;
while ( Expr2 )
{
for ( Expr1 ; Expr2 ; Expr3 ) Instr try {
    Instr ;
} finally { Expr3
}
}
    
```

## 5. Un exemple de traitement d'exceptions sur des fichiers

Créons une hiérarchie d'exceptions permettant de signaler des incidents sur des manipulations de fichiers. Pour cela on distingue essentiellement trois catégories d'incidents qui sont représentés par trois classes :

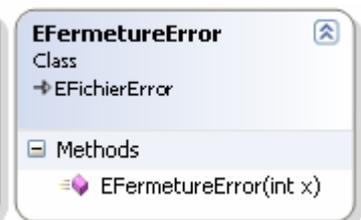
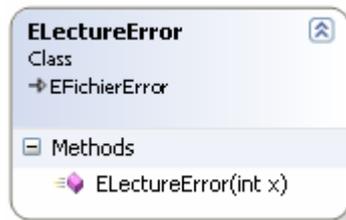
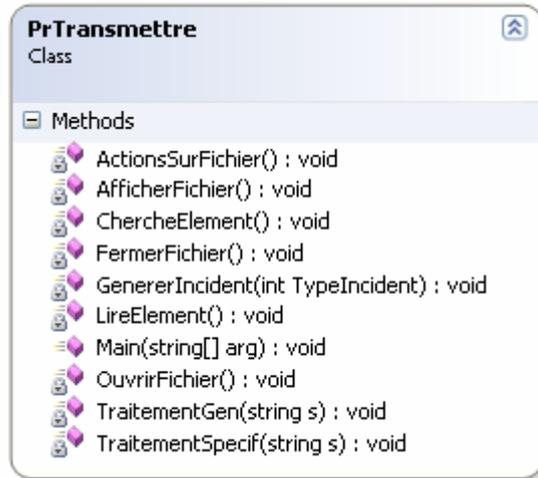


**Enoncé** : Nous nous proposons de mettre en oeuvre les concepts précédents sur un exemple simulant un traitement de fichier. L'application est composée d'un bloc principal <programme> qui appelle une suite de blocs imbriqués.

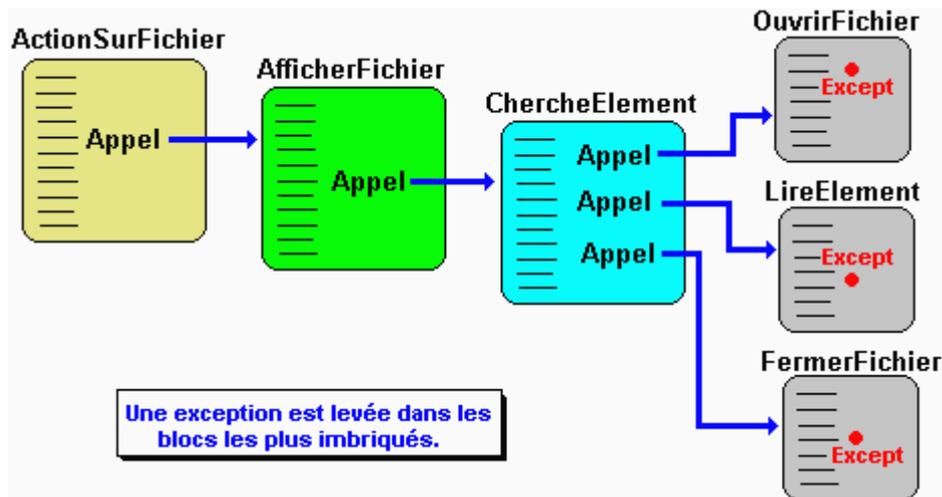
Les classes en jeu et les blocs de programmes (méthodes) acteurs dans le traitement des exceptions sont figurés dans les diagrammes UML de droite :

```

<programme: prTransmettre>
...<ActionsSurFichier>
.....<AfficheFichier>
.....<ChercheElement>
.....<OuvrirFichier> --> exception
.....<LireElement> --> exception
.....<FermerFichier> --> exception
  
```



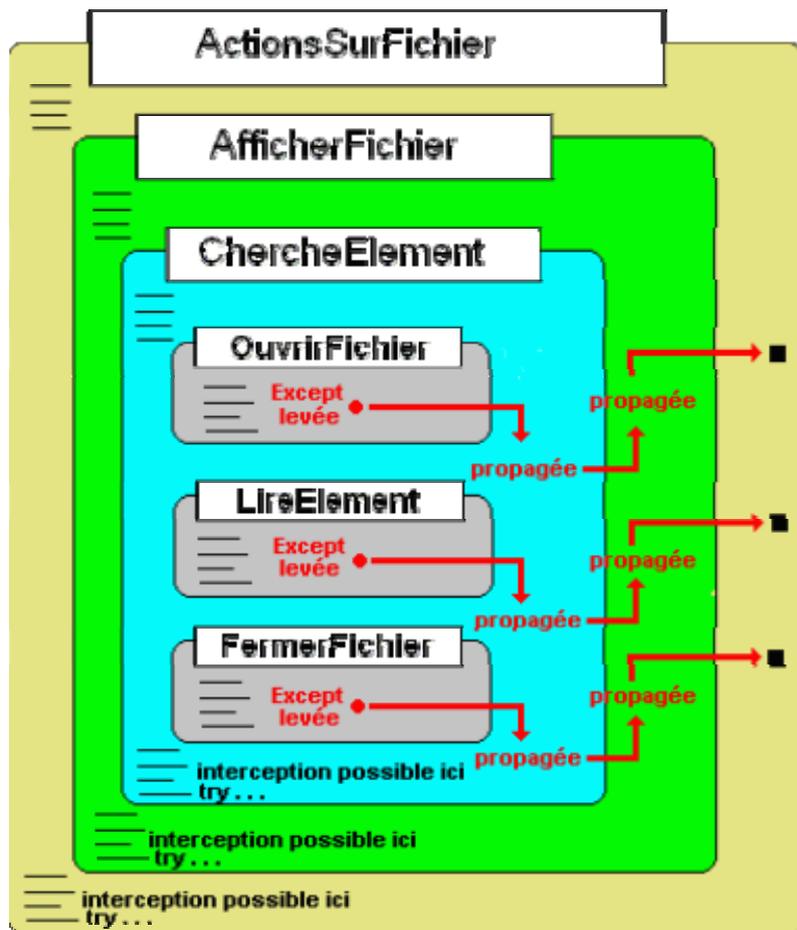
Les trois blocs du dernier niveau les plus internes <OuvrirFichier>, <LireElement> et <FermerFichier> peuvent lancer chacun une exception selon le schéma ci-après :



- **La démarche**

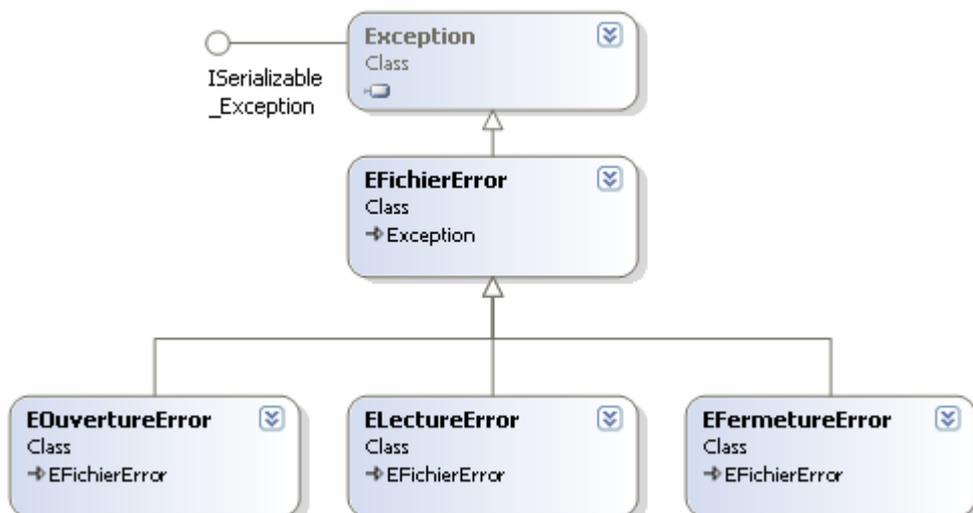
Les éventuelles exceptions lancées par les blocs <OuvrirFichier>, <LireElement> et <FermerFichier> doivent pouvoir se **propager** aux blocs de niveaux englobant afin d'être

interceptables à n'importe quel niveau.



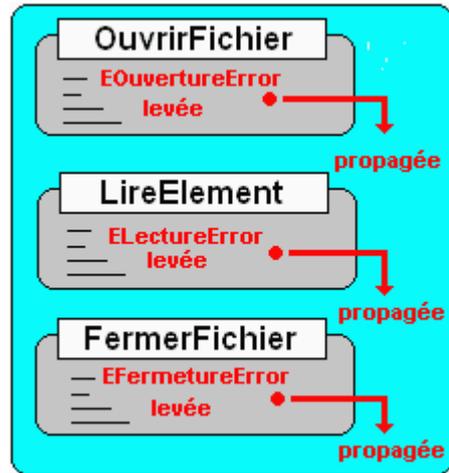
- *Les classes d'exception*

On propose de créer une classe générale d'exception **EFichierError** héritant de la classe des **Exception**, puis 3 classes d'exception héritant de cette classe **EFichierError** :



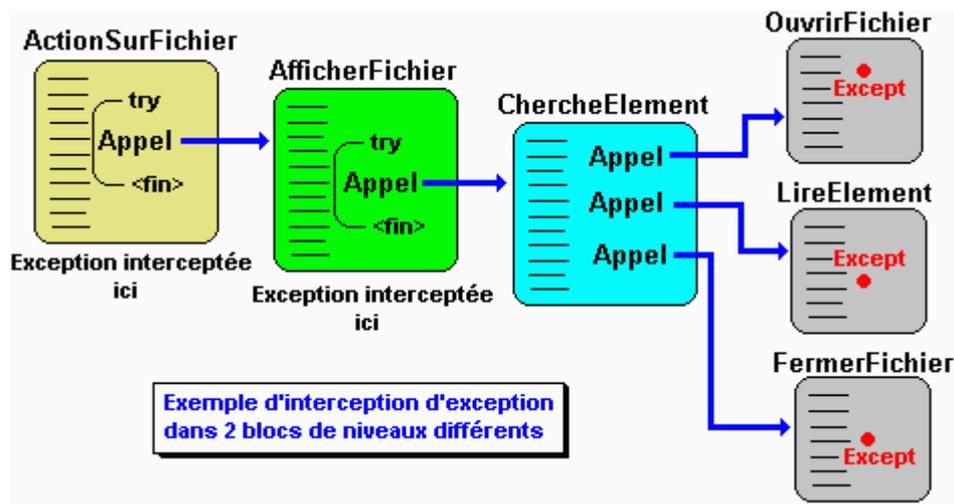
- *Les blocs lançant éventuellement une exception*

Chaque bloc le plus interne peut lancer (lever) une exception de classe différente et la propage au niveau supérieur :



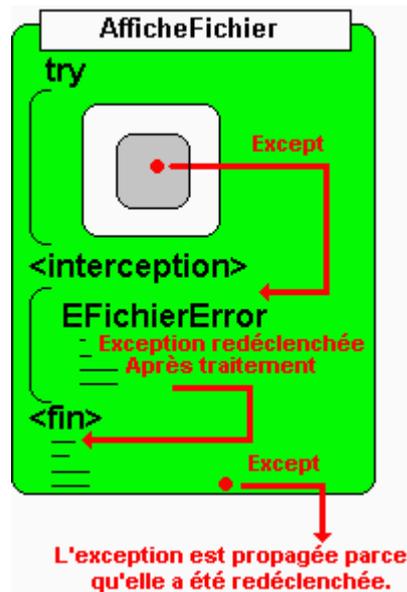
- *Les blocs interceptant les exceptions*

Nous proposons par exemple d'intercepter les exceptions dans les deux blocs <ActionsSurFichier> et <AfficheFichier> :



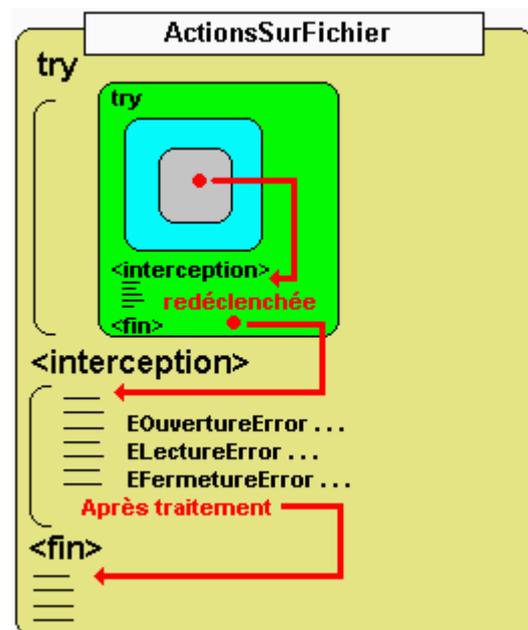
### Le bloc <AfficherFichier>

Ce bloc interceptera une exception de type EFichierError, puis la redéclenchera après traitement :



### Le bloc <ActionsSurFichier>

Ce bloc interceptera une exception de l'un des trois types EOuvertureError, ELectureError ou EFermetureError :



## 6. Une solution de l'exemple précédent en C#

