Langage C#

Initiation

Mourad MAHRANE
mmahrane@dawan.fr

Objectifs

- Apprendre à développer avec C#

- Créer des interfaces de gestion de bases

- Manipuler des objets de la plate-forme .NET

Bibliographie

MSDN : http://msdn.microsoft.com/fr-fr/default.aspx
Centre .NET Framework :
http://msdn.microsoft.com/fr-fr/netframework/default.aspx

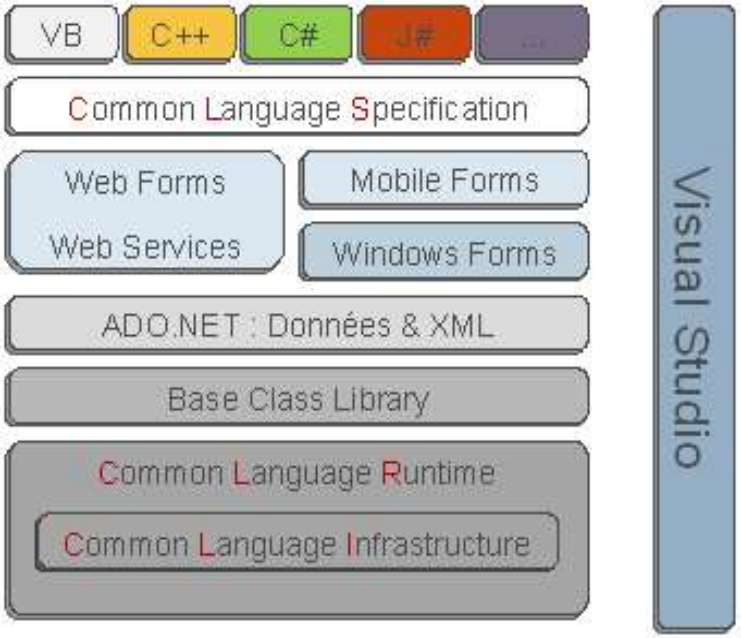
Plan

- Framework .NET
- Langage C#
- Syntaxe du langage
- Tableaux
- Méthodes et paramètres
- Gestion des exceptions
- Classes fondamentales
- Applications graphiques
- Programmation orientée objet
- ADO.NET

Framework .NET

Plateforme .NET

- **Utilisation :** Développement – Déploiement – Exécution
- **Applications :** Web, Windows, Mobile, serveurs, jeux
- **Langages supportés :** VB .NET, J#, C#, etc.
- **Gratuite**
- **Installation :** intégrée à certaines éditions Windows
  Téléchargeable via MSDN ou Windows Update

