# Microsoft® Visual C#®

Visual Studio

ESIGELEC

# Introduction

# Objectifs du cours



## Principaux objectifs du cours

- Connaître les possibilités du langage C#
- Connaître la syntaxe et la sémantique de C# comme
  - Langage *orienté objet* (encapsulation, héritage, interfaces, polymorphisme, associations et compositions)
  - Langage *orienté composant* (réutilisation via le *framework* .NET: interfaces de type bureau et web, opérations inter langages, opérations d'E/S(1) avec les flots séquentiels et la sérialisation, interfaçage avec des composants COM(2))
  - Langage *orienté données* (LINQ(3) et EF(4))
- Être capable de développer une application graphique dans l'environnement Visual Studio 2012 utilisant les concepts de la programmation orientée objet et les possibilités de C#

(1)Entrées/Sorties
(2)Component Object Model
(3)Language Integrated Query
(4)Entity Framework

# Plan du cours

Microsoft C#

1 Historique du C# et présentation de Visual Studio

2 Les fondamentaux du C#

3 Les interfaces utilisateur graphiques

4 Programmation Orientée Objet

5 Le framework .NET

6 Interfaces et polymorphisme

7 Développement de composants .NET

8 Accès aux bases de données

9 Possibilités supplémentaires du C#

Microsoft
C#

# Historique C#
# &
# Présentation Visual Studio

# Historique C# et présentation Visual Studio

Microsoft C#

- Historique C#
- Présentation Visual Studio
- Exercice 1
- Questions de révision

# Le style C

Microsoft C#

## A l'origine : le C

- 1971 : Dennis Ritchie et comparses aux laboratoires Bell (pour UNIX)
- Jusque dans les année 80 : 1er langage sur UNIX et PC
- Aujourd'hui : encore utilisé notamment en programmation système et en embarqué

## La syntaxe de style C

- Instructions terminées par un point-virgule ( **;** )
- Blocs d'instructions encadrés par des accolades ( **{** … **}** )
- Sensible à la casse (minuscules ≠ majuscules)
- Langage procédurale (utilisation de fonctions)

```
#include <stdio.h>
int main(void)
{
    printf("Hello world !\n");
    return 0;
}
```

*Adopter un style de codage clair, indenté et consistant*

# A partir du C…



## Evolution vers d'autres langages avec une syntaxe de base similaire comme

- **le C++**
  - Ajout des concepts objets (P.O.O.)
- **…puis Java**
  - Meilleure portabilité, plus facile à programmer
    - ± performances moindres
- **…puis JavaScript**
  - Ecriture de scripts et développement Web
- **…et puis…**

# Le C#

Microsoft C#

## Apparition en 2001 → 1er langage du 21e siècle

- Créé par Microsoft dans le cadre de .NET
- Dans la lignée C/C++/Java
- Signifie C augmenté (comme le # en musique)

## Objectifs

- + facile à écrire et moins sujet à erreur(s) que C++
- + efficace et performant que Java
- Pour tous types d'application
- Pour tous les niveaux d'environnement

## Principes

- Orienté Objet : tous les concepts de la P.O.O.
- Orienté Composant : caractéristiques intégrées au langage
- Orienté Données : manipulation de données par un langage de requêtes intégré nommé *LINQ* (Language Integrated Query)

# Versions



## Standard ECMA(1) et ISO(2)/IEC(3)

- N'est ***pas*** un langage Microsoft en lui-même

## Nom Microsoft : Visual C# .NET

- 2002 : V1, pré-standard
- 2005 : V2, totalement conforme avec ISO/IEC 23270 1e édition
- 2007 : V3, totalement conforme avec ECMA-344 4e édition
- 2010 : V4, totalement conforme, nouvelles fonctionnalités
- 2012 : V5, totalement conforme, nouvelles fonctionnalités

(1)European Computer Manufacturer's Association
(2)International Organization for Standardization
(3)International Electrotechnical Commission

# Premiers pas

## Hello World en C#

```
using System;
public class HelloWorld
{
    public static int Main()
    {
        Console.Write("Hello world !\n");
        return 0;
    }
}
```

## Syntaxe similaire et sortie identique au programme C

- La méthode `Main` est dans une *classe*
- Utilise la méthode `Console.Write` du Framework .NET
- Sinon, mêmes unités syntaxiques, mêmes mots clés

http://ent.esigelec.fr : Programmation en C# - Exemples du cours – Exemple Hello World

# Les commentaires

Microsoft C#

## En C#, les commentaires ne sont pas traités

- Le compilateur les considère comme des espaces

## 3 types de commentaires

- Multi-lignes encadrés par /* et */
- Mono-ligne commençant par //
- Documentation basée XML mono-ligne commençant par ///

```
///<summary>
/// Ceci sera inclus dans la documentation
///</summary>
public static int Main()
{
    /* commentaire sur
       plusieurs lignes */
    int compteur = 0; // commentaire jusqu'à la fin
    ...               // de la ligne
}
```

# Historique C# et présentation Visual Studio

Microsoft C#

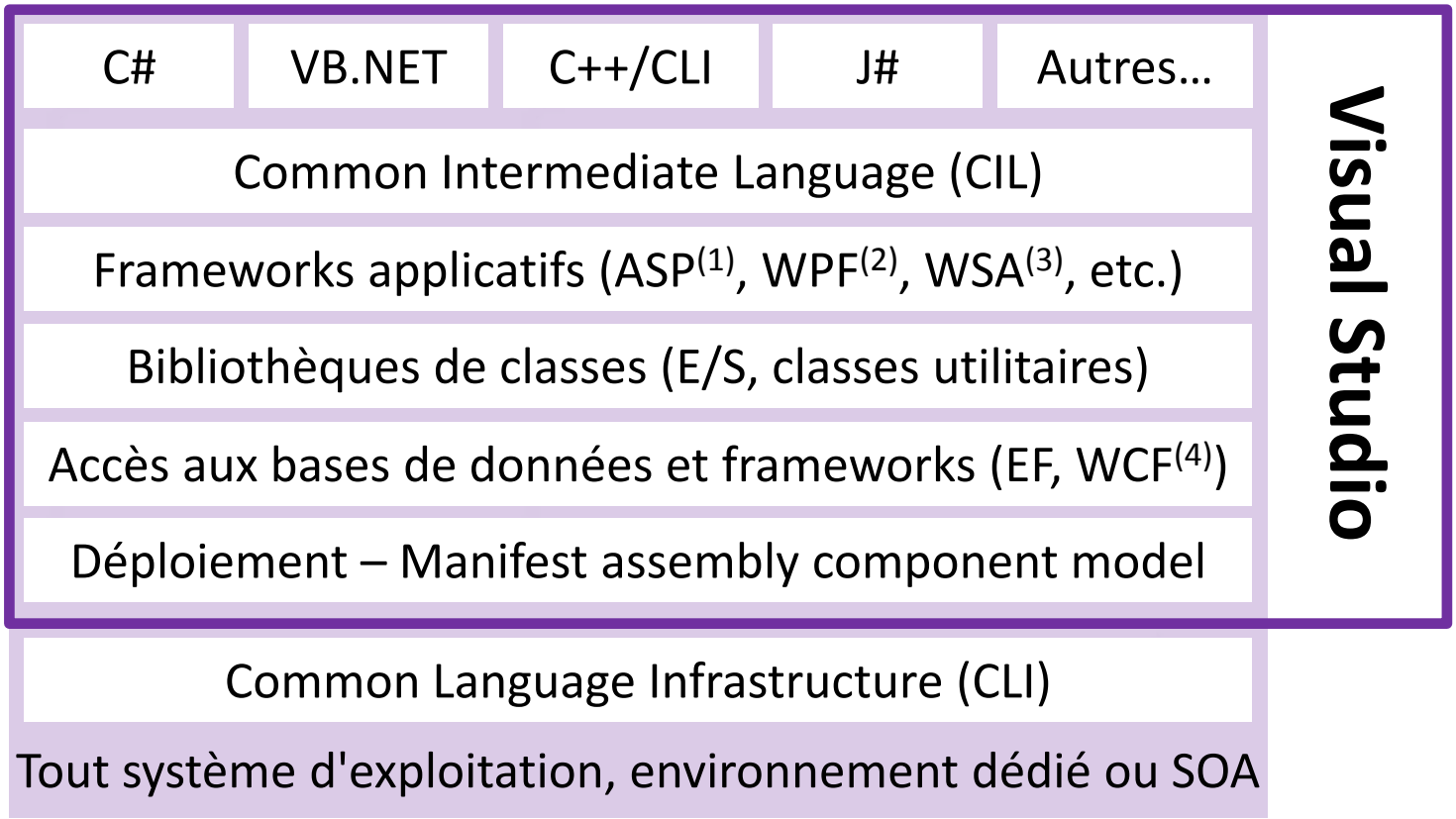
- Historique C#
- Présentation Visual Studio
- Exercice 1
- Questions de révision

# Architecture .NET

Microsoft C#

## Architecture globale organisée en couches logiques

- Pour les développements supportés par Visual Studio




(1)Active Server Page
(2)Windows Presentation Foundation
(3)Windows Store Apps (metro)
(4)Windows Communication Foundation

# Vue d'ensemble de Visual Studio

Microsoft C#

## Incontournable pour .NET et C#

- Différentes versions : Express, Professional, Ultimate, etc.
- www.microsoft.com

## Visual Studio 2012 Ultimate est installé

- Il inclut :
  - SQL Server 2012 Local DB
  - C# V5
  - Frameworks .NET jusqu'à la version 4.5
  - Entity Framework release 5

## Exercices compatibles avec des versions antérieures

- Sauf si le langage ou les bibliothèques ne le supportent pas
- Nécessitent généralement la création d'un nouveau projet (utiliser Add | Existing Item pour inclure un fichier d'un exercice)
- Si problème de version → indication dans le cours