```
using System ;
namespace PrTests
{
    /* pseudo-Traitement d'un fichier à plusieurs niveaux,
     * avec exception et relance d'exception
     */
    class EFichierError : Exception {
        public int typeErr ;
        public EFichierError ( String s, int x ): base ( s ) {
            typeErr = x ;
        }
    }
}
```

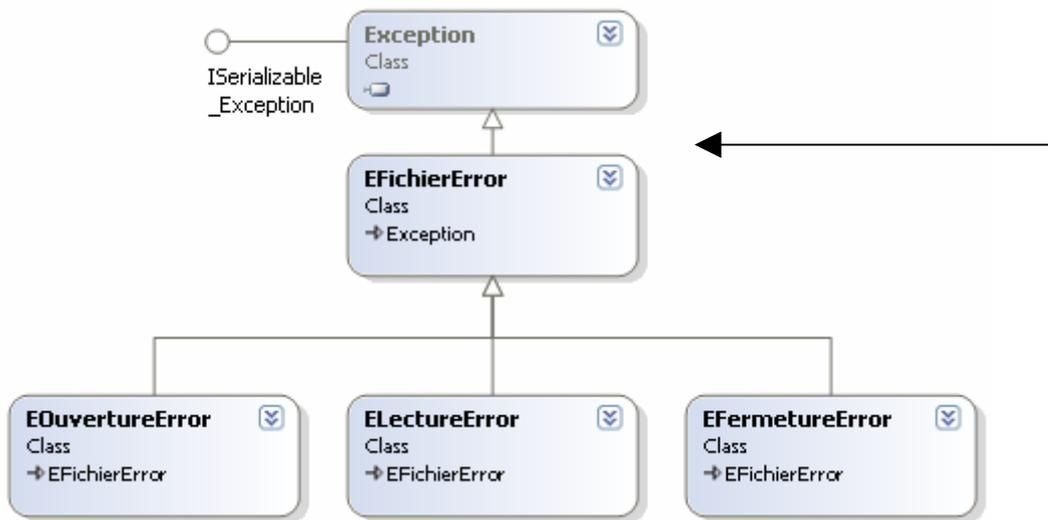
```

class EOuvertureError : EFichierError {
public EOuvertureError ( int x ): base ("Impossible d'ouvrir le fichier !" ,x) { }
}

class ELectureError : EFichierError{
public ELectureError ( int x ): base ("Impossible de lire le fichier !" ,x) { }
}

class EFermetureError : EFichierError{
public EFermetureError ( int x ): base ("Impossible de fermer le fichier !" ,x) { }
}

```



//-----

```

public class PrTransmettre {

void TraitementGen ( String s ) {
System.Console.WriteLine
("traitement general de l'erreur: " + s );
}

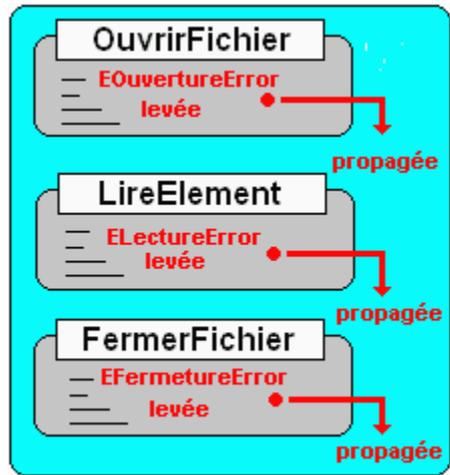
void TraitementSpecif ( String s ) {
System.Console.WriteLine
("traitement specifique de l'erreur: " + s );
}

void OuvrirFichier () {
System.Console.WriteLine (" >> Action ouverture...");
GenererIncident ( 1 );
System.Console.WriteLine (" >> Fin ouverture.");
}

void LireElement () {
System.Console.WriteLine (" >> Action lecture...");
GenererIncident ( 2 );
System.Console.WriteLine (" >> Fin lecture.");
}

void FermerFichier () {
System.Console.WriteLine (" >> Action fermeture...");
GenererIncident ( 3 );
System.Console.WriteLine (" >> Fin fermeture.");
}
}

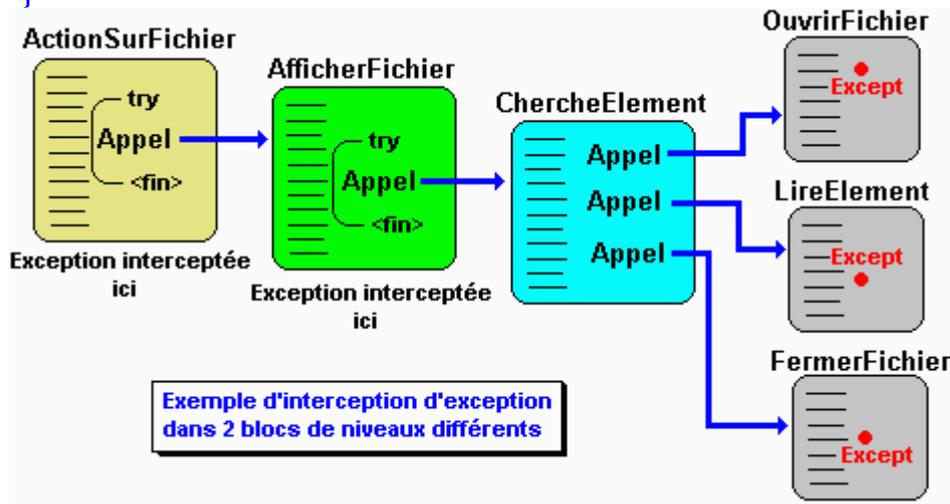
```



```

void ChercheElement () {
    OuvrirFichier ();
    LireElement ();
    FermerFichier ();
}

```



//-----

