



# C#, DÉVELOPPER EN .NET

Avec Visual Studio 2010/2012



# Benoît CHAUVET

1

- ❑ 10 ans d'expérience sur la plateforme .NET
- ❑ Ingénierie web et gestion de projets
  - Eurosport.com : Mise en place de la communauté et du player Eurosport (VOD et Streaming)
  - Externis : Gestion de projets B2B en Application Service Provider pour les process logistique et animation de grands comptes (Kraft, Unilever, Nestlé...)
- ❑ Formation et conseil
  - Gestion de projet (Méthodes agiles, Scrum)
  - Conception (UML, concepts objet)
  - Programmation .NET (C#, ASP.NET)
- ❑ [contact@benoitchauvet.com](mailto:contact@benoitchauvet.com)

# Table des matières

|    |                                    |    |
|----|------------------------------------|----|
| 1. | La plateforme .NET                 | 5  |
| 1. | Présentation                       | 6  |
| 2. | Versions du Framework              | 11 |
| 3. | Assembly                           | 16 |
| 4. | Namespace                          | 18 |
| 2. | L'environnement Visual Studio 2010 | 22 |
| 1. | Présentation                       | 23 |
| 2. | Modules de Visual Studio 2010      | 28 |
| 3. | Projets et solutions               | 40 |
| 3. | Syntaxe de base                    | 44 |
| 1. | Variables et opérateurs            | 45 |
| 2. | Structures de contrôle             | 65 |
| 3. | Méthodes                           | 71 |
| 4. | Gestion des exceptions             | 77 |
| 1. | Les exceptions                     | 78 |
| 2. | Création et déclenchement          | 83 |

|    |   |     |
|----|---|-----|
| 5. | Concepts objets en C#                       | 85  |
| 1. | Introduction                                | 86  |
| 2. | Classe et objet                             | 88  |
| 3. | Héritage et polymorphisme                   | 110 |
| 4. | Interfaces                                  | 125 |
| 5. | Délégués                                    | 132 |
| 6. | Evénements                                  | 136 |
| 7. | Attributs                                   | 146 |
| 6. | Objets et classes de base du framework .NET | 147 |
| 1. | Classes utilitaires                         | 148 |
| 2. | Collections                                 | 150 |
| 3. | Généricité                                  | 155 |
| 4. | Enumérations                                | 165 |
| 5. | Indexeurs                                   | 169 |
| 6. | Introduction à LINQ                         | 177 |

|     |                                     |     |
|-----|-------------------------------------|-----|
| 7.  | Accès aux données                   | 180 |
| 7.  | ADO.NET                             | 181 |
| 8.  | Mapping Objet-Relationnel           | 204 |
| 9.  | LINQ to SQL                         | 205 |
| 10. | LINQ to Entities                    | 209 |
| 8.  | Les différents types d'applications | 216 |
| 7.  | Applications Windows                | 217 |
| 8.  | Applications Web ASP.NET            | 228 |
| 9.  | Services Web ASP.NET                | 242 |

5

# La plateforme .net

Présentation de la plateforme .net, framework et architecture

# Présentation

6

- ❑ Développé à la fin des années 90 par Microsoft
- ❑ Première version sortie en 2002
- ❑ Destiné à remplacer les technologies précédentes (COM, DCOM, ActiveX)
- ❑ Reprend les principes de Java / J2EE
- ❑ Adaptation aux technologies web et systèmes distribués

# Caractéristiques

7

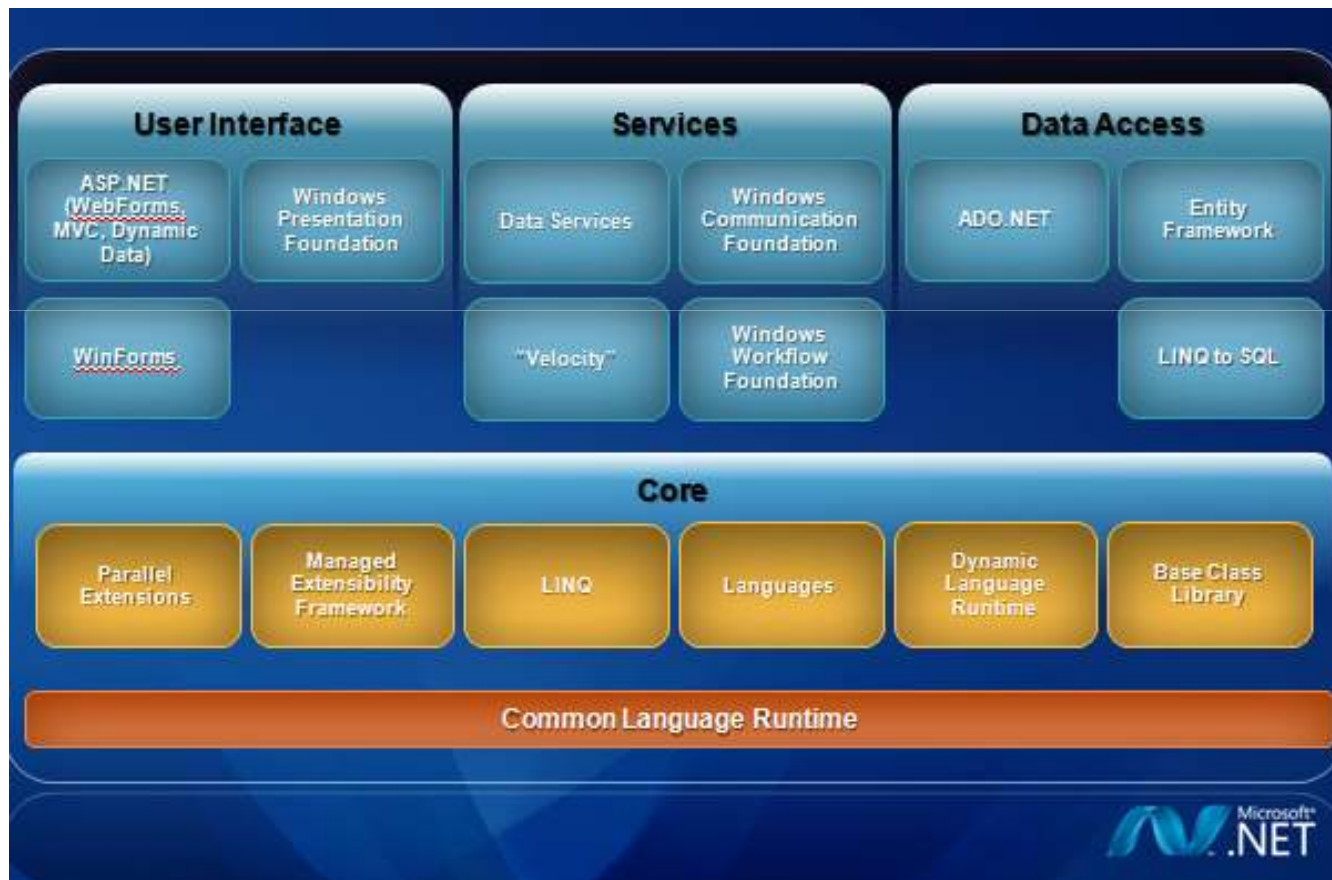
- ❑ Indépendant du système (Windows, Unix, ...)
- ❑ Basé sur la norme Common Language Infrastructure (CLI) : indépendante du langage
  - Plusieurs langages disponibles : C#, VB.net, F#...
  - Common Language Specification (CLS)
  - Common Type System (CTS)
  - Génération de code intermédiaire : Common Intermediate Language (CIL, anciennement MSIL)
- ❑ Exécution du CIL par une machine virtuelle : Common Language Runtime (CLR)
- ❑ Compilation Just In Time (JIT) par la CLR, à l'exécution (Runtime)



# Plateforme .NET

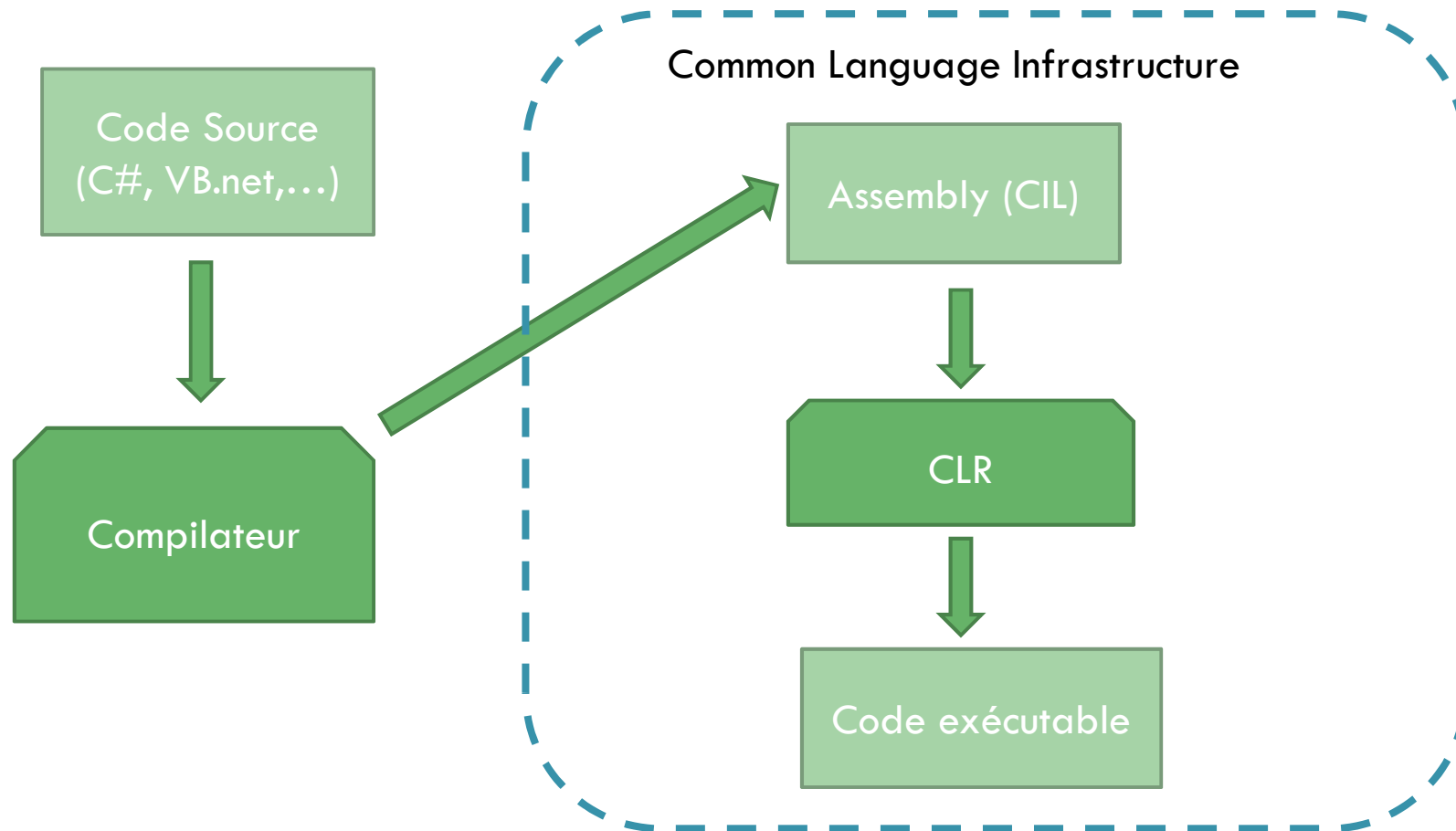
8

## Le Framework .net



# Plateforme .NET

9



(c) Benoit Chauvet 2013

# CLR et CTS

10

- ❑ Common Language Runtime (CLR)
  - Chargement des assemblies
  - Compilation Just In Time (JIT)
  - Exécution
  - Gestion de la mémoire
  - Gestion des threads
  - Gestion de la sécurité (bac à sable)
- ❑ Common Type Specification (CTS)
  - Règles de définition, utilisation et gestion des types par le CLR

# Versions du framework

11

| Version | Date de sortie | Outil de développement        | C#  |
|---------|----------------|-------------------------------|-----|
| 1.0     | 05/01/2002     | Visual Studio .NET (7.0)      | 1.0 |
| 1.1     | 01/04/2003     | Visual Studio .NET 2003 (7.1) |     |
| 2.0     | 07/11/2005     | Visual Studio 2005 (8.0)      | 2.0 |
| 3.0     | 06/11/2006     | Expression Blend              |     |
| 3.5     | 19/11/2007     | Visual Studio 2008 (9.0)      | 3.0 |
| 4.0     | 12/04/2010     | Visual Studio 2010 (10)       | 4.0 |
| 4.5     | 15/08/2012     | Visual Studio 2012 (11)       | 5   |

# Versions du framework

12

## ❑ Framework 1.0 : Version initiale

- Bibliothèque de classes
- CLR
- Winforms
- ADO.NET

## ❑ Framework 1.1

- Intégration ASP.NET, accès aux données ODBC et Oracle
- .NET Compact Framework (Windows CE)

## ❑ Framework 2.0

- Support des Generics
- Améliorations d'ASP.NET
- .NET Micro Framework

# Versions du framework

13

## ❑ Framework 3.0

- Windows Presentation Foundation (WPF) : interfaces graphiques vectorielles (XAML), 3D (Direct3D)
- Windows Communication Foundation (WCF) : messagerie orientée services (regroupe COM+, Web Services et .NET Remoting)
- Windows Workflow Foundation (WF) : transactions / tâches automatisées à l'aide de workflows
- Windows CardSpace : gestion sécurisée des identités

## ❑ Framework 3.5

- Language Integrated Query (LINQ) et Lambda expressions
- Améliorations ADO.NET
- Intégration d'ASP.NET Ajax
- Améliorations WCF et WF
- ASP.NET : intégration de MVC et améliorations

# Versions du framework

14

## ❑ Framework 4.0

- Entity Framework (accès aux données et modélisation)
- Améliorations ASP.NET, WCF, WPF et WF
- Nouveau CLR (parallélisme, améliorations réseau)

## ❑ Framework 4.5

- Intégration METRO (style Windows 8)
- Support HTML 5
- MVC 4
- Managed Extensibility Framework (MEF)
- Améliorations WPF

# Compatibilité

15

- ❑ Compatibilité descendante :
  - Applications écrites dans une version antérieure du framework peuvent s'exécuter sur une version plus récente.
- ❑ Compatibilité ascendante :
  - Applications écrites dans une version plus récente du framework peuvent s'exécuter sur des versions antérieures, à condition de ne pas utiliser des nouveautés inconnues.
- ❑ Exécution côte à côte : cohabitation de versions différentes d'une même application et de plusieurs versions de runtime sur un ordinateur (Assemblies avec attribution de noms forts)



# Assembly

16

- ❑ Un Assembly est **une unité de déploiement indivisible** (Fichier .exe ou .dll)
- ❑ Il se caractérise par son identité (propriétés de l'assembly)
  - un nom
  - une version
  - un identificateur de culture
  - une clé publique
- ❑ Il contient:
  - la liste de l'ensemble des fichiers (exe, dll, données, images, ressources)
  - les méta données (informations descriptives des Types et Classes publiques)
  - L'énumération des autres Assembly dont l'application dépend et leurs dépendances.
  - l'ensemble des autorisations requises pour que l'assembly fonctionne correctement.

# Global Assembly Cache (GAC)

17

- ❑ Stocke les assemblies partagés entre plusieurs applications
- ❑ Les assemblies doivent être dotés d'un nom fort avant de pouvoir être placés dans le GAC :
  - Nom textuel, numéro de version et informations de culture
  - Clé publique et signature numérique
- ❑ ATTENTION : Nouveau GAC à partir du Framework 4.0
  - Ancien : `C:\Windows\assembly`
  - Nouveau : `C:\Windows\Microsoft.NET\assembly`

# Namespace

18

- ❑ Les namespaces sont une organisation logique des composants d'une assembly.
- ❑ Séparation des noms par des points, de manière hiérarchique.
- ❑ Exemple :
  - System.Collections.Generic
  - System.Collections.Specialized
  - MonProjet.DataAccess
  - MonProjet.Business
  - MonProjet.Presentation.Formulaires
  - MonProjet.Presentation.Controles

# Namespace

19

- ❑ On peut appeler un composant d'un namespace via son nom pleinement qualifié :

```
System.IO.StreamWriter sw;
```

```
System.Collections.Specialized.OrderedDictionary od;
```

- ❑ Pour une syntaxe plus réduite, on peut utiliser mot clé `using` au début du fichier de code qui utilise les composants :

```
Using System.Collections.Specialized;
```

```
Using System.IO;
```

```
...
```

```
OrderedDictionary od;
```

```
StreamWriter sw;
```

# Namespace

20

- ❑ Il est possible d'avoir des classes portant le même nom dans des namespace différents
- ❑ Ex :  
    System.Windows.Forms.Label  
    System.Web.UI.Label
- ❑ Pour utiliser ces classes dans le même fichier source avec leur nom court (« Label »), il faut utiliser un alias :  
    using winCtrl = System.Windows.Forms;  
    using webCtrl = System.Web.UI;  
    ...  
    winCtrl.Label lbl1;  
    webCtrl.Label lbl2;

# Namespace

21

## ❑ Définition d'un namespace :

```
namespace MonProjet.Data
{
    namespace People
    {
        class Personne
        {
        }
    }
}
```

MonProjet.Data.People.Personne p;

## ❑ Namespace par défaut : propriétés du projet

Application

Configuration: N/A Platform: N/A

Assembly name: ConsoleApplication3

Default namespace: ConsoleApplication3

Target framework: .NET Framework 4.5

Output type: Console Application

# L'environnement Visual Studio

Présentation de Visual Studio 2010

Ecriture de code C#

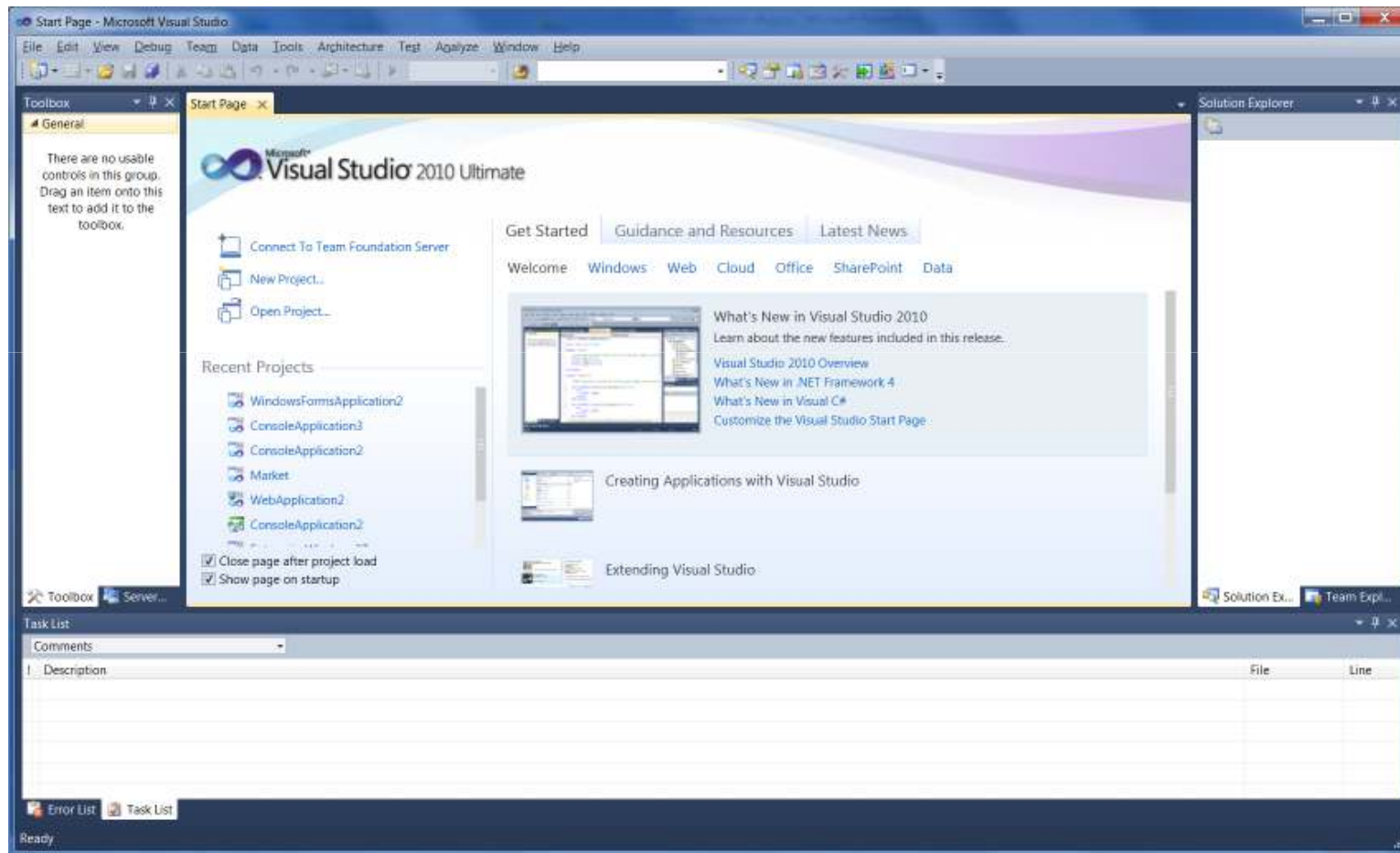
Débogage

Principaux outils

Projets et solutions

# Visual Studio 2010

23



(c) Benoit Chauvet 2013



# Visual Studio 2010

24

- ❑ IDE (Integrated Development Environment)
- ❑ Editeur de code avancé
  - Coloration syntaxique
  - Détection des erreurs à la volée
  - Intellisense
  - Outils de refactoring, génération de code
  - Editeur Wysiwyg pour les interfaces utilisateurs
- ❑ Multi fenêtré, paramétrage complet de l'affichage

# Visual Studio 2010

25

- ❑ Outils de débogage (pile des appels, espion, points d'arrêt...)
- ❑ Explorateur de serveurs intégré (Bases de données)
- ❑ Intégration de tests unitaires, tests d'interface
- ❑ Gestionnaire d'extensions
- ❑ Outils spécifiques ASP.net
  - Serveur web intégré
  - Outils de débogage javascript
  - Gestionnaire de publication (client FTP intégré)

# Versions complètes

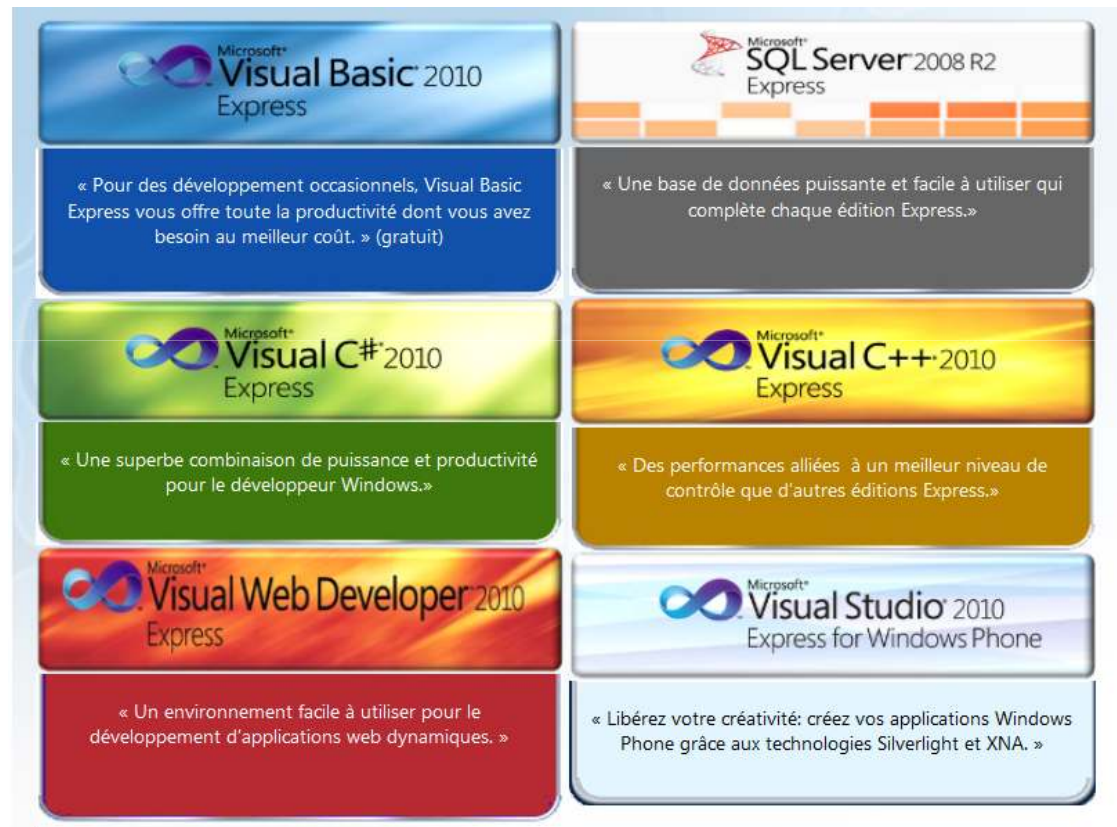
26

- ❑ **Visual Studio 2010 Professional** : la version de base de Visual Studio permettant aux développeurs de développer leurs applications web, windows, office, etc. (disponible sans abonnement MSDN).
- ❑ **Visual Studio 2010 Professional avec MSDN** : la version de base de Visual Studio permettant aux développeurs de développer leurs applications web, windows, office, etc. (avec abonnement MSDN).
- ❑ **Visual Studio 2010 Premium avec MSDN** : la version rassemblant les outils dédiés aux différents axes d'un projet : architecture, base de données, développement et test (avec abonnement MSDN).
- ❑ **Visual Studio 2010 Ultimate avec MSDN** : la suite étendue pour la gestion du cycle de vie des applications pour assurer une qualité accrue du design jusqu'au déploiement (avec abonnement MSDN).