# Visual Studio : Solutions et projets



**Une solution Visual Studio représente l'intégralité d'une application (.sln)**

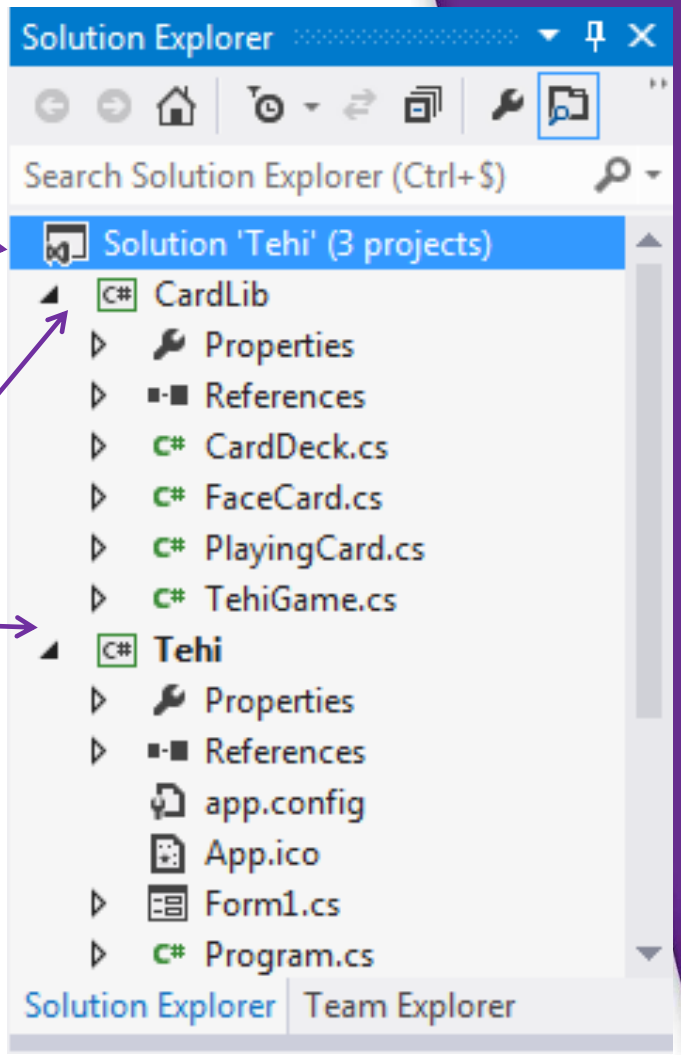
- Avec éventuellement plusieurs projets

**Un projet est une collection de fichiers fortement liés**

- Souvent une bibliothèque de classes (.dll)
- Ou un exécutable (.exe)

**☞ Le même projet peut être inclus dans différentes solutions**

**☞ Pour les exercices, "ouvrir la solution" ne signifie pas ouvrir le corrigé de l'exercice !**



# Du code *managé*

Microsoft C#

**Les compilateurs .NET produisent du CIL (Common Intermediate Language) et des métadonnées**

- Les métadonnées indiquent le type du contenu des fichiers (aucune copie ou inclusion ne sont nécessaires)

**Le compilateur Just-In-Time (JIT) convertit le CIL en code binaire natif lors de l'exécution**

**Pour exécuter un programme .NET, la CLI doit être présente**

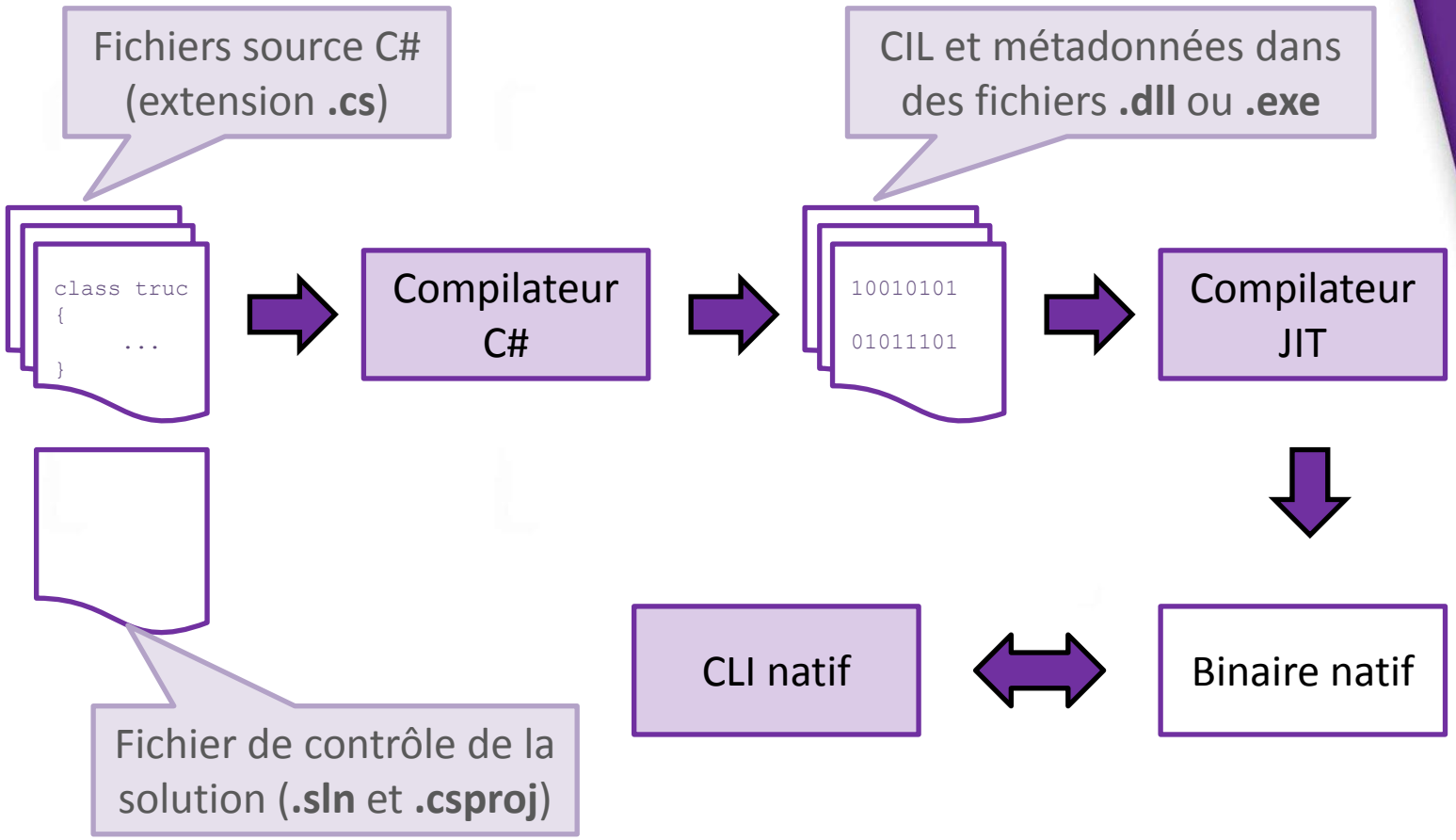
- L'implémentation Windows se nomme la CLR (Common Language Runtime)

**Le binaire natif et la CLI sont dans un dossier qui inclut le GAC (Global Assembly Cache)**

- Pour ce cours : C:\Windows\Microsoft.NET\Assembly

# Code *managé*

## Environnement de la compilation à l'exécution



# La CLI (Common Language Infrastructure)

Microsoft C#

## Pour exécuter un programme .NET, la CLI est nécessaire

- Windows : CLR installée de base dans les versions récentes
- Unix/Linux : la plus populaire est celle de Mono-Project (MPCLI)
- Référence Microsoft : SSCLI (Shared Source CLI)
- MonoTouch pour iPad, MonoDroid pour Android

## Services fournis à l'exécution par la CLI

- Intégration inter-langages
- Ramasse-miettes
- Accès et interaction avec des composants simples
- Sécurité et validation à l'exécution
- Connexion et communication avec les bases de données
- Autres fonctionnalités runtime typiques

# Les assemblies (assemblages)



## Un fichier .exe ou .dll est nommé *assembly*

- Parfois aussi nommé composant

## L'ensemble des assemblages est couramment placé dans un simple répertoire

- Avec une éventuelle arborescence
- Incluant souvent toutes les ressources nécessaires (images…)
- Les DLL externes peuvent y être placées aussi (ce que fait Visual Studio lors de l'ajout d'une référence)

## Déployer une application .NET = copier les assemblages dans l'emplacement d'installation

# Historique C# et présentation Visual Studio

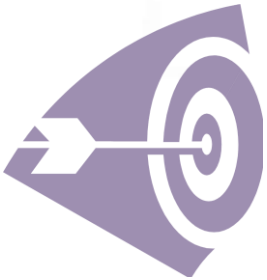
Microsoft C#

- Historique C#
- Présentation Visual Studio
- Exercice 1
- Questions de révision

# Exercice 1

## Objectifs de l'exercice

- La prise en main des outils de développement
- Travailler sur un petit programme de "conversion monétaire"
- Observer solutions, projets et assemblages



*Se reporter à l'énoncé de l'exercice 1 sur ent.esigelec.fr*

# Historique C# et présentation Visual Studio

Microsoft C#

- Historique C#
- Présentation Visual Studio
- Exercice 1
- Questions de révision

# Questions de révision

Microsoft C#

## Que produit le compilateur C# de Visual Studio ?

## Quelles sont les bonnes propositions ?

- ❑ Le standard C# est approuvé par ECMA
- ❑ Un programme écrit en C# ne peut s'exécuter que sous Windows
- ❑ Une solution Visual Studio peut contenir un ou plusieurs projets

## Comment déployer une solution .NET ?

## Que peut signifier "code managé" et qui le manage ?

# Les fondamentaux du C#

☞ http://msdn.microsoft.com/fr-fr/

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Organisation d'un programme

Microsoft C#

## Rappel de l'organisation du programme de conversion

```
namespace CurrencyConverter
{
    class Program
    {
        static void Main(string[] args)
        {
            /* instructions */
            System.Console.Write(…);
            System.Console.WriteLine(…);
        }
        static double GetAmount(string prompt)
        {
            /* instructions */
        }
        static double Convert(string ic, double d)
        {
            /* instructions */
        }
        static char SymbolFor(string currency)
        {
            /* instructions */
        }
    }
}
```

http://ent.esigelec.fr : Programmation en C# - Exercices du cours – Ex01

# Les espaces de noms

Microsoft C#

## Le programme est organisé en niveaux

## Le niveau le plus haut commence par la ligne :

```
namespace CurrencyConverter
```

- Regroupe les programmes développés ensemble
- Appartenance de plusieurs sources au même espace de noms
- Utilisation facultative sauf pour les composants mais