```

void GenererIncident (int TypeIncident) {
    int n;
    Random nbr = new Random ();
    switch (TypeIncident)
    {
        case 1 : n = nbr.Next () % 4;
            if (n == 0)
                throw new EOuvertureError (TypeIncident);
            break;
        case 2 : n = nbr.Next () % 3;
            if (n == 0)
                throw new ELectureError (TypeIncident);
            break;
        case 3 : n = nbr.Next () % 2;
            if (n == 0)
                throw new EFermetureError (TypeIncident);
            break;
    }
}

```

//-----

```
void ActionsSurFichier () {
    System.Console.WriteLine ("Debut du travail sur le fichier.");
```

```
try
{
    System.Console.WriteLine (".....");
    AfficherFichier ();
}
```

```
catch( EOuvertureError E )
{
    TraitementSpecif ( E.Message );
}
```

```
catch( ELectureError E )
{
    TraitementSpecif ( E.Message );
}
```

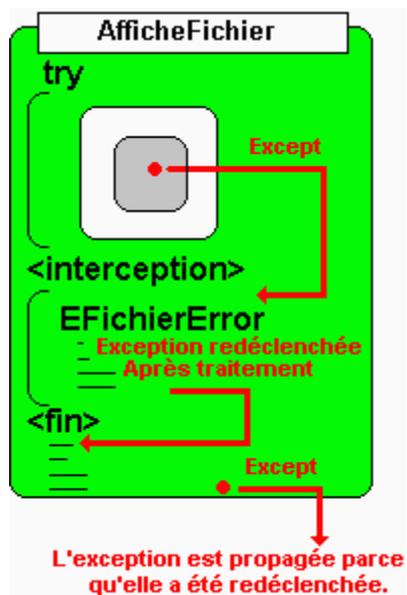
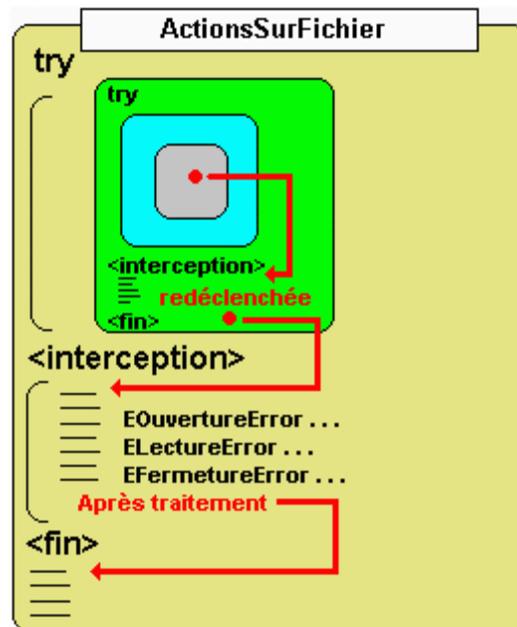
```
catch( EFermetureError E )
{
    TraitementSpecif ( E.Message );
}
```

```
System.Console.WriteLine ("Fin du travail sur le fichier.");
}
```

```
//-----
void AfficherFichier () {
```

```
try {
    ChercheElement ();
}
catch( EFichierError E ) {
    TraitementGen ( E.Message );
    throw E ;
}
}
```

```
public static void Main ( string [] arg ) {
    PrTransmettre Obj = new PrTransmettre ();
    try {
        Obj.ActionsSurFichier ();
    }
    catch( EFichierError E ) {
        System.Console.WriteLine ( " Autre type d'Erreur
générale Fichier !");
    }
    System.Console.ReadLine ();
}
}
```



## Exemples d'exécutions avec un incident de chaque type :

La méthode `GenererIncident` afin de simuler un incident, lance aléatoirement lors de l'exécution soit une exception de l'un des trois type **EOuvertureError** (incident lors de l'ouverture), **ELectureError** (incident lors de la lecture), **EFermetureError** (incident lors de la fermeture), soit ne lance pas d'exception pour indiquer le bon déroulement de toutes les opérations. Le mécanisme des `try...catch` mis en place dans le programme intercepte une éventuelle exception et la propage. Nous donnons ci-dessous les résultats console dans chacun des quatre cas.

1°) Aucune exception n'a été lancée (pas d'incident) :

```
Debut du travail sur le fichier.
.....
  >> Action ouverture...
  >> Fin ouverture.
  >> Action lecture...
  >> Fin lecture.
  >> Action fermeture...
  >> Fin fermeture.
Fin du travail sur le fichier.
```

2°) Une exception de type EOuvertureError a été lancée (incident lors de l'ouverture) :

```
Debut du travail sur le fichier.
.....
  >> Action ouverture...
traitement general de l'erreur: Impossible d'ouvrir le fichier !
traitement specifique de l'erreur: Impossible d'ouvrir le fichier !
Fin du travail sur le fichier.
```

3°) Une exception de type ELectureError a été lancée (incident lors de la lecture) :

```
Debut du travail sur le fichier.
.....
  >> Action ouverture...
  >> Fin ouverture.
  >> Action lecture...
traitement general de l'erreur: Impossible de lire le fichier !
traitement specifique de l'erreur: Impossible de lire le fichier !
Fin du travail sur le fichier.
```

4°) Une exception de type EFermetureError a été lancée (incident lors de la fermeture) :