# Visual Studio Express

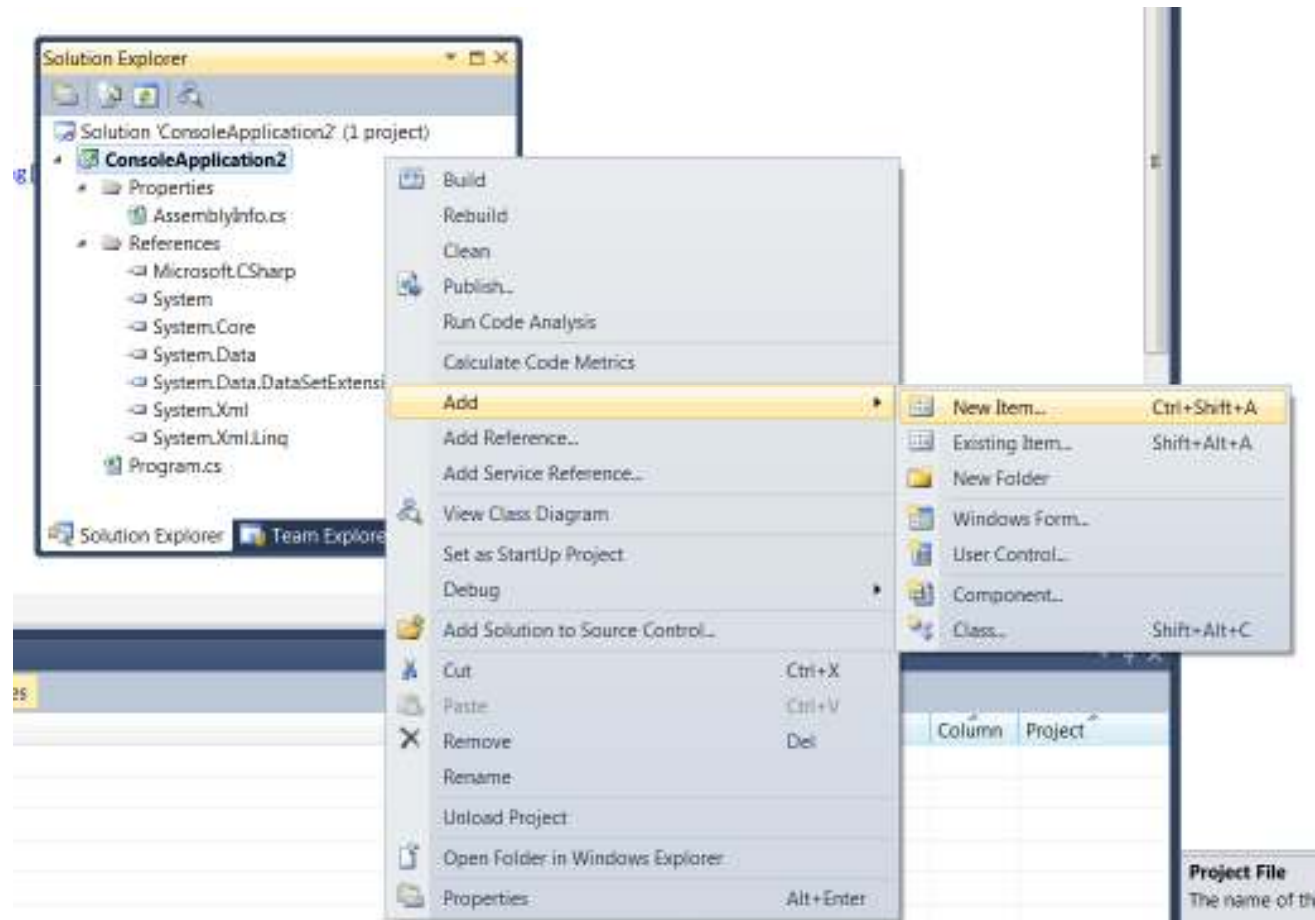
27

- ❑ Versions gratuites
- ❑ Spécifiques à chaque type d'applications (Web, Windows, Mobile)
- ❑ SQL Server express pour les bases de données
- ❑ <http://msdn.microsoft.com/fr-fr/express/aa975050.aspx>



# L'explorateur de solution

28

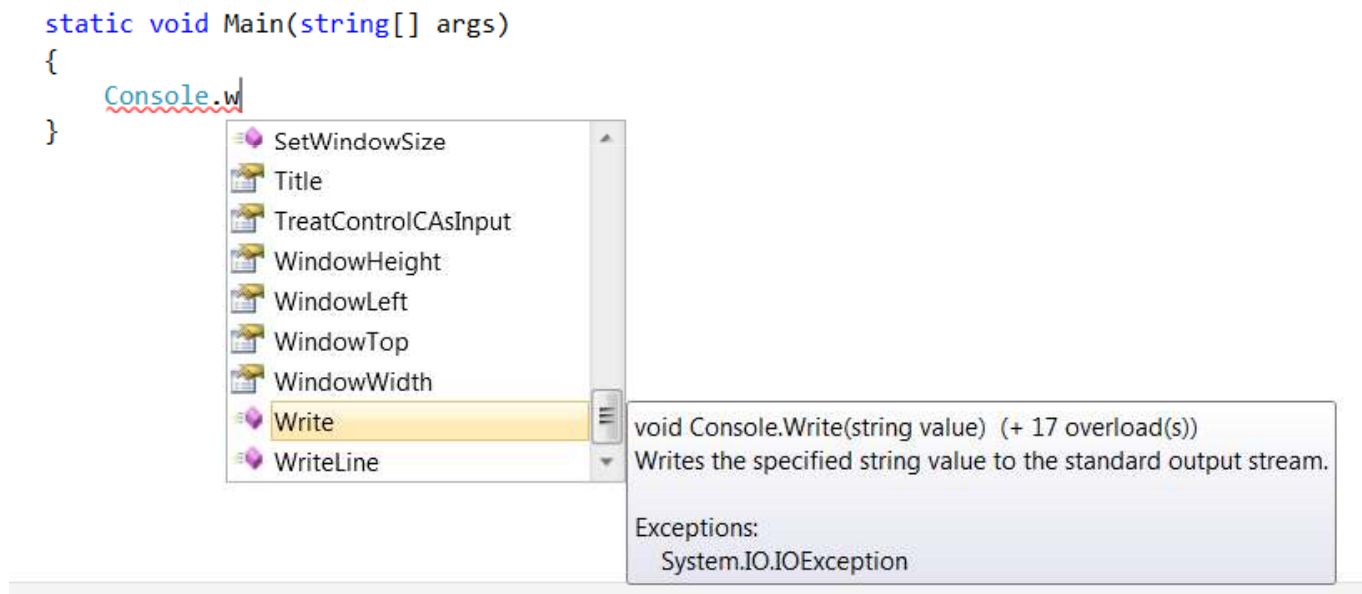


(c) Benoit Chauvet 2013

# Ecriture de code C#

29

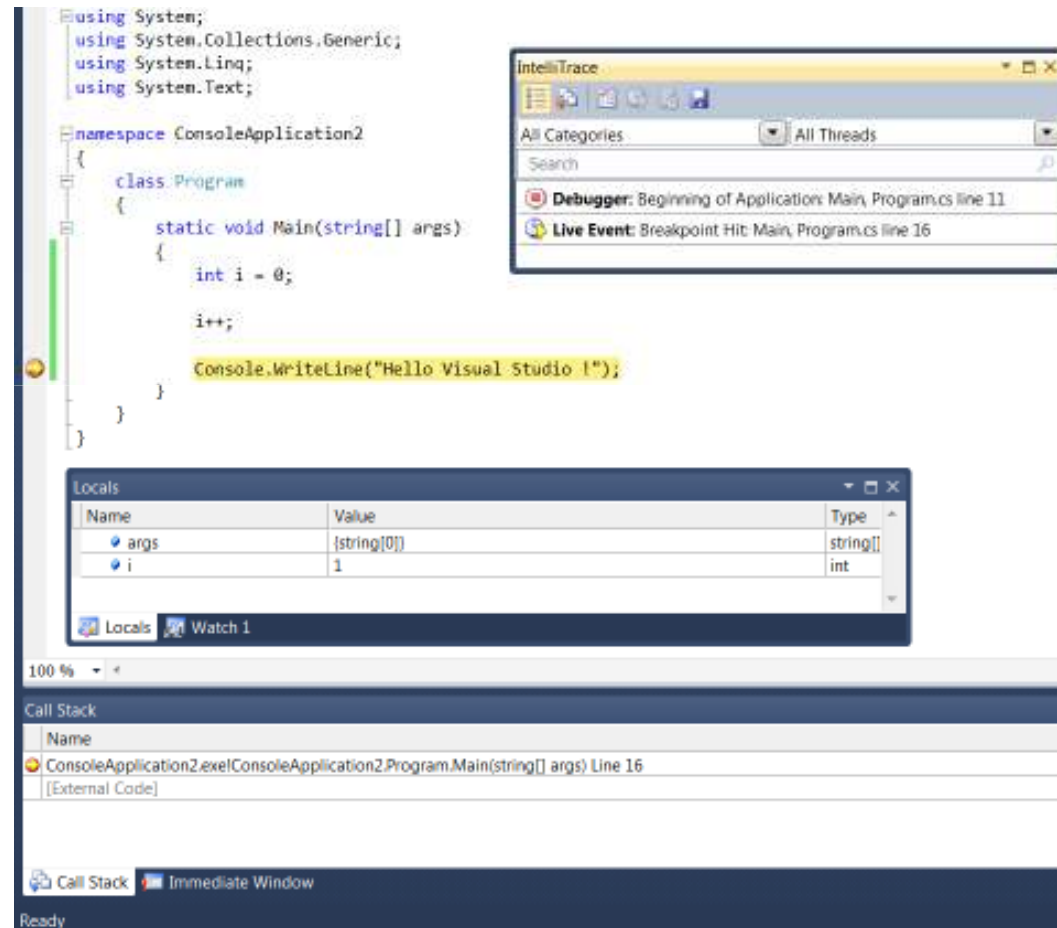
- ❑ IntelliSense permettant l'auto-complétion (touche TAB) et affiche le descriptif des éléments.



# Débogage

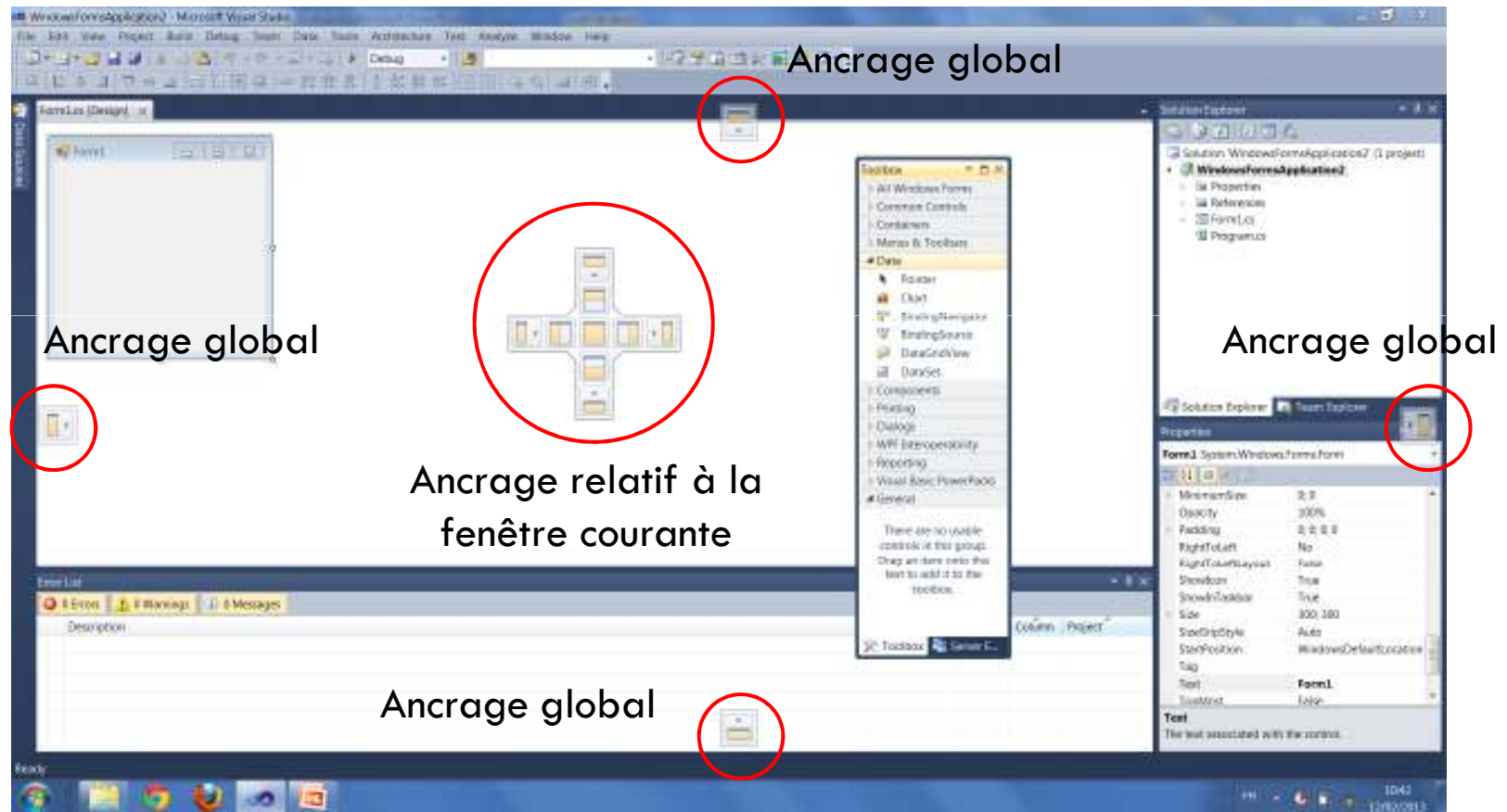
30

- ❑ Mise en place de points d'arrêt en mode DEBUG
- ❑ Visualisation de l'état des variables



# Ancrage des fenêtres

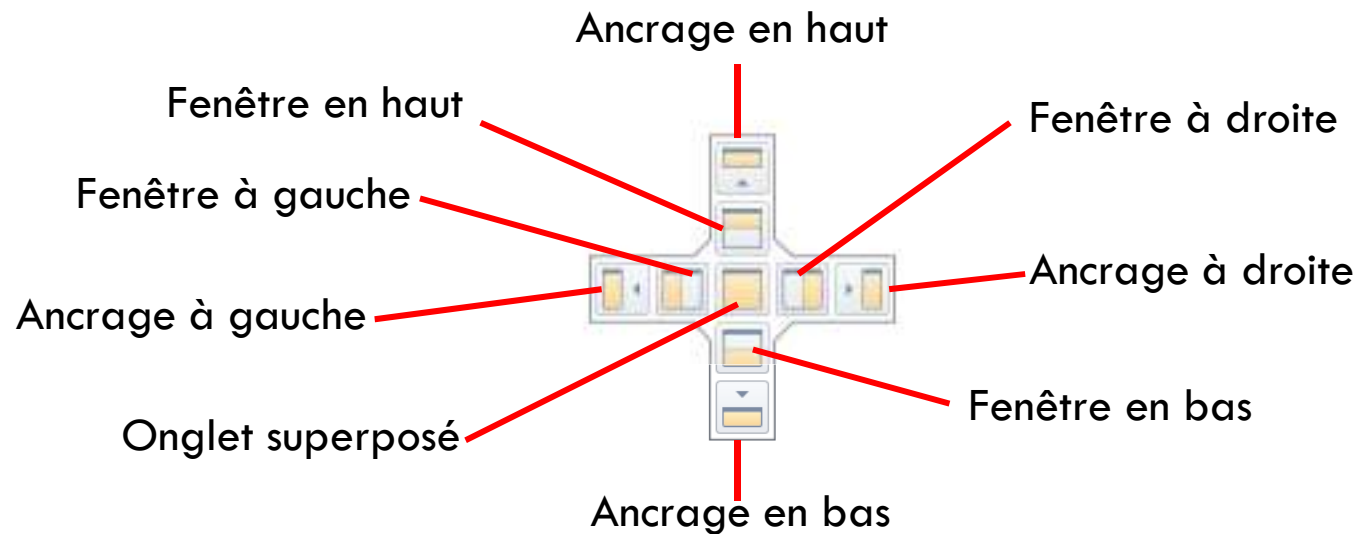
31





# Ancrage des fenêtres

32



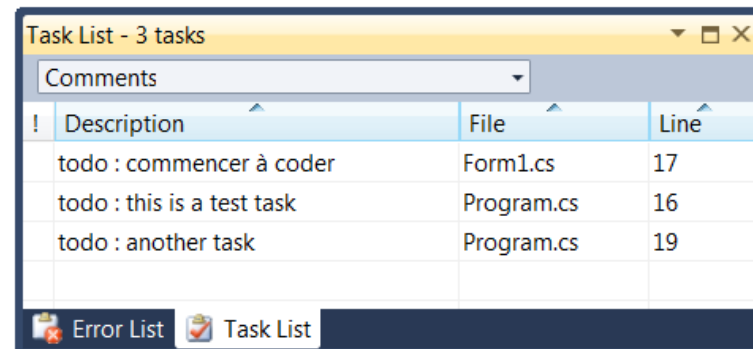
Bloquer / Débloquer une fenêtre ancrée

# Liste des tâches

33

- ❑ **Commentaires** : Les commentaires commençant par « todo » apparaissent automatiquement dans la liste des tâches de type « comment »
- ❑ La liste des tâches indique le fichier et la ligne concernés
- ❑ Double clic sur la tâche pour accéder au code

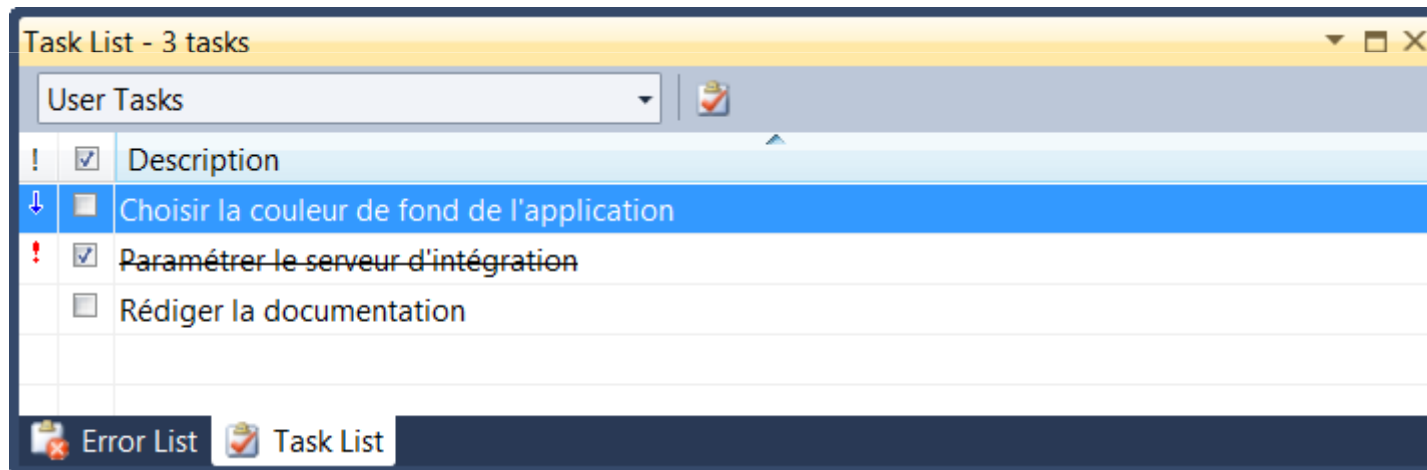
```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    // todo : this is a test task
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    // todo : another task
    Application.Run(new Form1());
}
```



# Liste des tâches

34

- ❑ Tâches utilisateurs : Saisie manuelle de tâches
- ❑ Possibilité d'affecter une priorité (Haute, Normale ou Basse)
- ❑ Case à cocher pour indiquer que la tâche est terminée (barrée)

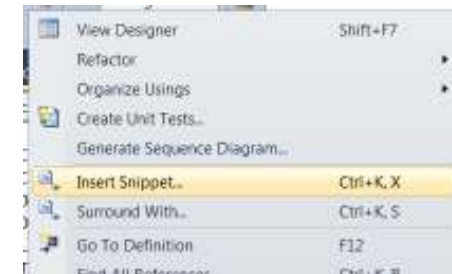


# Extraits de code (Snippets)

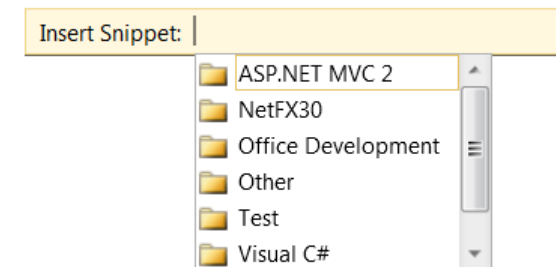
35

- ❑ Insertion automatique d'extraits de code
- ❑ Clic droit / insert snippet
- ❑ Choix de la catégorie
- ❑ Reste à remplacer les expressions surlignées en jaune par ses propres variables ou valeurs
- ❑ Répercussion automatique : par exemple en modifiant le 'i' surligné, les autres références à 'i' (encadrées en pointillés) se mettent à jour.

```
for (int i = 0; i < length; i++)  
{  
  
}
```



```
public Form1()  
{  
    InitializeComponent();  
  
}
```



# Snippets personnalisés

36

- ❑ Possibilité de créer ses propres snippets
- ❑ Fichier xml (extension .snippet)
- ❑ Ajout aux snippets via gestionnaire de snippets (Outils/Gestionnaire d'extraits de code/import)

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippet Format="1.0.0" xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <Header>
    <Title>BoucleEtIncrement</Title>
    <Author>Ben</Author>
    <Shortcut>boucleIncr</Shortcut>
    <Description>Effectue un boucle while et incrémente un compteur à chaque passage</Description>
    <SnippetTypes>
      <SnippetType>SurroundsWith</SnippetType>
      <SnippetType>Expansion</SnippetType>
    </SnippetTypes>
  </Header>
  <Content>
```

# Snippets personnalisés

37

```
<Snippet>
  <Declarations>
    <Literal>
      <ID>compteur</ID>
      <Default>i</Default>
    </Literal>
    <Literal>
      <ID>condition</ID>
      <Default>conditionARemplir</Default>
    </Literal>
  </Declarations>
  <Code Language="CSharp">
    <![CDATA[
      int $compteur$ = 0;
      while ($condition$)
      {
        // traitement à effectuer...

        $compteur$++;
      }
    ]]>
  </Code>
</Snippet>
</CodeSnippet>
```

```
int i = 0;
while (conditionARemplir)
{
  // traitement à effectuer...

  i++;
}
```

# Suivi des modifications

38

- ❑ Coloration dans la marge :
  - Jaune : code modifié/ajouté pas encore sauvegardé
  - Vert : Code modifié et sauvegardé



```
public Form1()
{
    InitializeComponent();

    Console.WriteLine("Hello World");
}

public Form1()
{
    InitializeComponent();

    MessageBox.Show("Hello World");
}

public Form1()
{
    InitializeComponent();

    MessageBox.Show("Hello World");
}
```

# Sélection de texte

39

- ❑ Sélection de plusieurs colonnes avec la touche Alt
- ❑ Modification simultanée du code saisi
- ❑ Intercaler du texte en sélectionnant une colonne de largeur 0

```
Console.WriteLine("Hello World");  
Console.WriteLine("Ligne 1");  
Console.WriteLine("Ligne 2");  
Console.WriteLine("Ligne 3");  
Console.WriteLine("Ligne 4");
```

```
Messa("Hello World");  
Messa("Ligne 1");  
Messa("Ligne 2");  
Messa("Ligne 3");  
Messa("Ligne 4");
```

```
MessageBox.Show("Hello World");  
MessageBox.Show("Ligne 1");  
MessageBox.Show("Ligne 2");  
MessageBox.Show("Ligne 3");  
MessageBox.Show("Ligne 4");
```



# Projets et solutions

40

- ❑ Un projet (.csproj) est un espace de travail qui regroupe les fichiers de code source, configuration et autres ressources nécessaires à l'écriture et la compilation d'une application.
- ❑ Plusieurs types de projets :
  - Applications (console, web, windows, WPF, WCF, Silverlight...)
  - Bibliothèques de classes
- ❑ Une solution (.sln) regroupe un ou plusieurs projets et définit des propriétés globales (compilation,...)
- ❑ Un projet peut être utilisé par plusieurs solutions

# Propriétés de Solution

41

- ❑ **Projet de démarrage** : projet(s) à démarrer au lancement du débogage de la solution
- ❑ **Dépendances du projet** : Gestion des dépendances entre les projets de la solution
- ❑ **Fichiers source pour le débogage** : Spécification des emplacements de code source utilisés lors du débogage
- ❑ **Configurations** : Configuration des différents projets :
  - Debug / Release
  - Plateforme cible

# Propriétés de projet

42

- ❑ **Application** : Comportement de l'application (nom d'assembly, namespace par défaut, framework cible, type de sortie, classe de démarrage, ressources) – cf. AssemblyInfo
- ❑ **Générer** : Option de génération (symboles de compilation conditionnelle, mode debug/release, plateforme, niveaux d'erreurs, chemin de sortie)
- ❑ **Événements de génération** : permet de spécifier des commandes à exécuter avant / après la génération
- ❑ **Propriétés de débogage** : Paramètres spécifiques au débogage (élément à démarrer, arguments de ligne de commande, répertoire de travail...)

# Propriétés de projet

43

- ❑ **Ressources** : permet de gérer un fichier de ressources, pour stocker des éléments de type texte (messages standard...). Définition de clés/valeurs. Accessibles dans le code par : *Properties.Resources.<clé de la ressource>*
- ❑ **Paramètres d'application** : Paramètres de configuration de l'application, chargés dynamiquement au démarrage. Accessibles dans le code par : *Properties.Settings.Default.<nom du paramètre>*
  - Type : String, DateTime, etc...
  - Portée :
    - Utilisateur (Modifiable pdt l'exécution)
    - Application (Modifiable uniquement via propriétés de projet)

44

# C# - Syntaxe de base

Variables

Opérateurs

Structures de contrôle

Méthodes

# Variables

45

- ❑ Permettent de stocker des valeurs pendant l'exécution du programme
- ❑ Conventions de nommage :
  - Commencent par une lettre
  - Max 1023 caractères
  - Composées de chiffres, lettres et '\_'
  - Sensible à la casse (maVariable != MaVariable)
  - Ne pas utiliser les mots clés du langage (if, for...)

# Types de variables

46

- ❑ Type de données que contient la variable
  - Types intégrés : int, char, bool, ...
  - Types utilisateur : interface, classe, énumération, ...
- ❑ Valeur ou référence
  - Types valeur : la variable contient la donnée
    - Pour les types primitifs (numériques, booléens, char)
    - Pour les énumérations et les structures
  - Type référence : la variable contient une référence vers l'emplacement mémoire de la donnée (notion de pointeur)
    - Pour les types complexes (String, classes utilisateur ou du framework)

# Types numériques

47

| Alias   | .NET Class | Type   | Bits | Plage  |
|---------|------------|--|------|--|
| byte    | Byte       | Entier non signé   | 8    | 0 à 255  |
| sbyte   | SByte      | Entier signé   | 8    | - 128 à 127  |
| int     | Int32      | Entier signé   | 32   | - 2 147 483 648 à 2 147 483 647                      |
| uint    | UInt32     | Entier non signé   | 32   | 0 à 4294967295                                       |
| short   | Int16      | Entier signé   | 16   | - 32 768 à 32 767                                    |
| ushort  | UInt16     | Entier non signé   | 16   | 0 à 65535  |
| long    | Int64      | Entier signé   | 64   | - 922337203685477508 à 922337203685477507            |
| ulong   | UInt64     | Entier non signé   | 64   | 0 à 18446744073709551615                             |
| float   | Single     | Type virgule flottante à simple précision  | 32   | -3.402823e38 à 3.402823e38                           |
| double  | Double     | Type virgule flottante à double précision  | 64   | -1.79769313486232e308 à 1.79769313486232e308         |
| decimal | Decimal    | Type fractionnaire ou intégral précis qui peut représenter des nombres décimaux avec 29 bits significatifs | 128  | $\pm 1,0 \times 10^{-28}$ à $\pm 7,9 \times 10^{28}$ |



# Autres types primitifs

48

## ❑ Caractère

- Type « **char** »
- Classe *Char*
- Stockage du code Unicode sur 2 octets
- Délimité par des guillemets simples
- Caractère d'échappement : « \ »
  - \', \", \\, \0 (nul), \a, \b, \f, \n, \r, \t, \v

## ❑ Booléen

- Type « **bool** »
- Classe *Boolean*
- Stockage sur 1 octet
- Valeur : *true* ou *false*

# Le type Object

49

- ❑ Le type Object est le plus universel dans le langage C#
- ❑ Tout type (y compris primitif) est issu du type Object
- ❑ On peut donc stocker n'importe quel élément dans un Object
- ❑ Object est un type par référence >> il stocke l'adresse mémoire de la donnée
- ❑ Méthodes :
  - ToString()
  - GetHashCode()
  - Equals(Object o)
  - GetType()

# Le type dynamic

50

- ❑ Permet de spécifier des variables dont le type n'est connu qu'à l'exécution
- ❑ Mot clef « dynamic »
- ❑ Les opérations (méthodes, propriétés) réalisées sur des types dynamiques sont résolues au moment de l'exécution
- ❑ Essentiellement utilisé pour manipuler les éléments issus d'un langage dynamique (IronRuby, IronPython) ou d'une API COM

```
dynamic a, b;  
a = 3;  
b = "3";  
Console.WriteLine(b.Length); // Propriété de string  
Console.WriteLine(a + b); // 33  
b = 3;  
Console.WriteLine(b.Length); // erreur : pas de propriété Length pour int  
Console.WriteLine(a + b); // 6
```

# Types Nullable

51

- ❑ Les types primitifs ne peuvent pas avoir de valeur null
- ❑ Par défaut ils ont une valeur (int : 0, float : 0, bool : false, char : \0)
- ❑ Le type Nullable permet de donner une valeur null à des variables de ce type (valeur par défaut)
- ❑ Syntaxe : <type>? <nomDeVariable>;
- ❑ Propriété HasValue >> pour vérifier si null ou pas
- ❑ Propriété Value >> renvoie la valeur ATTENTION : si null, l'appel à Value déclenche une erreur

```
int? age;  
age = 25;  
if (age.HasValue)  
{  
    Console.WriteLine(age.Value);  
}
```

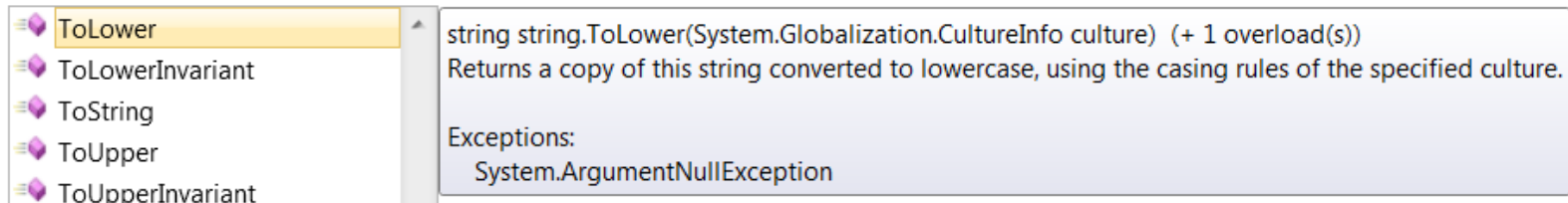
# Typage implicite

52

- ❑ Déclaration de variables avec le mot clé `var`
- ❑ La variable doit obligatoirement être initialisée à la déclaration
- ❑ Seulement pour des variables locales
- ❑ C'est la valeur donnée qui indique le type de la variable
- ❑ Évalué à la compilation (contrairement au type *dynamic*, qui est évalué à l'exécution)
- ❑ Utilisé notamment par LINQ

```
var x = "hello";
```

x.



|  |   |
|--|---|
| <ul style="list-style-type: none"><li>ToLower</li><li>ToLowerInvariant</li><li>ToString</li><li>ToUpper</li><li>ToUpperInvariant</li></ul> | <p>string string.ToLower(System.Globalization.CultureInfo culture) (+ 1 overload(s))<br/>Returns a copy of this string converted to lowercase, using the casing rules of the specified culture.</p> <p>Exceptions:<br/>System.ArgumentNullException</p> |
|--|---|

# Conversions

53

## ❑ Conversions implicites :

- Pas besoin de spécifier de conversion
- int vers long par exemple

## ❑ Conversions explicites :

- Opération de cast : type entre parenthèses
- (int)maVariable
- Selon le type d'origine et le type cible, il peut y avoir une perte de données :
- (int)2.53 deviendra 2

# Chaînes de caractères

54

## ❑ Le type string

- Alias : string
- Classe : String
- Méthodes de manipulation :
  - Recherche (IndexOf, Contains,
  - Modification (Insert, ToUpper, ToLower...)
  - Vérification (String.IsNullOrEmpty(string s), CompareTo...)
- Concaténation :
  - Fonctionne avec l'opérateur +
  - ATTENTION : l'utilisation du + sur les string est lente
  - >> Pour des opérations de concaténations nombreuses, on utilisera l'objet StringBuilder