## Prévention des collisions de noms

- Simplification du déploiement et facilité de réutilisation

## Possibilité d'organisation hiérarchique

- Sous-espaces de noms séparés par un point (ex : System.IO)

# Les espaces de noms (suite)



## On peut y placer un ou plusieurs :

- Classes : type référence défini par le développeur
- Structures : type valeur défini par le développeur
- Interfaces : spécification d'un service
- Énumérations : spécification d'un type énuméré
- Délégués : spécification d'une adresse de fonction
- Espaces de noms imbriqués

## Dans l'exercice, une seule classe `Program`

- Par convention, c'est le nom de la classe qui contient la méthode `Main`

# Les classes



## Chaque classe ou structure peut contenir des :

- Champs : données membres
- Méthodes : fonctions (ou procédures) membres
- Propriétés : syntaxe alternative pour lire / écrire un champ
- Opérateurs : symbole alternatif pour une méthode
- Events : permet le choix d'une fonction lors de l'exécution
- Énumérations : spécification d'un type énuméré
- Délégués : spécification d'une adresse de fonction
- Classes et structures imbriquées

## Dans l'exercice, seulement des méthodes

- `Main`, `GetAmount`, `SymbolFor` et `Convert`

## L'espace de noms peut être utilisé à l'appel d'une méthode :

- System.Console.Writeline(…);
- `System` : espace de noms, `Console` : classe de cet espace

# La directive *using*

Microsoft C#

## Alternative à l'exercice précédent

```
using System;
namespace CurrencyConverter;
{
    class Program
    {
        static void Main(string[] args)
        {
            /* instructions */
            Console.Write(…);
            Console.WriteLine(…);
        }
        /* autres méthodes*/
    }
}
```

Une directive `using` ici et …

… il n'est plus nécessaire d'utiliser le nom de l'espace ici

## Possibilité intéressante

- Évite de la saisie sans alourdir le programme
- Pas de collision avec les assemblies du framework mais attention avec ceux implémentés dans les programmes

http://ent.esigelec.fr : Programmation en C# - Exercices du cours – Ex02

# Les méthodes : définition

Microsoft C#

## Une méthode possède, en général, une définition en deux parties :

- Une spécification (ou en-tête ou interface) qui indique comment l'utiliser
- Un corps (ou bloc, ou implémentation) qui indique ce qu'elle fait

## Dans l'exercice précédent, `Main` est définie comme suit :

```
static void Main(string[] args)
{
    /* instructions */
}
```

Spécification

Corps

# Les méthodes : spécification

Microsoft C#

## La spécification indique comment une méthode est utilisable selon le format suivant :

```
[qualifieurs] [type de retour] Nom([paramètres])
```

## Dans l'exercice précédent :

- Le nom de la méthode est `Main`
- Le seul qualifieur est `static`
  - Il signifie accessible au niveau de la classe
    - ☞ Pas nécessaire de créer une instance (membre de classe)
    - ☞ La plupart des méthodes ne sont **pas** `static`
- Le seul paramètre est `string[] args`
  - Un tableau de chaînes est transmis à la méthode `Main`
- Le type de retour est `void`
  - Aucune donnée n'est retournée par la méthode

# Les méthodes : signature



## Le nom d'une méthode combiné avec la liste de ses paramètres est appelé *signature*

## L'identification d'une méthode se fait par sa signature

- Une signature différente indique une méthode différente
- Exemple : `faire(int i)` ≠ `faire(double d)`
  - Quelle méthode sera appelée par `faire(12.5)` ?

## Cette différenciation s'appelle la surcharge (overloading) de méthodes

*A première vue la surcharge peut sembler non nécessaire et potentiellement déroutante. Or, elle est indispensable et rend la programmation orientée objet plus claire et aisée*

# La méthode `Main`



## Une seule signature possible dans une même classe

- Un seul point d'entrée possible dans un programme

## Cependant, la méthode `Main` peut avoir une spécification différente selon le type d'application

## Par exemple :

- `static void Main()`
- `static int Main(string[] args)`

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Les types primaires

Microsoft C#

## Le compilateur interprète directement les types primaires

| Type | Description | V | R | Commentaires |
|---|---|---|---|---|
| `int` | 32 bits signé | X | | Autres : `byte`, `ulong`, `short`, etc. |
| `char` | 16 bits Unicode | X | | Cf. www.unicode.org |
| `bool` | `true` ou `false` | X | | |
| `double` | IEEE 64 bits | X | | Cf. www.ieee.org autre : `float` (32 bits) |
| `decimal` | 128 bits | X | | Précision de 28 décimales |
| `enum` | N/A | X | | Valeurs de type primaire |
| `string` | 16 bits Unicode | | X | **Pas** un tableau de caractères |
| `array` | N/A | | X | Tableau de type quelconque |
| `object` | N/A | | X | Classe de base pour tous les types |

V: Valeur R: Référence

# Les variables : valeur et référence



## Valeur signifie que la donnée est stockée et accédée directement

- Sur la pile

## Référence signifie que la donnée est stockée et accédée indirectement

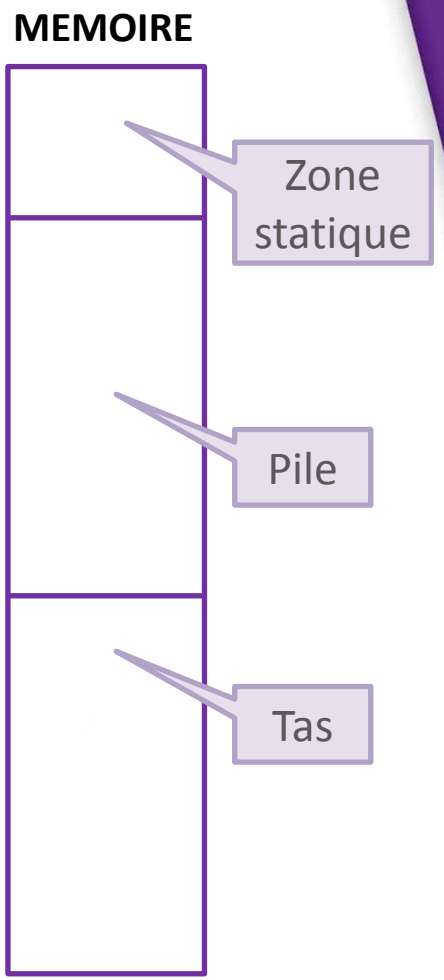
- Habituellement sur le tas

## Par exemple :

```
public static void Main()
{
    double inAmt, outAmt;
    string inCur = "USD";
    string outCur = "EUR";
    /* instructions */
}
```

Alloué sur ?

Alloué sur ?



# Types définis par l'utilisateur

Microsoft C#

## Les autres types de données sont les *class* et les *struct*

- Non prédéfinis dans le langage
- Définis par l'utilisateur ou obtenus depuis une bibliothèque

## Les structures sont *toujours* un type valeur

- Allouées automatiquement sur la pile

```
complex c;
```

## Les classes sont *toujours* un type référence

- Allouées usuellement sur le tas avec le mot-clé new

```
Compte c = new Compte();
```

## Une instance d'un type *class* ou *struct* est nommée objet

## Les structures sont plus rarement utilisées que les classes

# Les littéraux numériques

Microsoft C#

## Les littéraux entiers peuvent être en base 10 ou 16

- **Pas** en base 8
- `123, 0x53BD`
- Les littéraux longs sont suivis par un L

## Les littéraux flottants se représentent avec un `.` ou un `e`

- Ne peuvent pas se terminer par un `.`
- `1.0, 5E3, 3.53e-13`
- Les littéraux `float` sont suivis par un `f`
- Les littéraux `decimal` sont suivis par un `m`

# Les littéraux alphanumériques (textuels)



## Les littéraux caractères sont encadrés par des simples quotes (apostrophes) et c'est *Unicode* qui est employé

- `'X'`
- `'\u20AC'` est le symbole de l'Euro €
- `'\n'` est le passage à la ligne
- `'\\'` est le backslash

## Les littéraux chaînes de caractères sont encadrés par des doubles quotes (guillemets)

- `"C:\\temp\\log_prg1.txt"`
- `@"C:\temp\log_prg1.txt"`

☞ Toutes les polices de caractères des plateformes Windows ne supportent pas *Unicode*

http://ent.esigelec.fr : Programmation en C# - Exemples du cours – Exemple test Unicode

# Définition explicite et implicite



## C# est un langage à typage fort

- Variables initialisées ou affectées **avant** leur utilisation
- Les deux formes suivantes sont valides

```
string inCur;
inCur = "USD";
string outCur = "EUR";
```

Affectation

Initialisation

## ☝V3 +, typage implicite possible (mot-clé var)

- Possible seulement en cas d'initialisation

```
var aStr;
aStr = "ABC";
var eStr = "EFG";
```

Invalide

Valide

☞ Le typage implicite demeure un typage fort

http://ent.esigelec.fr : Programmation en C# - Exemples du cours – Exemple typage implicite

# Définition explicite et implicite (suite)

Microsoft C#

## Quels sont les types de données des définitions implicites suivantes ?

```
1. var x = 1.26m;
2. var y = '\u02A5';
3. var z = 11;
4. var c = new Compte();
5. var b;
```

## Le typage implicite nuit à la compréhension d'un programme

- Ajouté à C# seulement pour les cas particuliers où il peut-être difficile de déterminer par avance le type d'une variable
  - ☞ Utilisation de LINQ par exemple
  - ☞ Son emploi est éviter dans ce cours

http://ent.esigelec.fr : Programmation en C# - Exemples du cours – Exemple typage implicite