```
Debut du travail sur le fichier.
.....
  >> Action ouverture...
  >> Fin ouverture.
  >> Action lecture...
  >> Fin lecture.
  >> Action fermeture...
traitement general de l'erreur: Impossible de fermer le fichier !
traitement specifique de l'erreur: Impossible de fermer le fichier !
Fin du travail sur le fichier.
```

# Processus et multi-threading



---

## Plan général:

### 1. Rappels

- 1.1 La multiprogrammation
- 1.2 Multitâche et processus
- 1.3 Multi-threading et processus

### 2. C# autorise l'utilisation des processus et des threads

- 2.1 Les processus avec C#
- 2.2 Comment exécuter une application à partir d'une autre application
- 2.3 Comment atteindre un processus déjà lancé
- 2.4 Comment arrêter un processus

### 3. C# et les threads

- 3.1 Comment créer un Thread
- 3.2 Comment endormir, arrêter ou interrompre un Thread
- 3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation
- 3.4 Section critique en C# : lock
- 3.5 Section critique en C# : Monitor
- 3.6 Synchronisation commune aux processus et aux threads

# 1. Rappels

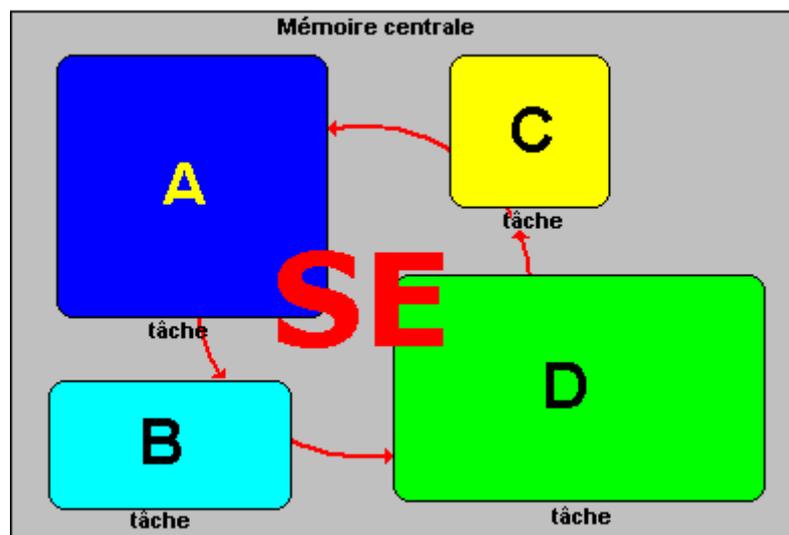
## 1.1 La multiprogrammation

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multiprogrammation ou multitâches, permettant l'exécution "simultanée" de plusieurs tâches.

Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'**un seul processeur**, en apparence **en même temps** ou encore en **parallèle**, en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur Internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

## 1.2 Multitâche et processus

Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un **processus** :



Rappelons la définition des **processus** donnée par A.Tanenbaum: un programme qui s'exécute et qui possède **son propre espace mémoire** : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multiprogrammation par la commutation entre processus effectuée par le processeur unique).

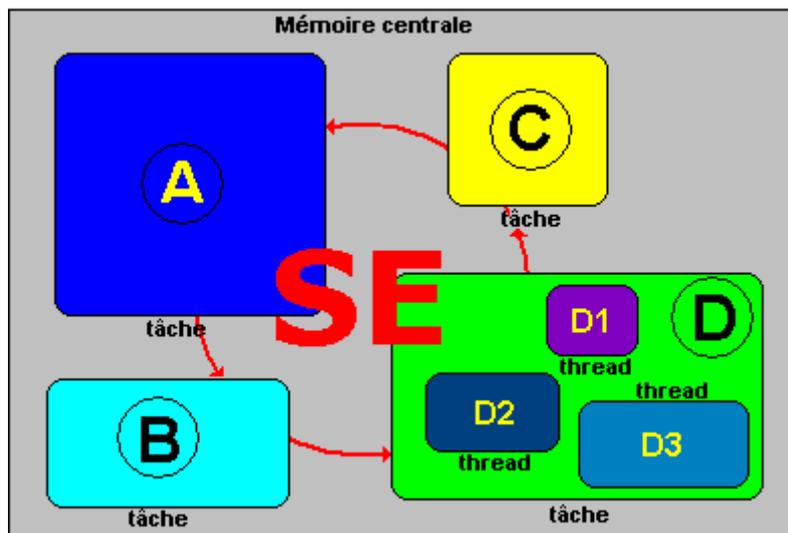
## Thread

En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multiprogrammation. Ces sous-tâches sont nommées "flux d'exécution" ou **Threads**.

Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :

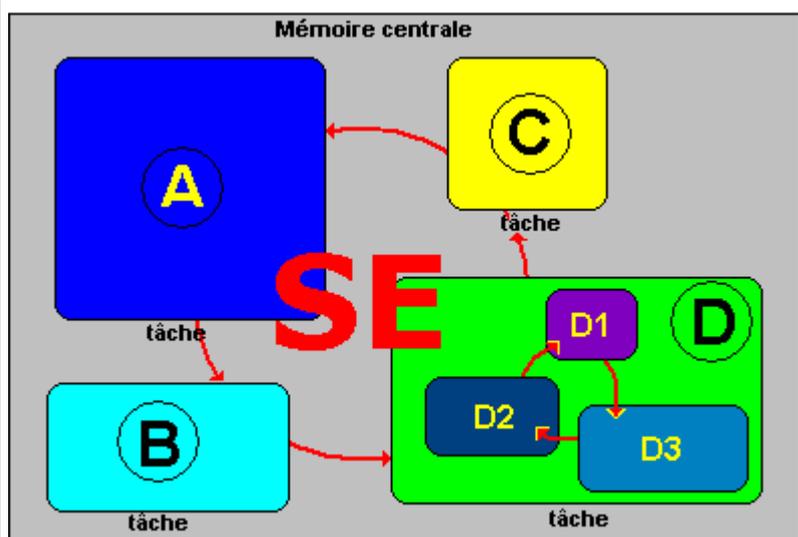


Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads) :



La commutation entre les threads d'un processus fonctionne de la même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps.

Le partage et la répartition du temps sont effectués **uniquement** par le système d'exploitation :



### 1.3 Multithreading et processus

#### Définition :

La majorité des systèmes d'exploitation (Windows, Linux, Solaris, MacOS,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de **Multi-threading**.

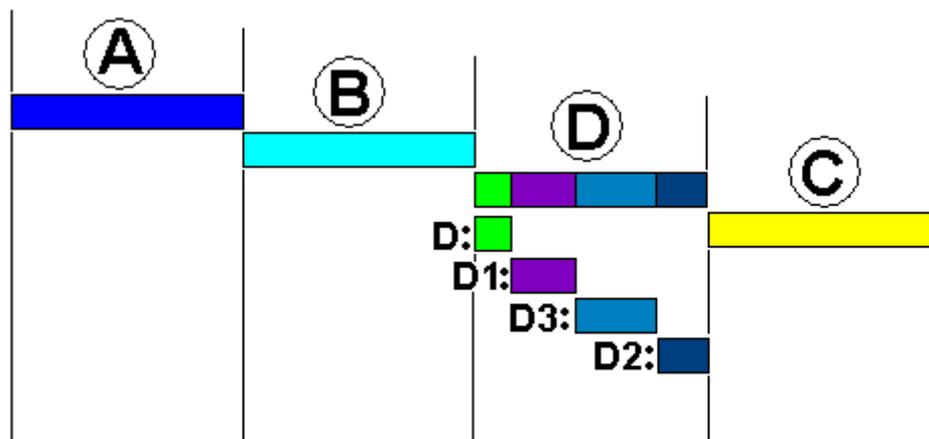
#### Différences entre threads et processus :

- Communication entre threads **plus rapide** que la communication entre processus,
- Les threads partagent un **même espace de mémoire** (de travail) entre eux,
- Les processus ont chacun un **espace mémoire personnel**.

Dans l'exemple précédent, figurons les processus A, B, C et le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.

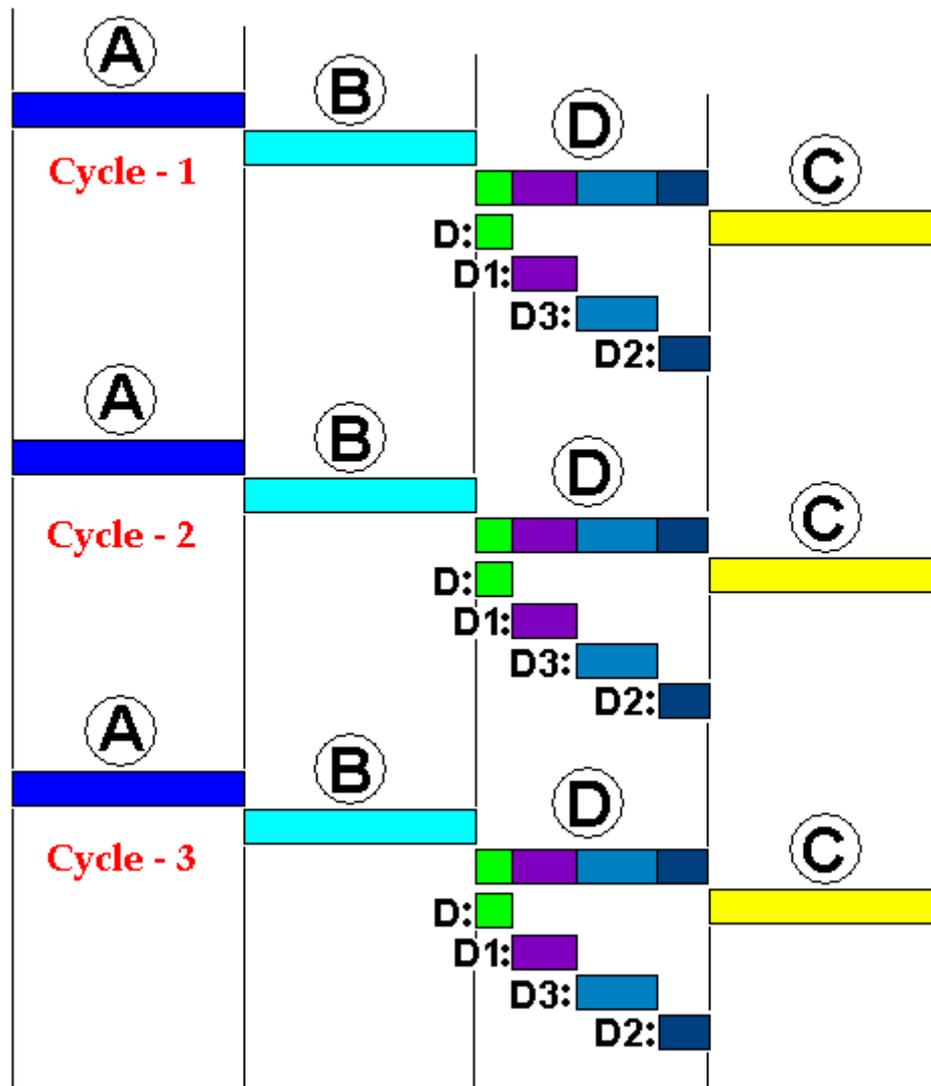


Le système ayant alloué le même temps d'exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :



Tranches de temps allouées pendant l'exécution

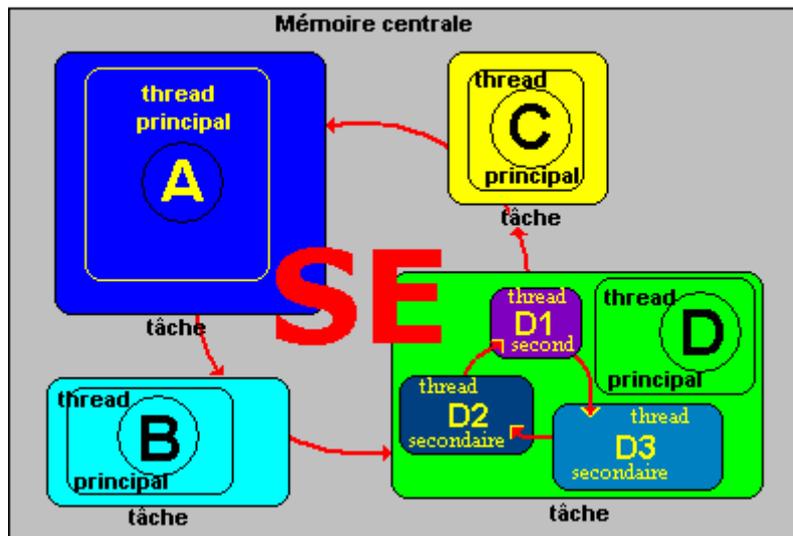
Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



Le langage C# dispose de classes permettant d'écrire et d'utiliser des threads dans vos applications.

## 2. C# autorise l'utilisation des processus et des threads

Lorsqu'un programme C# s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé **thread principal**. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et D soient toutes des applications C#, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



## 2.1 Les processus avec C#

Il existe dans .Net Framework une classe nommée `Process` dans l'espace de nom `System.Diagnostics`, permettant d'accéder à des processus locaux à la machine ou distants et de les manipuler (démarrer, surveiller, stopper, ...). Cette classe est bien documentée par la bibliothèque MSDN de Microsoft, nous allons dans ce paragraphe montrer comment l'utiliser à partir d'un cas pratique souvent rencontré par le débutant : comment lancer une autre application à partir d'une application déjà en cours d'exécution.

## 2.2 Comment exécuter une application à partir d'une autre

1°) Instancier un objet de classe `Process` :