```
StringBuilder sb = new StringBuilder();  
  
for (int i = 0; i < 1000; i++)  
{  
    sb.Append(i);  
}  
  
string result = sb.ToString();
```

# Conversions en chaînes de caractères

55

- ❑ Tout type a une méthode `ToString()`. Pour les types primitifs, renvoie la valeur de la variable au format string.
- ❑ Pour gérer le formatage :
  - méthode `String.Format(<format souhaité>, variable à formater)` renvoie la valeur formatée sous forme de string. ex : `String.Format("{0:c}", 123.45) >> 123,45 €`
  - Quelques formats :
    - Currency : `"{0:c}" >> 123.45 / 123,45 €`
    - Percent : `"{0:p}" >> 0.34 / 34.00%`
    - Standard : `"{0:s}" >> 12345.54321 / 12 345,54`



# Conversions en chaînes de caractères

56

## ❑ Formats numériques personnalisés :

- 0 : emplacement pour un caractère. Zéros non significatifs affichés
- # : emplacement pour un caractère. Zéros non significatifs masqués
- . : emplacement pour séparateur décimal
- , : emplacement pour séparateur de milliers

## ❑ Exemple :

- `String.Format("{0:#,###,###.000000 pièces d'or}", 12345678.32) >> 12 345 678,320000 pièces d'or`

# Conversions en chaînes de caractères

57

## ❑ Formats de Date et Heure prédéfinis :

- {0:d} : date court – 28/06/2012
- {0:D} : date longue – mardi 28 juin 2012
- {0:T} : heure – 13:45:23
- {0:G} : date court et heure - 28/06/2012 13:45:23
- {0:s} : format triable – 2012-06-28T13:45:23

## ❑ Formats personnalisés :

- Jour : d, dd, ddd, dddd - Mois : M, MM, MMM, MMMM
- Année : y, yy, yyyy
- Heure : h, hh (12h), H, HH (24h) - Minute : m, mm - Seconde : s, ss
- Décalage UTC : zzz
- Exemple : `String.Format("{0:ddd dd/MMMM/yyyy HH:mm:ss zzz}", DateTime.Now)`  
    >> mar. 11/décembre/2012 15:17:05 +01:00

# Conversions depuis une chaîne

58

- ❑ Chaque type primitif propose une méthode *Parse* qui convertit la chaîne passée en paramètre dans le type voulu.
- ❑ Exemple : `float prix = float.Parse("12,35");`
- ❑ ATTENTION : erreur si la chaîne donnée n'est pas convertible. Ex : `int.parse("toto")`
- ❑ Pour éviter cela, utiliser la méthode `TryParse()`

```
Console.Write("Saisissez votre age : ");
string strAge = Console.ReadLine();

int age;
if (int.TryParse(strAge, out age))
{
    Console.WriteLine("Vous avez {0} ans", age);
}
else
{
    Console.WriteLine("Saisie incorrecte");
}

Console.ReadLine();
```

# Portée des variables

59

- ❑ Bloc d'instructions : code compris entre { et }
- ❑ Une variable est accessible à l'intérieur du bloc dans lequel elle est définie et dans les blocs sous jacents.
- ❑ Sa durée de vie est définie de la même manière.

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        // i et j sont accessibles
    }

    // j n'est plus accessible, et est éliminée de la mémoire
    // i est encore accessible
}

// i n'est plus accessible, et est éliminée de la mémoire
```

# Constantes

60

- ❑ Une constante se déclare comme une variable, en ajoutant devant le mot-clef *const*
- ❑ Une constante doit être initialisée lors de sa déclaration
- ❑ Une constante n'est pas modifiable

```
const string message = "Bonjour";  
const string messageMadame = message + " Madame";
```

# Enumérations

61

- ❑ Définition d'un ensemble de constantes sous forme de liste : Mot-clef *enum*
- ❑ Chaque élément d'une enum correspond à un entier (par défaut 0,1,2.... Dans l'ordre de déclaration)
- ❑ Possibilité de spécifier la valeur
- ❑ S'utilise comme un type

```
Civilite civ = Civilite.Monsieur;

Console.WriteLine(civ); // "Monsieur"
Console.WriteLine((int)civ); // "2"
}

enum Civilite
{
    Madame, // equivaut à : Madame = 0,
    Mademoiselle, // equivaut à : Mademoiselle = 1,
    Monsieur // equivaut à : Monsieur = 2
}
```

```
enum Contenance
{
    quart = 25,
    tiers = 33,
    demi = 50,
    litre = 100
}
```

# Structures

62

- ❑ Permet de créer des types agrégés à partir de plusieurs données
- ❑ Utile pour regrouper des données dans une variable
- ❑ Mot clé *struct*
- ❑ Possibilité d'ajouter des méthodes
- ❑ Les structures sont un type de données par **valeur**
- ❑ Les membres de la structure doivent être déclarés public pour être accessibles

```
struct Personne
{
    public string nom;
    public string prenom;
    public int age;

    public string NomComplet()
    {
        return nom + " " + prenom;
    }
}
```

```
Personne p;
p.nom = "Messi";
p.prenom = "Lionel";
p.age = 25;
Console.WriteLine(p.NomComplet());
```

# Tableaux

63

- ❑ Stocke un ensemble de variables d'un même type
- ❑ Utilisation d'un index pour accéder aux valeurs d'un tableau (0 = premier élément)
- ❑ Une ou plusieurs dimensions
- ❑ Taille du tableau spécifiée à la création

```
int[] valeurs = new int[3];
valeurs[0] = 10;
valeurs[1] = 20;
valeurs[2] = 30;

string[] saisons = { "printemps", "été", "automne", "hiver" };

for (int i = 0; i < saisons.Length; i++)
{
    Console.WriteLine(saisons[i]);
}

// tableaux à 2 dimensions
int[,] matrice1 = new int[3, 3];
matrice1[0, 0] = 1;
matrice1[0, 1] = 2;
matrice1[1, 2] = 6;
int[,] matrice2 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

// tableau à 3 dimensions
int[,,] cube;
cube = new int[4, 4, 4];
cube[0, 1, 2] = 36;
```



# Opérateurs

64

- ❑ Opérateur d'affectation : =
- ❑ Opérateurs arithmétiques : +, -, \*, /, %
  - Ex : 19 / 3 : 6.3333333 19 % 3 : 1
- ❑ Opérateurs binaires : niveau du bit (sur entiers uniquement)
  - & (et), | (ou), ^ (ou exclusif), ~ (négation)
- ❑ Opérateurs de comparaison : booléens
  - == (égalité), != (inégalité), < (inf.), > (sup.), <= (inf. ou égal), >= (sup. ou égal), Is (comparaison sur un type)

```
Cool cSharp = new Cool();
```

```
Console.WriteLine(cSharp is Cool); // true
```

- ❑ Opérateurs Logiques : booléens ( test1 [opérateur] test2)
  - ❑ & (et), | (ou), ^ (ou exclusif), ! (négation)
  - ❑ && (et) évalue test2 uniquement si test1 est vrai
  - ❑ || (ou) évalue test2 uniquement si test1 est faux

# Structures de contrôle - if

65

*If (condition) instruction ;*

```
If (condition)  
{  
    bloc d'instructions  
}
```

```
If (condition) instruction;  
    else instruction;
```

```
If (condition)  
{  
    bloc d'instructions  
}  
else  
{  
    bloc d'instructions  
}
```

# Structures de contrôle : switch

66

```
switch (expression)
{
    case valeur1 :
        [case valeur2 :]
            bloc d'instructions
            [break;]
    case valeur3 :
        [case valeur4 :]
            bloc d'instructions
            [break;]
    ...
    default :
        bloc d'instructions
}
```

```
Console.Write("langage préféré : ");
string langage = Console.ReadLine();

switch (langage.ToLower())
{
    case "java" :
        Console.WriteLine("pas mal... essaie le C# !");
        break;
    case "c#":
    case "csharp" :
        Console.WriteLine("Très bon choix !");
        break;
    default :
        Console.WriteLine("bof");
        break;
}
```

# Structures de contrôle - boucles

67

- ❑ **While** : Evaluation avant premier passage :

```
while (condition)
{
    bloc d'instructions
}
```

- ❑ **Do...while** : Premier passage avant évaluation :

```
do
{
    bloc d'instructions
}
while (condition)
```

# Structures de contrôle - boucles

68

- ❑ **For** : Contrôle du nombre de passages :  
for (*initialisation; condition; instruction d'itération*)  
{  
    *bloc d'instructions*  
}

*Initialisation* :

Exécutée une seule fois avant le premier passage

*Condition* :

Évaluée avant chaque passage

*Instruction d'itération* :

Exécutée après chaque passage

# Structures de contrôle - boucles

69

- ❑ **Foreach** : Parcours d'un tableau ou d'une collection

`foreach (element in tableau)`

`{`

*bloc d'instructions*

`}`

```
string[] saisons = { "hiver", "printemps", "été", "automne" };
```

```
foreach (string s in saisons)
```

```
{
```

```
    Console.WriteLine(s);
```

```
}
```

```
for (int i = 0; i < saisons.Length; i++)
```

```
{
```

```
    Console.WriteLine(saisons[i]);
```

```
}
```

# Structures de contrôle - using

70

- ❑ Le mot clé *using* s'utilise pour un bloc d'instruction qui utilise une ressource externe. La structure *using* libère automatiquement la ressource à sa sortie
- ❑ Libération de la ressource : appel implicite à sa méthode *Dispose()* >> interface *IDisposable*

```
using (StreamWriter sw = File.CreateText("fichier.txt"))
{
    sw.WriteLine(DateTime.Now.ToString("dd/MM/yyyy hh:mm:ss") + " : ok");
}
```

# Méthodes

71

- ❑ `[static] typeDeRetour NomMethode([paramètres])`
  - Mot clé *static* : pour des méthodes communes à toutes les instances (voir concepts objet)
  - Type de retour : type du résultat renvoyé par la méthode (fonction), *void* si pas de retour (procédure)
  - Nom des méthodes en casse pascal : **NomMethode()**
  - Une fonction qui a un type de retour doit contenir au moins une instruction *return*
  - Paramètres : *type param1, type param2...*



# Méthodes - Exemple

72

```
static int PlusGrand(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}

static void EcrireFichier(string chemin, string message)
{
    using (StreamWriter sw = File.CreateText(chemin))
    {
        sw.WriteLine(message);
    }
}
```

# Méthodes - paramètres

73

- ❑ Passage de paramètres par valeur ou par référence
- ❑ Par défaut, défini selon le type de paramètre :
  - Numérique, bool, structure : **valeur**
  - Cas particulier des string : type référence, mais comportement par **valeur** en tant que paramètre (immutable)
  - Autres : **référence**
- ❑ Les paramètres **valeur** modifiés dans une méthode **ne sont pas modifiés** dans le code appelant
- ❑ Les paramètres **référence** modifiés dans une méthode **sont modifiés** dans le code appelant

# Paramètres - exemple

74

```
static void TestValeurReference(int i, String s, int[] t)
{
    i++;
    s = s + "def";
    t[0]++;
}

static void Main(string[] args)
{
    int[] tableau = { 1, 2, 3 };
    int i = 0;
    String s = "abc";
    TestValeurReference(i, s, tableau);
    Console.WriteLine(i); // 0
    Console.WriteLine(s); // abc
    Console.WriteLine(tableau[0]); // 2

    Console.ReadLine();
}
```

# Paramètres par référence

75

- ❑ Mot clé *ref* : spécifie que le paramètre doit être passé par référence
  - `void MaMethode(ref int x){...}`
  - Doit être utilisé lors de l'appel : `MaMethode(ref i);`
  - La variable doit être initialisée avant l'appel.
- ❑ Mot clé *out* : similaire à *ref*, mais l'initialisation n'est pas obligatoire

# Paramètres par référence - exemple

76

```
static void Incrementer(ref int i)
{
    i++;
}

static void Initialiser(out string s)
{
    s = "Hello !";
}

static void Main(string[] args)
{
    int i = 0;
    Incrementer(ref i);

    string s;
    Initialiser(out s);

    Console.WriteLine(i); // 1
    Console.WriteLine(s); // Hello
}
```

77

# Gestion des exceptions

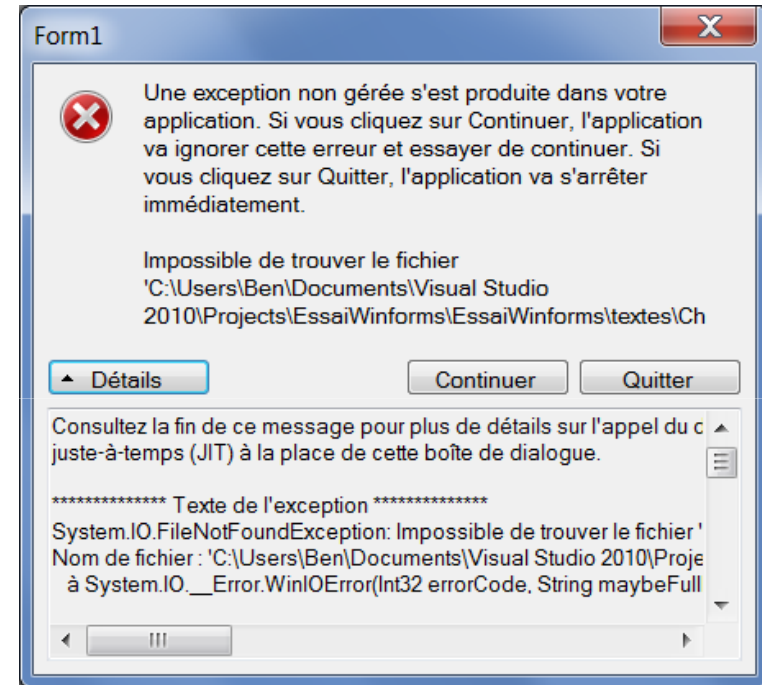
Récupération d'exceptions

Création et déclenchement d'exceptions

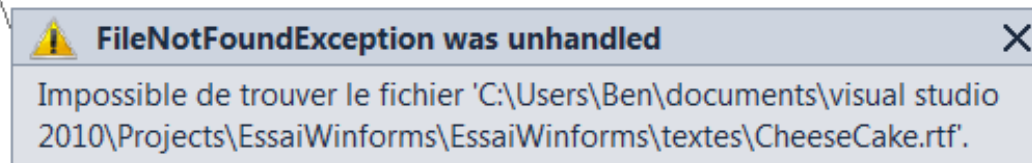
# Les exceptions

78

- ❑ A l'exécution, de nombreuses erreurs peuvent se produire (division par zéro, ouverture d'un fichier qui n'existe plus, accès à une variable null...). On les appelle des Exceptions
- ❑ En release, message d'erreur standard
- ❑ En debug, point d'arrêt et message d'erreur sur la ligne de code concernée



```
t.Dock = DockStyle.Fill;  
t.richTextBox1.LoadFile("../..../textes/" + e.Node.Name + ".rtf");  
t.Show();  
break;
```



# Récupération des exceptions

79

- ❑ Il est possible de protéger un bloc de code susceptible de déclencher une exception, avec les instructions `try / catch / finally` :

```
try
{
    // bloc de code à surveiller
}
catch (Exception e)
{
    // traitement à effectuer en cas d'exception
}
finally
{
    // traitement effectué avec ou sans exception
}
```



# La classe Exception

80

- ❑ Lorsqu'une exception est levée, un objet de type Exception est récupéré par le bloc catch.
- ❑ La classe Exception expose des propriétés et méthodes permettant d'obtenir des informations détaillées sur l'erreur. Les plus courantes sont :
  - Message : message d'erreur
  - StackTrace : Pile des appels sous forme de string
  - Source : nom de l'élément qui a déclenché l'erreur
  - InnerException : exception à l'origine de l'exception en cours

```
catch (Exception e)
{
    MyLog.AddMessage(DateTime.Now + " : Exception : " + e.Message);
    Console.WriteLine("Une erreur est survenue. Tout est sous contrôle");
}
```

# Exceptions spécialisées

81

- ❑ Le framework propose de nombreuses classes dérivées d'Exception, qui proposent des propriétés spécifiques en fonction de leur spécificité.
- ❑ Quelques exemples :
  - System.NullReferenceException
  - System.NotImplementedException
  - System.StackOverflowException
  - System.DivideByZeroException
  - System.InvalidCastException
  - System.IO.DriveNotFoundException
  - System.IO.FileNotFoundException
  - System.IO.DirectoryNotFoundException

# Exceptions spécialisées

82

- ❑ Possibilité de mettre plusieurs blocs catch successifs pour adapter le traitement aux types d'exceptions :

```
catch (FileNotFoundException e)
{
    Console.WriteLine("Le fichier " + e.FileName + " n'existe pas");
}
catch (IOException e)
{
    Console.WriteLine("Erreur d'entrée sortie");
}
catch (Exception e)
{
    Console.WriteLine("Une erreur est survenue");
}
```

# Création d'exceptions

83

- ❑ Le mécanisme des exceptions est un moyen fiable et élégant pour gérer des erreurs de type métier.
- ❑ Il est possible de définir ses propres classes d'exceptions, en les dérivant de la classe `Exception` :

```
public class AgeIncoherentException : Exception
{
    public AgeIncoherentException(string message) : base(message)
    {
    }
}
```

# Déclenchement d'exceptions

84

- ❑ On peut lever explicitement une exception (mot clé *throw*), qu'elle soit personnalisée ou issue du framework

```
public class Personne
{
    public Personne(string nom, DateTime dateNaissance)
    {
        this.nom = nom;
        if (dateNaissance > DateTime.Now)
        {
            throw new AgeIncoherentException("date de naissance dans le futur !");
        }
        this.dateNaissance = dateNaissance;
    }

    public void TraitementTresImportant()
    {
        throw new NotImplementedException();
    }
}
```

85

# Concepts Objet en C#

Classe et Objet

Héritage et polymorphisme

# Concepts objet - Introduction

86

- ❑ Décomposer un programme en entités logique (objets)
- ❑ Avantages :
  - Modularité, souplesse
  - Maintenance facilitée
  - Réutilisation de code
  - Structure plus logique et plus cohérente, plus proche du réel.
  - Supporte mieux les évolutions
- ❑ Notion de classe : une classe décrit une entité par des propriétés (valeurs) et des méthodes (traitements).
- ❑ Encapsulation : L'implémentation n'a pas besoin d'être connue pour utiliser une classe : Chaque classe est responsable de sa structure et de ses traitements internes.
- ❑ Notion d'instance : Un objet est une instance de classe : "exemplaire" concret d'une classe. Exemple : "l'objet Zinedine Zidane est une instance de la classe footballeur".
- ❑ Notion d'héritage : il est possible de définir des classes "mères" et des classes "filles", les secondes héritant des comportements des premières. Ainsi on favorise la réutilisation de code et évite de le dupliquer inutilement.

# Niveau d'accès

87

- ❑ Détermine quelles parties du code ont accès à un élément (classe, méthode, propriété...)
- ❑ **public** : accessible par tout le monde
- ❑ **protected** : utilisé sur les membres d'une classe. Restreint l'accès à la classe et aux classes qui en héritent
- ❑ **internal** : accessible uniquement à l'intérieur de l'assembly (projet)
- ❑ **protected internal** : combine protected et internal
- ❑ **private** : accessible uniquement à l'intérieur de la classe (par défaut)



# Déclaration d'une classe

88

- ❑ Mot clef *class*
- ❑ *La classe est délimitée par des accolades : { }*
- ❑ Recommandations :
  - 1 classe par fichier (NomClasse.cs)
  - Commencer par une majuscule
- ❑ Exemple (Vehicule.cs) :

```
class Vehicule
{

}
```

# Champs

89

- ❑ Les champs sont les attributs de la classe, auxquels des valeurs sont associées.
- ❑ Par exemple, un véhicule a une longueur et une largeur.
- ❑ Chaque champ a un type, par exemple entier (int) dans le cas des dimensions.
- ❑ Exemple :

```
class Vehicule
{
    int longueur;
    int largeur;
}
```

# Propriétés

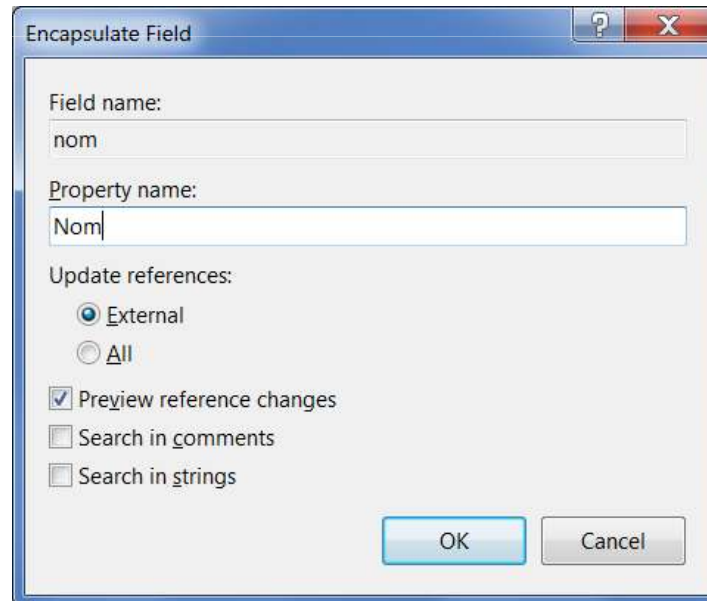
90

- ❑ Les propriétés sont une spécificité de C# qui permettent d'exposer une représentation spécifique des champs, en lecture ("get") et/ou en, écriture ("set").
- ❑ Une bonne pratique consiste à utiliser les propriétés pour accéder aux champs, en lecture comme en écriture, et empêcher l'accès direct aux champs depuis l'extérieur de la classe. Dans ce cas, on déclare les champs en "private" et les propriétés en "public" (voir portée).
- ❑ Une propriété peut contenir plusieurs instructions.
- ❑ Utile pour contrôler des entrées (set) et pour donner des informations calculées (set)

# Propriétés

91

- ❑ Convention de nommage :
  - Champ : `dateNaissance`
  - Propriété : `DateNaissance`
- ❑ Visual Studio : Génération automatique à partir du champ : clic droit sur le champ/ refactoriser/ « encapsuler le champ »



# Propriétés - Exemple

92

```
class Personne
{
    private string nom;
    private DateTime dateNaissance;

    public string Nom
    {
        get { return nom; }
        set { nom = value; }
    }

    public DateTime DateNaissance
    {
        set {
            if (value < DateTime.Now) {
                dateNaissance = value;
            }
        }
    }

    public int Age
    {
        get {
            TimeSpan ts = DateTime.Now.Subtract(dateNaissance);
            return ts.Days/365;
        }
    }
}
```

# Constructeurs

93

- ❑ Instanciation d'une classe >> appel au constructeur
- ❑ Constructeur : fonction qui crée et initialise une nouvelle instance
- ❑ Syntaxe : [portée] <NomClasse> ([paramètres])
- ❑ On peut avoir plusieurs constructeurs avec des paramètres différents pour répondre à divers cas d'utilisations >> Notion de surcharge
- ❑ Par exemple, on peut créer une personne en précisant ses nom et date de naissance, ou simplement créer une personne sans préciser ses valeurs si on souhaite les renseigner plus tard

# Constructeurs

94

- ❑ Le mot clef "this" est une référence à l'instance courante, il permet d'accéder à l'objet en cours au moment de l'appel au constructeur (valable également pour les méthodes)
- ❑ Pour qu'un constructeur soit appelé par n'importe quelle partie du programme (à l'extérieur de la classe), il doit être déclaré public
- ❑ Si aucun constructeur précisé >> constructeur par défaut est sans paramètre
- ❑ Un ou plusieurs constructeurs définis >> plus de constructeur par défaut

# Constructeurs - Exemple

95

```
class Personne
{
    /// <summary>
    /// Constructeur sans paramètre
    /// </summary>
    public Personne()
    {

    }

    /// <summary>
    /// Constructeur avec paramètres
    /// </summary>
    /// <param name="nom">nom de la personne</param>
    /// <param name="dateNaissance">date de naissance</param>
    public Personne(string nom, DateTime dateNaissance)
    {
        this.nom = nom;
        this.dateNaissance = dateNaissance;
    }
}
```



# Constructeurs

96

- ❑ Possibilité d'appeler un constructeur à partir d'une surcharge :

```
public Personne()  
{  
    this.dateCreation = DateTime.Now;  
}  
  
public Personne(string nom, DateTime dateNaissance) : this()  
{  
    this.nom = nom;  
    this.dateNaissance = dateNaissance;  
}  
  
private DateTime dateCreation;
```

# Destructeurs

97

- ❑ Quand il n'est plus référencé, un objet n'a plus lieu d'exister, et doit être déchargé de la mémoire
- ❑ Lorsqu'un objet est déchargé de la mémoire (garbage collector), son destructeur est appelé
- ❑ On peut spécifier un destructeur pour y ajouter des traitements, comme par exemple la libération de ressources
- ❑ Syntaxe : `~NomDeClasse(){...}`
- ❑ ATTENTION : On ne contrôle pas le moment exact où l'objet sera supprimé, c'est le garbage collector qui gère son activité.

```
~Personne()  
{  
    // libération des ressources  
}
```

# Méthodes

98

- ❑ Les méthodes représentent les différents traitements qui sont associés à la classe.
- ❑ Une méthode doit préciser un type de retour (valeur retournée à l'issue du traitement), ou "void" s'il n'y a pas de valeur retournée.
- ❑ Une méthode peut avoir des paramètres, qui sont utilisables dans le corps de la méthode.

```
public string Decrire()  
{  
    return "je m'appelle " + this.nom + " et j'ai " + this.Age + " ans.";  
}  
  
Personne p = new Personne("Benoit", new DateTime(1979, 06, 27));  
p.Nom = "Arthur";  
string description = p.Decrire();  
Console.WriteLine(description); // je m'appelle Arthur et j'ai 33 ans
```

# Surcharge de méthodes

99

- ❑ Comme pour les constructeurs, on peut surcharger une méthode, c'est à dire conserver le même nom de méthode mais spécifier des attributs différents.
- ❑ Selon les arguments passés lors de l'appel, le programme appellera la fonction correspondante. La différence entre ces méthodes se fait par le nombre de paramètres, leur type et leur ordre.
- ❑ Les surcharges sont souvent utilisées pour donner le choix entre une méthode avec ses paramètres par défaut et une version qui laisse le choix des paramètres
- ❑ Quand c'est possible, privilégier la refactorisation

# Surcharge de méthodes - Exemple

100

## ❑ Surcharge :

```
public void Enregistrer()
{
    using (StreamWriter sw = File.CreateText("personne.txt"))
    {
        sw.WriteLine(this.Nom + " - " + this.dateNaissance);
    }
}

public void Enregistrer(string chemin)
{
    using (StreamWriter sw = File.CreateText(chemin))
    {
        sw.WriteLine(this.Nom + " - " + this.dateNaissance);
    }
}
```

# Surcharge de méthodes - Exemple

101

- ❑ Après refactorisation :

```
public void Enregistrer()
{
    Enregistrer("personne.txt");
}

public void Enregistrer(string chemin)
{
    using (StreamWriter sw = File.CreateText(chemin))
    {
        sw.WriteLine(this.Nom + " - " + this.dateNaissance);
    }
}
```

# Instanciación

102

- ❑ A partir d'une classe, on peut instancier (= "créer") des objets n'importe où dans le programme. On utilise le mot clé *new*
- ❑ Un objet se déclare et s'instancie de la manière suivante :  

```
Vehicule v;  
v = new Vehicule();
```
- ❑ On peut également déclarer et instancier l'objet en même temps :  

```
Vehicule v2 = new Vehicule(400, 180);
```
- ❑ A partir du moment où on dispose d'une instance d'objet, on peut utiliser à toutes ses propriétés et méthodes publiques. Par exemple :  

```
v.Longueur = 350;  
v.Largeur = 140;  
Console.WriteLine(v2.ObtenirDimensions());
```

# Membres static

103

- ❑ Au niveau d'une classe, on peut avoir besoin de manipuler des valeurs ou méthodes génériques, qui restent constantes quelque soit l'instance créée. Pour définir de tels champs ou méthodes, on utilise le mot clé *static*
- ❑ On l'utilise souvent pour des méthodes utilitaires, pour des calculs par exemple (cf classe Math)
- ❑ Par exemple, on peut l'utiliser pour définir une largeur maximale, et pour l'afficher, sans avoir besoin d'instancier un objet Vehicule

```
public static int largeurMaximum = 300;

public static string ObtenirLargeurMaximum()
{
    return "La largeur maximale d'un véhicule est de " + largeurMaximum.ToString();
}
```

- ❑ Pour accéder à un membre static, il faut préciser le nom de la classe

```
string s = Vehicule.ObtenirLargeurMaximum();
```



# Membres static

104

- ❑ On peut aussi utiliser un attribut static pour maintenir une variable globale à la classe, par exemple le nombre de véhicules instanciés
- ❑ On mettra à jour cette valeur à chaque création de véhicule, donc dans le constructeur

```
public static int nbVehicules = 0;
```

```
public Vehicule()  
{  
    nbVehicules++;  
}
```

# Classes static

105

- ❑ On peut aussi déclarer une classe comme static
  - Une classe static ne contient que des membres static
  - Elle ne peut pas être instanciée
  - Elle ne peut pas être héritée (sealed)
  - Elle ne peut pas contenir de constructeurs d'instance
- ❑ Il est possible de déclarer un constructeur static
  - Il ne peut pas avoir de modificateur d'accès
  - Il n'a aucun paramètre
  - Appelé implicitement lors du premier appel à un membre de la classe static.