# Les fondamentaux du C#

Microsoft C#

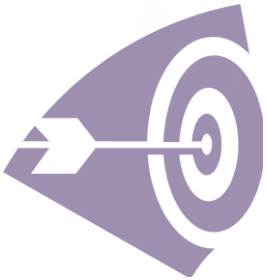
- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Exercice 2



## Objectifs de l'exercice

- Simplifier un programme en employant la directive `using`
- Utiliser un composant de bibliothèque dans un autre espace de noms



*Se reporter à l'énoncé de l'exercice 2 sur ent.esigelec.fr*

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Les expressions

## Fonction de conversion de l'exercice précédent

```
public static double Convert(string ic, double amt)
{
    const double convRate = 1.395;
    double convValue;
    if (ic == "EUR")
        convValue = amt * convRate;
    else
        convValue = amt / convRate;
    return convValue;
}
```

**const** : convRate doit être initialisé dans sa définition, sa valeur ne peut pas changer

## L'instruction (`amt / convRate`) est une expression

- Possède un *type* et une *valeur*
- Utilisable directement. Exemple :

```
if (ic == "EUR")
    return amt * convRate;
else
    return amt / convRate;
```

http://ent.esigelec.fr : Programmation en C# - Exercices du cours – Ex02

# Les opérateurs



## Principaux opérateurs de C# :

| Type | Symboles | Description |
|---|---|---|
| Arithmétique | + – * / % | Précédence standard |
| Affectation | = | Ou initialisation (☝ affectation étendue) |
| Incrément | ++ –– | Ajoute ou soustrait 1 de l'opérande |
| Égalité | == != | Ne pas confondre = et == |
| Relationnel | > < >= <= | |
| Logique | && \|\| ! | |
| Bit à bit | & \| ^ ~ | Ne pas confondre & et &&, etc. |
| Décalage | << >> | Décalage des bits à gauche et à droite |
| Conditionnel | ?: | Inline – assez peu employé en C# |
| Lambda | => | Raccourci pour une méthode – ☝**V3+** |
| Nullable | ? ?? | Pertinent pour les SGBD |

# Les opérateurs (suite)

Microsoft C#

## Les opérateurs symboliques ont généralement un comportement identique aux autres langages

- Usage, précédence et associativité similaires
- Groupement modifié avec des parenthèses : ( )

## La division (/) entre entiers tronque

- Exemple : `11/4` donne `2` et non `3` (comme ferait VB)
- Opère comme l'`\` de VB

## Affectation étendue (ou composée) : combinaison des opérateurs arithmétiques ou logiques avec l'affectation (=)

- Exemple : `i += 2` ⇔ `i = i + 2`
  `j %= 3` ⇔ `j = j % 3`

## Si affectation type valeur, alors copie de la valeur, idem réf.

# L'opérateur d'égalité

Microsoft C#

## La comparaison de types valeur compare les valeurs

## La comparaison de types référence compare aussi les valeurs ( = adresses !)

- La surcharge permettra de résoudre ce problème
- Exemple :

```
public static void Main()
{
    int i = 5;
    int j = 5;
    if(i == j) ...    // VRAI

    Compte c1 = new Compte(123);
    Compte c2 = new Compte(123);
    if(c1 == c2) ... // FAUX, pourquoi ?
}
```

# Opération multi-types



## Opérations multi-types possibles si aucun risque de corruption de données

- Du plus "petit" vers le + "grand" : fonctionne généralement
- Types flottants + "grands" que les entiers
- Conversion implicite : la valeur du + "petit" est convertie temporairement en + "grand" et le résultat est du type du + "grand"
- Exemple :

```
public static void Main()
{
     int i = 5;
     long n = 4;
     double x; ;
     x = n * i;  // valeur de i convertie en long
                 // le résultat long est converti en double
}
```

# Sous-typage (downcast)

Microsoft C#

## Si perte de données possible, alors sous-typage

- Habituellement pour passer d'un "grand" type à un + "petit"
- Exemple :

```
bool b; char c; int i; long n; double d; decimal x;
c = 97;          //erreur: pas de conversion implicite
c = (char)97;    //coercition: c devient l'Unicode 'a'
i = (int)5.8;    //coercition: i devient 5
b = i;           //erreur: pas de conversion int vers bool
b = (bool)i;     //erreur: pas de conversion int vers bool
n = i;           //ok: conversion implicite (petit vers grand)
d = n;           //ok: conversion implicite (petit vers grand)
x = d;           //erreur: pas de conversion implicite
d = x;           //erreur: pas de conversion implicite
d = (double)x; //ok: coercition
```

## Attention aux sous-typages

- Affecter un flottant à un entier tronque la valeur fractionnaire
- Affecter un grand entier à un petit fait perdre les bits de poids fort

# Suite exercice 2 : coercition

Microsoft C#

**Récupérer la correction de l'exercice 2 et remplacer les types `double` par `decimal`**

- `decimal` est plus adapté pour les montants financiers
- Utiliser "Edit | Quick Replace" (sur "current document" !)

**Lancer la compilation : erreurs ! Trouver les instructions d'initialisation et les arguments auxquels des erreurs sont associées. Modifier les littéraux en `decimal` (suffixe `m`)**

**Lancer la compilation : erreurs ! (`EuroTable` retourne des résultats de type `double`. Faire la coercition des valeurs retournées en `decimal` .**

**Lancer la compilation : OK, avec une meilleure précision**

# Les énumérations : exemple

Microsoft C#

## Une `enum` en C# peut être de n'importe quel type intégré (`int` par défaut)

- La première valeur n'a pas à être 0 ou 1, aucune séquence imposée

## Placée dans la portée d'un `namespace` ou d'une `class`

- Ne peut être dans la portée d'une méthode

```
public class Banque
{
    public enum CodeMonnaie {EUR = 1, USD, CAD, GBP, JPY}
    public static void Main()
    {
        int ic = (int)CodeMonnaie.CAD;      // Coercition OK
        CodeMonnaie code = CodeMonnaie.JPY;
        // CodeMonnaie.cm1 = 1;             // ERREUR !
        CodeMonnaie cm2 = (CodeMonnaie) 4; // Coercition OK
        Console.WriteLine(cm2);             // Affiche ?
    }
}
```

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Les tableaux



## C# gère des tableaux de n'importe quel type

- Stockage linéaire d'éléments accessibles par un indice

## Tableaux alloués dynamiquement sur le tas avec `new`

- `int[] tab = new int[50];`

## Tableaux créés en utilisant un bloc d'initialisation

- `int[] premiers = {2, 3, 5, 7, 11, 13};`

## Pour connaître la taille d'un tableau : propriété `Length`

- `int dim = premiers.Length;`

## Une fois alloué, l'indexation et l'usage sont standards

- `tab[9] = 17; //10e élément (1er indice = 0)`

## Quelle conséquence a l'utilisation d'un indice hors bornes ?

# Les tableaux multidimensionnels



**Comme en Java, C# accepte les tableaux de "taille irrégulière". Leur allocation se fait ligne par ligne.**

```
double[][] matrice = new double[2][];
matrice[0] = new double[12];
matrice[1] = new double[31];
```

**Valeurs de `matrice.Length`, `matrice[1].Length` ?**

**Le C# supporte les tableaux multidimensionnels "vrais" de type Fortran**

```
int[,] tab2D = new int[5,10];
```

**Valeur de `tab2D.Length` ?**

# Les chaînes de caractères

Microsoft C#

## Elles ne sont pas un tableau au sens C/C++

- Un indice peut être utilisé pour **lire** un caractère
- 
```
string maChaine = "Hello";
char x = maChaine[1]; // x ?
```

## Pour connaître la taille d'une chaîne: propriété `Length`

- 
```
int taille = maChaine.Length; // taille ?
```

## Les `string` sont non-modifiables (*immuables*)

- 
```
maChaine.ToUpper(); // ne fait rien
```
- 
```
maChaine = maChaine.ToUpper(); // OK
```

## Concaténation (+) avec une autre chaîne ou tout autre type

- 
```
string resultat = "Le total vaut " + somme;
```

## Comparaisons : égalité (==), différence (!=)

# Les chaînes de caractères (suite)

Microsoft C#

## Opérations nommées (méthodes)

- Exemple (extraction d'une sous-chaîne) :

```
string immat = "1234AB56";
string dept = immat.Substring(6,2);
```

## Formatage d'une chaîne (`Format`)

- Exemple :

```
double x = 1234.5678;
string os = string.Format("Mt = ${0:N2}", x);
```

- Produit : `Mt = $1 234.57`
- Elément de mise en forme : `{index,[alignement]:[format]}`
- Quelques spécificateurs de format :
  - Plus sur http://msdn.microsoft.com/fr-fr/library/system.string.format.aspx :
  - D : décimal, convertit la sortie en format entier
  - E : exponentiel, format scientifique
  - F : point décimal fixé, précision après le point
  - N : numérique, semblable à F mais avec séparation par milliers

# Les chaînes de caractères (suite)

Microsoft C#

## Largeur et justification peuvent être précisées

- `string os = string.Format("{0,-5}", data);`
- Champ justifié à gauche (-) avec 5 caractères

## Largeur et formatage combinables et exploitables dans les E/S

- `Console.WriteLine(">{0,12:F3}<", 1234.567890);`
- Sortie ?