```
| Process AppliAexecuter = new Process();
```

2°) Paramétrer les informations de lancement de l'application à exécuter :

Parmi les nombreuses propriétés de la classe `Process` la propriété `StartInfo` (**public** `ProcessStartInfo StartInfo {get; set;}`) est incontournable, car elle permet ce paramétrage. Supposons que notre application se nomme "Autreappli.exe" qu'elle soit située sur le disque C: dans le dossier "Travail", qu'elle nécessite au démarrage comme paramètre le nom d'un fichier contenant des données, par exemple le fichier "donnees.txt" situé dans le dossier "C:\infos". La propriété `StartInfo` s'utilise alors comme suit afin de préparer le lancement de l'application :

```
| AppliAexecuter.StartInfo.FileName = "c:\\Travail\\Autreappli.exe";  
| AppliAexecuter.StartInfo.UseShellExecute = false;  
| AppliAexecuter.StartInfo.RedirectStandardOutput = false;  
| AppliAexecuter.StartInfo.Arguments = "c:\\infos\\donnees.txt";
```

Remarques :

- `UseShellExecute = false`: permet de lancer directement l'application sans avoir à utiliser l'interface shell du système d'exploitation.

- `RedirectStandardOutput = false`: la sortie standard du processus reste dirigée vers l'écran par défaut.

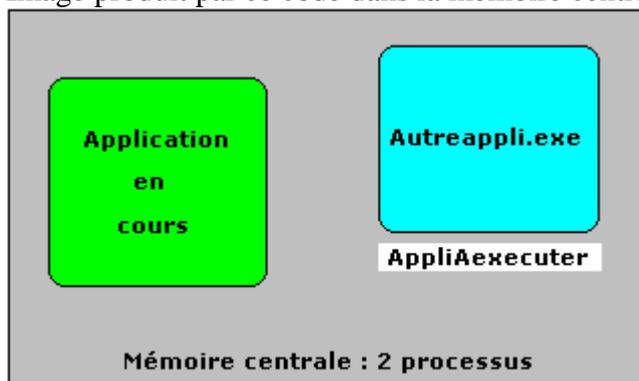
3°) Démarrer le processus par exemple par la surcharge d'instance de la méthode `Start()` de la classe `Process` :

```
| AppliAexecuter.Start( );
```

Code récapitulatif de la méthode `Main` pour lancer un nouveau processus dans une application en cours d'exécution :

```
public static void Main( ) {  
    Process AppliAexecuter = new Process( );  
    AppliAexecuter.StartInfo.FileName = "c:\\Travail\\Autreappli.exe";  
    AppliAexecuter.StartInfo.UseShellExecute = false;  
    AppliAexecuter.StartInfo.RedirectStandardOutput = false;  
    AppliAexecuter.StartInfo.Arguments = "c:\\infos\\donnees.txt";  
    AppliAexecuter.Start( );  
}
```

Image produit par ce code dans la mémoire centrale :



### 2.3 Comment atteindre un processus déjà lancé

Atteindre le processus courant de l'application appelante :

```
| Process Courant = Process.GetCurrentProcess();
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur l'ordinateur local :

```
| Process [] localProcess = Process.GetProcessesByName("Autreappli");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant de nom "ComputerDistant" :

```
| Process [] localProcess = Process.GetProcessesByName("Autreappli", "ComputerDistant");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18":

```
| Process [] localProcess = Process.GetProcessesByName("Autreappli","101.22.34.18");
```

Atteindre un processus grâce à l'identificateur unique sur un ordinateur local de ce processus par exemple 9875 :

```
| Process localProcess = Process.GetProcessById(9875);
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant de nom "ComputerDistant" :

```
| Process localProcess = Process.GetProcessById(9875, "ComputerDistant");
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18" :

```
| Process localProcess = Process.GetProcessById(9875, "101.22.34.18");
```

## 2.4 Comment arrêter un processus

*Seuls les processus locaux peuvent être stoppés dans .Net Framework, les processus distants ne peuvent qu'être surveillés.*

**Si le processus est une application fenêtrée** (application possédant une interface IHM)

La méthode "**public bool** CloseMainWindow()" est la méthode à employer

```
| Process AppliAexecuter = new Process();  
| ....  
| AppliAexecuter.CloseMainWindow();
```

Cette méthode renvoie un booléen qui indique :

- ❑ **True** si la fermeture a été correctement envoyée et le processus est stoppé.
- ❑ **False** si le processus n'est pas stoppé soit parcequ'il y a eu un incident, soit parce que le processus n'était une application fenêtrée (application console).

**Si le processus n'est pas une application fenêtrée** (application console)

La méthode "**public void** Kill()" est la méthode à employer

```
| Process AppliAexecuter = new Process();  
| ....  
| AppliAexecuter.Kill();
```

*Ou bien :*

```
| if( !AppliAexecuter.CloseMainWindow() )  
|     AppliAexecuter.Kill();
```

## 3. C# et les threads

Comme l'avons déjà signalé tous les systèmes d'exploitation modernes permettent la programmation en multi-threading, le .Net Framework contient une classe réservée à cet usage dans l'espace de noms `System.Threading` : la classe non héritable `Thread` qui implémente l'interface `_Thread`.

```
public sealed class Thread: _Thread
```

Cette classe `Thread` sert à créer, contrôler, et modifier les priorités de threads.

### 3.1 Comment créer un Thread

Le C# 2.0 propose quatre surcharges du constructeur de `Thread`, toutes utilisent la notion de **delegate** pour préciser le code à exécuter dans le thread, nous examinons celle qui est la plus utilisée depuis la version 1.0.

```
public Thread ( ThreadStart start );
```

Le paramètre `start` de type `ThreadStart` est un objet **delegate** sans paramètre qui *pointe sur* (equiv : *fait référence à* ) la méthode à appeler à chaque exécution du thread ainsi créé :

```
public delegate void ThreadStart ( );
```

Vous devez donc écrire une méthode de classe ou d'instance ayant la même signature que le delegate `void ThreadStart ( )`, puis créer l'objet **delegate** qui pointera vers cette méthode. Vous pouvez nommer cette méthode du nom que vous voulez, pour rester dans le style Java nous la dénommerons `run( )`. Ci-dessous un pseudo-code C# de création d'un thread à partir d'un délégué pointant sur une méthode de classe :

```
public class ChargerDonnees {  
    public static void run(){ ... }  
}
```

```
public class AppliPrincipale {  
    public static void Main( ){  
        ThreadStart ChargerDelegate = new ThreadStart (ChargerDonnees.run);  
        Thread thrdChargement = new Thread(ChargerDelegate);  
    }  
}
```

Notons qu'il est possible d'alléger le code d'instanciation du thread en créant un objet délégué anonyme qui est passé en paramètre au constructeur de `Thread` :

```
Thread thrdChargement = new Thread( new ThreadStart (ChargerDonnees.run) );
```

Dans l'exemple qui suit nous lançons trois threads en plus du thread principal automatiquement construit par le CLR pour chaque processus, l'un à partir d'une méthode **static** `run0` de la classe `Program` et les deux autres à partir d'une méthode **static** `run1` et d'une méthode d'instance `run2` de

la classe `AfficherDonnees` :

```
public class AfficherDonnees
{
    public static void run1()
    {
        for (int i1 = 1; i1 < 100; i1++)
            System.Console.WriteLine(">>> thread1 = " + i1);
    }
    public void run2()
    {
        for (int i2 = 1; i2 < 100; i2++)
            System.Console.WriteLine("*** thread2 = " + i2);
    }
}
public class Program
{
    public static void run0()
    {
        for (int i0 = 1; i0 < 100; i0++)
            System.Console.WriteLine("... thread0 = " + i0);
    }
    static void Main(string[] args)
    {
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread thread0 = new Thread(new ThreadStart( run0 ));
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread0.Start();
        thread1.Start();
        thread2.Start();
        for (int i = 1; i < 100; i++)
            System.Console.WriteLine(" i = " + i);
        System.Console.WriteLine("fin de tous les threads.");
    }
}
```

*Résultats de l'exécution du programme précédent (dépendants de votre configuration machine+OS) :*

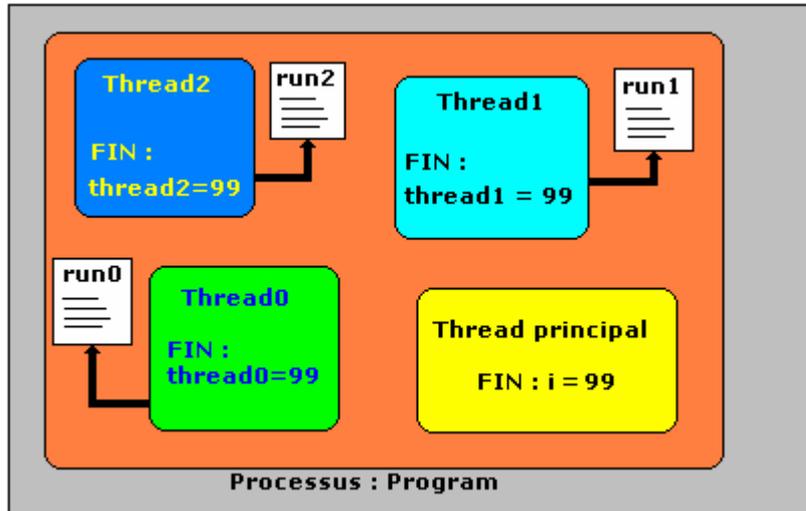
**Lancement des threads :**

```
.... thread0 = 1
>>> thread1 = 1
i = 1
*** thread2 = 1
.... thread0 = 2
>>> thread1 = 2
i = 2
*** thread2 = 2
.... thread0 = 3
.....
i = 98
*** thread2 = 98
.... thread0 = 99
>>> thread1 = 99
i = 99
*** thread2 = 99
fin de tous les threads.
```

L'exécution précédente montre bien que chacun des `thread0`, `thread1` et `thread2` se voient allouer une tranche de temps pendant laquelle chacun d'eux exécute une partie de sa boucle `for` et

imprime ses informations sur la console. Chaque thread se termine lorsque le code de la méthode run vers laquelle il pointe a fini de s'exécuter.