# Classes static - Exemple

106

```
static class MyLog
{
    static MyLog()
    {
        Console.WriteLine("Je crée le fichier de log");
        // todo : création d'un fichier texte nommé avec date/heure
    }

    public static void AddMessage(string message)
    {
        Console.WriteLine("Nouveau message loggé : " + message);
        // ajout de la ligne dans le fichier
    }
}
```

```
MyLog.AddMessage("message 1");
MyLog.AddMessage("message 2");
```

L'exécution de ces 2 lignes renverra :

```
Je crée le fichier de log
Nouveau message loggé : message 1
Nouveau message loggé : message 2
```

# Redéfinition d'opérateurs

107

- ❑ Dans la déclaration d'une classe, il est possible de redéfinir des opérateurs, pour pouvoir comparer des instances de cette classe ( $<$ ,  $>$ ,  $==$ , ...), ou effectuer des opérations ( $+$ ,  $-$ , ...).
- ❑ On utilise le mot clé *operator* suivi de l'opérateur à redéfinir et de la signature correspondante
- ❑ Quand on redéfinit un opérateur de comparaison, on doit obligatoirement redéfinir aussi son opposé :
  - $==$  et  $!=$
  - $<$  et  $>$
  - $<=$  et  $>=$

# Redéfinition d'opérateurs : exemple

108

```
public static bool operator == (Vehicule a, Vehicule b)
{
    if (a.longueur == b.longueur && a.largeur == b.largeur)
    {
        return true;
    }
    else
        return false;
}

public static bool operator != (Vehicule a, Vehicule b)
{
    if (a.longueur != b.longueur || a.largeur != b.largeur)
    {
        return true;
    }
    else
        return false;
}

Vehicule v1 = new Vehicule(300, 185);
Vehicule v2 = new Vehicule(347, 203);

if (v1 == v2)
{
    Console.WriteLine("identiques");
}
```

# Redéfinition d'opérateurs : exemple

109

```
public static Vehicule operator + (Vehicule a, Vehicule b)
{
    Vehicule result = new Vehicule();

    result.largeur = a.largeur + b.largeur;
    result.longueur = a.longueur + b.longueur;

    return result;
}
```

```
Vehicule v3 = v1 + v2;
```

# Héritage et polymorphisme

110

- ❑ Héritage : Notion fondamentale de la programmation orientée objet
- ❑ Consiste à définir une classe à partir d'une classe existante. La classe dérivée ("classe fille") bénéficie ainsi de toutes les fonctionnalités de la classe mère.
- ❑ Par exemple, on peut créer une classe Voiture, et une classe Bateau qui héritent de Vehicule, et possèdent donc les attributs et les méthodes de Vehicule
  - Factorisation du code commun
  - Généricité
- ❑ Polymorphisme : Définition d'un comportement commun (méthode), mais traitement différent pour chaque classe fille
- ❑ En C#, il n'est possible d'hériter que d'une seule classe (dans certains langages, on peut hériter de plusieurs classes, on parle alors d'héritage multiple).

# Héritage - déclaration

111

- ❑ La déclaration d'une classe dérivée se fait de la manière suivante

```
public class Voiture : Vehicule  
{  
  
}
```

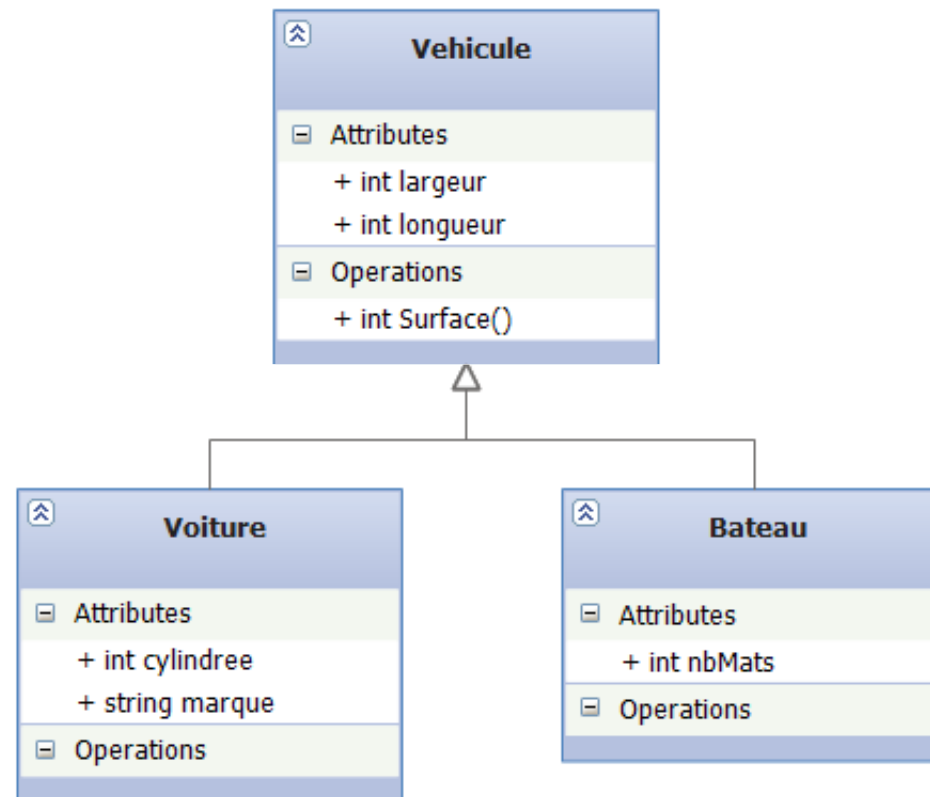
- ❑ La classe Voiture hérite de la classe Vehicule.



# Héritage - membres

112

- ❑ Automatiquement, la classe fille hérite des membres de la classe mère (champs, propriétés, méthodes)
- ❑ Dans la classe fille, on peut déclarer des champs supplémentaires qui lui seront propres
- ❑ La classe fille peut accéder aux membres public, protected et protected internal de sa classe mère



# Héritage - Constructeur

113

- ❑ L'appel au constructeur d'une classe Fille déclenche automatiquement l'appel au constructeur standard de la classe mère (celui qui n'a pas de paramètres).

```
public Voiture()  
{...}
```

>> appel implicite au constructeur de Vehicule()

- ❑ Possibilité de spécifier l'appel à un constructeur particulier de la classe mère : mot clé *base*
- ❑ Le mot clé *base* permet de faire référence à la classe mère
- ❑ Ex : Appel au constructeur de Vehicule à 2 paramètres (longueur, largeur)

```
public Voiture(int longueur, int largeur, string marque, int cylindree) : base(longueur, largeur)  
{  
    this.marque = marque;  
    this.cylindree = cylindree;  
}
```

# Héritage – Classes abstraites

114

- ❑ Une classe peut être déclarée comme "abstract", c'est à dire qu'elle peut définir des méthodes ou des propriétés sans implémentation (juste une signature)
- ❑ L'implémentation est confiée aux classes dérivées
- ❑ Les classes dérivées doivent obligatoirement implémenter ces méthodes/propriétés (mot clé *override*)
- ❑ Mot clé *abstract* devant chaque méthode/propriété abstraite
- ❑ La classe doit elle aussi être déclarée abstract
- ❑ Une classe abstraite peut avoir des membres non abstraits
- ❑ Une classe abstraite ne peut pas être instanciée directement

# Classes abstraites - exemple

115

```
public abstract class Vehicule
{
    public abstract bool EstTerrestre
    {
        get;
    }
    public abstract void Demarrer();
}

public class Voiture : Vehicule
{
    public override bool EstTerrestre
    {
        get { return true; }
    }
    public override void Demarrer()
    {
    }
}
```

# Héritage – Méthodes virtuelles

116

- ❑ Une classe peut déclarer des méthodes/propriétés virtuelles, à l'aide du mot clé *virtual*
- ❑ Un membre virtuel peut être redéfini dans les classes dérivées
- ❑ Contrairement à l'abstract, un membre virtual (méthode ou propriété) contient un corps, et la redéfinition n'est pas obligatoire
- ❑ Ce mécanisme permet de définir un comportement par défaut dans la classe mère, et de laisser la possibilité de le remplacer ou de le compléter dans les classes filles

# Méthodes virtuelles - exemple

117

```
public class Vehicule
{
    public virtual void Decrire()
    {
        Console.WriteLine("je suis un véhicule de " + longueur + "m sur " + largeur);
    }

    public override void Decrire()
    {
        // appel à la méthode Decrire() de Vehicule (facultatif)
        base.Decrire();
        // traitement spécifique Voiture :
        Console.WriteLine("Je suis une voiture de marque " + this.marque);
    }
}
```

# Héritage – Classes sealed

118

- ❑ Une classe est déclarée *sealed* lorsqu'on souhaite empêcher de la dériver
  - ❑ Aucune classe ne peut hériter d'une classe sealed
  - ❑ Une classe sealed ne peut pas définir de membres abstract ou virtual
- ```
public sealed class Neutron
{
    ...
}
```
- ❑ Le mot clé *sealed* peut aussi être utilisé sur une méthode, pour empêcher sa redéfinition dans les classes filles.

# Polymorphisme

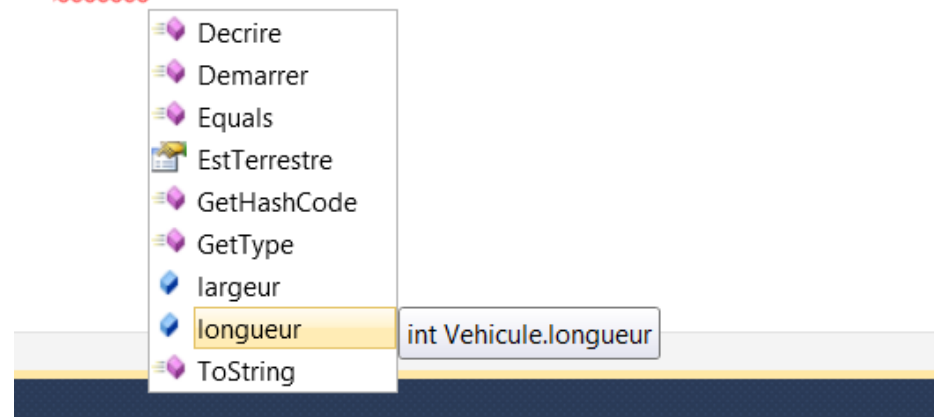
119

- ❑ Le polymorphisme consiste à manipuler des objets dérivés en sous la forme de leur classe mère : une partie du code peut avoir besoin de manipuler des Vehicules, sans savoir s'il s'agit de bateaux, voitures, motos, etc...
- ❑ Dans ce cas, seuls les membres de la classe mère sont accessibles

```
Voiture merco = new Voiture(400, 200, "Mercedes", 3200);
```

```
Vehicule engin = new Voiture(300, 160, "Renault", 1200);
```

engin.





# Substitution de méthodes

120

- ❑ Les méthodes virtual peuvent être redéfinies dans les classes filles
- ❑ On utilise le mot clé *override* pour effectuer cette substitution
- ❑ Le mot clé *override* est aussi utilisé pour l'implémentation de méthodes abstract
- ❑ Une méthode virtual peut être substituée dans les classes dérivées, à tous les niveaux de parenté
- ❑ Pour bloquer la substitution vers les classes filles, on utilise le mot clé *sealed* :  

```
public override sealed void Decrire()  
{...}
```

# Substitution - exemple

121

```
public class Vehicule
{
    public virtual void Decrire()
    {
        Console.WriteLine("je suis un véhicule de " + longueur + "m sur " + largeur);
    }

    public override void Decrire()
    {
        // appel à la méthode Decrire() de Vehicule (facultatif)
        base.Decrire();
        // traitement spécifique Voiture :
        Console.WriteLine("Je suis une voiture de marque " + this.marque);
    }
}
```

# Masquage de méthodes

122

- ❑ Dans le cas de l'override, c'est le type de l'instance qui détermine la méthode à appeler
- ❑ Au lieu de l'override, on peut utiliser le mot clé *new* pour masquer les substitutions
- ❑ Dans ce cas, c'est le type de la variable qui détermine la méthode à appeler
- ❑ Il est possible de masquer une méthode qui n'est pas virtual
- ❑ L'intérêt du masquage est de garantir le fonctionnement d'un code existant quand on ajoute des classes dérivées. A manipuler avec précaution...

# Masquage de méthodes - exemple

123

```
public abstract class Vehicule
{
    public virtual void DecrireOverride()
    {
        Console.WriteLine("vehicule");
    }

    public virtual void DecrireNew()
    {
        Console.WriteLine("vehicule");
    }
}

public class Voiture : Vehicule
{
    public override void DecrireOverride()
    {
        Console.WriteLine("voiture");
    }

    public new void DecrireNew()
    {
        Console.WriteLine("voiture");
    }
}

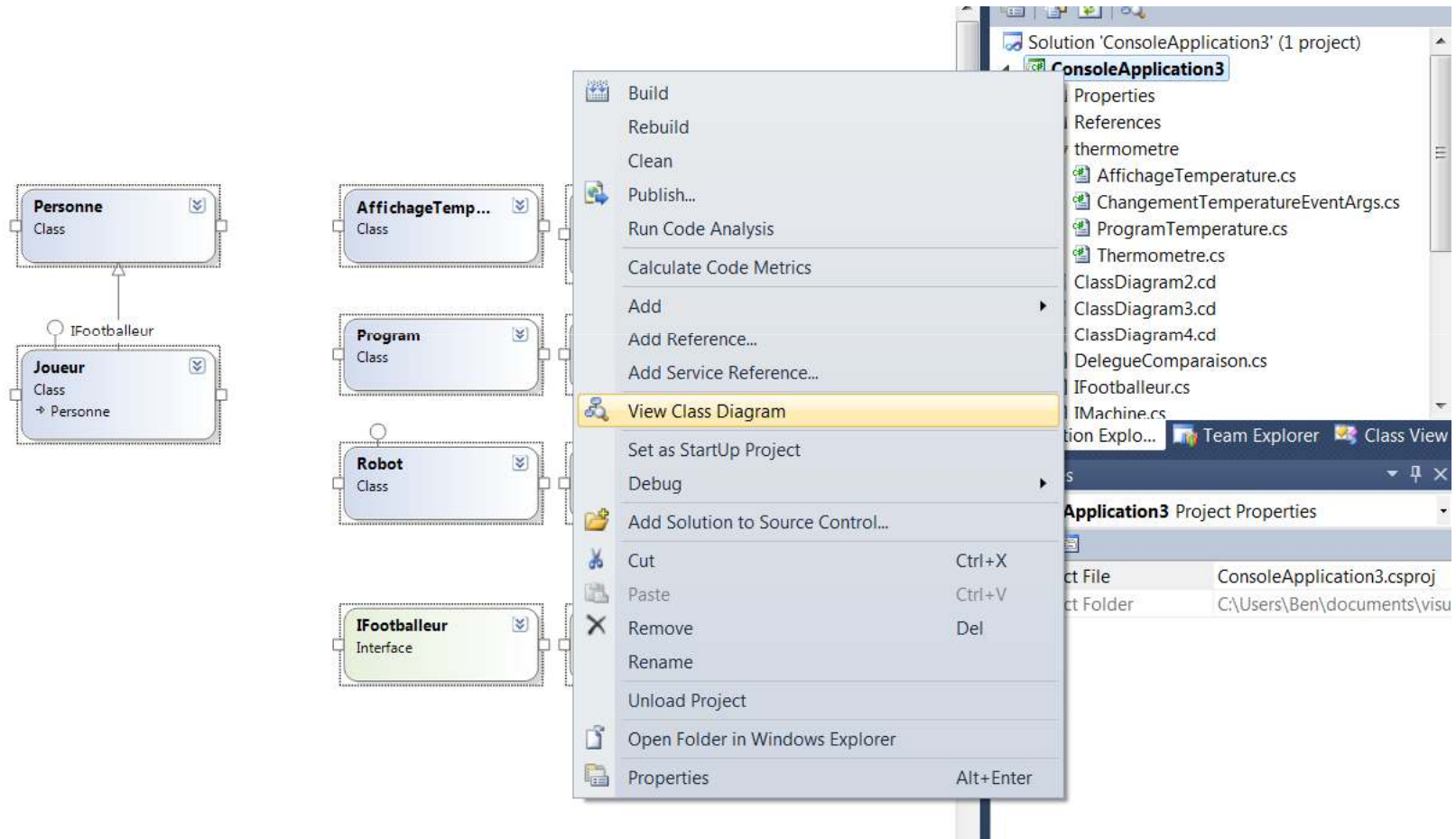
Voiture merco = new Voiture(400, 200, "Mercedes", 3200);
Vehicule engen = new Voiture(300, 160, "Renault", 1200);

engen.DecrireOverride(); // voiture
engen.DecrireNew();      // vehicule

merco.DecrireOverride(); // voiture
merco.DecrireNew();      // voiture
```

# Affichage du diagramme de classes

124



# Interfaces

125

- ❑ Une interface définit un ensemble de méthodes et propriétés à implémenter (comme pour une classe abstraite).
- ❑ L'interface constitue un « contrat » à remplir par les classes qui l'implémentent.
- ❑ Toute classe qui implémente une interface doit définir l'intégralité de ses méthodes et propriétés.
- ❑ Il est possible d'implémenter plusieurs interfaces
- ❑ Une interface peut « hériter » d'autres interfaces
- ❑ C# : préfixe en i

# Interfaces

126

- ❑ Le framework .NET propose de nombreuses interfaces qui sont utilisées par des classes du framework, et utilisables par le développeur
- ❑ Quelques exemples :
  - **Comparable** : Méthode CompareTo qui permet de comparer deux instances
  - **Serializable** : Méthode GetObjectData permettant de personnaliser le format de sérialisation des données d'un objet
  - **ICollection** : Jeu de méthodes et de propriétés pour les classes de type collection comme ArrayList, Stack et Queue
  - **IDictionary** : Jeu de méthodes et de propriétés pour les classes de type collection basées sur une clé comme HashTable et SortedList

# Interface - exemple

127

```
namespace ConsoleApplication3
{
    interface IFootballeur
    {
        int Endurance { get; set; }
        int Vitesse { get; set; }
        int Technique { get; set; }
        int NumeroMaillot { get; set; }

        void Dribbler();
        void Passer();
        void Tirer();
    }
}
```

```
public class Joueur : Personne, IFootballeur
{
    public int Endurance { get; set; }
    public int Vitesse { get; set; }
    public int Technique { get; set; }
    public int NumeroMaillot { get; set; }

    public void Dribbler()
    {
        Console.WriteLine("hop hop hop");
    }

    public void Passer()
    {
        Console.WriteLine("choppe !");
    }

    public void Tirer()
    {
        Console.WriteLine("vlan !");
    }
}
```



# Interface - exemple

128

```
public class Robot : IFootballeur, IMachine
{
    public int Endurance { get; set; }
    public int Vitesse { get; set; }
    public int Technique { get; set; }
    public int NumeroMaillot { get; set; }

    public void Dribbler()
    {
        Console.WriteLine("Schling schling");
    }

    public void Passer()
    {
        Console.WriteLine("Clac");
    }

    public void Tirer()
    {
        Console.WriteLine("Bzzzzz !");
    }

    public void Huiler()
    {
        // implémentation
    }

    public void Recharger()
    {
        // implémentation
    }
}
```

```
interface IMachine
{
    void Huiler();
    void Recharger();
}
```

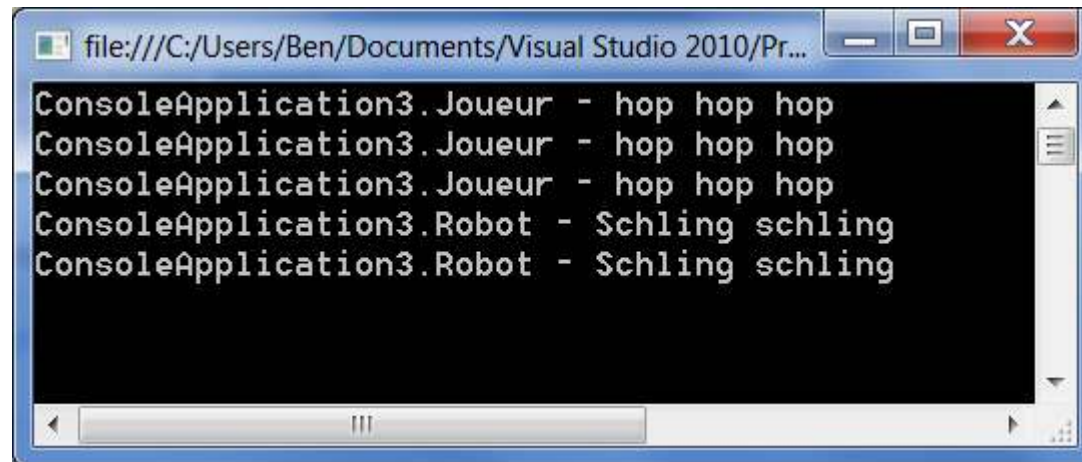
# Interface - exemple

129

```
IFootballeur[] equipe = new IFootballeur[5];

equipe[0] = new Joueur(5,3,7,12,"Florian");
equipe[1] = new Joueur(9,9,9,18,"Zlatan");
equipe[2] = new Joueur(11,11,11,10,"Lionel");
equipe[3] = new Robot(5,3,7,7);
equipe[4] = new Robot(5,3,7,6);

foreach (IFootballeur f in equipe)
{
    Console.WriteLine(f.GetType().ToString() + " - ");
    f.Dribbler();
}
```

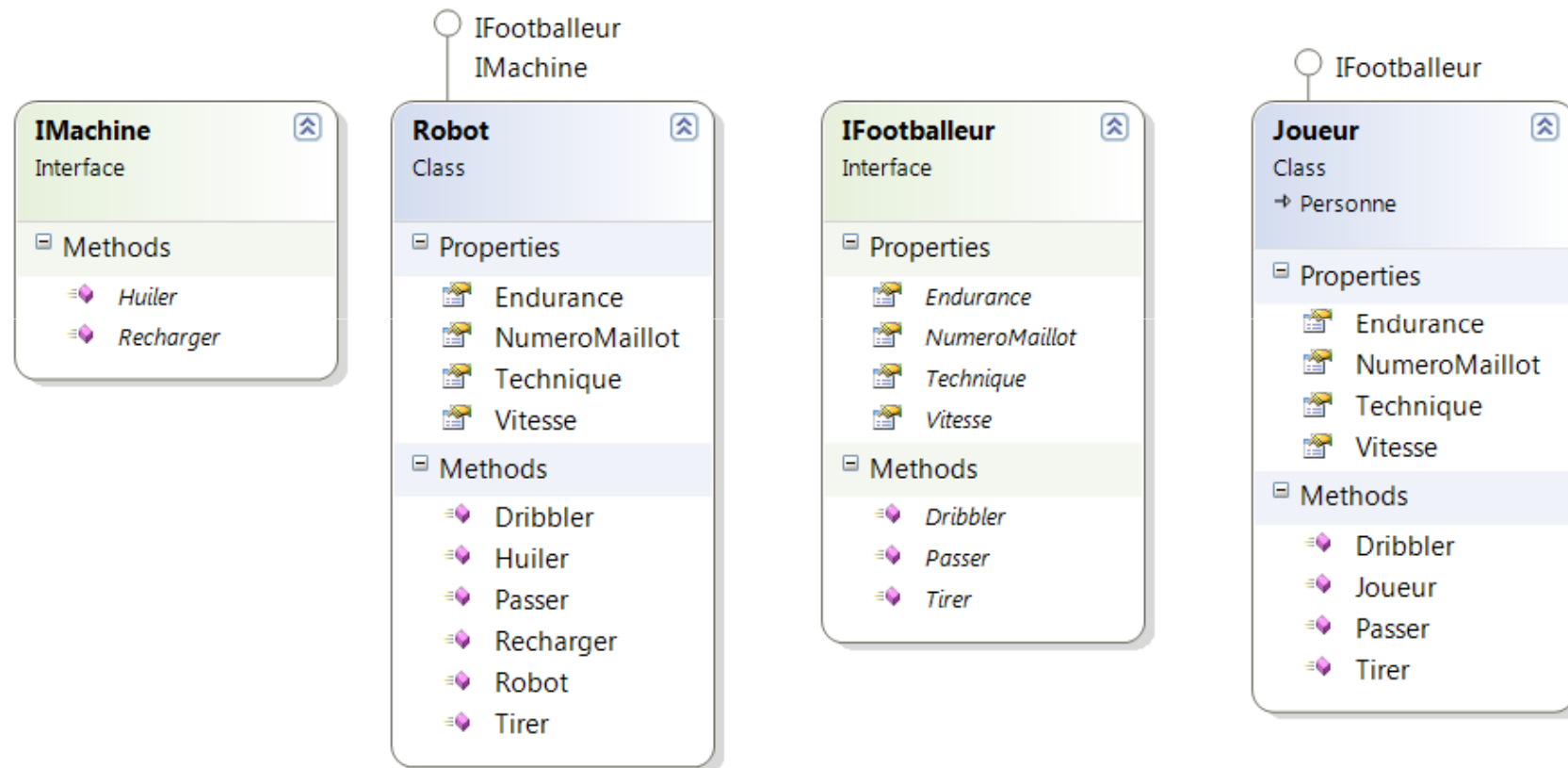


```
file:///C:/Users/Ben/Documents/Visual Studio 2010/Pr...
ConsoleApplication3.Joueur - hop hop hop
ConsoleApplication3.Joueur - hop hop hop
ConsoleApplication3.Joueur - hop hop hop
ConsoleApplication3.Robot - Schling schling
ConsoleApplication3.Robot - Schling schling
```

(c) Benoit Chauvet 2013

# Interfaces - exemple

130



# Conversions

131

- ❑ Conversion en référence de classe de base
  - Classe dérivée vers classe de base
  - (Vehicule)maVoiture
- ❑ Conversion en référence de classe dérivée
  - Classe de base vers classe dérivée
  - (Voiture)monVehicule
  - Fonctionne seulement si monVehicule contient une instance de Voiture, InvalidCastException sinon

# Délégués

132

- ❑ Le mécanisme de délégué permet de passer en paramètre non pas un type, mais une méthode.
- ❑ Un délégué représente la définition du paramètre de méthode. On le déclare avec le mot clé *delegate*
- ❑ Par exemple, on peut utiliser un délégué pour trier un tableau de Voitures en spécifiant la méthode de comparaison que l'on souhaite utiliser (par longueur ou par cylindrée).
- ❑ 3 étapes :
  - Déclaration du délégué
  - Création des méthodes qui respectent la signature du délégué
  - Utilisation du délégué dans la méthode de tri

# Délégués - exemple

133

```
// 1 - Déclaration du délégué :
public delegate int comparaison(Voiture a, Voiture b);

// 2 - Création des méthodes de comparaison
public static int ComparerLongueur(Voiture a, Voiture b)
{
    if (a.longueur < b.longueur)
    {
        return -1;
    }
    else
    {
        return 1;
    }
}

public static int ComparerCylindree(Voiture a, Voiture b)
{
    if (a.cylindree < b.cylindree)
    {
        return -1;
    }
    else
    {
        return 1;
    }
}
```

# Délégués - exemple

134

```
// 3 - Méthode de tri :
public static void TrierVoitures(Voiture[] voitures, comparaison comparateur)
{
    Voiture v;
    bool permut;

    do{
        permut = false;
        for (int i=0; i < voitures.Length - 1; i++)
        {
            // appel du délégué pour la comparaison :
            if (comparateur(voitures[i], voitures[i+1]) < 0)
            {
                v = voitures[i];
                voitures[i] = voitures[i+1];
                voitures[i+1] = v;
                permut = true;
            }
        }
    } while (permut == true);
}
```

# Délégués - exemple

135

```
// 4 - utilisation :
public static void ExempleDelegate()
{
    Voiture[] voitures = new Voiture[5];
    voitures[0] = new Voiture(400, 200, "Mercedes", 3200);
    voitures[1] = new Voiture(300, 200, "Porsche", 4000);
    voitures[2] = new Voiture(600, 200, "Ferrari", 6000);
    voitures[3] = new Voiture(250, 200, "Fiat", 1200);
    voitures[4] = new Voiture(330, 200, "Renault", 1100);
    // tri par longueur :
    TrierVoitures(voitures, ComparerLongueur);
    // tri par cylindree :
    TrierVoitures(voitures, ComparerCylindree);
}
```



# Événements

136

- ❑ Pour interagir avec des objets, on peut appeler leurs méthodes, mais on peut avoir besoin de d'être « notifié » d'un changement par l'objet lui-même
- ❑ Par exemple, un objet thermomètre met à jour en interne sa propriété température, et on crée un deuxième objet affichage température qui affiche la température actuelle du thermomètre
- ❑ Problématique : pour que l'affichage soit correct, l'affichage température doit être informé en temps réel des variations captées par le thermomètre