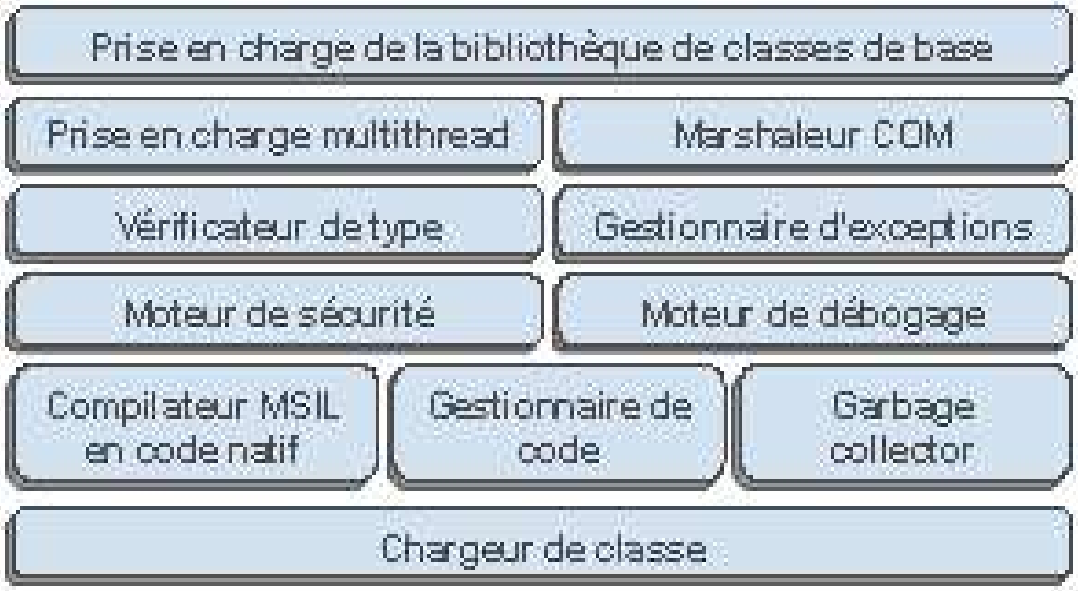


Versions

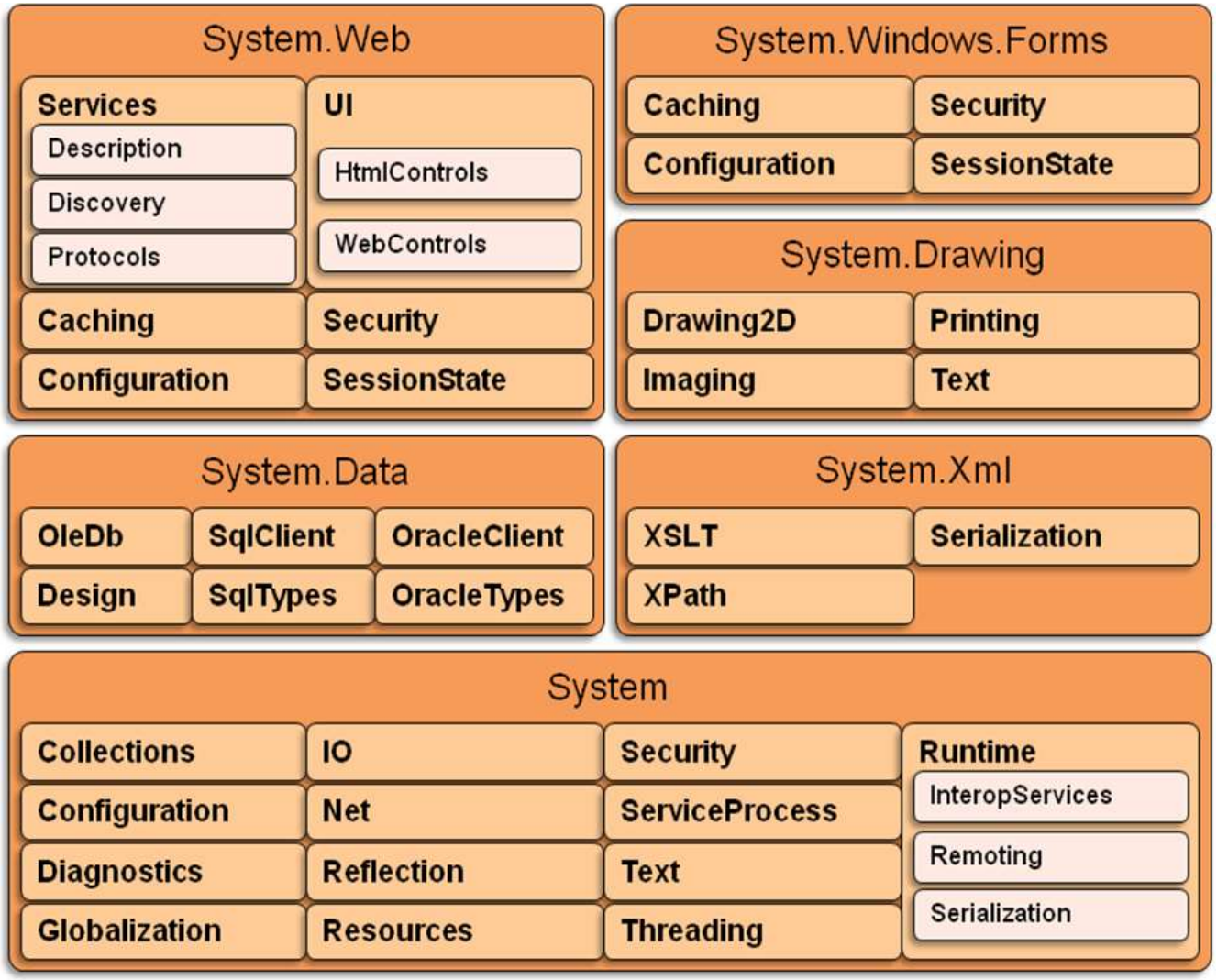
- Juin 2000 : Lancement du développement
- Février 2002 : .NET Framework 1
- Mars 2003 : .NET Framework 1.1
- Novembre 2006 : .NET Framework 2.0
- Novembre 2007 : .NET Framework 3.0
- 2008 : .NET Framework 3.5
- 2010 : .NET Framework 4.0
- 2012 : .NET Framework 4.5