---

## Tous les types de données convertissent une chaîne en leur type

- Exemple (conversion d'une `string` en `int`):

```
string is = "12345";
int i = int.Parse(is);
```

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Transmission d'arguments

Microsoft C#

## En C#, paramètres et arguments peuvent être :

- De type valeur
- De type référence

## Pour les types valeur, la donnée est copiée

- Transmission par *valeur*

## Pour les types référence, l'adresse de la donnée est passée

- Transmission par *valeur de référence*

## Lors du retour d'une donnée

- Les types valeur sont *retournés par valeur*
- Les types référence sont *retournés par valeur de référence*

# Transmission de type valeur

Microsoft C#

## Exemple :

```
public static void Main()
{
    int[] code = new int[2];
    code[0] = 1234;
    code[1] = 5678;
    int num = 9876;
    ChangeCode(num, code);
    Console.WriteLine("num = " + num);
    Console.WriteLine("code[0] = " + code[0]);
    Console.WriteLine("code[1] = " + code[1]);
}

public static void ChangeCode(int n, int[] c)
{
    c[0] = c[1];
    c[1] = n++;
}
```

## Qu'est-il affiché pour num et code ?

# Transmission de type valeur par référence

Microsoft C#

## Exemple :

```
public static void Main()
{
    int[] code = new int[2];
    code[0] = 1234;
    code[1] = 5678;
    int num = 9876;
    ChangeCode(ref num, code);
    Console.WriteLine("num = " + num);
    Console.WriteLine("code[0] = " + code[0]);
    Console.WriteLine("code[1] = " + code[1]);
}

public static void ChangeCode(ref int n, int[] c)
{
    c[0] = c[1];
    c[1] = n++;
}
```

Doit être utilisé par l'appelant

Doit faire partie de la signature

## Qu'est-il affiché pour num et code ?

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Boucles et conditions

Microsoft C#

## Boucles et instructions conditionnelles traditionnelles

| Instruction | Exemple |
|---|---|
| `if` | `if(a > b) c = d;` |
| `if...else` | `if(a > b) c = d`<br>`else c = e;` |
| `switch` | Voir pages suivantes |
| `while` | `while(a > b) b += 2;` |
| `do...while` | `do(a = AFaire();) while(a > b);` |
| `for` | `for(i = 0; i < n; i++) AFaire(i);` |
| `foreach` | Voir pages suivantes |
| `goto` | `label: ...code...`<br>`...`<br>`goto label;` |

# La boucle foreach



## Appliquer la même opération à tous les éléments d'une collection

- Les tableaux constituent un exemple de collection
- La boucle foreach rend le codage très simple

## Exemple :

```
public void CalculPaie(string[] employes)
{
    foreach(string personne in employes)
    {
        Console.WriteLine("Paye de " + personne);
        EffectueCalcul(personne);
    }
}
```

# La sélection switch

Microsoft C#

## Pour éviter les if...else en cascade : switch

- Un case sans break permet de passer au case suivant

## Exemple :

```
... code ...
int nbj = GetNbJoueursTarot();
switch(nbj)
{
    case 2  : Console.WriteLine("Jeu impossible !");
              break;
    case 3  :
    case 4  : Console.WriteLine("Jeu à 3 ou 4 : écart de 6 cartes");
              break;
    case 5  : Console.WriteLine("Jeu à 5 : écart de 3 cartes");
              break;
    default : Console.Write("Jeu à 5 + " + (nbj - 5) + " mort(s)");
              Console.WriteLine(" : écart de 3 cartes");
              break;
}
... code ...
```

Donnée simple (pas un objet ni une expression)

Passage au "case" suivant légal

# Les fondamentaux du C#

Microsoft C#

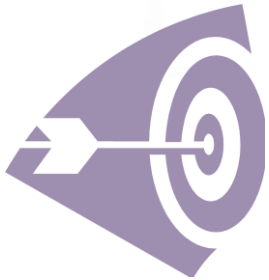
- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Exercice 3



## Objectifs de l'exercice

- Utilisation des arguments de la ligne de commande



*Se reporter à l'énoncé de l'exercice 3 sur ent.esigelec.fr*

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Bugs cachés

Microsoft C#

**Erreurs dans le programme de conversion monétaire !**

**Récupérer la correction de l'exercice 3**

**Exécuter le programme**

**À la demande, entrez `cent` au lieu de `100`. Que se passe-t-il ?**

- Cliquer Non si ouverture d'une boîte de dialogue

**Laisser la solution ouverte pour l'instant**

# Exceptions

Microsoft C#

## Mécanisme sûr et standardisé pour traiter les erreurs

- Utilisé extensivement dans les bibliothèques du framework .NET (un très grand nombre d'exceptions y sont définies)
  - System.OverflowException
  - System.ArithmeticException
  - System.IO.IOException
  - System.IO.FileNotFoundException
  - …

## Objet de type *classe* hérité de `System.Exception`

## Le développeur peut définir ses propres exceptions

- Ne pas traiter les exceptions génère des programmes peu fiables

## ☞ L'héritage est abordé dans un prochain chapitre

# Bug cachés (suite)

Microsoft C#

## Le programme aurait dû traiter les exceptions :

```
public static double GetAmount(string prompt)
{
    double data = 0.00;
    bool noData = true;
    while(noData)
    {
        try
        {
            System.Console.Write(prompt);
            string is = System.Console.ReadLine();
            data = double.Parse(is);
            if(data > 0.00) noData = false;
        }
        catch(FormatException e)
         {
             System.Console.WriteLine(e.Message);
         }
    }
    return data;
}
```

## Inclure la gestion des exceptions dans le programme

# try...catch...finally



## Forme générale du traitement d'une exception :

```
try
{
    //code qui peut lever exception_1 ou exception_2
}
catch(exception_1 nom_instance)
{
    //arrive ici si exception_1 est levee dans le bloc try
}
catch(exception_2 nom_instance) //plusieurs catch !
{
    //arrive ici si exception_2 est levee dans le bloc try
}
finally   //optionnel
{
    //arrive ici dans tous les cas, exception ou pas
}
```

# Levée d'une exception



## Enchaînement des événements suivants :

- Un nouvel objet exception est créé
- Le flot du programme saute toutes les instructions depuis l'endroit où l'exception a été levée jusqu'à la clause `catch` convenable la plus proche
- Si aucune clause `catch` convenable n'est trouvée, le programme se termine
  - Un message indique qu'une exception est levée
  - Une trace de la pile des appels est affichée

## La clause `finally` est exécutée dans tous les cas

# Les blocs `using`

Microsoft C#

## Un bloc `using` existe pour les objets `Disposable`

- Alternative simple au `try-catch-finally`
- Équivalent à un `try-finally` sans `catch`

```
using(DisposableObject dobj = new DisposableObject())
{
    // logique susceptible de lever une exception
}
```

- Tels que ceux des classes d'E/S, comme `StreamWriter`

**L'objet est automatiquement *disposé* (fermé), comme s'il l'était dans un bloc `finally`, à la fermeture du bloc `using`**

**Possibilité de capturer ultérieurement l'exception levée si cela est nécessaire**

# Lever une exception

Microsoft C#

## Mécanisme préféré de traitement des erreurs en C#

## Utilisation du mot-clé `throw`

## Exemple :

```
public void Deposer(decimal montant)
{
    ... code ...
    if(montant <= 0.00M)
    {
        throw new ArgumentException("Montant négatif !");
    }
     .. code ...
}
```

## Exécution poursuivie au plus proche bloc `catch`

- Ou fin du programme s'il n'y en a pas

# Eviter une exception non voulue

Microsoft C#

## Possibilité en C# d'activer ou désactiver des exceptions dans les blocs `checked` / `unchecked` "imbriquables"

## Exemple :

```
public static void Main()
{
     int val = int.MaxValue;
     unchecked
     {
          val++;
     }
       .. code ...
}
```

L'exception liée au dépassement de capacité arithmétique ne sera pas levée

- **Déclenchement des exceptions arithmétiques contrôlé par une option du compilateur**
  - Par défaut, le compilateur Visual Studio ne les lève pas
  - Peut être différent pour d'autres compilateurs

# Les fondamentaux du C#

Microsoft C#

- Espaces de noms et méthodes
- Types de données et littéraux
- Exercice 2
- Expressions et opérateurs
- Tableaux et chaînes
- Transmission d'arguments
- Boucles et conditions
- Exercice 3
- Exceptions
- Questions de révision

# Questions de révision

Microsoft C#

Qu'est-ce qu'un espace de noms ?

_____

Qu'est-ce que la surcharge ?

_____

Comme les variables, les expressions ont un(e) _____ et un(e) _____

Quelle est la syntaxe d'une boucle `foreach` ?

_____

_____

# Questions de révision (suite)

Microsoft C#

**Dans quel(s) cas la conversion implicite de type est-elle permise ?**

**Qu'est-ce qu'un sous-typage ?**

**Quels sont les six mots-clés associés aux exceptions ?**

Microsoft
C#

# Applications graphiques

# Applications graphiques



- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Évolution des interfaces utilisateur



## Jusqu'en 1984 les interfaces de style console étaient la norme

- Encore utilisées de nos jours…
- Ligne de commande Windows (exemple de reliquat)

## En 1984, Apple lance le Macintosh

- Bureau "intuitif" : interface orientée souris

## En 1995, apparition de Windows 95 (interface semblable à celle du Mac

- Domine en 2001 plus de 87% du marché

## En 1995, le Web prend sa forme actuelle

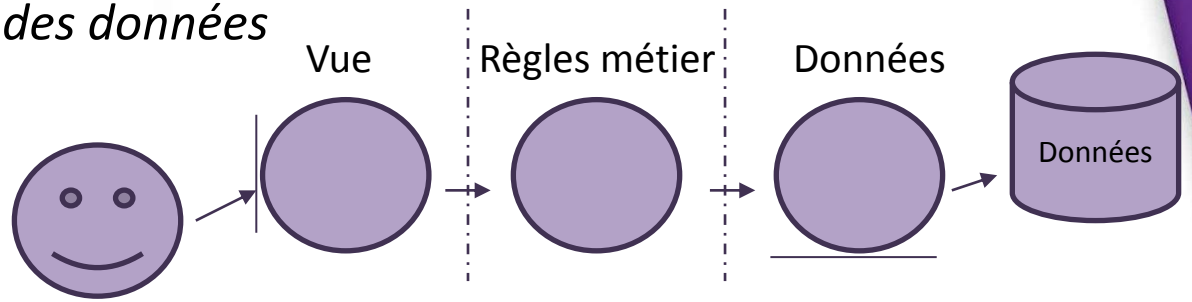
- Modèle *client lourd* et *client léger*

# Architecture multi niveaux



## Par expérience, il est préférable de développer des applications avec au minimum 3 couches

- La *vue* ou *couche présentation*
- La *couche des règles métier*
- La *couche des données*




## Idéalement : indépendance des niveaux (remplacement possible)

- = *couplage faible*

## Séparation des couches = maintenabilité et extensibilité

- Facilitation des changements de la vue ou de l'ajout de services ou de tests

# Applications graphiques

Microsoft C#

- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Concepts d'une interface graphique