# Événements

137

- ❑ Pour répondre à ce besoin, C# introduit le mécanisme d'événement
- ❑ On retrouve notamment ce mécanisme dans les applications graphiques (winforms...)
- ❑ C'est le principe d'un « abonnement » : l'objet qui désire être notifié (affichage température) se déclare auprès de l'objet observé (thermomètre) en lui fournissant une méthode à appeler (sous forme de délégué) quand un changement survient (déclenchement d'un événement)

# Événements

138

- ❑ Création du gestionnaire d'événements :
  - Dans la classe thermomètre, déclaration d'un délégué de type « EventHandler » avec signature spécifique :
  - **public event EventHandler(Object sender, EventArgs e) temperatureChangee;**
  - event : delegate spécial pour les événements
  - Object sender : objet qui déclenche l'événement (Thermomètre)
  - EventArgs e : informations spécifiques à fournir à l'observateur (Affichage température). Si pas d'infos, utilisation du type EventArgs, sinon, création d'une classe héritée de EventArgs pour stocker les infos à notifier
  - temperatureChangee : nom de l'événement

# Événements

139

- ❑ Création d'un EventArgs spécifique :
  - La température est un membre privé de la classe thermomètre.
  - On veut la transmettre à l'affichage température via un EventArgs personnalisé : *ChangementTemperatureEventArgs*, qui contient une propriété *Temperature*
  - La déclaration du gestionnaire d'événement devient :
  - `public event EventHandler(Object sender, ChangementTemperatureEventArgs e) temperatureChangee;`

# Événements

140

- ❑ Dans la classe Thermomètre, nous pouvons maintenant déclencher l'événement quand la température change, seulement s'il y a un gestionnaire actif (inscription de l'affichage température) :

```
public void ModifierTemperature(int variation)
{
    this.temperature += variation;

    ChangementTemperatureEventArgs e = new
        ChangementTemperatureEventArgs(this.temperature);
    if (temperatureChangee != null)
    {
        temperatureChangee(this, e);
    }
}
```

# Événements

141

- ❑ Dans la classe Affichage température, on définit une méthode destinée à traiter l'événement :

```
public void AfficherTemperature(object sender,  
    ChangementTemperatureEventArgs e)  
{  
    Console.WriteLine("La température est de "  
        + e.Temperature + "°c");  
}
```

# Événements

142

- ❑ Il reste à connecter le gestionnaire d'événement à la méthode d'affichage (création de « l'abonnement ») :

```
Thermometre thermo = new Thermometre();  
AffichageTemperature affich = new AffichageTemperature();  
thermo.temperatureChangee += new  
    EventHandler<ChangementTemperatureEventArgs>  
    (affich.AfficherTemperature);
```

- ❑ A partir de là, l'appel à `thermo.ModifierTemperature()` déclenchera un événement et la méthode d'affichage sera appelée automatiquement

# Événements - Exemple

143

```
public class ChangementTemperatureEventArgs : EventArgs
{
    int temperature;

    public int Temperature
    {
        get { return temperature; }
    }

    public ChangementTemperatureEventArgs(int temp)
    {
        temperature = temp;
    }
}
```



# Événements - exemple

144

```
public class Thermometre
{
    // Déclaration de l'événement :
    public event EventHandler<ChangementTemperatureEventArgs> temperatureChangee;

    private int temperature = 0;

    public void ModifierTemperature(int variation)
    {
        this.temperature += variation;
        // Création de l'event args qui contient la température à jour :
        ChangementTemperatureEventArgs e = new ChangementTemperatureEventArgs(this.temperature);

        // Déclenche l'événement seulement s'il y a un handler pour le traiter :
        if (temperatureChangee != null)
        {
            temperatureChangee(this, e);
        }
    }
}
```

# Événements - exemple

145

```
public class AffichageTemperature
{
    // traitement de l'événement :
    public void AfficherTemperature(object sender, ChangementTemperatureEventArgs e)
    {
        Console.WriteLine("La température est de " + e.Temperature + "°c");
    }
}

static void Main(string[] args)
{
    Thermometre thermo = new Thermometre();

    AffichageTemperature affich = new AffichageTemperature();

    thermo.temperatureChangee += new EventHandler
        <ChangementTemperatureEventArgs>(affich.AfficherTemperature);

    thermo.ModifierTemperature(2);
}
```

# Attributs

146

- ❑ Les attributs sont des informations complémentaires (metadata) aux éléments du code
- ❑ Stockés dans les métadonnées de l'assembly et utilisés par le runtime.
- ❑ Syntaxe : [attribut(paramètres)]
- ❑ Quelques attributs courants :
  - [Serializable()] : indique qu'une classe peut être sérialisée
  - [Obsolete("message", true)] : indique qu'une méthode est obsolète, avec message d'avertissement ou d'erreur à la compilation (selon 2eme paramètre)
  - [WebMethod()] : marque une méthode de service web

```
// Déclenchera un message d'alerte à la compilation si cette fonction est appelée dans le code
[Obsolete("Utiliser MaNouvelleMethode() à la place", false)]
public void MaMethode()
{
    //...
}
```

147

# Objets et classes de base

Traitement des dates et durées

Les classes Math et Random

Listes et collections

Généricité

IEnumerable et IEnumerator

Indexeurs

Présentation de LINQ to Object

# Traitement des dates et durées

148

## ❑ DateTime

- Stockage des dates / heures
- Propriété DateTime.Now : donne la date du jour
- Méthodes intégrées (AddDay(), AddMonth(),...)

## ❑ TimeSpan

- Intervalle de temps
- Résultat d'opérations entre dates (ex : Subtract...)
- Propriétés d'extraction (Ticks, Seconds, Days...)

# Classes Math et Random

149

- ❑ Classe Math : classe static pour opérations mathématiques
  - Math.PI, Math.Cos(), Math.Round(), Math.Abs(), etc...
- ❑ Classe Random : Gestion de nombres aléatoires

```
Random deASixFaces = new Random();  
int resultat = deASixFaces.Next(1, 7);
```

# Collections - présentation

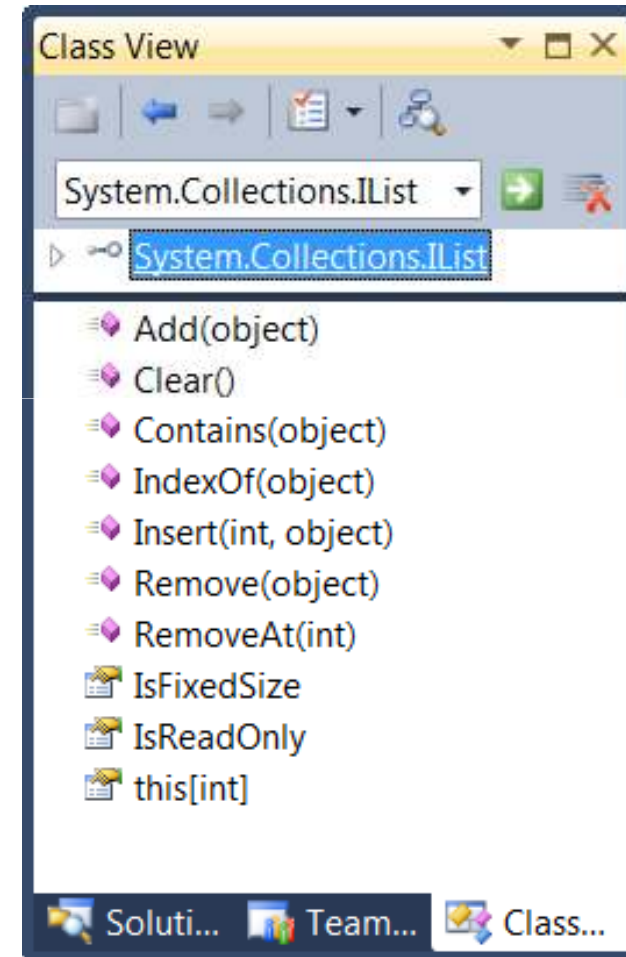
150

- ❑ En plus du type tableau, le framework .Net propose de nombreuses classes pour manipuler des listes ou collections d'objets, avec des fonctionnalités avancées pour les gérer
- ❑ Namespace System.Collections
- ❑ Différents types de collections (Array, ArrayList, List, Hashtable, Dictionary, Queue, Stack) chacune ayant des fonctionnalités spécifiques.
- ❑ Implémentent les interfaces IList ou IDictionary qui leur donnent des fonctionnalités communes
- ❑ Le namespace System.Collections.Specialized propose des collections pour couvrir des besoins spécifiques (NameValueCollection, OrderedDictionary, ...)

# Interface IList

151

- ❑ Gestion des collections ordonnées
- ❑ Implémentée par :
  - Array
  - ArrayList
  - StringCollection
  - TreeNodeCollection

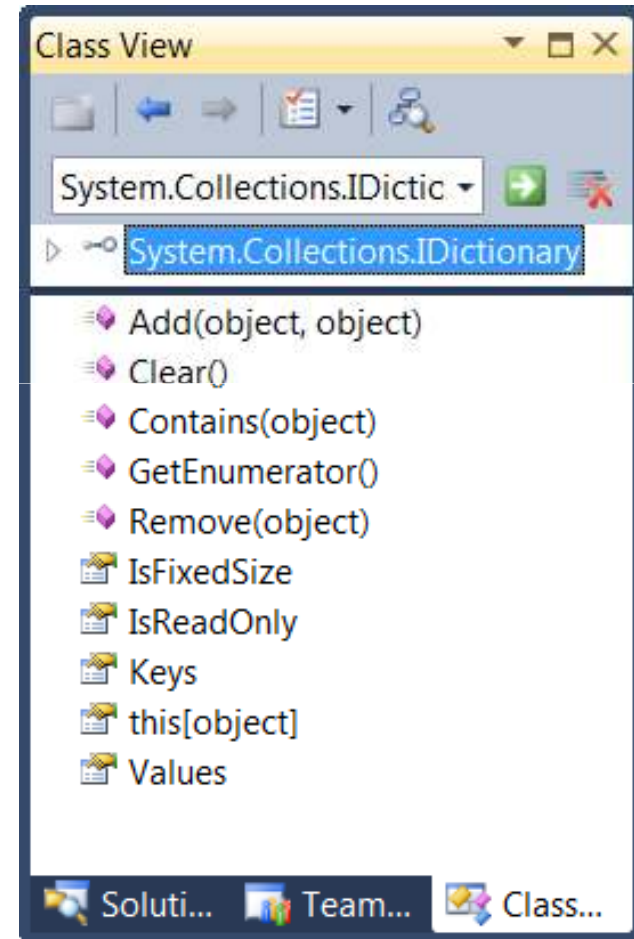




# Interface IDictionary

152

- ❑ Gestion de collections de type clé/valeur
- ❑ Implémentée par
  - Hashtable
  - DictionaryBase
  - SortedList



# Principales Collections

153

## ❑ System.Array

- Tableaux de taille fixe
- Plusieurs dimensions possibles
- Length : taille du tableau
- Rank : nombre de dimensions

## ❑ ArrayList

- Taille dynamique
- Une seule dimension
- Méthodes d'ajout/insertion/suppression de plusieurs éléments simultanément (AddRange(), ...)

# Principales Collections

154

## ❑ Hashtable

- Couples clé/valeur
- Basée sur le code de Hashage de la clé (`Object.GetHashCode()`);

## ❑ Queue

- File d'attente (FiFo)
- Méthodes Enqueue, Dequeue et Peek

## ❑ Stack

- Pile (LiFo)
- Méthodes Push, Pop, Peek

# Généricité

155

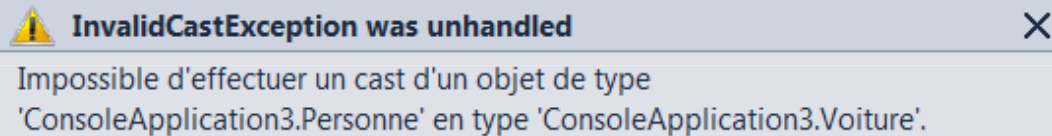
- ❑ Pour pouvoir gérer tout type de contenu, les collections de `System.Collection` gèrent des éléments de type `Object` : principe de généralisation
  - Conversion explicite nécessaire pour utiliser les éléments
  - Pas de contrôle sur le type de contenu (« liaison tardive »)
- ❑ Le framework 2.0 introduit la notion de générique pour définir au moment de la déclaration le type d'objets contenus dans la liste :
  - Ex : `List<Personne> maListe = new List<Personne>();`
  - Namespace `System.Collections.Generic`
  - Plus besoin de conversion explicite
  - Le type de contenu est contrôlé

# Généricité - Exemple

156

```
// Généralisation :  
ArrayList al = new ArrayList();  
al.Add(new Personne());  
// cast pour récupérer le type Personne :  
Personne p = (Personne)al[0];  
// pas de contrôle à la compilation, mais erreur à l'exécution :
```

```
Voiture v = (Voiture)al[0];
```



```
//Généricité :  
List<Personne> l = new List<Personne>();  
l.Add(new Personne());  
// pas besoin de cast :  
Personne p = l[0];  
// contrôle à la compilation :  
Voiture v = l[0];
```

# Création d'une classe générique

157

- ❑ Au-delà des collections génériques, il est possible de créer ses propres classes génériques
- ❑ On peut également créer de méthodes génériques (voir suite du cours), ainsi que des interfaces et des délégués génériques (non traité dans ce cours)
- ❑ Objectif : Rendre le code indépendant du type d'objets manipulés, tout en maintenant un contrôle optimal à la compilation
- ❑ Exemple : création d'une classe équipe de foot générique qui pourra gérer une équipe d'un type au choix (joueur ou robot)

# Création d'une classe générique

158

```
// Déclaration de la classe générique :
public class EquipeDeFoot<typeEquipier>
{
    private string nom;
    private ArrayList equipe = new ArrayList();

    public EquipeDeFoot(string nom)
    {
        this.nom = nom;
    }

    public void Ajouter(typeEquipier e)
    {
        equipe.Add(e);
    }
}
```

# Utilisation d'une classe générique

159

```
public class GeneriqueProgram
{
    static void Main(string[] args)
    {
        EquipeDeFoot<Joueur> equipeHumaine = new EquipeDeFoot<Joueur>("C# United");

        for (int i = 1; i < 12; i++)
        {
            equipeHumaine.Ajouter(new Joueur(i));
        }

        EquipeDeFoot<Robot> equipeTerminator = new EquipeDeFoot<Robot>("Empire Clones");

        for (int i = 1; i < 12; i++)
        {
            equipeTerminator.Ajouter(new Robot(i));
        }
    }
}
```



# Méthodes génériques

160

- ❑ On peut aussi créer des méthodes génériques, même dans une classe non générique

```
public static void Echanger<typeDonnee>(ref typeDonnee a, ref typeDonnee b)
{
    typeDonnee o;
    o = a;
    a = b;
    b = o;
}
```

```
static void Main(string[] args)
{
    Voiture v1 = new Voiture(300, 150, "Renault", 1200);
    Voiture v2 = new Voiture(400, 200, "Peugeot", 1600);
    Echanger(ref v1, ref v2);

    Personne p1 = new Personne();
    Personne p2 = new Personne();
    Echanger(ref p1, ref p2);
}
```

# Contraintes génériques

161

- ❑ Il est possible de spécifier des contraintes sur les types de données acceptées par un générique
- ❑ Where typeDonnee : struct
  - Impose que le paramètre soit de type valeur et non nullable
- ❑ Where typeDonnee : class
  - Impose que le paramètre soit de type référence (classe, interface, tableau ou délégué)
- ❑ Where typeDonnee : nom de classe
  - Impose que le paramètre soit du type de la classe indiquée, ou une de ses sous classes
- ❑ Where typeDonnee : interface1 [, interface 2...]
  - Impose que le paramètre implémente la ou les interfaces indiquées
- ❑ Where typeDonnee : new()
  - Impose que le paramètre possède un constructeur public et sans paramètre (doit être en dernier si plusieurs contraintes)
- ❑ Cumul possible de plusieurs contraintes

# Contraintes génériques

162

```
// Déclaration de la classe générique :
public class EquipeDeFoot<typeEquipier>
    where typeEquipier : class, IFootballeur, new()
{
    private string nom;
    private ArrayList equipe = new ArrayList();

    public EquipeDeFoot(string nom)
    {
        this.nom = nom;

        for (int i = 1; i < 12; i++)
        {
            // la contrainte new() permet l'instanciation :
            typeEquipier e = new typeEquipier();
            // la contrainte IFootballeur permet d'utiliser les membres d'interface :
            e.NumeroMaillot = i;
            equipe.Add(e);
        }
    }

    public void Ajouter(typeEquipier e)
    {
        equipe.Add(e);
    }
}
```

# Variance et classes génériques

163

## ❑ Contravariance :

- Le paramètre fourni est un sur-type du type attendu
- Utilisation de la classe Personne là où la classe Joueur est attendue
- Non autorisé par les classes génériques

## ❑ Covariance

- Le paramètre fourni est un sous-type du type attendu
- Utilisation de la classe Joueur là où la classe Personne est attendue
- Autorisé par les classes génériques

# Variance et classes génériques

164

```
public class Variance<typeDonnee>
{
    public void Afficher(typeDonnee a)
    {
        Console.WriteLine(a.ToString());
    }
}

Personne p = new Personne("toto", DateTime.Now);
Joueur j = new Joueur();

// contravariance
Variance<Joueur> contravariance = new Variance<Joueur>();
contravariance.Afficher(p); // erreur de compilation

// covariance
Variance<Personne> covariance = new Variance<Personne>();
covariance.Afficher(j); // OK
```

# Interfaces IEnumerable et IEnumerator

165

## ❑ Interface IEnumerable

- Méthode GetEnumerator() qui renvoie un IEnumerator
- Permet d'énumérer le contenu d'une collection
- La méthode GetEnumerator() est appelée par l'instruction foreach

## ❑ Interface IEnumerator

- Contrôle du déplacement dans l'énumérateur
- Méthodes MoveNext() et Reset() : déplacement / raz du curseur
- Propriété Current : élément en cours

# IEnumerable : mise en œuvre

166

```
// implémente l'interface IEnumerable :  
// Génération automatique des signatures de méthodes  
public class EquipeDeFoot : IEnumerable  
{  
    private string nom;  
    private ArrayList equipe = new ArrayList();  
  
    IEnumerator IEnumerable.GetEnumerator()  
    {  
        return new EquipeEnumerator(this.equipe);  
    }  
}
```

# IEnumerator : mise en œuvre (1)

167

```
public class EquipeEnumerator : IEnumerator
{
    private ArrayList equipe;
    private int currentIndex;

    public EquipeEnumerator(ArrayList equipe)
    {
        this.equipe = equipe;
        currentIndex = -1;
    }

    object IEnumerator.Current
    {
        get { return equipe[currentIndex]; }
    }
}
```



# IEnumerator : mise en œuvre (2)

168

```
bool IEnumerator.MoveNext()
{
    bool result = false;
    if (currentIndex < equipe.Count - 1)
    {
        currentIndex++;
        result = true;
    }
    return result;
}

void IEnumerator.Reset()
{
    currentIndex = -1;
}
}
```

# Indexeurs

169

- ❑ Les indexeurs permettent de donner accès à des propriétés d'une classe à la façon d'un tableau
- ❑ Par exemple, l'accès aux joueurs d'une équipe en passant par un index : `equipe[8]`, ...
- ❑ Se définissent comme une propriété, nommée *this*
- ❑ Possibilité d'utiliser des index non numériques
- ❑ Possibilité d'avoir plusieurs indexeurs pour une même classe (le type d'index doit être différent)

# Indexeurs - création

170

```
public class EquipeDeFoot
{
    private List<Joueur> equipe = new List<Joueur>();

    public Joueur this[int index]
    {
        get { return equipe[index]; }
        set { equipe[index] = value; }
    }

    // renvoie la liste des joueurs pour un poste donné :
    public List<Joueur> this[string poste]
    {
        get { return equipe.Where(j => j.Poste == poste).ToList<Joueur>(); }
    }

    public void Acheter(Joueur j)
    {
        equipe.Add(j);
    }
}
```

# Indexeurs - Utilisation

171

```
static void Main(string[] args)
{
    EquipeDeFoot psg = new EquipeDeFoot();

    psg.Acheter(new Joueur("Zlatan", 10, "attaquant"));
    psg.Acheter(new Joueur("David", 18, "attaquant"));
    psg.Acheter(new Joueur("Salvatore", 1, "gardien"));

    // indexeur pour lecture :
    Joueur j = psg[0]; // Zlatan
    // indexeur pour ecriture :
    psg[0] = new Joueur("Cristiano", 10, "attaquant");

    // indexeur par nom de poste :
    List<Joueur> attaquants = psg["attaquant"]; // Cristiano, David
```

# LINQ - Présentation

172

- ❑ Language Integrated Query
- ❑ Extension du langage C# pour manipuler des données (objets) avec une syntaxe proche du SQL
- ❑ Domaines d'application :
  - **LINQ To Object** : manipulation d'objets
  - **LINQ To XML** : manipulation de fichiers XML
  - **LINQ To SQL** : manipulation de bases de données
  - **LINQ To Entities** : manipulation d'entités
- ❑ Fonctionnalités de C# utilisées par LINQ :
  - Typage implicite des variables
  - Initialisation d'instances à la déclaration
  - Les méthodes d'extension
  - Types anonymes
  - Lambda Expressions

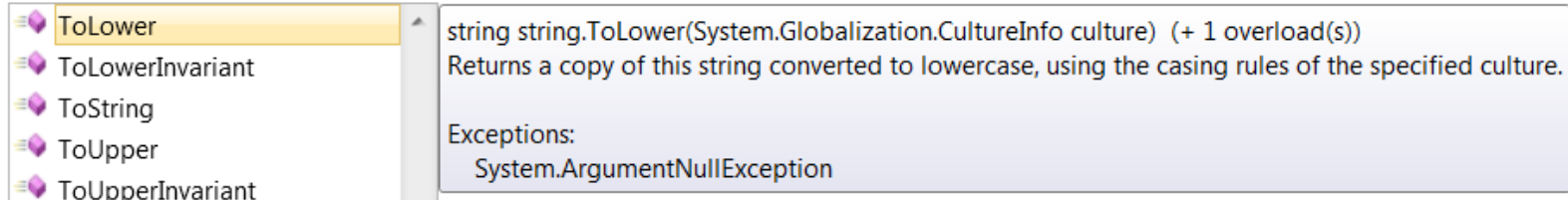
# Typage implicite des variables

173

- ❑ « Inférence de type »
- ❑ Déclaration de variables avec le mot clé *var*
- ❑ La variable doit obligatoirement être initialisée à la déclaration
- ❑ Seulement pour des variables locales
- ❑ C'est la valeur donnée qui indique le type de la variable
- ❑ Évalué à la compilation (contrairement au type *dynamic*, qui est évalué à l'exécution)

```
var x = "hello";
```

x.



The screenshot shows an IDE with a list of string methods on the left and a tooltip for the `ToLower` method on the right. The list includes `ToLower`, `ToLowerInvariant`, `ToString`, `ToUpper`, and `ToUpperInvariant`. The tooltip for `ToLower` shows the signature `string string.ToLower(System.Globalization.CultureInfo culture) (+ 1 overload(s))`, a description "Returns a copy of this string converted to lowercase, using the casing rules of the specified culture.", and an exception `System.ArgumentNullException`.

# Initialisation à la déclaration

174

- ❑ Syntaxe permettant de créer une instance et d'initialiser ses propriétés en une seule instruction

```
// initialisation classique :
```

```
Personne p = new Personne();
```

```
p.DateNaissance = new DateTime(1979, 06, 27);
```

```
p.Nom = "Bob" ;
```

```
// initialisation à la volée :
```

```
Personne p = new Personne() { DateNaissance = new DateTime(1979, 06, 27), Nom = "Bob" };
```

# Méthodes d'extension

175

- ❑ Permettent d'ajouter des méthodes à un type existant sans en modifier la définition
- ❑ Méthodes static et mot clé this devant le premier paramètre
- ❑ Appelées sur une instance du type

```
public static class IntExtension
{
    public static int Puissance(this int valeur, int puiss)
    {
        int result = valeur;
        for (int i = 0; i < puiss - 1; i++ )
        {
            result = result * valeur;
        }
        return result;
    }
}
```

```
int i = 3;
Console.WriteLine(i.Puissance(3));
```

i.

- Equals
- GetHashCode
- GetType
- GetTypeCode
- Puissance**
- ToString

(extension) int int.Puissance(int puiss)



# Types anonymes

176

- ❑ Définition d'une classe à la volée, à la création de l'instance
- ❑ Utilise l'inférence de type
- ❑ Restreint aux variables locales

```
var entite = new { Name = "Lilianne B.", Numero = 123456, Solde = 10000000.00f };
```

# Lambda Expressions

177

- ❑ Méthodes anonymes, utilisées à la place d'un délégué
- ❑ Implémentation d'un traitement à la volée
- ❑ Signe `=>` pour la déclaration

```
List<Personne> groupe = new List<Personne>();  
// ...  
List<Personne> sousGroupe = groupe.FindAll(p => p.Nom.StartsWith("Ch"));  
  
bool EstFeminin = groupe.TrueForAll(p => p.Sexe == "F");  
  
List<Personne> groupeMineurs = groupe.Where(p => p.Age < 18).ToList<Personne>();  
List<Personne> groupeRetraites = groupe.Where(p => p.Age > 60).ToList<Personne>();  
List<Personne> groupeReduc = groupeMineurs.Union(groupeRetraites).ToList<Personne>();
```

# Opérateurs LINQ

178

- ❑ Très proches du SQL :
  - **Source** : from *element* in *sourceDeDonnées*
  - **Conditions** : where *condition(s)*
  - **Tri** : Order by *champ(s)*
  - **Champs** : select *champ(s)*

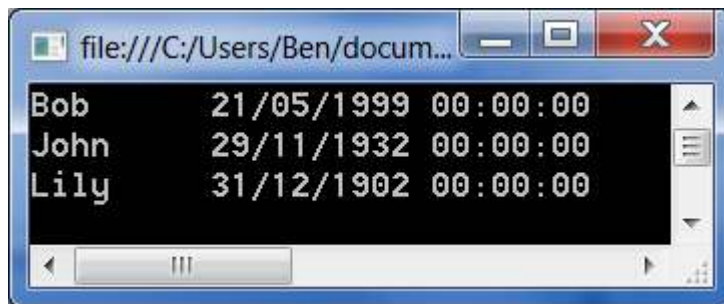
# LINQ to Object - Exemple

179

```
List<Personne> groupe = new List<Personne>()
{
    new Personne("Bob", new DateTime(1999, 05, 21)),
    new Personne("John", new DateTime(1932, 11, 29)),
    new Personne("Agatha", new DateTime(1980, 02, 12)),
    new Personne("Lily", new DateTime(1902, 12, 31))
};

var liste = from pers in groupe
            where pers.Age < 18 | pers.Age > 60
            orderby pers.Nom
            select new { pers.Nom, pers.DateNaissance };

liste.ToList().ForEach(p => Console.WriteLine(p.Nom + " \t " + p.DateNaissance));
```



(c) Benoit Chauvet 2013

180

# Accès aux données

ADO.NET

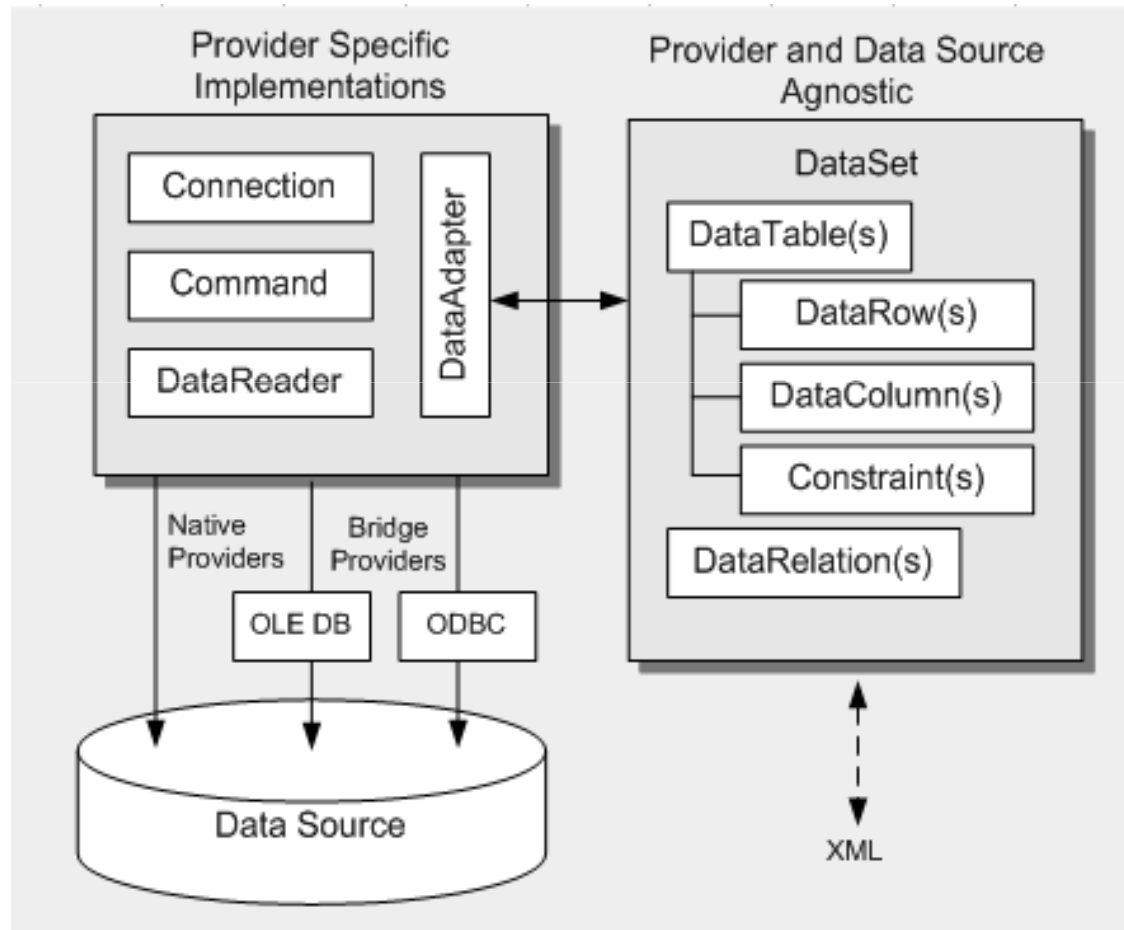
LINQ to SQL

LINQ to Entities

# ADO.NET - Présentation

181

- ❑ ADO.net :  
ensemble de composants du framework pour la manipulation de données
- ❑ Namespace System.Data