Nous figurons ci-après l'image de la mémoire centrale pour le processus **Program** avec chacun des 3 threads instanciés pointant vers la méthode qu'il exécute :



### 3.2 Comment endormir, arrêter ou interrompre un Thread

La société Microsoft a déprécié depuis la version 2.0 du .Net Framework les méthodes `Suspend()` et `Resume()` qui permettaient d'effectuer la synchronisation entre les threads, toutefois elles n'ont pas été supprimées. En ce sens Microsoft adopte la même attitude que Sun pour Java, afin de ne pas encourager les développeurs à utiliser des outils qui se sont montrés sensibles aux blocages du type verrou mortel. Nous ne proposerons donc pas ces méthodes dangereuses au lecteur, mais plutôt des méthodes sûres.

#### Endormir : Méthode `Sleep ( ... )`

On parle d'endormir un thread pendant un certain temps  $t$ , lorsque l'exécution de ce thread est arrêtée pendant ce temps  $t$ , c'est à dire que le thread est retiré de la file d'attente de l'algorithme d'ordonnancement du système. A la fin du temps  $t$ , le thread est automatiquement "réveillé" par le système, c'est à dire qu'il est replacé dans la file d'attente de l'ordonnanceur et donc son exécution repart.

La méthode "**public static void Sleep( int millisecondsTimeout )**" sert à endormir un thread pendant un temps exprimé en millisecondes. Dans la méthode `run0( )` de l'exemple précédent si nous rajoutons l'instruction "**Thread.Sleep(2);**", nous "ralentissons" l'exécution de la boucle, puisque à tous les tours de boucles nous bloquons le thread qui l'exécute pendant 2 ms.

```
public static void run0( )
{
    for (int i0 = 1; i0 < 100; i0++)
    {
        System.Console.WriteLine("... thread0 = " + i0);
        Thread.Sleep(2);
    }
}
```

```
}
```

Résultats de l'exécution du programme précédent avec `Thread.Sleep(2)` dans la méthode `run0()` :

Lancement des threads :

```
.... thread0 = 1
```

```
>>> thread1 = 1
```

```
i = 1
```

```
*** thread2 = 1
```

```
.....
```

```
i = 98
```

```
*** thread2 = 98
```

```
.... thread0 = 79
```

```
>>> thread1 = 99
```

```
i = 99
```

```
*** thread2 = 99
```

```
.... thread0 = 80
```

```
fin de tous les threads.
```

```
.... thread0 = 81
```

```
.... thread0 = 82
```

```
.... thread0 = 83
```

```
.... thread0 = 84
```

```
.... thread0 = 85
```

```
.... thread0 = 86
```

```
.... thread0 = 87
```

```
.... thread0 = 88
```

```
.... thread0 = 89
```

```
.... thread0 = 90
```

```
.... thread0 = 91
```

```
.... thread0 = 92
```

```
.... thread0 = 93
```

```
.... thread0 = 94
```

```
.... thread0 = 95
```

```
.... thread0 = 96
```

```
.... thread0 = 97
```

```
.... thread0 = 98
```

```
.... thread0 = 99
```

*(résultats dépendants de votre configuration machine+OS)*

Notons que cette exécution est semblable à la précédente, du moins au départ, car nous constatons vers la fin que le `thread0` a pris un léger retard sur ses collègues puisque le `thread1` et le `thread2` se termine avec la valeur 99, le thread principal affiche la phrase "**fin de tous les threads**" alors que le `thread0` n'a pas encore dépassé la valeur 80.

Les trois threads `thread1`, `thread2` et le thread principal sont en fait terminés seul le `thread0` continue son exécution jusqu'à la valeur 99 qui clôt son activité.

## Arrêter : Méthode `Abort` ( ... )

Dans l'exemple précédent le message "**fin de tous les threads**" n'est pas conforme à la réalité puisque le programme est bien arrivé à la fin du thread principal, mais `thread0` continue son exécution. Il est possible de demander au système d'arrêter définitivement l'exécution d'un thread, cette demande est introduite par la méthode d'instance `Abort` de la classe `Thread` qui lance le processus d'arrêt du thread qui l'appelle.

Nous reprenons le programme précédent dans lequel nous lançons une demande d'arrêt du `thread0` par l'instruction : `thread0.Abort()`;

```
public class AfficherDonnees
{
    public static void run1()
    {
        for (int i1 = 1; i1 < 100; i1++)
            System.Console.WriteLine(">>> thread1 = " + i1);
    }
    public void run2()
    {
        for (int i2 = 1; i2 < 100; i2++)
            System.Console.WriteLine("*** thread2 = " + i2);
    }
}
public class Program
```

```

{
    public static void run0()
    {
        for (int i0 = 1; i0 < 100; i0++)
            System.Console.WriteLine(".... thread0 = " + i0);
        Thread.Sleep(2);
    }
    static void Main(string[] args)
    {
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread thread0 = new Thread(new ThreadStart( run0 ));
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread0.Start();
        thread1.Start();
        thread2.Start();
        for (int i = 1; i < 100; i++)
            System.Console.WriteLine(" i = " + i);
        thread0.Abort();
        System.Console.WriteLine("fin de tous les threads.");
    }
}

```

*Résultats de l'exécution du programme précédent avec Thread.Sleep(2) et thread0.Abort() :*

**Lancement des threads :**

.... thread0 = 1

>>> thread1 = 1

i = 1

\*\*\* thread2 = 1

.....

i = 98

\*\*\* thread2 = 98

.... thread0 = 79

>>> thread1 = 99

i = 99

\*\*\* thread2 = 99

.... thread0 = 80

**fin de tous les threads. (Résultats dépendants de votre configuration machine+OS)**

Le thread0 a bien été arrêté avant sa fin normale, avec comme dernière valeur 80 pour l'indice de la boucle **for**.

## Attendre : Méthode Join ( ... )

Un thread peut se trouver dans des états d'exécution différents selon qu'il est actuellement en cours d'exécution, qu'il attend, qu'il est arrêté etc... Il existe en C# un type énuméré **ThreadState** qui liste toutes les valeurs possibles des états d'un thread :

```

public enum ThreadState { Running = 0, StopRequested = 1, SuspendRequested = 2, Background = 4, Unstarted = 8, Stopped = 16, WaitSleepJoin = 32, Suspended = 64, AbortRequested = 128, Aborted = 256 }

```

Dans la classe **Thread**, nous trouvons une propriété "**public ThreadState ThreadState {get;}**" en lecture seule qui fournit pour un thread donné son état d'exécution. En consultant cette propriété le développeur peut connaître l'état "en direct" du thread, notons que cet état peut varier au cours du

temps.

On peut faire attendre la fin d'exécution complète d'un thread pour qu'un autre puisse continuer son exécution, cette attente est lancée par l'une des trois surcharges de la méthode d'instance **Join** de la classe **Thread** :

```
public void Join();
```

Dans le programme précédent rappelons-nous que le thread0 prend du "retard" sur les autres car nous l'avons ralenti avec un `Sleep(2)`. Au lieu de l'arrêter définitivement avant la dernière instruction de la méthode `Main` remplaçons l'instruction `"thread0.Abort();"` par l'instruction `"thread0.Join();"`.

Que se passe-t-il : Le thread principal qui exécute la méthode `main`, invoque la méthode **Join** du thread0 avant de terminer son exécution, ce qui signifie que le thread principal bloque tant que le thread0 n'a pas fini complètement son exécution. Nous ajoutons au programme précédent une méthode **public static void** `etatsThreads` qui affiche l'état d'exécution du thread principal et des 3 threads instanciés :

```
public static void etatsThreads(Thread principal, Thread thrd1, Thread thrd2, Thread thrd3 )
{
    System.Console.WriteLine(principal.Name + " : " + principal.ThreadState);
    System.Console.WriteLine(thrd1.Name + " : " + thrd1.ThreadState);
    System.Console.WriteLine(thrd2.Name + " : " + thrd2.ThreadState);
    System.Console.WriteLine(thrd3.Name + " : " + thrd3.ThreadState);
}

static void Main(string[] args)
{
    Console.WriteLine("Lancement des threads :");
    AfficherDonnees obj = new AfficherDonnees();
    Thread principal = Thread.CurrentThread;
    principal.Name = "principal";
    Thread thread0 = new Thread(new ThreadStart(run0));
    thread0.Name = "thread0";
    Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
    thread1.Name = "thread1";
    Thread thread2 = new Thread(new ThreadStart(obj.run2));
    thread2.Name = "thread2";
    etatsThreads(principal, thread0, thread1, thread2);
    thread0.Start();
    thread1.Start();
    thread2.Start();
    etatsThreads(principal, thread0, thread1, thread2);
    for (int i = 1; i < 100; i++)
    {
        System.Console.WriteLine(" i = " + i);
    }
    etatsThreads(principal, thread0, thread1, thread2);
    thread0.Join();
    etatsThreads(principal, thread0, thread1, thread2);
    System.Console.WriteLine("fin de tous les threads.");
}
```

Nous obtenons une référence sur le thread principal par la propriété `CurrentThread` dans l'instruction : `Thread principal = Thread.CurrentThread`.