Microsoft C#

## Pilotée par les *événements*

- Réponse aux requêtes de l'utilisateur en temps et dans l'ordre
- Dispositif de pointage (souris…)

## Programmation fondamentalement *orientée objet*

## Dans VS : outil RAD(1) facilitant l'agencement des contrôles

- Sélectionner un contrôle et le glisser sur la **Form**
- Modifier ses propriétés selon le besoin

## Les espaces de noms

- `System.Windows.Forms` pour les applications de bureau
- `System.Web.UI` pour les applications Web ASP.NET

(1)Rapid Application Development

# Les contrôles

Microsoft C#

## Les contrôles Windows Forms et ASP.NET sont très semblables

- Certains virtuellement identiques (`ListBox`)
- Certains propres aux Windows Forms (`Timer`)
- D'autres propres à ASP.NET (`LinkButton`)

## Toutefois…

## Les contrôles ASP.NET apparaissent dans un document HTML avec des balises (`<asp:ListBox ...>`)

- Le HTML pur est généré 'automagiquement" lors du rendu au navigateur avec de nombreux contrôles HTML plus du JavaScript

# Les propriétés de dessin



**Tous les contrôles ont des *propriétés* associées.**

**Espace de noms `System.Drawing`**

**Cela inclut notamment**

- La police de caractères
- La couleur du fond
- La couleur des caractères
- La visibilité…

**Propriété `(Name)` = nom logique du contrôle. Fixée pendant le développement dans VS en utilisant la *feuille de propriété* (*property sheet*)**

**La plupart des propriétés sont modifiables par programme**

```
button1.BackColor = Color.LightCoral;
```

# Applications graphiques

Microsoft C#

- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Contrôles container



## Avec les Windows Forms, une *Form* est un container sur lequel on peut placer d'autres contrôles

- "Fenêtre"
- Possède un *cadre*, une *barre de titre*, des contrôles *minimize* et *maximise*, etc.

## Autre container : le *panneau* (*panel*)

- Se place dans une Form ou dans un autre panneau, lui-même contenu dans une Form ou dans un autre panneau, lui-même…

# Ancrage et arrimage

Microsoft C#

**La taille et la position d'un contrôle dans un container peuvent être fixées de façon absolue en terme de coordonnées x et y**

- Peu pratique si une Form est redimensionnée par l'utilisateur

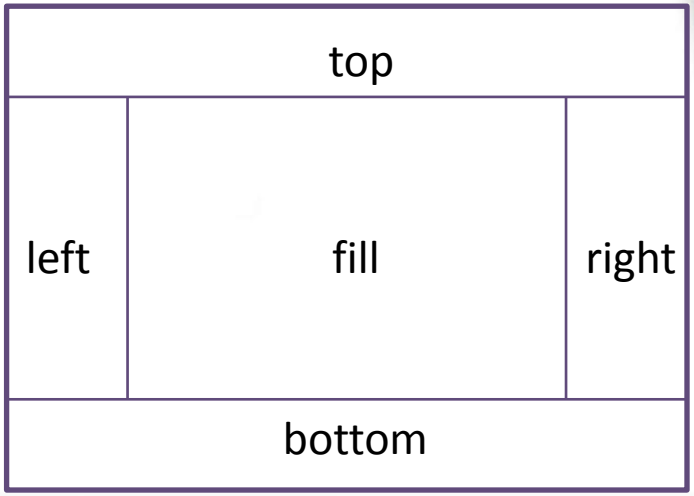
**L'*ancrage* pilote la position d'attachement du contrôle**

- Par exemple : top, left
- Les contrôles peuvent être étirés

**L'*arrimage* divise le container en 5 quadrants (voir figure)**

- Le contrôle est redimensionné pour correspondre au quadrant
- Classiquement, un panneau est placé dans chaque quadrant et les contrôles sont placés dans les panneaux

Quadrants d'arrimage



# Gestion des applications Windows Forms

Microsoft C#

## L'exécution des applications Windows Forms est contrôlée par la classe `Application`

- `Application.Run(... application ...)` lance le gestionnaire de messages pour que les événements soient distribués à l'application concernée
- `Application.Exit()` stoppe le gestionnaire de messages, ferme la fenêtre et termine l'exécution

## Ne pas confondre la classe `Application` des Windows Forms avec l'objet `Application` d'ASP.NET

- Avec ASP.NET, il est utilisé pour mémoriser des données, pas pour contrôler l'exécution d'un programme

# Applications graphiques

Microsoft C#

- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Interactions ASP.NET



Les interactions diffèrent entre une application de bureau et une page Web

En fonction de l'agencement effectué, du pur HTML et du JavaScript sont transmis au navigateur

Une interaction basée sur les événements est possible entre un utilisateur (via son navigateur) et l'application

# Agencement avec ASP.NET

Microsoft C#

## Une *Form* est une page Web. Son développement est similaire à celui d'une application de bureau

- Notions d'arrimage et d'ancrage inexistantes

## Possibilité d'éliminer le besoin de coder du HTML

- Séparation entre HTML et code C#
- Le développeur se concentre sur la logique en C#

## La page peut être visualisée en

- Vue conception
- Vue HTML
- Montrant le code C# (plus courant)

## Extension pour les pages Web ASP.NET :

- `nompage.aspx` pour la page elle-même
- `nompage.aspx.cs` pour le code C# sous-jacent

# La navigation (Web)



## Déplacement page par page

## Peut être effectué via un hyperlien

- L'URL(1) indique où l'utilisateur doit être redirigé

## Navigation possible à partir du code

- Le plus courant est : Response.Redirect(…URL …)
- Informe le navigateur d'aller chercher une page (l'URL spécifiée)
- L'URL est visible dans le navigateur

(1)Uniform Resource Locator

# Pré-requis ASP.NET



## Pour exécuter une application ASP.NET, il faut :

### Sur le serveur

- Microsoft IIS[(1)] (WWWPS[(2)]) sur une plate-forme Windows (ou une alternative appropriée sous Linux ou Unix)
- L'infrastructure Common Language .NET (qui inclut la bibliothèque de ASP.NET)

### Sur le client

- Un navigateur Web (browser) raisonnablement moderne

[(1)]Internet Information Services

[(2)]World Wide Web Publishing Service

# Applications graphiques



- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Traitement des événements



**Dessin d'une Form = apparence future de l'application (*maquette*) mais aucune logique de traitement (rien ne s'exécute si l'on clique sur un bouton par exemple)**

**Codage de la logique de l'application en C# = traitement des événements (interactions avec l'utilisateur)**

# A propos des événements

Microsoft C#

## Interaction d'un utilisateur avec un contrôle
## ➔ génération d'un événement

- Appui et relâchement d'un bouton, sélection d'un item dans une liste, chargement d'une Form…

## Un événement contient de l'information

- Le composant qui l'a généré et le type de l'événement

## ASP.NET : événements côté serveur au renvoi de la page

- Clic sur un bouton de soumission
- Déclenchés par JavaScript (possibilité de faire en sorte que les pages soient renvoyées automatiquement)

## Les événements sont traités par gestionnaire d'événements (event handler), mais…

- Délégation du traitement à une *méthode* que nous devons écrire

# Applications graphiques



- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Le code caché



**Visual Studio génère une grande partie du code pour nous (application Web ou bureau)**

- Reste à écrire les méthodes évènementielles (de traitement)

**Les méthodes événementielles et les définitions de contrôles sont placées dans des segments différents d'une classe `partial`**

- Dans des fichiers à part

# Le code caché (suite)

Microsoft C#

## Visual Studio génère un code ressemblant au suivant :

```
namespace SomeNamespace
{
    public partial class SomeForm : System.Windows.Forms.Form
    {
        ...
    }
}
```

Héritage pour les Windows Forms

```
namespace SomeNamespace
{
    public partial class SomePage : System.Web.UI.Page
    {
        ...
    }
}
```

Héritage pour une page Web

# Ajout d'un contrôle



## Au placement d'un contrôle et de la valorisation de ses propriétés, Visual Studio génère le code pour :

- Créer une instance du contrôle
- Initialiser ses propriétés

## Exemple :

```
partial class SomeForm
{
    ...autre code ...
    void InitializeComponent()
    {
        ...initialisation des autres contrôles et propriétés...
        button1 = new Button();
        button1.Text = "Aide";
        button1.BackColor = Color.LightCoral;
    }
}
```

# Ajout d'un traitement d'événement

Microsoft C#

**Lors d'un double-clique en mode design, Visual Studio ajoute un gestionnaire d'événements dans le code caché de la manière suivante :**

- Il crée un gestionnaire d'événement (`System.EventHandler`)
- Il transmet "l'adresse" de notre méthode événementielle au gestionnaire d'événements : c'est un *délégué* (*delegate*)
- Il enregistre le gestionnaire d'événements en le connectant à la propriété du contrôle qui générera cet événement

# Ajout d'un traitement d'événement (suite)

Microsoft C#

## Exemple (à noter l'utilisation de l'opérateur +=):

```
...idem exemple précédent...
void InitializeComponent()
{
    ...création des contrôles et autre code...
    Button1.click += new EventHandler(button1_Click);
}
```

Enregistrement du gestionnaire

Création du gestionnaire d'événements

Délégué vers notre méthode événementielle

```
protected void button1_Click(object sender, EventArgs e)
{
    this.Text = "Pas d'aide pour vous"; // Dire la vérité
    button1.text = 'Bof...";            // Reconfigurer
}
```