# ADO.NET – Fournisseurs de données

182

- ❑ Fournisseurs disponibles dans le framework :
  - Fournisseur SQL Serveur – (SQL)
  - Fournisseur OLE DB – (OLEDB)
  - Fournisseur ODBC – (ODBC)
- ❑ Fournisseurs tiers :
  - Oracle Data Provider (ODP)
  - MySql...
  - Assemblies téléchargeables chez les tiers
- ❑ Proposent des implémentations d'ADO.NET basées sur les mêmes objets de base (préfixés).
  - SQL : `System.Data.SqlClient`
  - ODBC : `System.Data.Odbc`
  - OLE DB : `System.Data.OleDb`
  - Oracle : `System.Data.OracleClient`

# Connexion à une base de données

183

- ❑ **Objet Connection** : gère la connexion à la base à partir d'une chaîne de connexion
  - Chaîne de connexion (propriété `ConnectionString`)
  - Méthodes `Open ()` & `Close()`

```
// using System.Data.SqlClient;

// Création d'une connexion
SqlConnection cnx = new SqlConnection();
// Affectation d'une chaîne de connexion :
cnx.ConnectionString = "Data Source=localhost;Initial Catalog=Market;Integrated Security=True";
// Ouverture de la connexion :
cnx.Open();
// ...
// échanges avec la base de données
// ...
// Fermeture de la connexion :
cnx.Close();
```



# Chaînes de connexion

184

- ❑ Représente les paramètres de connexion à la base, sous forme couples clé/valeur, stockés dans une string
- ❑ Gestion intégrée de pool de connexions
- ❑ Paramètres courants :
  - **Data source** : ip ou nom du serveur de données
  - **Initial catalog** : nom de la base de données
  - **Integrated security** :
    - true : utilisation du compte windows pour l'authentification
    - false : nom d'utilisateur et mot de passe à fournir
  - **User ID** : nom d'utilisateur (integrated security = false)
  - **Pwd** : mot de passe (integrated security = false)

# Chaînes de connexion

185

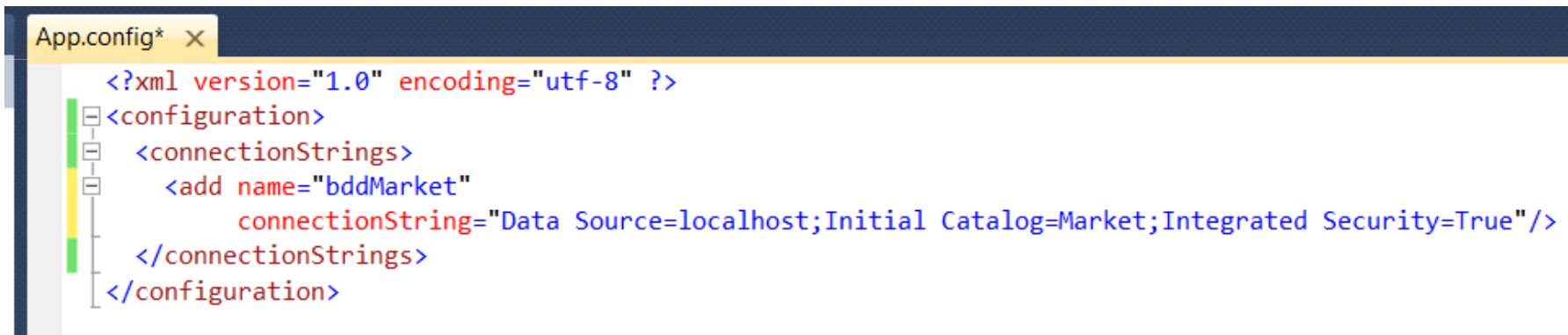
## ❑ Paramètres courants (suite)

- **Connect Timeout** : durée en secondes d'attente de réponse du serveur à la demande de connexion.
- **Connection LifeTime** : durée de vie d'une connection dans un pool de connections (0 = infini)
- **Max Pool Size** : nombre max de connections dans le pool
- **Min Pool Size** : nombre min de connections dans le pool
- **Connection Reset** : indique si la connection est réinitialisée lors de sa remise dans le pool
- **Pooling** : indique si la connection peut être extraite d'un pool

# Chaînes de connexion

186

- ❑ Dans une application .net, on stocke les chaînes de connexion dans un fichier de configuration (app.config)
- ❑ Pour l'utiliser, il faut ajouter dans le projet une référence à l'assembly System.Configuration



```
App.config* X
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="bddMarket"
          connectionString="Data Source=localhost;Initial Catalog=Market;Integrated Security=True"/>
  </connectionStrings>
</configuration>
```

```
// Affectation d'une chaîne de connexion :
cnx.ConnectionString = System.Configuration.ConfigurationManager
                        .ConnectionStrings["bddMarket"].ConnectionString ;
```

# Mode connecté

187

- ❑ Ouverture d'une connexion à chaque lecture et mise à jour de la base
- ❑ Lecture basée sur le DataReader
- ❑ SqlDataReader, OdbcDataReader, OracleDataReader....
- ❑ Envoi d'une commande à la base
- ❑ Objet Command : Gère la commande envoyée à la base de données (requête, procédure stockée...) et les éventuels paramètres associés

# Objet Command

188

- ❑ Propriété **Connection** : connexion associée
- ❑ Propriété **CommandText** : texte de la commande
- ❑ Propriété **CommandType**
  - **Text** : requête (par défaut)
  - **StoredProcedure** : procédure stockée
- ❑ Méthodes d'exécution :
  - **ExecuteScalar** : Requête renvoyant une seule donnée (ex :  
SELECT COUNT(\*) FROM CLIENTS)
  - **ExecuteReader** : Requête renvoyant un jeu de résultats dans  
un DataReader (ex : SELECT \* FROM CLIENTS)
  - **ExecuteNonQuery** : Requête de mise à jour (ex : DELETE  
FROM CLIENTS WHERE id = 1)

# Objet Command - Exemple

189

```
SqlCommand cmd = new SqlCommand();  
cmd.Connection = cnx;  
cmd.CommandText = "SELECT * FROM ARTICLES";  
cmd.CommandType = System.Data.CommandType.Text;  
// exécution de la commande :  
SqlDataReader dr = cmd.ExecuteReader();
```

```
SqlCommand cmd2 = new SqlCommand();  
cmd2.Connection = cnx;  
cmd2.CommandText = "SELECT COUNT(*) FROM ARTICLES";  
cmd2.CommandType = System.Data.CommandType.Text;  
// exécution de la commande :  
int nbArticles = (int)cmd.ExecuteScalar();
```

```
SqlCommand cmd3 = new SqlCommand();  
cmd3.Connection = cnx;  
cmd3.CommandText = "UPDATE ARTICLES SET prix = 20 WHERE id = 1";  
cmd3.CommandType = System.Data.CommandType.Text;  
// exécution de la commande :  
int nbLignesAffectees = (int)cmd.ExecuteNonQuery();
```

# Objet Command - paramètres

190

- ❑ Passage de paramètres à une commande :  
collection de SqlParameter

```
SqlCommand cmd3 = new SqlCommand();  
cmd3.Connection = cnx;  
cmd3.CommandText = "UPDATE ARTICLE SET prix = 20 WHERE id = @idArticle";  
// ajout d'un paramètre :  
SqlParameter p_idArticle = new SqlParameter("@idArticle", 1);  
cmd3.Parameters.Add(p_idArticle);  
  
cmd3.CommandType = System.Data.CommandType.Text;  
int nbLignesAffectees = (int)cmd3.ExecuteNonQuery();
```

# DataReader - Lecture

191

- ❑ Dans le cas du `ExecuteReader()`, on récupère un `DataReader`
- ❑ Méthode `Read()`
  - `true` si ligne lue
  - `false` si fin du jeu d'enregistrements
- ❑ Accès aux champs de l'enregistrement en cours par un indexeur (index ou libellé de la colonne)
- ❑ Méthode `Close()` : à appeler à la fin de la lecture



# DataReader - Exemple

192

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = cnx;
cmd.CommandText = "SELECT id, libelle, prix FROM ARTICLE";
cmd.CommandType = System.Data.CommandType.Text;
// exécution de la commande :
SqlDataReader dr = cmd.ExecuteReader();
while (dr.Read())
{
    string libelleArticle = (string)dr["libelle"]; // ou dr.GetString(1);
    int idArticle = (int)dr["id"]; // ou dr.GetInt32(0);
    float prixArticle = (float)dr["prix"]; // ou dr.GetFloat(2);
}
dr.Close();
```

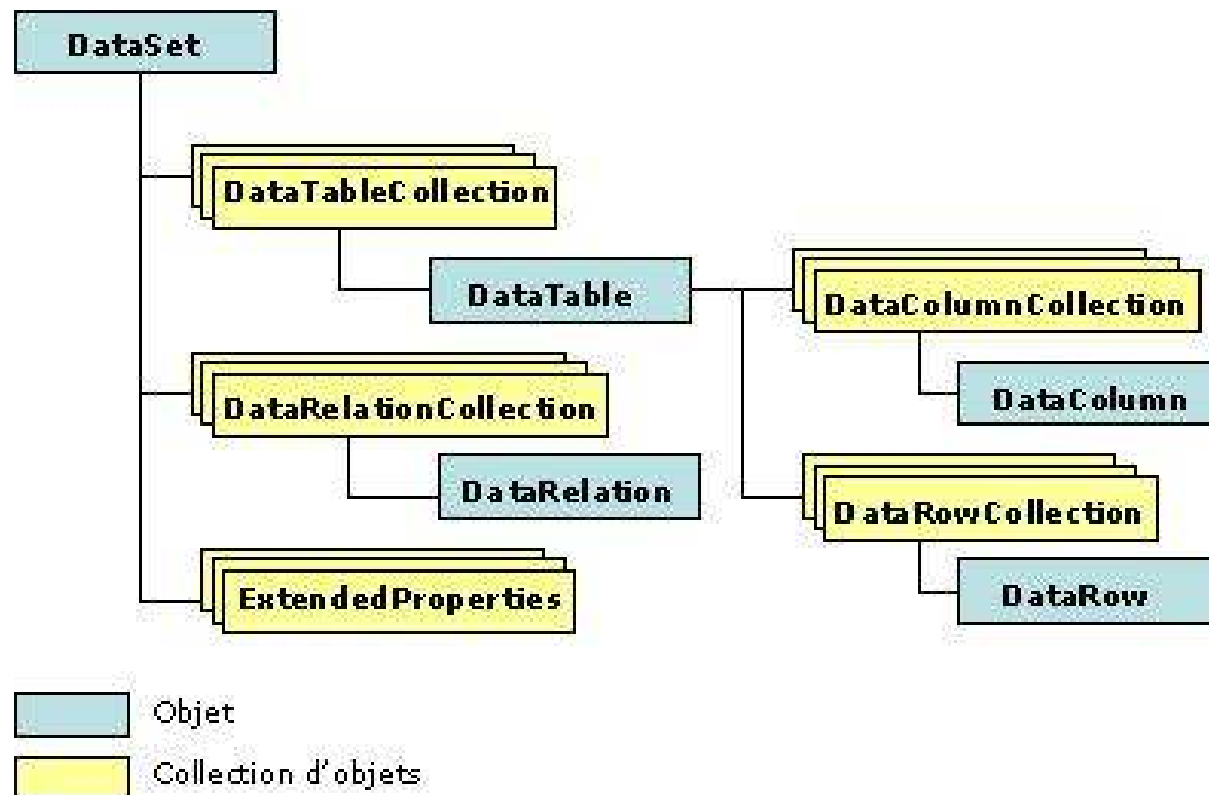
# Mode déconnecté – Le DataSet

193

- ❑ Utilisation de l'objet DataSet qui représente un extrait de la base de données
- ❑ Namespace System.Data
- ❑ Accès initial à la base pour créer le DataSet
- ❑ Hors connexion, modifications sur le DataSet
- ❑ Possibilité de persistance des DataSet (enregistrement sous forme de fichier)
- ❑ Synchronisation du DataSet avec la base de données
- ❑ Un DataSet peut être alimenté par d'autres moyens qu'une base de données

# DataSet

194



# DataSet - Chargement

195

- ❑ Pour alimenter un DataSet, on utilise l'objet DataAdapter
- ❑ Propriété SelectCommand : requête de sélection (objet Command)

```
// Création d'une connexion
SqlConnection cnx = new SqlConnection();
// Affectation d'une chaîne de connexion :
cnx.ConnectionString = System.Configuration.ConfigurationManager
                        .ConnectionStrings["bddMarket"].ConnectionString;

// création du DataAdapter :
SqlDataAdapter da = new SqlDataAdapter();
// Affectation de SelectCommand :
da.SelectCommand = new SqlCommand("SELECT id, libelle, prix FROM ARTICLE", cnx);
// Création d'un DataSet :
DataSet ds = new DataSet();
//Remplissage du DataSet
da.Fill(ds, "Article");
// récupération de la DataTable chargée dans le DataSet :
DataTable dt = ds.Tables["Article"];
```

# Modification d'un DataSet

196

Modifications sur le DataSet, en mode non connecté : à ce stade, la base de données n'est pas impactée

```
// Ajout d'un enregistrement :
DataRow dr = ds.Tables["Article"].NewRow();
dr["libelle"] = "nouvel article";
dr["prix"] = 12.5f;
ds.Tables["Article"].Rows.Add(dr);

// modification d'un enregistrement :
DataRow dr2 = ds.Tables["Article"].Rows[2];
dr2.BeginEdit();
dr2["Libelle"] = "lunettes de piscine";
dr2.EndEdit();

// suppression d'un enregistrement :
ds.Tables["Article"].Rows[1].Delete();
```

# DataAdapter - Update

197

- ❑ Propriété UpdateCommand pour répercuter les modifications :

```
SqlCommand ucmd = new SqlCommand();
ucmd.CommandText = "UPDATE ARTICLE SET libelle = @libelle, prix = @prix WHERE id = @id";
ucmd.Connection = cnx;
// ajout d'un paramètre :
ucmd.Parameters.Add(new SqlParameter("@libelle", SqlDbType.NVarChar, 50, "libelle"));
// variante 1 :
ucmd.Parameters.Add("@prix", SqlDbType.Float, 8, "prix");
// variante 2 :
SqlParameter p_id = new SqlParameter();
p_id.ParameterName = "@id";
p_id.SqlDbType = SqlDbType.Int;
p_id.Size = 4;
p_id.SourceColumn = "id";
ucmd.Parameters.Add(p_id);
// ajout de la commande au DataAdapter :
da.UpdateCommand = ucmd;
```

# DataAdapter - Insert

198

- ❑ Propriété InsertCommand pour répercuter les insertions :

```
SqlCommand icmd = new SqlCommand();  
icmd.CommandText = "INSERT INTO ARTICLE (libelle, prix) VALUES (@libelle, @prix)";  
icmd.Connection = cnx;  
// ajout d'un paramètre :  
icmd.Parameters.Add(new SqlParameter("@libelle", SqlDbType.NVarChar, 50, "libelle"));  
// variante 1 :  
icmd.Parameters.Add("@prix", SqlDbType.Float, 8, "prix");  
// ajout de la commande au DataAdapter :  
da.InsertCommand = icmd;
```

# DataAdapter - Delete

199

- ❑ Propriété DeleteCommand pour répercuter les suppressions :

```
SqlCommand dcmd = new SqlCommand();  
dcmd.CommandText = "DELETE FROM ARTICLE WHERE id = @id";  
dcmd.Connection = cnx;  
// ajout d'un paramètre :  
dcmd.Parameters.Add(new SqlParameter("@id", SqlDbType.Int, 4, "id"));  
// ajout de la commande au DataAdapter :  
da.DeleteCommand = dcmd;
```



# CommandBuilder

200

- ❑ Alternative à la création manuelle des commandes :  
objet `CommandBuilder` :

```
SqlCommandBuilder b = new SqlCommandBuilder(da);
```

```
da.InsertCommand = b.GetInsertCommand();
```

```
da.UpdateCommand = b.GetUpdateCommand();
```

```
da.DeleteCommand = b.GetDeleteCommand();
```

# Envoi des modifications à la base

201

- ❑ Méthode Update() du DataAdapter – *ne pas confondre avec UpdateCommand*
- ❑ Exécute les Insert, Update et Delete en fonction de l'état du DataSet
- ❑ Propriété RowState d'une DataRow :
  - Unchanged
  - Added
  - Deleted
  - Modified

```
// mise à jour de la base :  
da.Update(ds, "Article");
```

# Sauvegarde d'un DataSet

202

- ❑ Possibilité d'enregistrer un DataSet dans un fichier xml et de le charger à partir de ce fichier

```
// Enregistrement des données d'un dataset :  
ds.WriteXml(@"../../dataset/ds.xml");
```

```
// Chargement des données xml dans un dataset :  
DataSet ds2 = new DataSet();  
ds2.ReadXml(@"../../dataset/ds.xml");
```

# DataView

203

- ❑ Possibilité d'extraire différentes vues à partir d'un DataSet
- ❑ Plusieurs vues possibles pour une même table

```
DataTable dt = ds.Tables["Article"];  
// création du DataView :  
DataView dv = new DataView(dt);  
// Filtres :  
dv.RowFilter = "prix > 12";  
// Tri :  
dv.Sort = "libelle asc";  
// Lignes à afficher :  
dv.RowStateFilter = DataViewRowState.CurrentRows;
```

# Mapping Objet Relationnel (ORM)

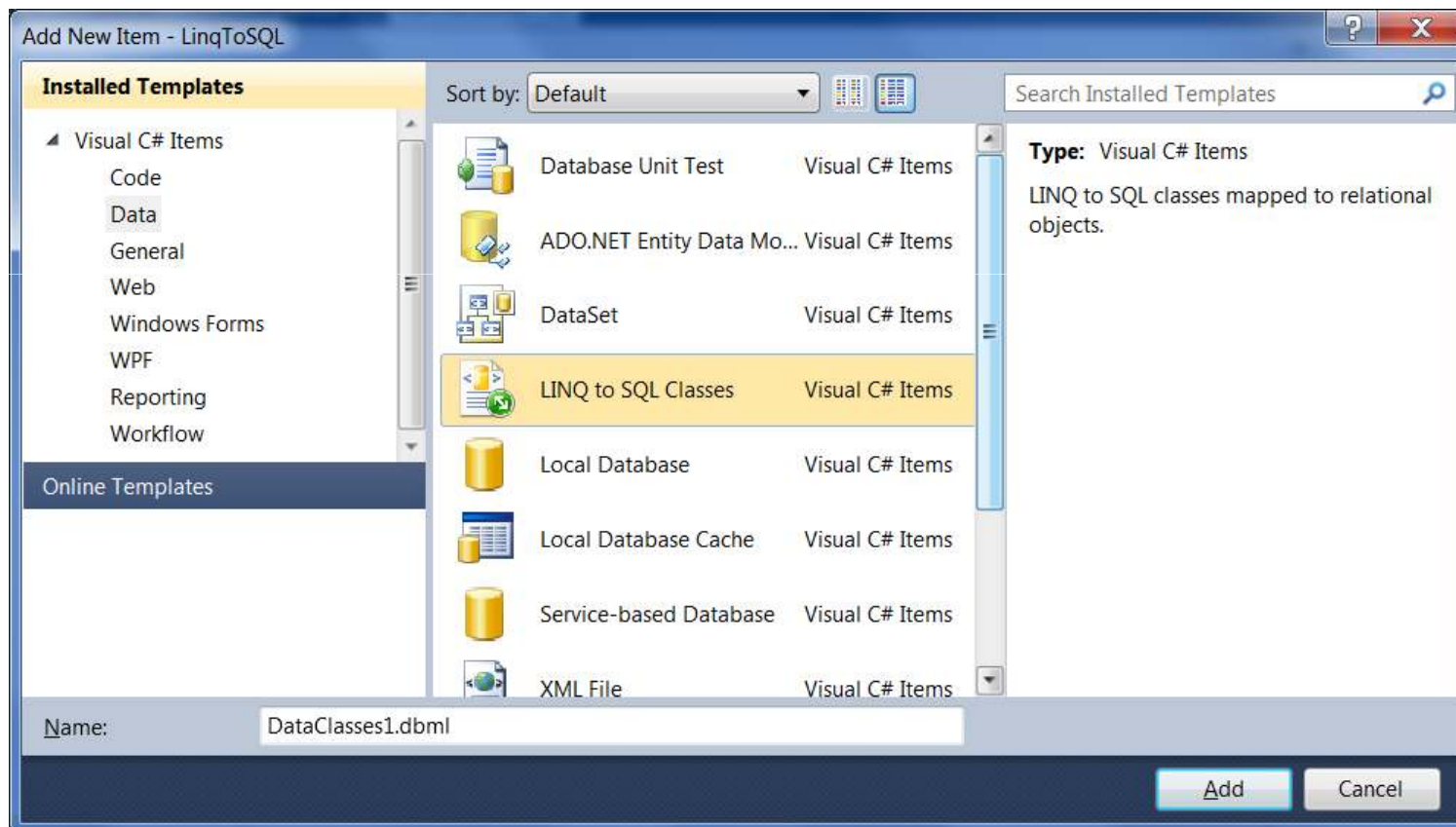
204

- ❑ Techniques de correspondance entre le monde objet et le monde relationnel (Bdd)
- ❑ Création d'objets qui représentent les données et se chargent des accès à la base de données
- ❑ CRUD : Create, Read, Update, Delete
- ❑ Deux approches sont proposées par le framework .NET :
  - LINQ to SQL
  - LINQ to Entities (Entity framework)

# LINQ to SQL - Création

205

- Ajout d'un élément LINQ to SQL à un projet



(c) Benoit Chauvet 2013

# LINQ to SQL - Création

206

Department

- Properties
  - DepartmentID
  - Name
  - Budget
  - StartDate
  - Administrator

Course

- Properties
  - CourseID
  - Title
  - Credits
  - DepartmentID

GetDepartmentName (System.Int32 iD, ref System.String name)

Ajout de classes(table) et de méthodes (procédures stockées) par le designer (Drag & drop à partir du server Explorer)

(c) Benoit Chauvet 2013

# LINQ to SQL - Utilisation

207

```
using (SchoolDataContext dc = new SchoolDataContext())
{
    // LECTURE DES DONNEES
    foreach (Course c in dc.Courses)
    {
        Console.Write(c.CourseID + " - " + c.Title);
        // APPEL DE PROCEDURE STOCKEE :
        string dptName = "";
        dc.GetDepartmentName(c.DepartmentID, ref dptName);
        Console.WriteLine(" - " + dptName);
        // UTILISATION DU MAPPING :
        Department d = c.Department;
        // ...
    }
}
```



# LINQ to SQL - Utilisation

208

```
using (SchoolDataContext dc = new SchoolDataContext())
{
    // ECRITURE
    Department nvDpt = new Department { Name="IT", StartDate=DateTime.Now, Budget=150000 };
    dc.Departments.InsertOnSubmit(nvDpt);
    Course nvCourse = new Course { CourseID=3333, Title=".NET", Department = nvDpt, Credits=3 };
    dc.Courses.InsertOnSubmit(nvCourse);
    dc.SubmitChanges();

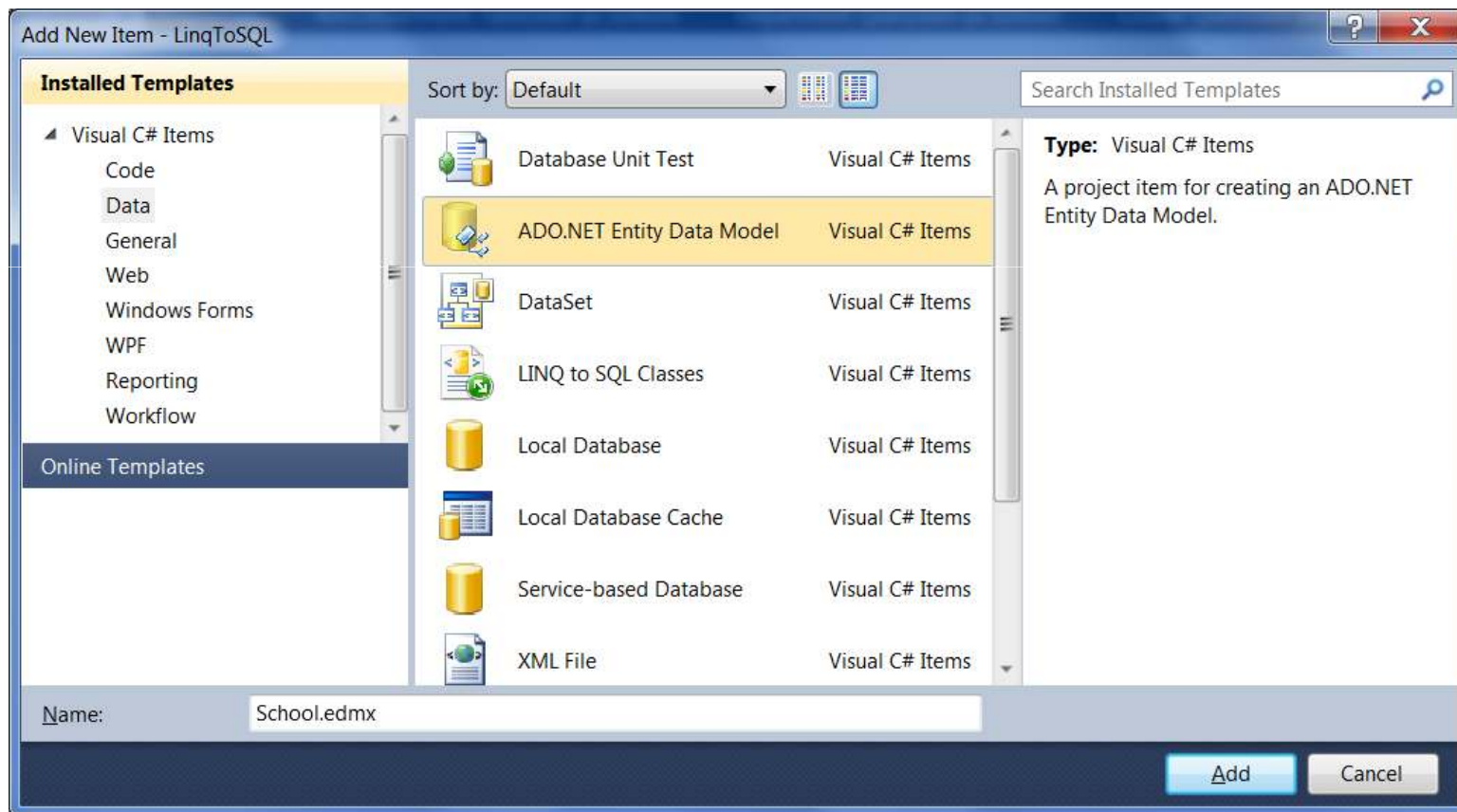
    // MISE A JOUR
    var q = from c in dc.Courses where c.Title == ".NET" select c;
    Course modCourse = q.FirstOrDefault();
    modCourse.Title = "C# .NET";
    dc.SubmitChanges();