Common Language Runtime

- Implémentation du standard « Common Language Infrastructure »
- Concepts : Debug - Typage (Common Type System) - Exceptions...
- Fonctions : Gestion du contexte d'exécution et de la mémoire - Versioning des applications
- Sécurité et intégrité des applications (signatures)
- Interopérabilité COM



Bibliothèque de classes



Avantages

- Normes et pratiques du web.
- Modèles d'application unifiés.
- Classes extensibles.

N.B. : le Framework .NET ne fonctionne que sous Windows

Utiliser Mono ou DotGNU pour d'autres plateformes

Présentation

- Langage orienté objet de type sécurisé
- Très proche du C++ et du Java
- RAD
- Multi-plateformes (IL)
- Plusieurs versions successives

Développement C#

- Applications Windows
- Pages ASP.NET
- Services Web
- Services Windows

C#.Net Multi-plateformes :

Win32 - Win64 - ~~WinCE~~ – ~~WinMobile~~

IDE :

- Microsoft Visual Studio payant ou version Community
- SharpDevelop ou MonoDevelop gratuits mais moins performants


Programme C#

- Espace de noms
- Classes
- Méthode main
- Classe Console (ReadLine et WriteLine)

```
using System;

namespace MyProgram
{
   class HelloWord
   {
      static void Main()
      {
         Console.WriteLine("Hello World !");
      }
   }
}
```

Débogage et Exécution

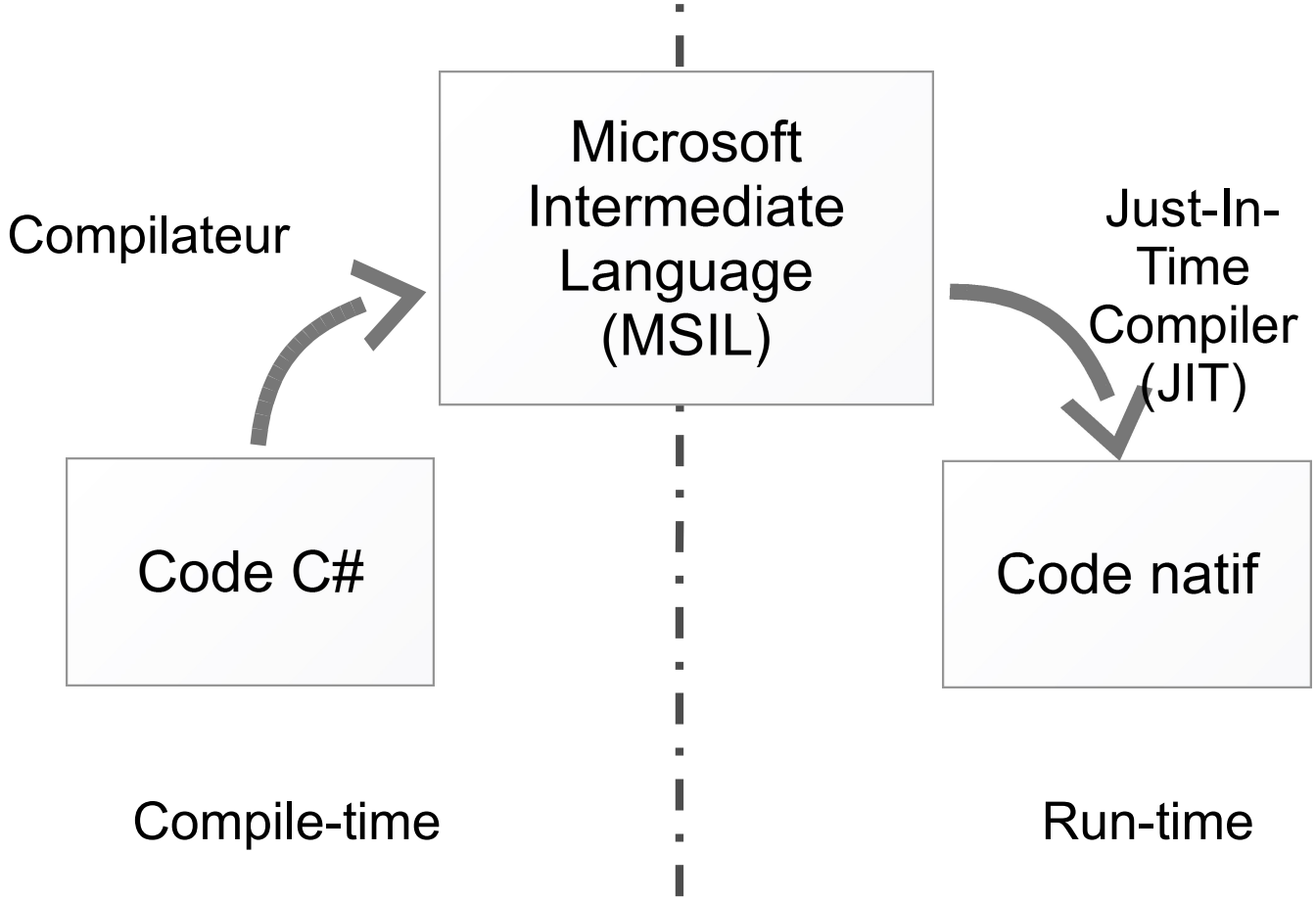
Localisation et correction des erreurs :