## À quoi va ressembler l'affichage après un clic sur le bouton ?

# Applications graphiques

Microsoft C#

- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Les boîtes de message

Microsoft C#

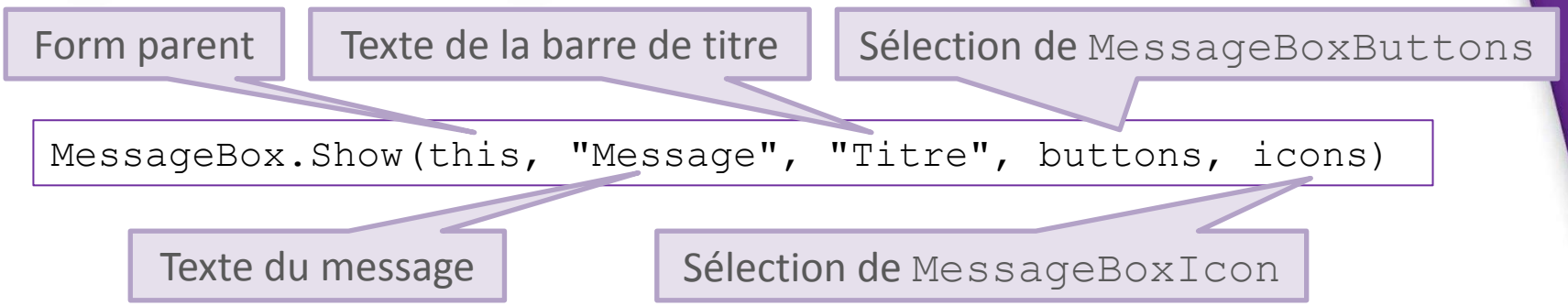
## Ou *boîte de dialogue* : interface particulière

- Apparaît généralement au-dessus d'une fenêtre principale

## Possède une méthode `static` nommée Show pour rendre son usage aisé

- Les méthodes statiques peuvent être invoquées sans créer d'instance
- Retourne une énumération `DialogResult`

## Nombreuses surcharges disponibles, la plus commune est :

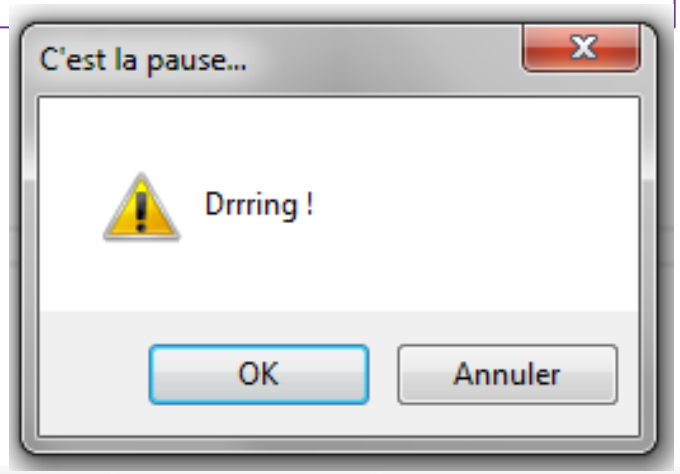


# Exemple de MessageBox

Microsoft C#

## Une alerte…

```
void UneMethode()
{
    ...instructions...
    DialogResult dr = MessageBox.Show (this,
                                       "Drrring !",
                                       "C'est la pause...",
                                       MessageBoxButtons.OKCancel,
                                       MessageBoxIcon.Warning);
    if(dr == DialogResult.Cancel) return;
    else this.Text = "Get out !!!";
    ...autres instructions...
}
```



# Applications graphiques

Microsoft C#

- Évolution des interfaces utilisateur
- Contrôles et propriétés
- Agencement des IHM de bureau
- Agencement des IHM Web
- Événements
- Code de traitement
- Boîtes de message
- Exercice 4
- Questions de révision

# Questions de révision

Microsoft C#

## Qu'est-ce qu'un délégué ?

## Qu'est-ce qu'un événement Windows Forms ou ASP.NET ?

## Qu'est-ce qu'un DialogResult ?

# Définition de types de données utilisateur

# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision

# Rappels de P.O.O.



## Style de programmation

- Ne nécessite pas obligatoirement un langage O.O. mais…

## Se résume essentiellement à l' "abstraction de données"

- i.e. : définir de nouveaux types de données correspondant en général à des entités du monde réel

## Les objets du monde réel ont trois caractéristiques importantes :

- Un état
- Un comportement
- Une identité

# Rappels de P.O.O.

Microsoft C#

## C# fournit un support pour les trois concepts principaux de la programmation orientée objet

- *Encapsulation* : masquer l'état d'une classe derrière son comportement
- *Héritage* : définir une classe qui étend une classe précédemment définie
- *Polymorphisme* : accéder à des méthodes de classes dérivées en utilisant l'interface de la classe dont elles dérivent

## Quels sont les avantages d'une bonne P.O.O. ?

# Définir ses propres types de données

Microsoft C#

## C# possède deux constructions semblables pour définir des types de données (comme un emporte-pièce)

- Une *classe* si le type va être utilisé par référence
- Une *structure* si le type va être utilisé par valeur (≠ structure C)

## Un objet est une entité en mémoire

- Les objets de types classes sont créés sur le tas (avec **`new`**)
- Les objets de types structures sont en général créés sur la pile

# Création de structures

Microsoft C#

## Moins employées que les classes, utilisées pour :

- Enregistrements en mémoire passés comme un tout
- Petits types mathématiques dotés d'opérateurs symboliques

```
namespace ComplexMath
{
    public struct complex
    {
        public double real;
        public double imag;
        ...
    }
}
```

```
namespace ComplexMath
{
    public class Program
    {
        public static void Main()
        complex c1;
        c1.real = 1.234;
        c1.imag = 5.678;
        Calcul(c1);
        ...
    }
}
```

Création sur la pile

Transmis par valeur comme un élément unique

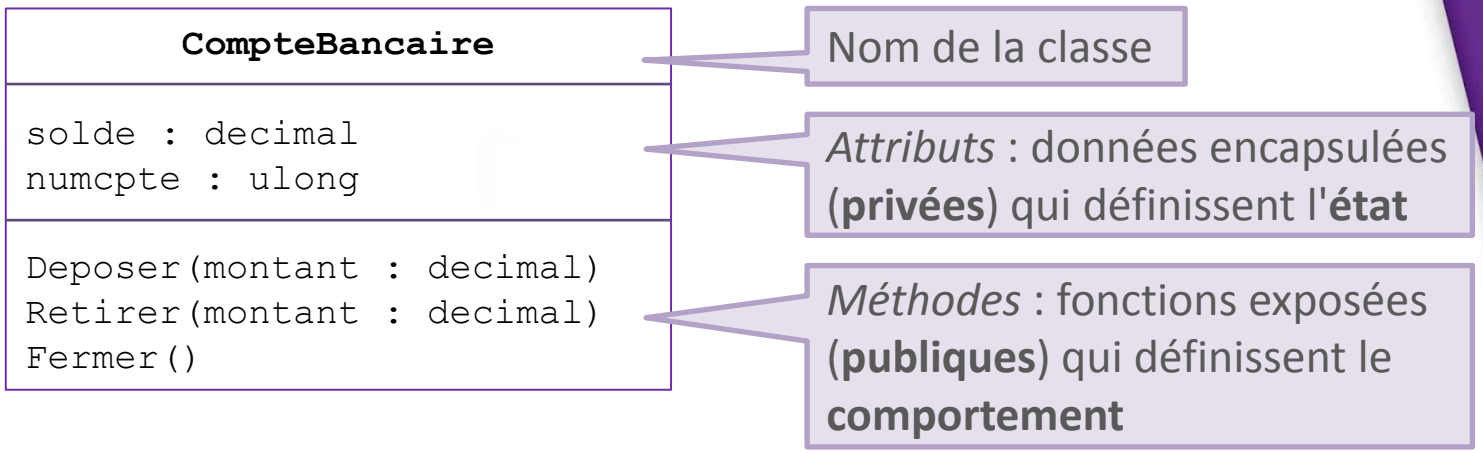
## Notre étude portera essentiellement sur les classes

- Certaines fonctionnalités sont valables pour les structures

# UML

Microsoft C#

## Notation UML pour une classe. Exemple :




## L'*interface* d'une classe est la liste de toutes ses méthodes

## Les règles métier sont dans les méthodes

- L'encapsulation force l'utilisateur à respecter les règles métier de l'interface

# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision

# Définition de classes en C#

Microsoft C#

## La forme générale d'une définition de classe en C# est la suivante :

```
[Modifiers] class ClassName [: [BaseClass[,]] [Interfaces[,]]]
{
    ... attributs privés ...
    ... méthodes et propriétés publiques ...
}
```

**`Modifiers` : informent sur la nature de la classe**

- Public : disponible de n'importe quel assembly
- Internal (par défaut) : disponible au sein de son assembly

**Les *méthodes* et *propriétés publiques* sont accessibles de partout (selon visibilité de la classe)**

**Les *Attributs privés* sont accessibles au sein de la classe**

**`BaseClass` : classe dont on hérite**

**`Interfaces` : implémentées par la classe**

# La classe CompteBancaire

Microsoft C#

## Code C# de la classe CompteBancaire (exemple UML)

```
using System;
namespace Banque
{
    public class CompteBancaire
    {
        private decimal solde;
        private ulong numcpte;

        public void Deposer(decimal montant)
        {
            solde += montant;
        }

        public void Retirer(decimal montant)
        {
            if(solde >= montant)
            {
                solde -= montant;
            }
            else throw new InvalidArgumentException("zut");
        }
        ... autres méthodes ...
    }
}
```

Aucune classe de base fournie ➔ `CompteBancaire` hérite de `object`

Attributs privés

**Cette classe ne contient que des champs et des méthodes**

Toutes les méthodes peuvent accéder aux attributs privés

# Utilisation de la classe CompteBancaire

Microsoft C#

## Une fois définie, la classe peut être instanciée

```
using System;
namespace Banque
{
    public class Program
    {
        public static void Main()
        {
            CompteBancaire c1 = new CompteBancaire();
            CompteBancaire c2 = new CompteBancaire();

            c1.Deposer(100.00m);
            c2.Deposer(500.00m);
            c2.Retirer(50.00m);
            c1.Deposer(50.00m);

            // c2.solde = 1000.00m;
        }
    }
}
```