Résultats de l'exécution du programme précédent avec `Thread.Sleep(2)` et `thread0.Join()` :

<pre>Lancement des threads principal : Running thread0 : Unstarted thread1 : Unstarted thread2 : Unstarted .... thread0 = 1 &gt;&gt;&gt; thread1 = 1 principal : Running *** thread2 = 1 &gt;&gt;&gt; thread1 = 2 .... thread0 = 2 thread0 : WaitSleepJoin *** thread2 = 2 &gt;&gt;&gt; thread1 = 3 .... thread0 = 3 thread1 : WaitSleepJoin *** thread2 = 3 &gt;&gt;&gt; thread1 = 4 thread2 : WaitSleepJoin .... thread0 = 4 *** thread2 = 4 &gt;&gt;&gt; thread1 = 5 i = 1 .....</pre>	<pre>i = 99 principal : Running .... thread0 = 85 thread0 : WaitSleepJoin thread1 : Stopped .... thread0 = 86 thread2 : Stopped .... thread0 = 87 .... thread0 = 88 .... thread0 = 89 .... thread0 = 90 .... thread0 = 91 .... thread0 = 92 .... thread0 = 93 .... thread0 = 94 .... thread0 = 95 .... thread0 = 96 .... thread0 = 97 .... thread0 = 98 .... thread0 = 99 principal : Running thread0 : Stopped thread1 : Stopped thread2 : Stopped fin de tous les threads.</pre>
---	--

*(Résultats dépendants de votre configuration machine+OS)*

Au début de l'exécution les états sont :

```
principal : Running
thread0 : Unstarted
thread1 : Unstarted
thread2 : Unstarted
```

Seul le thread principal est en état d'exécution, les 3 autres nonencore démarrés.

Après invocation de la méthode **Start** de chaque thread, la tour revient au thread principal pour exécuter la boucle **for** (`int i = 1; i < 100; i++`), les 3 autres threads sont dans la file d'attente :

```
principal : Running
thread0 : WaitSleepJoin
thread1 : WaitSleepJoin
thread2 : WaitSleepJoin
```

Après la fin de l'exécution de la boucle **for** (`int i = 1; i < 100; i++`) du thread principal celui-ci est sur le point de s'arrêter, les thread1 et thread2 ont fini leur exécution, le thread0 continue son exécution car ralenti par le `Sleep(2)` à chaque tour de boucle :

```
principal : Running
thread0 : WaitSleepJoin
thread1 : Stopped
thread2 : Stopped
```

Après la terminaison du décompte de la boucle du thread0 jusqu'à la valeur 99, le thread0 se termine et le thread principal est sur le point de se terminer (il ne lui reste plus la dernière instruction d'affichage "`System.Console.WriteLine("fin de tous les threads.")`" à exécuter) :

principal : **Running**  
thread0 : **Stopped**  
thread1 : **Stopped**  
thread2 : **Stopped**  
fin de tous les threads.

## Interrompre-réveiller : Méthode Interrupt ( ... )

Si le thread est dans l'état **WaitSleepJoin** c'est à dire soit endormi (**Sleep**), soit dans la file d'attente attendant son tour (**Wait**), soit en attente de la fin d'un autre thread (**Join**), il est alors possible d'interrompre son état d'attente grâce à la méthode **Interrupt**. Cette méthode interrompt temporairement le thread qui l'invoque et lance une exception du type **ThreadInterruptedException**.

Dans le programme précédent nous ajoutons l'instruction "**thread0.Interrupt()**" dans la méthode Main, juste après la fin de la boucle **for** (**int** i = 1; i < 100; i++) {...}. Lors de l'exécution, dès que le thread0 se met en mode **WaitSleepJoin** par invocation de la méthode **Sleep(2)**, il est interrompu :

```
public static void run0()
{
    for (int i1 = 1; i1 < 100; i1++)
    {
        System.Console.WriteLine("... thread0 = " + i1);
        Thread.Sleep(2);
    }
}
```

**ThreadInterruptedException was unhandled**  
Thread interrompu à partir d'un état d'attente.

**Troubleshooting tips:**  
[Get general help for this exception.](#)

[Search for more Help Online...](#)

**Actions:**  
[View Detail...](#)  
[Copy exception detail to the clipboard](#)

Quick Console

```
i = 93
*** thread2 = 97
... thread0 = 84
>>> thread1 = 98
i = 94
*** thread2 = 98
>>> thread1 = 99
... thread0 = 85
i = 95
*** thread2 = 99
i = 96
... thread0 = 86
i = 97
i = 98
... thread0 = 87
i = 99
principal : Running
thread0 : Running
```

Une exception **ThreadInterruptedException** a bien été lancée et peut être interceptée dans le thread principal.

Nous listons ci-dessous le code source de la méthode Main produisant l'interruption du thread0 figurée précédemment :

```

static void Main(string[] args)
{
    Console.WriteLine("Lancement des threads :");
    AfficherDonnees obj = new AfficherDonnees();
    Thread principal = Thread.CurrentThread;
    principal.Name = "principal";
    Thread thread0 = new Thread(new ThreadStart(run0));
    thread0.Name = "thread0";
    Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
    thread1.Name = "thread1";
    Thread thread2 = new Thread(new ThreadStart(obj.run2));
    thread2.Name = "thread2";
    etatsThreads(principal, thread0, thread1, thread2);
    thread0.Start();
    thread1.Start();
    thread2.Start();
    etatsThreads(principal, thread0, thread1, thread2);
    for (int i = 1; i < 100; i++)
    {
        System.Console.WriteLine(" i = " + i);
    }
    thread0.Interrupt();
    etatsThreads(principal, thread0, thread1, thread2);
    thread0.Join();
    etatsThreads(principal, thread0, thread1, thread2);
    System.Console.WriteLine("fin de tous les threads.");
}

```

*Attention : La rapidité du processeur, l'influence du CLR et la charge instantanée du système **changent** considérablement **les résultats obtenus** ! Il faut donc n'utiliser ces outils que pour du parallélisme réel et non pour du séquentiel (cf. la notion de synchronisation paragraphe suivant)*

### 3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation

#### ressource partagée

D'un point de vue général, plusieurs processus peuvent accéder en lecture et en écriture à un **même espace mémoire** par exemple : accéder à un spooler d'imprimante, réserver une page en mémoire centrale, etc... Cet espace mémoire partagé par plusieurs processus se dénomme **une ressource partagée**.

#### concurrence

Lorsque le résultat final après exécution des processus à l'intérieur sur une ressource partagée n'est pas déterministe, mais dépend de l'ordre dans lequel le système d'exploitation a procédé à l'exécution de chacun des processus, on dit que l'on est en situation de **concurrence**.

#### synchronisation

Lorsqu'une structure de données est accédée par des processus en situation de concurrence, il est impossible de prévoir le comportement des threads sur cette structure. Si l'on veut obtenir un comportement déterministe, il faut ordonner les exécutions des processus d'une manière séquentielle afin d'être sûr qu'**un seul** processus accède à toute la structure **jusqu'à la fin** de son exécution, puis laisse la main au processus suivant etc. Cette régulation des exécutions des processus s'appelle la **synchronisation**.

#### Exclusion mutuelle

Lorsque plusieurs processus travaillent sur une ressource partagée, la synchronisation entre les divers processus sous-entend que cette ressource partagée est exclusivement à la disposition d'un processus pendant sa durée complète d'exécution. Le mécanisme qui permet à un seul processus de s'exécuter sur une ressource partagée à l'exclusion de tout autre, se dénomme l'**exclusion mutuelle**.

## section critique

Lorsqu'un bloc de lignes de code traite d'accès par threads synchronisés à une ressource partagée on dénomme ce bloc de code une **section critique**.

Tout ce que nous venons de voir sur les processus se reporte intégralement aux **threads** qui sont des processus légers.

Dans un système d'exploitation de multiprogrammation, ces situations de concurrence sont très courantes et depuis les années 60, les informaticiens ont mis en œuvre un arsenal de réponses d'exclusion mutuelle ; ces réponses sont fondées sur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

Le développeur peut avoir besoin dans ses programmes de gérer des situations de concurrence comme par exemple dans un programme de réservation de place de train et d'édition de billet de transport voyageur. Le multi-threading et les outils de synchronisation que le langage de programmation fournira seront une aide très précieuse au développeur dans ce style de programmation.