    // SUPPRESSION
    dc.Courses.DeleteOnSubmit(modCourse);
    dc.Departments.DeleteOnSubmit(nvDpt);
    dc.SubmitChanges();
}
```

# LINQ to Entities - Création

209

## ❑ Ajout d'entités à un projet

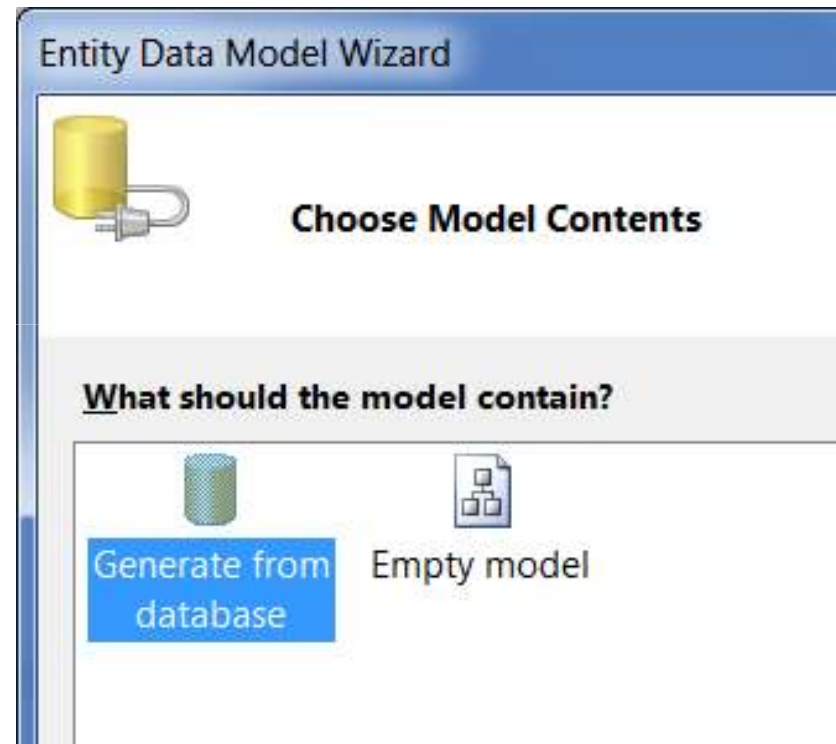


(c) Benoit Chauvet 2013

# LINQ to Entities - Création

210

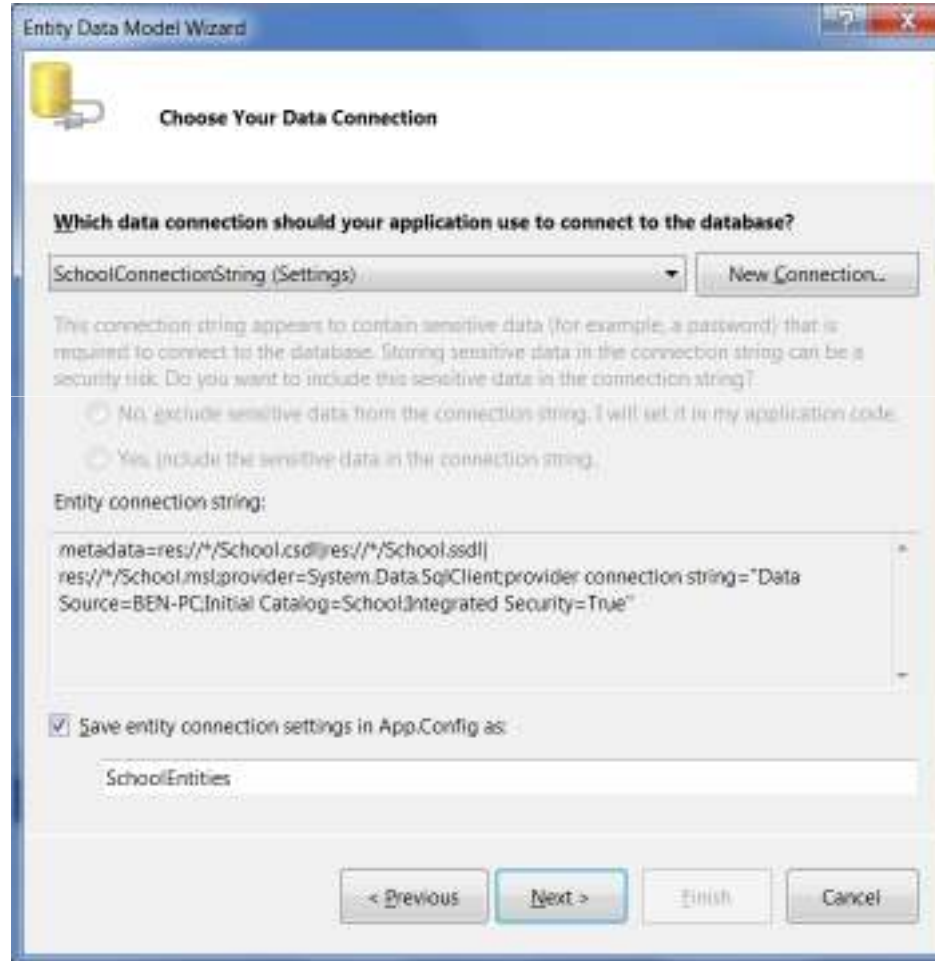
- ❑ A la création, 2 possibilités :
  - Génération des entités à partir d'une base de données existante
  - Création « manuelle » des entités (via designer) et possibilité de générer un script de création de base de données



# LINQ to Entities - Création

211

- ❑ Choix de la connexion (dans le cas d'une base de données existante)

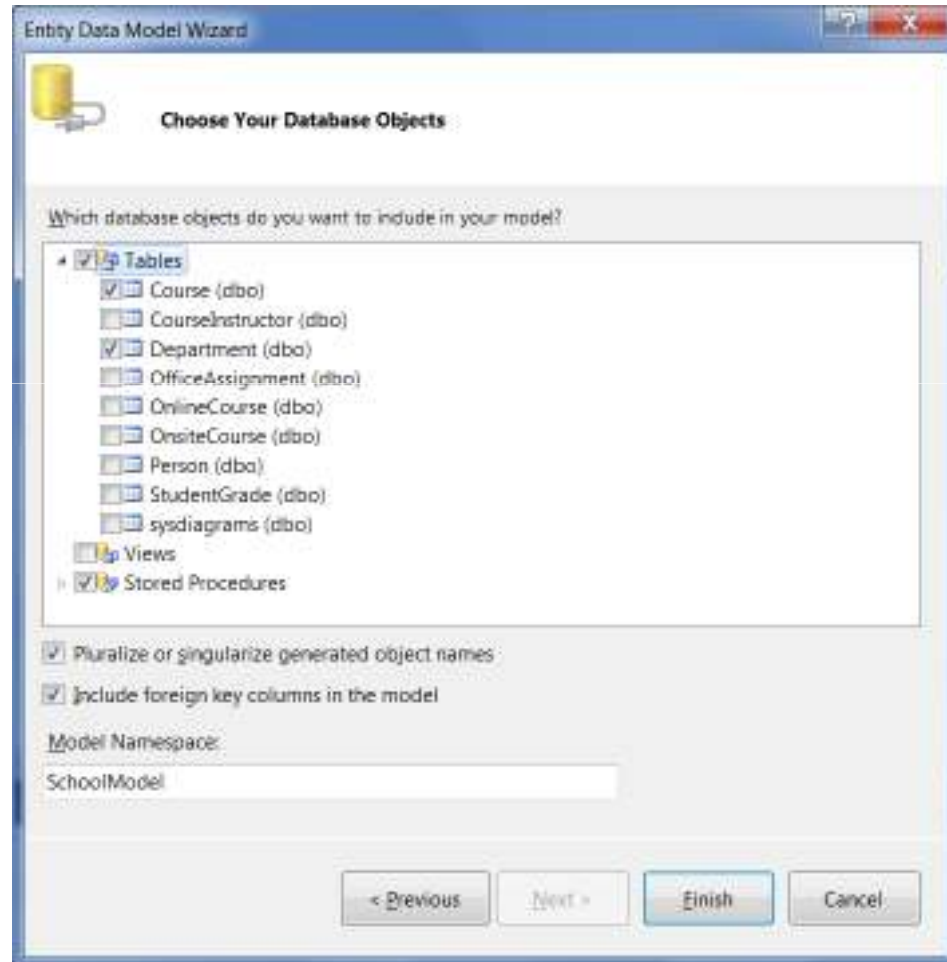


(c) Benoit Chauvet 2013

# LINQ to Entities - Création

212

- ❑ Choix des objets à mapper (dans le cas d'une base de données existante)

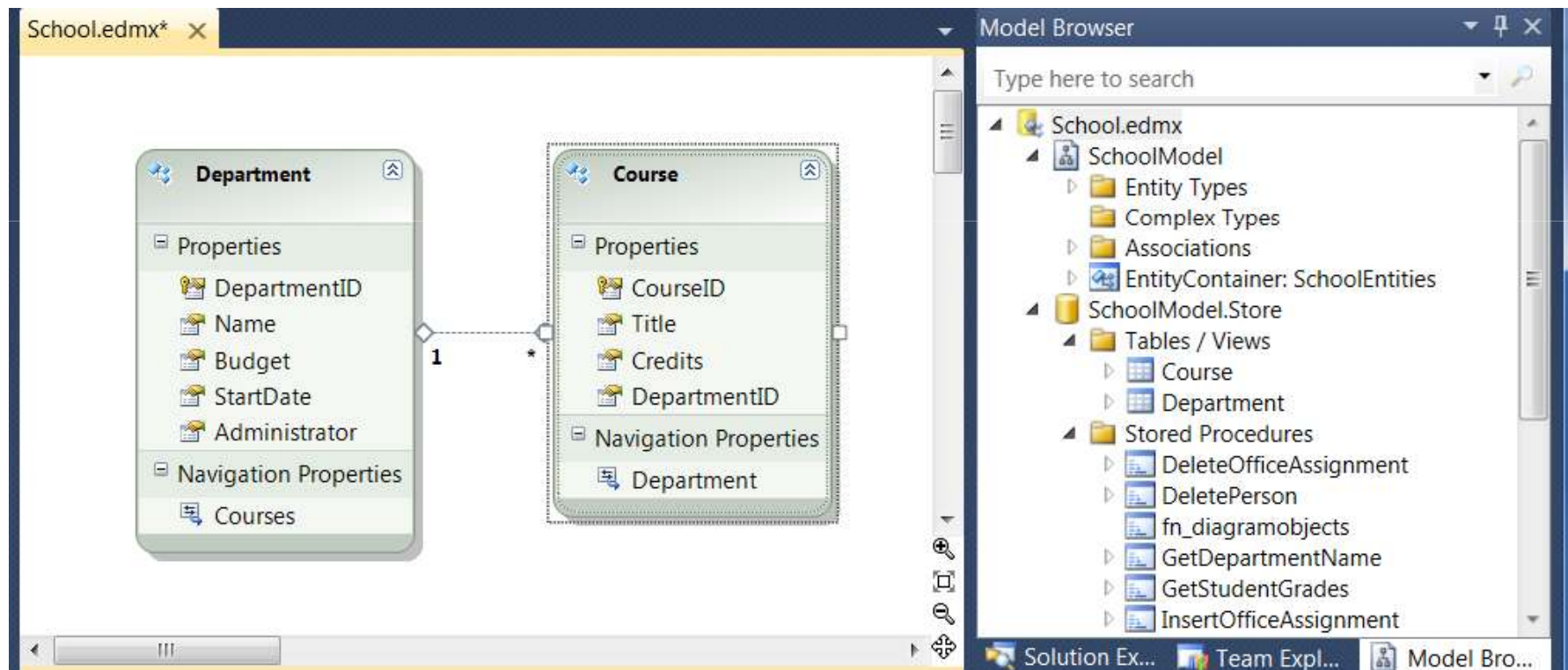


(c) Benoit Chauvet 2013

# LINQ to Entities - Création

213

## □ Designer et explorateur de modèle



# LINQ to Entities - Utilisation

214

```
using (SchoolEntities ctx = new SchoolEntities())
{
    // LECTURE DES DONNEES :
    foreach (Course c in ctx.Courses)
    {
        Console.Write(c.CourseID + " - " + c.Title);
        // APPEL DE PROCEDURE STOCKEE :
        System.Data.Objects.ObjectParameter dptName =
            new System.Data.Objects.ObjectParameter("Name", "");
        ctx.GetDepartmentName(c.DepartmentID, dptName);
        Console.WriteLine(" - " + dptName.Value);
        // UTILISATION DU MAPPING :
        Department d = c.Department;
        //...
    }
}
```

# LINQ to Entities - Utilisation

215

```
using (SchoolEntities ctx = new SchoolEntities())
{
    // ECRITURE :
    Department nvDpt = new Department { Name = "IT", StartDate = DateTime.Now, Budget = 300000 };
    ctx.Departments.AddObject(nvDpt);
    Course nvCourse = new Course { CourseID = 4444, Title = "Entity Framework",
                                   Department = nvDpt, Credits = 3 };
    ctx.Courses.AddObject(nvCourse);
    ctx.SaveChanges();

    // MISE A JOUR :
    var q = from c in ctx.Courses where c.Title == "Entity Framework" select c;
    Course modCourse = q.FirstOrDefault();
    modCourse.Title = "LINQ to Entities";
    ctx.SaveChanges();

    // SUPPRESSION :
    ctx.DeleteObject(modCourse);
    ctx.DeleteObject(nvDpt);
    ctx.SaveChanges();
}
```



216

## Les différents types d'applications

Applications Windows

Présentation d'ASP.NET

Applications web ASP.NET

Services Web ASP.NET

# Applications Windows

217

- ❑ Windows Forms
- ❑ Projets de type **Windows Forms Application**
- ❑ Une fenêtre est une classe dérivée de `System.Windows.Forms.Form`
- ❑ Le programme principal (Main) instancie et ouvre la première fenêtre de l'appli

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

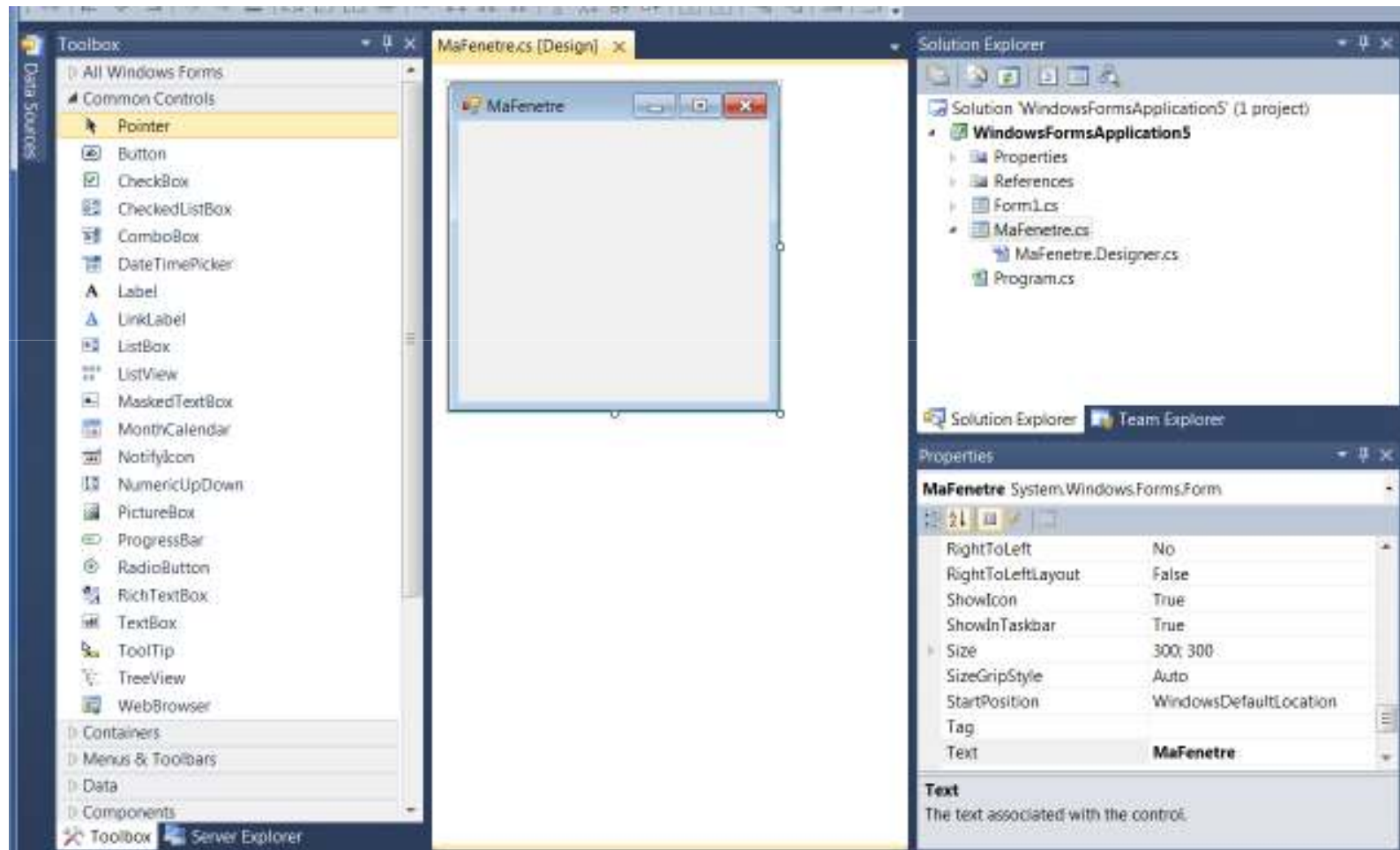
# Les fenêtres

218

- ❑ Quand on ajoute une fenêtre (Windows form) au projet, visual studio crée 2 fichiers :
  - **MaFenetre.cs** : code du formulaire à la charge du développeur (pour y accéder, clic droit sur le fichier, « view code »)
  - **MaFenetre.designer.cs** : code généré automatiquement par les actions sur le mode design
- ❑ Le développement des fonctionnalités se fera donc uniquement dans le .cs
- ❑ Ces 2 fichiers décrivent la même classe *MaFenetre*, grâce au mécanisme de classes partielles (public partial class MaFenetre)

# Fenêtre en mode design

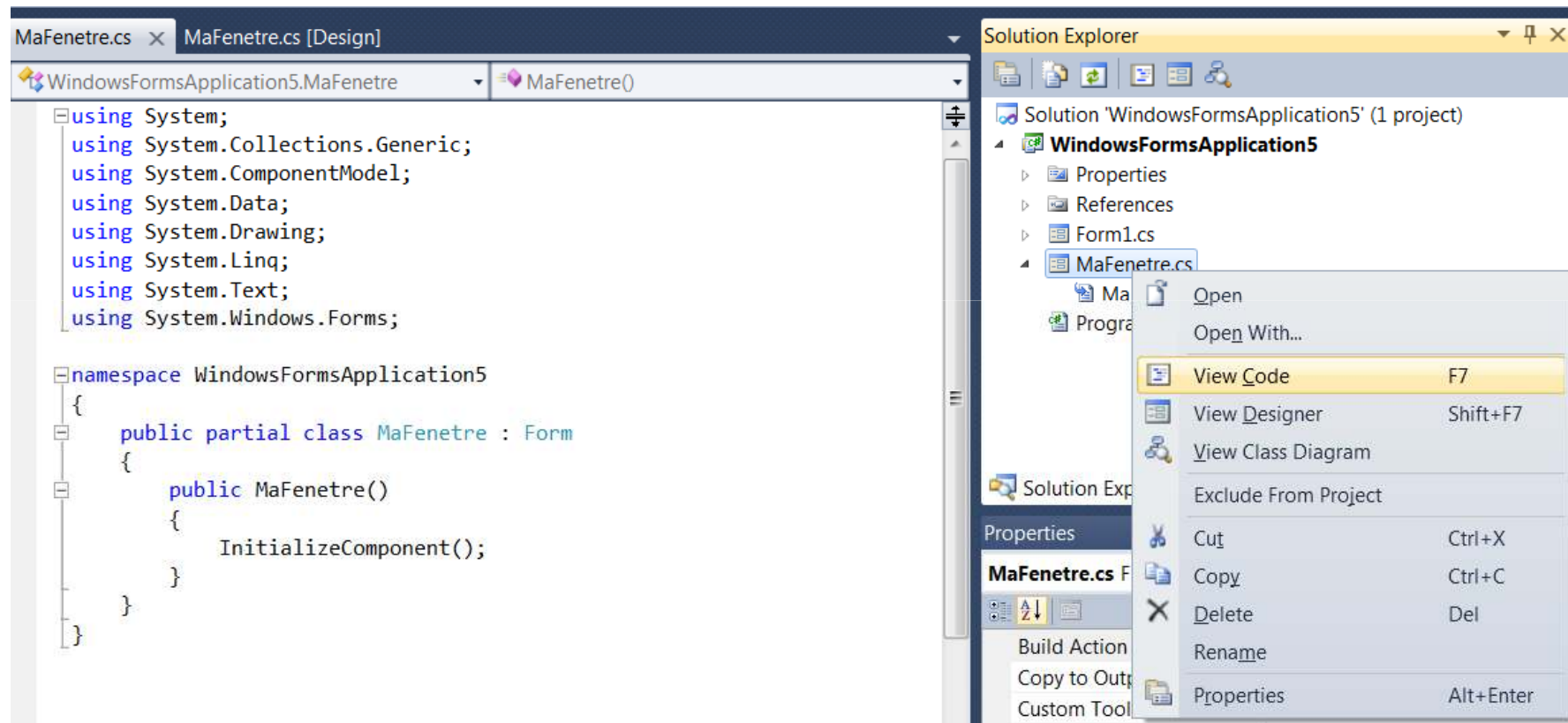
219



(c) Benoit Chauvet 2013

# Code de la fenêtre (.cs)

220



# Code généré (.designer.cs)

221

```
namespace WindowsFormsApplication5
{
    partial class MaFenetre
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

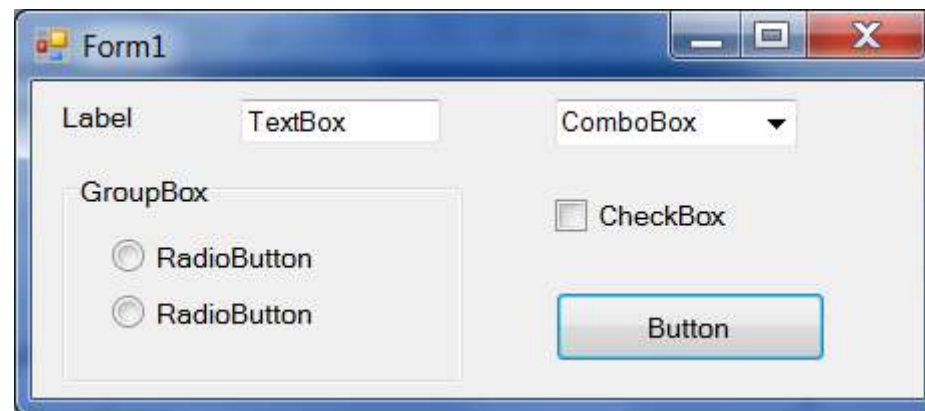
        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "MaFenetre";
        }
    }
}
```

# Les contrôles

222

- ❑ Boutons, étiquettes, zones de texte, listes, images, cadres ...
- ❑ Accessibles depuis la boîte à outils (drag & drop)
- ❑ Redimensionnement et positionnement à la souris
- ❑ Fenêtre propriétés pour personnaliser les contrôles
- ❑ Classes dérivées de `System.Windows.Forms.Control`
- ❑ Déclarés dans le fichier `.designer.cs`

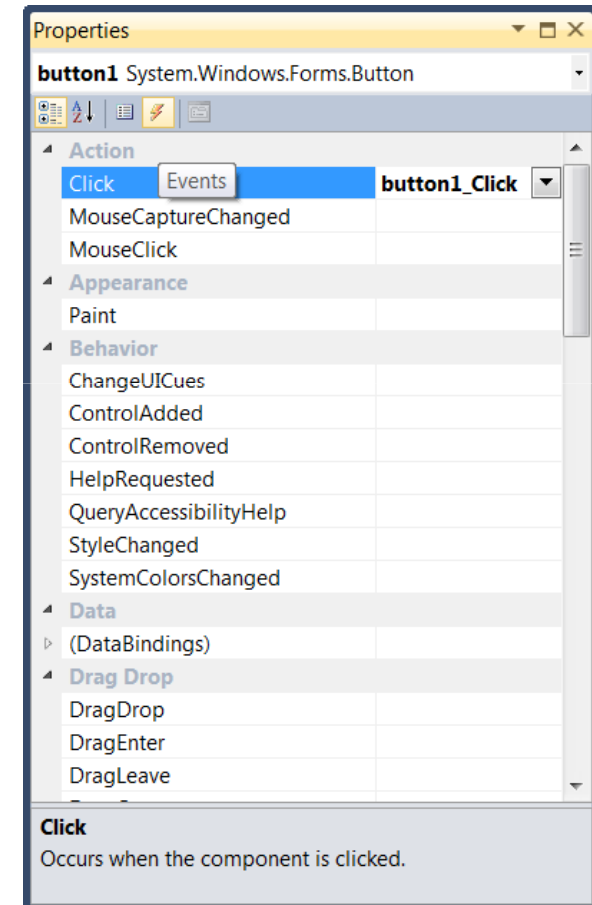


# Événements

223

- ❑ Gestion des événements générés par des actions de la souris et du clavier (click, mouseEnter,...)
- ❑ Certains contrôles ont des événements spécifiques (ex : selectedValueChanged pour la ComboBox...)
- ❑ Liste des événements dans la fenêtre propriétés
- ❑ Double-clic sur événement pour l'utiliser : génère la méthode dans le fichier.cs

```
private void button1_Click(object sender, EventArgs e)
{
}
}
```





# Liaison de données (databinding)

224

- ❑ Certains contrôles peuvent être liés à une source de données pour leur contenu :
  - Le contrôle DataGridView permet d'afficher des données sous forme de grille (tableau)
  - Les contrôles de type liste (ListBox, comboBox...) affichent une liste de valeurs attachées à un identifiant
- ❑ Propriété DataSource :
  - Tableau, Liste, Collection, DataTable...

# DataBinding exemple

225

```
public enum Saisons { printemps, été, automne, hiver }

public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();

        // remplissage de ComboBox à partir d'une enum :
        foreach (string s in Enum.GetNames(typeof(Saisons)))
        {
            comboBox1.Items.Add(s);
        }

        // remplissage de DataGridView à partir d'un dataset :
        SqlConnection cnx = new SqlConnection("Data Source=BEN-PC;Initial Catalog=Market;Integrated Security=True");
        SqlDataAdapter da = new SqlDataAdapter("select * from article", cnx);
        DataSet ds = new DataSet();
        da.Fill(ds, "article");
        dataGridView1.DataSource = ds.Tables["article"];
    }

    // récupération de l'id de la ligne courante sur clic dans un DataGridView :
    private void gv_CellClick(object sender, DataGridViewCellEventArgs e)
    {
        if (e.RowIndex >= 0)
        {
            int ligneCourante = e.RowIndex;
            int id = int.Parse(dataGridView1.Rows[ligneCourante].Cells[0].Value.ToString());
            Detail d = new Detail(id);
            d.Show();
        }
    }
}
```

# Interfaces SDI / MDI

226

- ❑ SDI : Single Document Interface : interface monodocument (une seule fenêtre dans l'appli)
- ❑ MDI : Multiple Documents Interface : interface multi documents (plusieurs fenêtres à la fois dans l'appli)
  - Fenêtre parent / Fenêtre(s) enfant(s)
  - Propriété **IsMdiContainer** (bool) sur la fenêtre parent
  - Propriété **MdiParent** (Form) sur les fenêtres enfants

# Interface MDI - Exemple

227

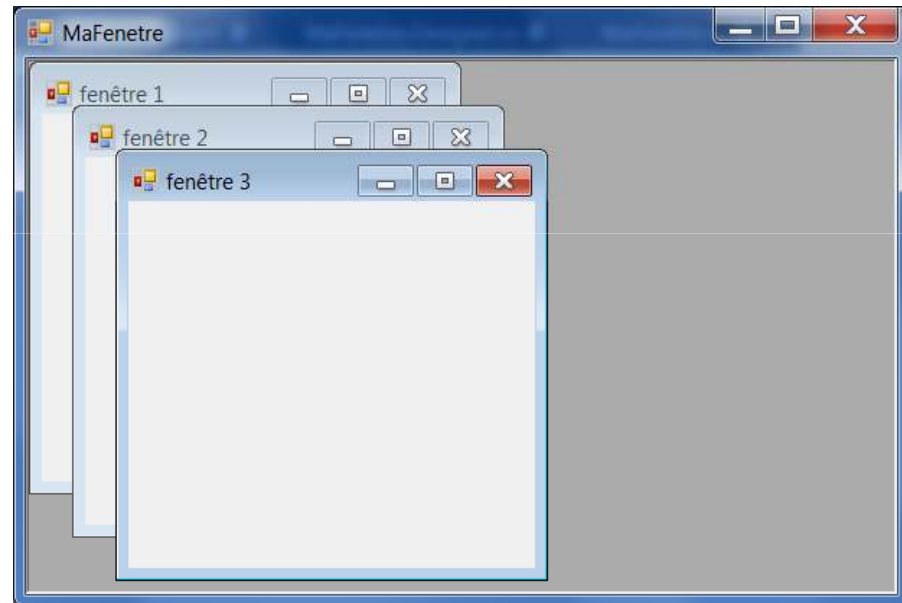
```
public partial class MaFenetre : Form
{
    public MaFenetre()
    {
        InitializeComponent();

        this.IsMdiContainer = true;

        Form fenetre1 = new Form();
        fenetre1.Text = "fenêtre 1";
        fenetre1.MdiParent = this;
        fenetre1.Show();

        Form fenetre2 = new Form();
        fenetre2.Text = "fenêtre 2";
        fenetre2.MdiParent = this;
        fenetre2.Show();