Création d'instances (objets) de la classe CompteBancaire

Accès aux données en appelant des méthodes

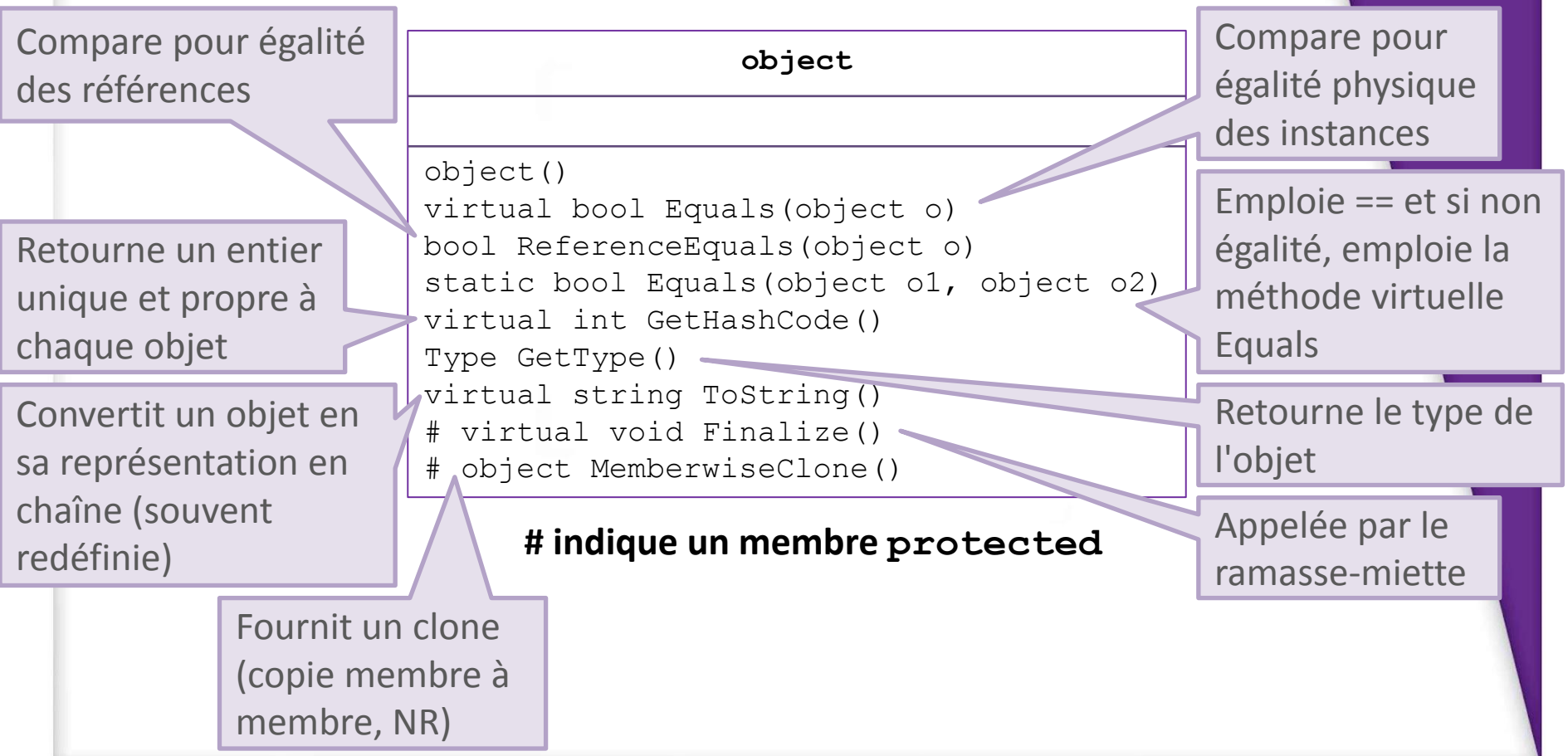
Erreur : violation de l'encapsulation

# La classe `object`

Microsoft C#

## En C#, si une classe n'hérite pas explicitement d'une autre, elle hérite implicitement de `object`

- `object`: alias reconnu par C# pour la classe `Object` de .NET




**# indique un membre `protected`**

# Redéfinir `ToString()`

Microsoft C#

## Il est courant d'implémenter une méthode `ToString` dans une classe (conversion en chaîne de caractères)

```
using System;
namespace Banque
{
    public class CompteBancaire
    {
        private decimal solde;
        private ulong numcpte;

        ... comme avant ...

        public override string ToString()
        {
            return "Compte : " + numcpte + ", solde : " + solde;
        }
    }
}
```

`override` indique au compilateur que pour un `CompteBancaire`, cette méthode se substitue à celle de la classe `object`

# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision

# Les constructeurs

Microsoft C#

## Nom identique à celui de la classe

## Pas de type de retour (pas même `void`)

## Objectif : initialiser les attributs du nouvel objet

## Constructeur par défaut (unique) :

- Aucun paramètre, appelé sans valeurs initiales
- Appelle les constructeurs par défaut pour chaque donnée membre (`0` pour les numériques, `false` pour les booléens et `null` pour les références) et le constructeur par défaut de la classe de base

# Surcharge des constructeurs

Microsoft C#

**Constructeurs par valeurs (plusieurs sont possibles) :**

- Initialisent un ou plusieurs attributs

**Constructeur par recopie (unique) :**

- Recopie les valeurs des attributs d'une instance passée en argument

**Une classe dont seuls des constructeurs par valeur ou par recopie ont été implémentés impose au code client de fournir des valeurs d'initialisation pour les attributs d'instance**

**Un attribut déclaré `readonly` ne peut être affecté que dans un constructeur ou lors de sa déclaration**

# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision

# La référence `this`

Microsoft C#

**Quand une méthode est appelée sur un objet, une référence spéciale, nommée `this`, est positionnée pour pointer sur cet objet**

**`this` est un mot-clé qui n'a de sens que dans le corps des méthodes d'instance**

**Utilisation de `this` :**

- Seule possibilité pour passer l'instance courante à une méthode d'une autre classe
- Permet de lever une ambiguïté entre un paramètre d'une méthode et un attribut de la classe auquel elle appartient

# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision

# Champs publics ?



**Pour la qualité de l'encapsulation, les champs publics sont indésirables**

**Dans les langages OO traditionnels, une classe a souvent des méthodes *get** et *set****

- Nommées accesseurs et mutateurs
- Usage rapidement pénible :

```
truc.SetValeur(truc.GetValeur() + 100.00);
```

- Il serait plus simple de pouvoir écrire :

```
truc.Valeur += 100.00;
```

**Les *propriétés* permettent un usage simple sans sacrifier l'encapsulation**

# Implémentation des propriétés

Microsoft C#

## Syntaxe générale des propriétés en C#:

```
visibilité type nom
{
    visibilité get
    {
        ...logique...
        return field;
    }
    visibilité set
    {
        ...logique...
        field = value;
    }
}
```

- *visibilité* est le type d'accès pour la propriété : `public`, `protected`
- *type* est le type de donnée de la propriété (pas nécessairement le type de l'attribut)
- *field* est l'attribut apparemment manipulé
- `value` est un mot-clé représentant la valeur du type fournie

# Implémentation des propriétés (exemple)

Microsoft C#

```
using System;
namespace Banque
{
    public class CompteBancaire
    {
        private decimal solde;
        private string titulaire;
        ...autres champs et méthodes...
        public decimal Solde { get { return solde; } }
        public string Titulaire
        {
            set
            {
                if(value.Length != 0)
                    titulaire = value;
                else
                    throw new Exception("Titulaire obligatoire !");
            }
            get { return titulaire; }
        }
    }
}
```

Un attribut commence par une minuscule

Une propriété commence par une majuscule

```
c2.Titulaire = "John Doe";
Console.WriteLine("Solde : " + c2.Solde);
```

# A propos des propriétés



**Les propriétés n'ont pas besoin de correspondre à un attribut**

**Si `set` n'est pas fournie, la propriété est read-only**

- write-only pour `get`, occasionnellement utile

**Les propriétés peuvent être `static`**

**Convention de nommage Microsoft : une propriété commence par une majuscule**

**Ex : les composants graphiques visuels de Visual Studio (fenêtre des propriétés !)**

# Lourdeur des propriétés

Microsoft C#

## Difficulté de relire des classes en raison du code des propriétés qui ne représentent souvent qu'une simple encapsulation

## Utilisation des *propriétés auto-implémentées*

```
public class CompteBancaire
{
    public decimal Solde { get; private set; };
    public ulong Numcpte { get; private set; };
    public decimal Taux { get; set; };

    public void Deposer(decimal montant)
    {
        Solde += montant;
    }
}
```

`private` indique get uniquement

Les méthodes référencent les propriétés auto-implémentées

## Syntaxe alternative

- A transformer si une logique supplémentaire doit être encapsulée

# Propriétés vs. méthodes

Microsoft C#

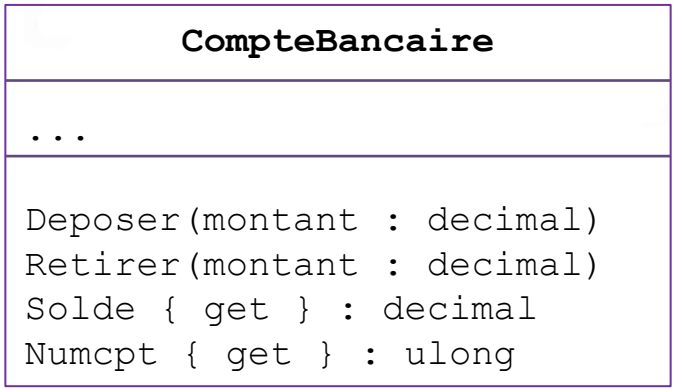
## Quand utiliser une propriété ou une méthode ?

- Les propriétés servent pour des accès simples aux données
- Les méthodes implémentent la logique métier

## Dans la classe CompteBancaire :

- `Deposer` et `Retirer` sont des méthodes
- `Solde` pourrait être une propriété en lecture seule

## Pas de syntaxe UML pour les propriétés : nous utiliserons donc `{ get, set }` dans le compartiment des méthodes



# Définition de types de données utilisateur

Microsoft C#

- Philosophie orientée objet
- Définition de classes C#
- Constructeurs
- La référence `this`
- Exercice 4.1
- Propriétés
- Exercice 4.2
- Héritage
- Exercice 4.3
- Questions de révision