- Erreurs et débogage JIT
- Points d'arrêts et pas-à-pas
- Examen et modifications des variables

Exécution :

- IDE (Start Without Debugging)
- Ligne de commande (nom de l'application)

Etapes de compilation



En ligne de commande :

csc.exe

msbuild.exe

Syntaxe du langage

Base du langage

- Les erreurs de syntaxes sont interdites et vérifiées à la compilation
- Instructions les unes après les autres séparées par des « ; », code en UTF-8
- Casse sensitive : différences entre majuscules et minuscules
- Commentaires :
  - `//fin de la ligne`
  - `/* documentation */`

Types de données

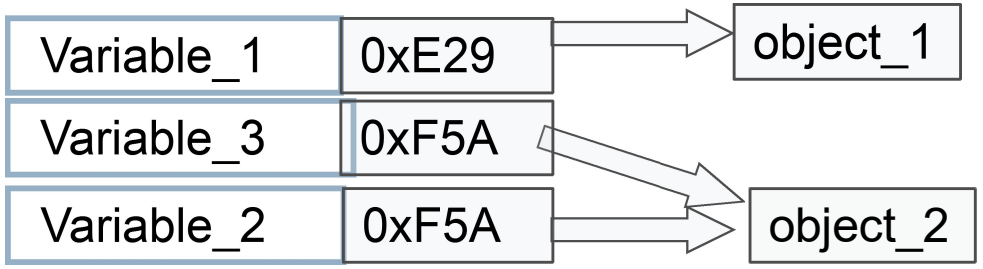
- Types communs (System)
- Types simples et Types références
- Déclaration de variables, affectation de valeurs, constantes …
- Tableaux
- Enumérations (enum)
- Structures (struct)

Types de données

- Types communs (System)
  - Entiers : Byte/SByte (8 bits), Short/UShort (16 bits), Integer/UInteger (32 bits), Long/ULong (64 bits)
  - Flottants : Single (32 bits), Double (64 bits)
  - Decimal (128 bits)
  - Autres : Boolean, Date, Char, String

- Types simples et Types références

Variable_1	45
Variable_2	true
Variable_3	c
Variable_4	56.8



Types de données

- Déclaration de variables, affectation de valeurs, constantes …

```
//Declaration de variables
int i;
string s;
//Affectation de valeurs
i = 15;
s = "ceci est une chaine";
//Affectation lors de la declaration
double d = 2 * i;
//Declaration de constante
const string CONSTANTE = "constante";
// Typage déterminé par le compilateur
var x = 2;
```

Types de données

- Enumérations

```
enum CouleurCarte {Pique, Coeur, Carreau, Trefle}

static void Main() {

    CouleurCarte maCouleur = CouleurCarte.Coeur;

    Int c = (int)maCouleur;

}
```

```
[Flags]

enum Options { ToitOuvrant=1, Climatisation=10,
FeuxAntiBrouillard=100, JantesAlu=1000 }

static void Main() {

    Options opts = Options.ToitOuvrant|
Options.JantesAlu;

    Console.WriteLine (opts);

}
```

Types de données

- Structures

```
structure Automobile {
    public int puissance;
    Public string couleur;
    Public string marque;
}
Automobile maVoiture;
MaVoiture.puissance = 6;
```

- Nullable/HasValue

```
//Declaration de variables
int? Variable2;
Variable1 = null;
Variable1.HasValue(); //retourne faux
```

Opérateurs

- Arithmétiques : + - * / % << >> ~
- Egalité/Inégalité : == !=
- Relationnels : < <= >= >
- Logiques : | & ^ ~ || && !
- Assignation : = += -= *= /= %= <<= >>= |= &= ^=
- Incrémentation, décrémentation : ++ --
- Autres : . [] () new + ??