Pour programmer de la synchronisation entre threads, le langage C# met à la disposition du développeur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

### 3.4 Section critique en C# : lock

## L'instruction lock (...) { .... }

Le mot clef **lock** détermine un bloc d'instructions en tant que **section critique**. Le verrouillage de cette section critique est obtenu par exclusion mutuelle sur un objet spécifique nommé verrou qui peut être dans deux états : soit disponible ou libre, soit verrouillé. Ci-dessous la syntaxe C# de l'instruction lock :

```
object verrou = new object();
lock ( verrou )
{
    ... lignes de code de la section critique
}
```

Lorsqu'un thread Th1 veut entrer dans la section critique délimitée par lock, nous sommes en face de deux possibilités selon que l'objet verrou est libre ou non :

- ❑ **Si l'objet verrou est libre** (c'est à dire qu'aucun autre thread n'exécute le code de la section critique) alors on dit que le thread Th1 acquiert le verrou, d'autre part il le verrouille pour tout autre thread. Dès que le thread Th1 finit d'exécuter la dernière instruction de la section critique, il libère le verrou qui devient disponible pour un autre thread.

- ❑ **Si l'objet verrou n'est pas libre** et qu'un thread Th2 demande à acquérir ce verrou (veut entrer dans la section critique) pendant que Th1 est dans la section critique, le thread Th2 est alors mis dans la file d'attente associée à l'objet verrou par le CLR (le thread est donc bloqué en attente). Chaque nouveau thread demandant à acquérir ce verrou est rangé dans la file d'attente du verrou tant que ce dernier n'est pas libéré. Dès que le verrou devient libre le CLR autorise le thread en tête de file à acquérir le verrou et ce thread est retiré de la file d'attente des threads du verrou.
- ❑ **Si une exception est levée** dans la section critique, le verrou est automatiquement libéré par l'instruction lock pour ce thread.

### Exemple C# de synchronisation avec lock :

1°) Partons d'un exemple où le fait qu'il n'y ait pas de synchronisation provoque un comportement erratique. Soit un tableau d'entiers "int[ ] datas = new int[50]" tous à la valeur 1, nous construisons trois threads modifiant ce tableau, le premier rajoute 1 à chaque cellule du tableau, le second multiplie par 2 le contenu de chaque cellule du tableau, le troisième soustrait 1 à chaque cellule du tableau.

Nous créons une classe `ThreadModifieurDonnees` dont la vocation est de permettre à des délégués, à qui nous donnerons un nom lors de leur instantiation, de travailler sur le tableau d'entiers à travers une méthode `run()`.

Dans cette classe `ThreadModifieurDonnees` la méthode `run()` effectue une action différente selon le nom du délégué qui l'invoque (ajouter 1, multiplier par 2, soustraire 1).

Afin de souligner la concurrence entre les 3 threads nous déséquilibrons les temps alloués aux threads par un endormissement différent pour le thread1 (`Thread.Sleep(1);`) et pour le thread2 (`Thread.Sleep(0);`), le thread3 restant indemne de toute modification temporelle :

```
public class ThreadModifieurDonnees
{
    private int[ ] donnees;
    private string nom;
    public ThreadModifieurDonnees(int[ ] donnees, string nom)
    {
        this.donnees = donnees;
        this.nom = nom;
    }
    public void run()
    {
        for (int i = 0; i < donnees.Length; i++)
        {
            if (nom == "modif1")
            {
                donnees[i] += 1;
                Thread.Sleep(1);
            }
            else
            if (nom == "modif2")
            {
                donnees[i] *= 2;
                Thread.Sleep(0);
            }
            else
            if (nom == "modif3")
                donnees[i] -= 1;
        }
    }
}
```







occupée par un autre thread :

```
private object verrou = new object();

public void methode1()
{
    Monitor.Enter ( verrou );
    ... lignes de code de la section critique
    Monitor.Exit ( verrou );
}

.....
public void methode2()
{
    if (Monitor.TryEnter ( verrou )
        methode1 ( );
    else
        ...Autres actions
}
```

Dans le cas où une exception serait levée dans une section critique le verrou n'est pas automatiquement levé comme avec l'instruction `lock(){ ... }`, Microsoft conseille au développeur de protéger son code par un `try...finally`. Voici le code minimal permettant d'assurer cette protection semblablement à un lock :

```
object verrou = new object();
Monitor.Enter ( verrou );
try
{
    ... lignes de code de la section critique
}
finally
{
    Monitor.Exit ( verrou );
}
```

Dans l'exemple précédent de section critique mettant à jour les 50 cellules d'un tableau d'entiers, nous remplaçons l'instruction `lock` par un appel aux méthodes `static Enter` et `Exit` de la classe `Monitor` sans protection du code :

```
public class ThreadModifierDonnees
{
    private int[] donnees;
    private string nom;
    private static object verrou = new object();

    public ThreadModifierDonnees(int[] donnees, string nom)
    {
        this.donnees = donnees;
        this.nom = nom;
    }
    public void run()
    {
        Monitor.Enter ( verrou );
```

```

for (int i = 0; i < donnees.Length; i++)
{
    if (nom == "modif1")
    {
        donnees[i] += 1;
        Thread.Sleep(1);
    }
    else
    if (nom == "modif2")
    {
        donnees[i] *= 2;
        Thread.Sleep(0);
    }
    else
    if (nom == "modif3")
    {
        donnees[i] -= 1;
    }
}
Monitor.Exit ( verrou );
}
}

```

Même programme avec appel aux méthodes **static** *Enter* et *Exit* de la classe **Monitor** et protection du code par **try...finally** dans la méthode *run()*:

```

public void run() {
    Monitor.Enter ( verrou );
    try
    {
        for (int i = 0; i < donnees.Length; i++) {
            if (nom == "modif1")
            {
                donnees[i] += 1;
                Thread.Sleep(1);
            }
            else
            if (nom == "modif2")
            {
                donnees[i] *= 2;
                Thread.Sleep(0);
            }
            else
            if (nom == "modif3")
            {
                donnees[i] -= 1;
            }
        }
    }
    finally { Monitor.Exit ( verrou ); }
}
}

```

### 3.6 Synchronisation commune aux processus et aux threads

Il existe dans la version C# 2.0 deux classes permettant de travailler aussi bien avec des threads qu'avec des processus.

## La classe Semaphore

```
public sealed class Semaphore : WaitHandle
```

**Semaphore** est une classe dédiée à l'accès à une ressource partagée non pas par un seul thread mais par un nombre déterminé de threads lors de la création du sémaphore à travers son constructeur dont nous donnons une surcharge :

```
public Semaphore ( int initialCount, int maximumCount );
```

`initialCount` = le nombre de thread autorisés à posséder le sémaphore en plus du thread principal, la valeur 0 indique que seul le thread principal possède le sémaphore au départ.

`MaximumCount` = le nombre maximal de thread autorisés à posséder le sémaphore.

Pour essayer de simuler une section critique accessible par un seul thread à la fois par la notion de verrou, il faut instancier un sémaphore avec un `maximumCount = 1` et utiliser les méthodes `WaitOne` et `Release` de la classe `Semaphore` :

```
Semaphore verrou = new Semaphore (0, 1);  
verrou.WaitOne ();  
... lignes de code de la section critique  
verrou.Release ();
```

Toutefois l'utilisation d'un sémaphore même avec `maximumCount = 1` ne permet pas d'ordonner les accès à la section critique, dans ce cas le sémaphore sert seulement à assurer qu'une section critique est accédée entièrement par un seul thread. Illustrons ce propos avec notre exemple de code partagé mettant à jour les 50 cellules d'un tableau d'entiers.

Nous créons un sémaphore **public static** dans la classe `ThreadModifierDonnees` que nous nommons `verrou`, nous supprimons les endormissements `Sleep(...)`, nousinstancions le sémaphore `Semaphore verrou = new Semaphore (0, 1)`, enfin nous encadrons la section critique par les appels des méthodes `WaitOne` et `Release` :

```
public class ThreadModifierDonnees  
{  
    private int[] donnees;  
    private string nom;  
    public static Semaphore verrou = new Semaphore (0, 1);  
  
    public ThreadModifierDonnees(int[] donnees, string nom)  
    {  
        this.donnees = donnees;  
        this.nom = nom;  
    }  
    public void run()  
    {  
        verrou.WaitOne();  
        for (int i = 0; i < donnees.Length; i++)  
        {  
            if (nom == "modif1")
```





```

public static Mutex verrou = new Mutex ( );

public ThreadModifierDonnees(int[] donnees, string nom)
{
    this.donnees = donnees;
    this.nom = nom;
}
public void run()
{
    verrou.WaitOne( );
    for (int i = 0; i < donnees.Length; i++)
    {
        if (nom == "modif1")
        {
            donnees[i] += 1;
            Thread.Sleep(1);
        }
        else
        if (nom == "modif2")
        {
            donnees[i] *= 2;
            Thread.Sleep(0);
        }
        else
        if (nom == "modif3")
        {
            donnees[i] -= 1;
        }
    }
    verrou.ReleaseMutex( );
}
}

```

Nous retrouvons dans la méthode Main de la classe `Program`, un code identique à celui correspondant à une utilisation d'un **lock** ou d'un **Monitor** :

```

static void Main(string[] args)
{
    initDatas();
    ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas, "modif1");
    ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas, "modif2");
    ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
    afficherDatas();
    Thread thread1 = new Thread(new ThreadStart(modif_1.run));
    Thread thread2 = new Thread(new ThreadStart(modif_2.run));
    Thread thread3 = new Thread(new ThreadStart(modif_3.run));
    thread1.Start();
    thread2.Start();
    thread3.Start();
    afficherDatas();
    System.Console.ReadLine();
}
}

```

Nous donnons les résultats obtenus selon l'ordre de lancement des thread :

1°) Pour l'ordre de lancement des threads suivant :

```

thread1.Start();
thread2.Start();
thread3.Start();

```