        Form fenetre3 = new Form();
        fenetre3.Text = "fenêtre 3";
        fenetre3.MdiParent = this;
        fenetre3.Show();
    }
}
```



(c) Benoit Chauvet 2013

# Présentation d'ASP.NET

228

- ❑ Framework de technologies web du Framework.NET
- ❑ Successeur de ASP (Active Server Pages)
- ❑ Permet de créer des applications et des services web
- ❑ Basé sur un serveur web (IIS ou serveur de développement ASP.NET intégré à Visual Studio)
- ❑ Intégration de code compilé (C#, VB.NET...) permettant de tirer parti du framework.NET

# Serveur Web

229

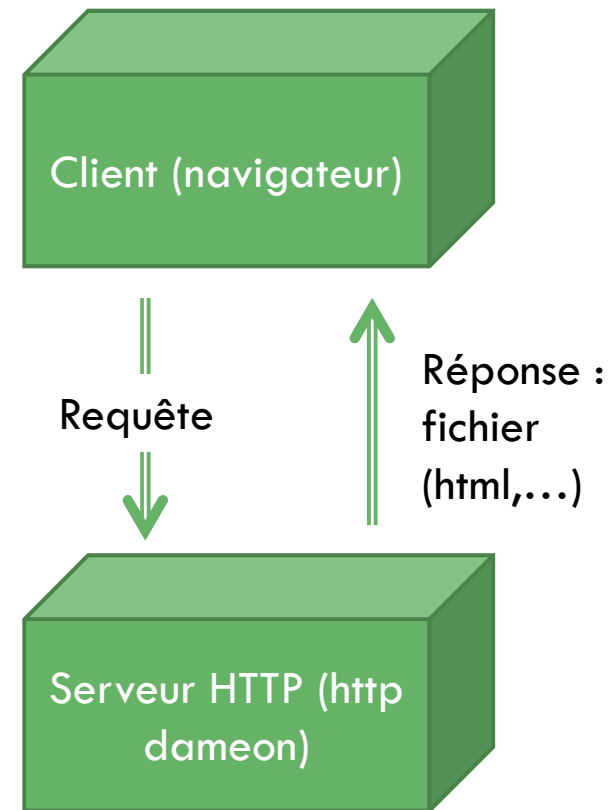
- ❑ Serveur web : protocole de communication client-serveur HTTP (HyperText Transfer Protocol)
- ❑ Requête / Réponse
- ❑ Quelques serveurs web :
  - Apache HTTP Server
  - Apache Tomcat (évolution d'apache pour J2EE)
  - IIS (Internet Information Services)
  - ...

# Sites statiques

230

## ❑ Sites statiques

- Serveur web = simple serveur de fichiers
- Simple
- Robuste
- Interactions limitées aux liens hypertexte
- Client essentiellement passif



# Sites dynamiques

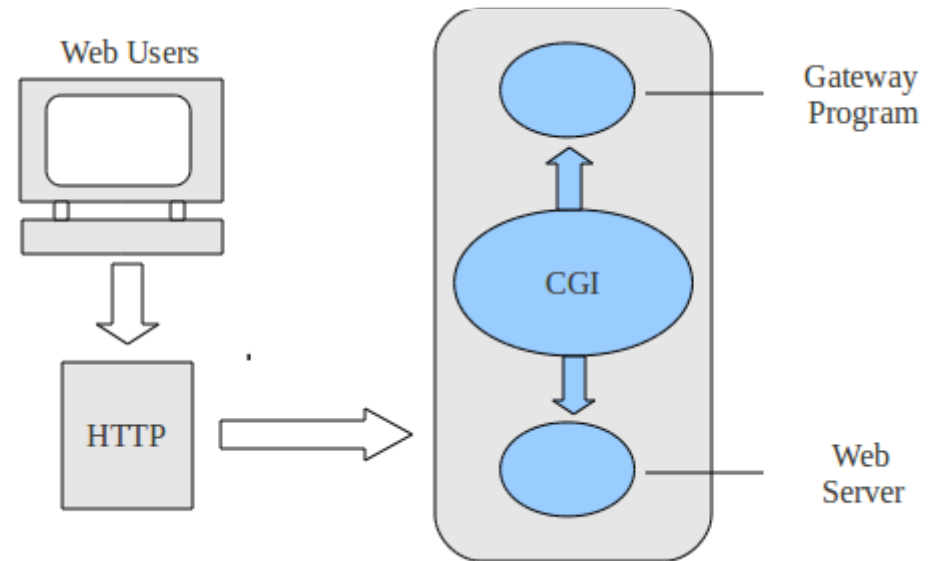
231

## ❑ Sites dynamiques

- Pages web générées à la demande
- Traitements côté serveur
- Liaisons avec bases de données

## ❑ Quelques exemples :

- CGI (Common Gateway Interface)
- Php
- JSP/Servlets
- Asp
- Asp.net...



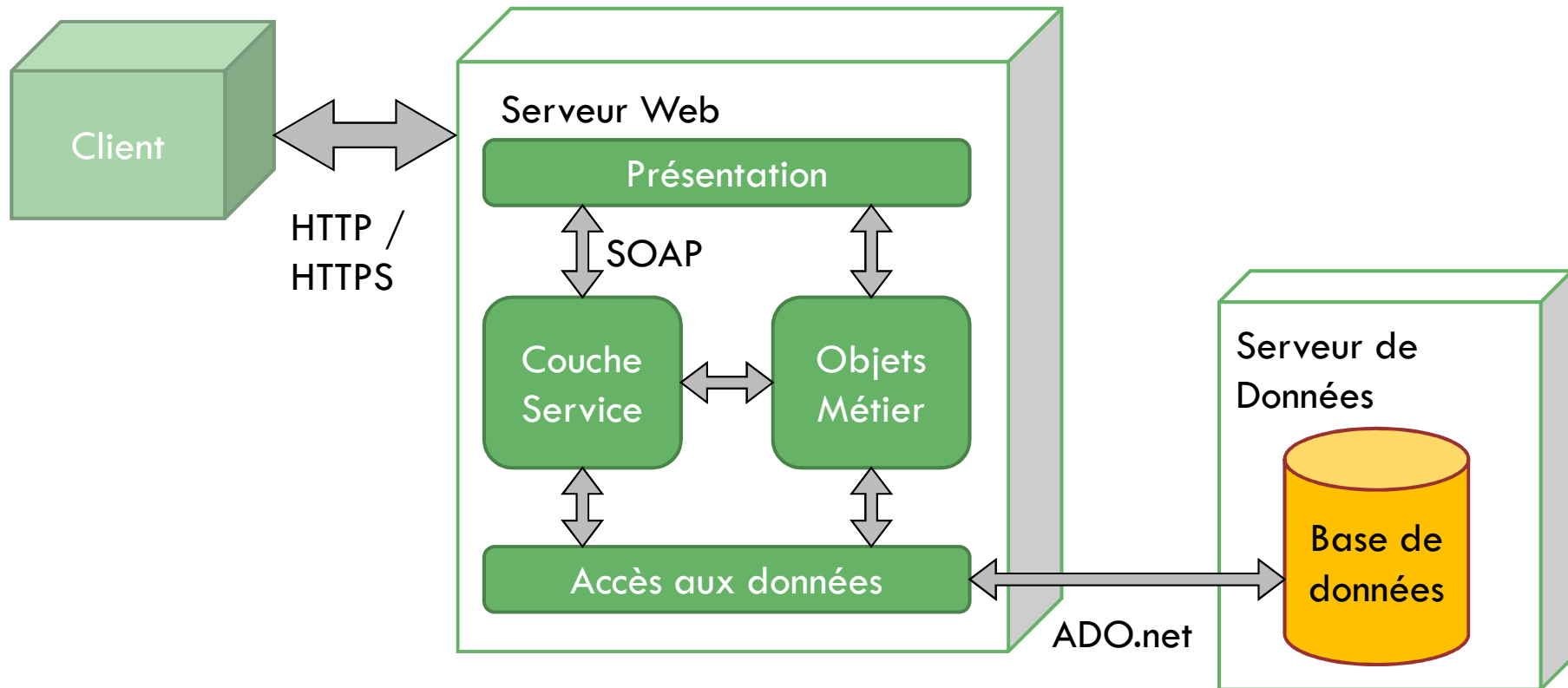
(c) Benoit Chauvet 2013



# Architecture d'une application web

232

Exemple d'architecture 3 tiers (Présentation / Métier / Données)



# Principes des pages dynamiques

233

- ❑ Page dynamique : générée à la demande
- ❑ Contenu variable
  - Paramètres contenus dans la requête
  - Date / Heure
  - Contenu de la base de données de référence...
  - ...
- ❑ Mécanisme :
  - Envoi d'une requête au serveur http
  - Transmission au logiciel (interpréteur) correspondant à la requête (php, asp.net, jsp...)
  - Génération du contenu
  - Envoi de la réponse sous forme de page statique
- ❑ Problématique de l'indexation par les moteurs de recherche (« deep web » ou « web profond »)

# Principe des WebForms

234

- ❑ ASP.net introduit la notion de **code behind** : le code côté serveur est séparé du code html, dans un fichier distinct.
- ❑ Notion de classe partielle : Séparation du code behind en 2 fichiers
  - code généré
  - code utilisateur
- ❑ Une page ASP.net (Webform) est donc constituée de 3 fichiers :
  - Balises html (« concepteur de vues ») : *maPage.aspx*
  - Code behind généré : *maPage.aspx.designer.cs*
  - Code behind utilisateur : *maPage.aspx.cs*
- ❑ Les modifications du .aspx entraînent des modifs dans le .aspx.designer.cs (déclarations, événements...)
- ❑ Chaque page asp.net est une classe, dérivée de *System.Web.UI.Page*

# Structure d'un Webform

235

maPage.aspx

```
<!-- Directive de page -->
```

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="maPage.aspx.cs" Inherits="W
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/:
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head runat="server">
```

```
  <title></title>
```

```
</head>
```

```
<body>
```

```
  <form id="form1" runat="server">
```

```
    <div>
```

```
      <!-- TextBox ajoutée via la boîte à outils -->
```

```
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
```

```
    </div>
```

```
  </form>
```

```
</body>
```

```
</html>
```

# Structure d'un Webform

236

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

maPage.aspx.cs (code behind)

```
namespace WebApplication2
{
    // Code Behind "utilisateur", classe PARTIELLE, dérivée de System.Web.UI.Page :
    public partial class maPage : System.Web.UI.Page
    {
        // Méthode de chargement (pour ajouter le code lié au chargement de la page)
        protected void Page_Load(object sender, EventArgs e)
        {
        }
    }
}
```

# Structure d'un Webform

237

## maPage.aspx.designer.cs (code généré)

```
//-----  
// <génééré automatiquement>  
// Ce code a été généré par un outil.  
//  
// Les modifications apportées à ce fichier peuvent provoquer un comportement incorrect et seront perdues si  
// le code est régénéré.  
// </génééré automatiquement>  
//-----  
  
namespace WebApplication2 {  
  
    // Code behind "designer" : généré automatiquement, classe PARTIELLE  
    public partial class maPage {  
  
        /// <summary>  
        /// Contrôle form1.  
        /// </summary>  
        /// <remarks>  
        /// Champ généré automatiquement.  
        /// Pour modifier, déplacez la déclaration de champ du fichier de concepteur dans le fichier code-behind.  
        /// </remarks>  
        protected global::System.Web.UI.HtmlControls.HtmlForm form1;  
  
        /// <summary>  
        /// Contrôle TextBox1.  
        /// </summary>  
        /// <remarks>  
        /// Champ généré automatiquement.  
        /// Pour modifier, déplacez la déclaration de champ du fichier de concepteur dans le fichier code-behind.  
        /// </remarks>  
        protected global::System.Web.UI.WebControls.TextBox TextBox1;  
    }  
}
```

# Utilisation des contrôles serveur

238

- ❑ Les contrôles serveur : éléments accessibles depuis le code behind.
- ❑ Déclaration générée dans le .aspx.designer.cs lors de l'ajout dans une page aspx.
- ❑ Définis par l'attribut **runat="server"**
- ❑ Référencés dans le code behind par leur ID
- ❑ Contrôles serveur HTML ou asp.net (web)

```
aspx    <input id="Text1" type="text" runat="server" />
        <asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
```

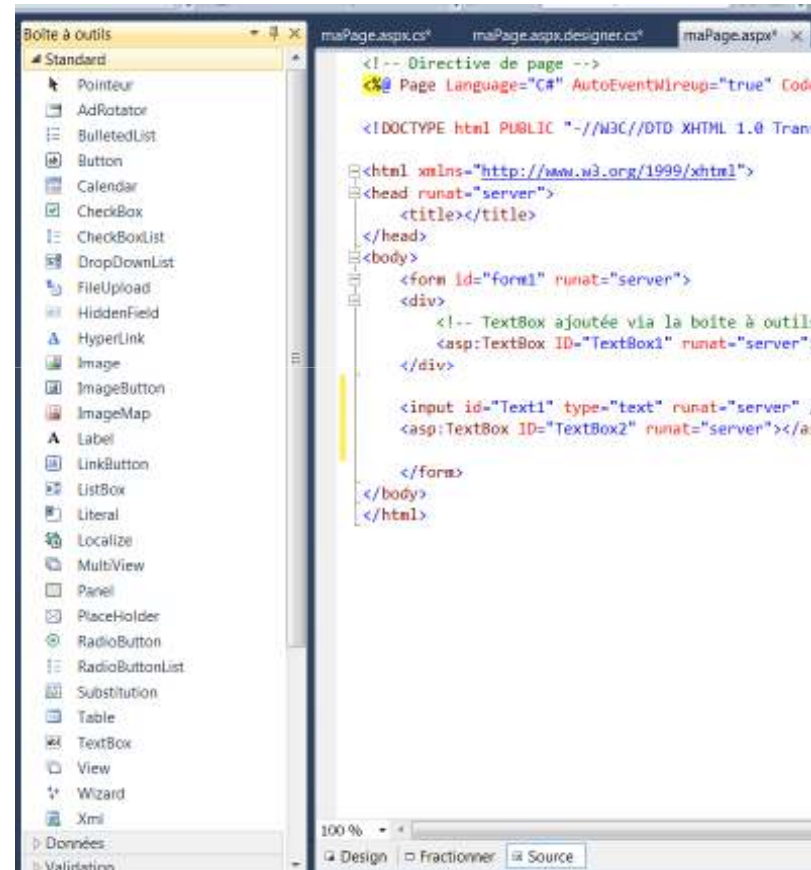
```
aspx.designer.cs    protected global::System.Web.UI.HtmlControls.HtmlInputText Text1;
                    protected global::System.Web.UI.WebControls.TextBox TextBox2;
```

```
aspx.cs    Text1.Value = "Hello";
            TextBox2.Text = "World";
```

# Insertion de contrôles

239

- Via la boîte à outils Visual Studio
  - Glisser/déplacer vers la fenêtre source ou design
  - Génération automatique du code behind de la déclaration





# Structure d'une application ASP.NET

240

## ❑ Properties

- AssemblyInfo : Paramètres de l'assembly (description, guid, version,...)

## ❑ References

- Références des assemblies utilisées par l'application (définies dans le .sln)

## ❑ Account

- Modèles de page pour l'authentification (Login, Register, ChangePassword...)

## ❑ App\_Data : répertoire prévu pour stocker des données de type fichier (XML,...)

# Structure d'une application ASP.NET

241

- ❑ Scripts

- scripts client, contient par défaut la librairie jquery (ajax)

- ❑ Styles

- css

- ❑ Global.asax

- gestion des événements de l'application

- ❑ Site.Master

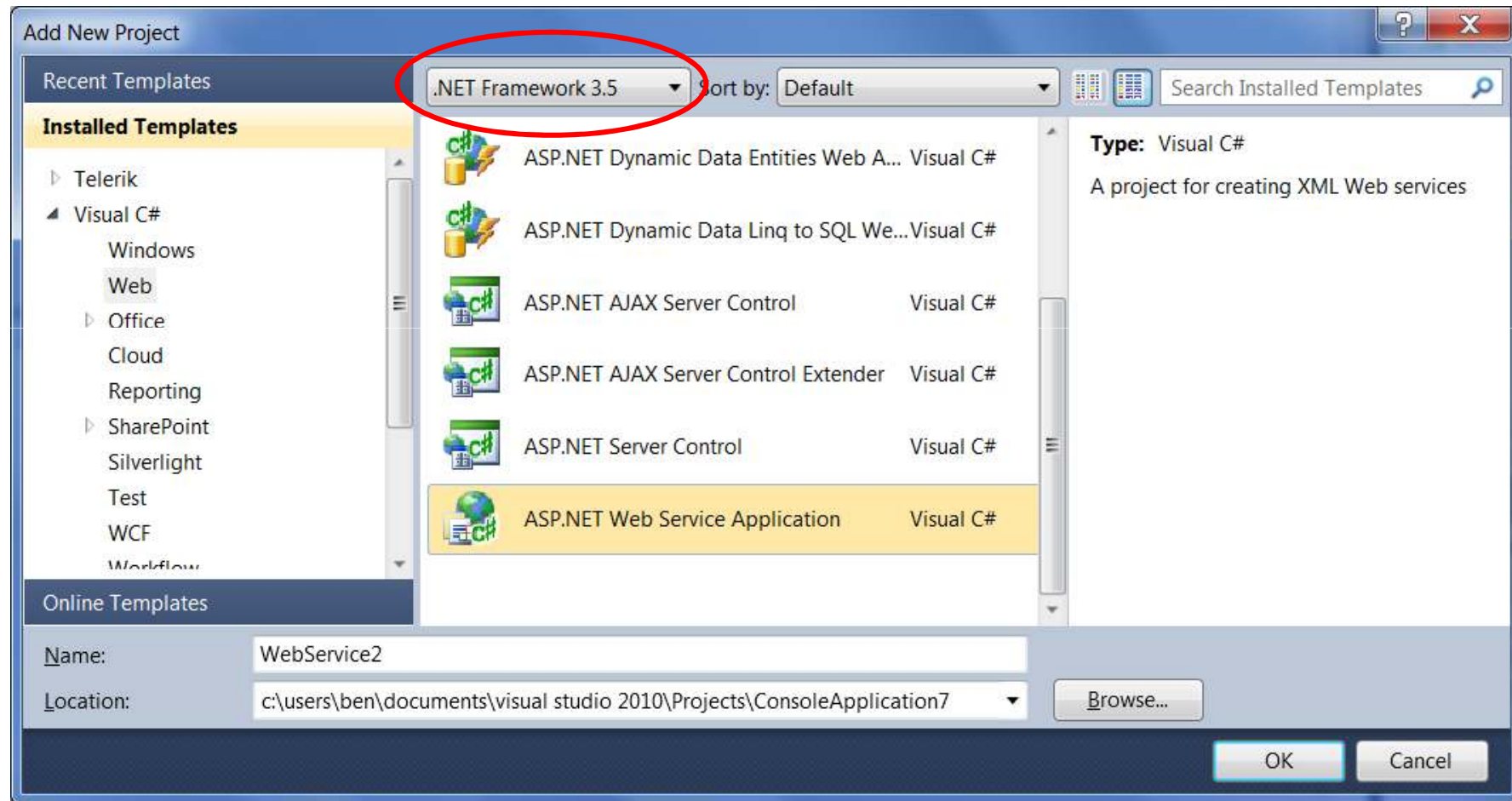
- masterpage (template d'affichage)

- ❑ Web.config

- configuration de l'application web

# Services web ASP.NET - Création

242



(c) Benoit Chauvet 2013

# Services web ASP.NET - Création

243

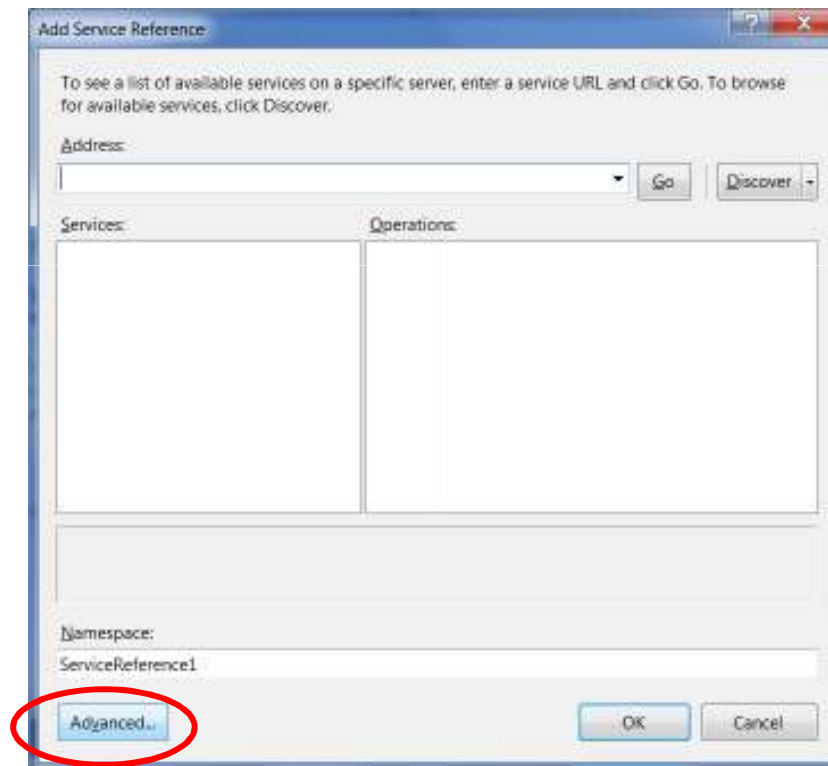
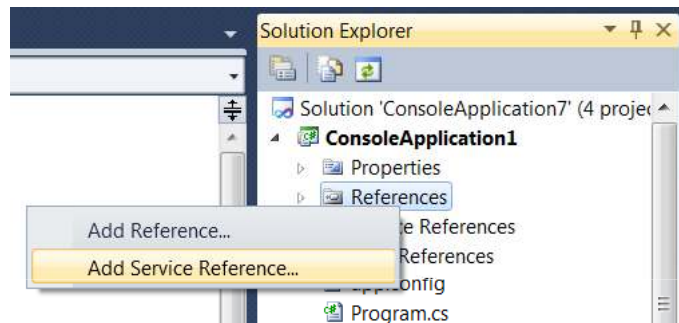
## ❑ Implémentation du service web (Service1.asmx.cs)

```
/// <summary>
/// Summary description for Service1
/// </summary>
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
// To allow this Web Service to be called from script, using ASP.NET AJAX, uncomment
// [System.Web.Script.Services.ScriptService]
public class Service1 : System.Web.Services.WebService
{
    [WebMethod]
    public int Addition(int a, int b)
    {
        return a + b;
    }
}
```

# Services Web - Utilisation

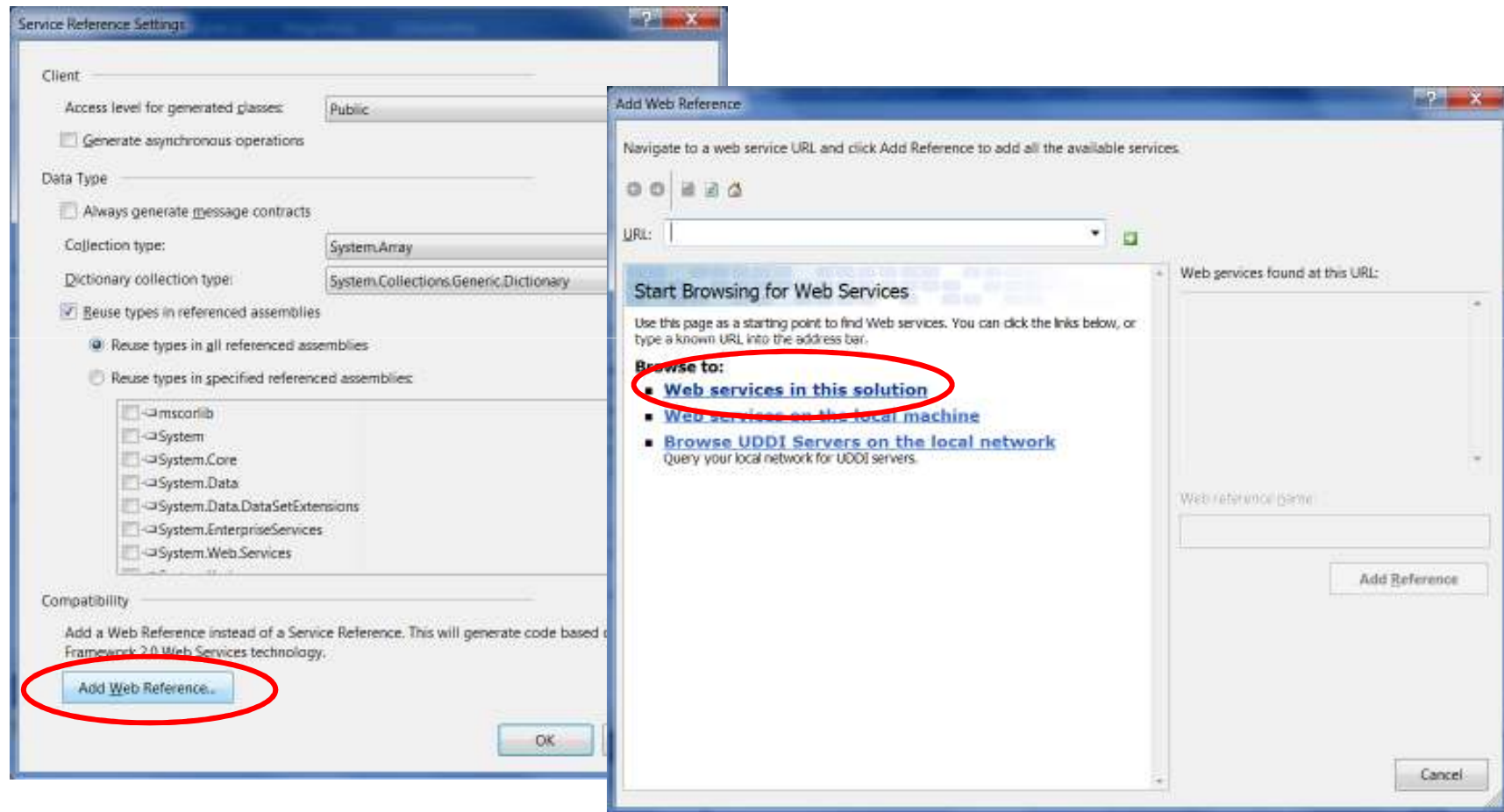
244

- Ajout d'une référence au service dans le projet client



# Services Web - Utilisation

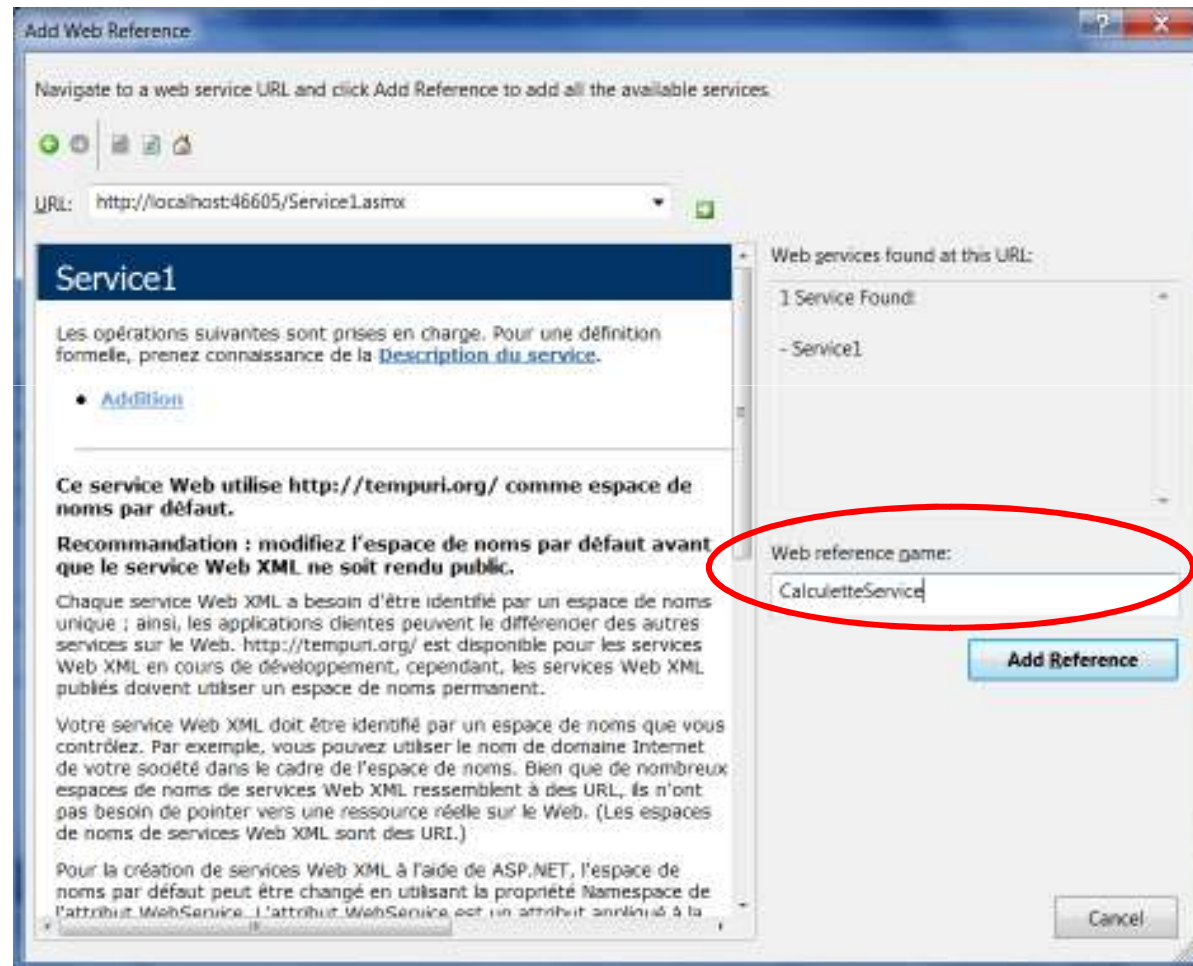
245



(c) Benoit Chauvet 2013

# Services Web - Utilisation

246



(c) Benoit Chauvet 2013

# Services Web - Utilisation

247

- ❑ Appel des méthodes du service dans le programme client :

```
static void Main(string[] args)
{
    CalculetteService.Service1 calc = new CalculetteService.Service1();

    int resultat = calc.Addition(3, 4);
}
```