Transtypage

- Conversions implicites (automatique)

```
int i = 3;
double d = 2 * i;
```

- Conversions explicites (cast)

```
float f = 3.45f;
byte b = (byte)f;
```

- Fonctions de conversions (Convert, parse...)

```
string s = "2.81";
double d = Convert.ToDouble(s);
```

Structures Conditionnelles

- if / else

```
int i = 25;
if (i == 22) {
// traitement 1
} else if (i == 25) {
// traitement 2
} else {
// traitement par défaut
}
```

Structures Conditionnelles

- Switch/Case

```
switch(n)
{
   case 1:
   case 2:
      x = 4;
      break;
   default:
      x = 5;
      break;
}
```

Structures Conditionnelles

- opérateur ternaire

```
int monEntier = 25;

string resultatTest;

resultatTest = (monEntier < 25)?"Variable inferieure
a 25":"Variable superieure a 25";
```

Équivalent à

```
int monEntier = 25;

String resultatTest;

if (monEntier < 22) {

    resultatTest = "Variable inferieure a 25";

} else {

    resultatTest = "Variable superieure a 25";

}
```

Boucles

- for

```
for (int i=1;i<=117;i++) {
        //Instructions
}
```

- while/do...while

```
int i=1 ;
while (i<=10) {
        //Instructions
        i++ ;
}
```

```
int i=1 ;
do {
        //Instructions
        i++ ;
} while (i<=10)
```

- foreach

```
foreach (string s in tabString) {
        //Instructions
}
```

Instructions de saut

- break : termine le traitement de boucle ou de switch courant

- continue : passe à l'itération suivante dans un traitement de boucle

- goto : se débranche vers un case particulier dans un switch

Tableaux

Tableaux

- Déclaration d'un tableau

```
int[] Tableau =  new int[3];
int[] Tableau =  new int[] {2,3,8};
```

- Accès à un élément d'un tableau

```
Tableau[indice]
```

- Taille d'un tableau

```
Tableau.Length
```

Tableaux

- Tableau à 2 dimensions

```
int[,] matrice = new int[2,4];

Matrice[1,3] = 5;
```

- Tableau multidimensionnels :

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

cube[1,3,4] = 5;
```

- Alternative : tableaux de tableaux (tableaux dentelés) :

```
double[][] t = new double[2][];
t[0] = new double[4];
t[1] = new double[2];
t[0][2] = 0.0;
```

Méthodes et paramètres

Méthodes

- C# est procédural, il permet de regrouper des instructions sous un même nom :
  - Diviser le code en morceaux (réutilisabilité, clarté, travail en groupe)
  - Factoriser le code (maintenabilité, clarté)
- Une méthode (« procédure », « fonction ») a un nom, des paramètres et peut retourner une valeur (dans le cas d'une fonction)
- Un appel de méthode indique son nom, envoie et/ou utilise les paramètres, utilise la valeur de retour éventuelle

Méthode

- Déclaration :

```
void meth1(int x)
{
   Console.WriteLine(((x+4)*3-12)/3);
}
```

- Appel :

```
int x = 4;
meth1(x);
```

Fonction

- Déclaration :

```
int meth1(int x)
{
   return ((x+4)*3-12)/3;
}
```

- Appel :

```
int x = 4;
x = meth1(x);

X = meth1(meth1(x));
```

Passage de paramètres

- Paramètres en entrée, en sortie (*out*), voire les deux (*ref)*
- Déclaration :

```
void methode(int i, ref int j, out
int k)
{
   j = j * i;
   k = i + 1;
}
```

- Appel :

```
int x = 2, y;
methode(5, ref x, out y);
```

Passage de paramètres

Paramètre facultatif et paramètre nommé

```
void MethodOptParams(int required, string
   optionalstr = "default string", int optionalint
   = 10)
{
      Console.WriteLine("{0} {1} {2}", required,
   optionalstr, optionalint);
}
```

Appel :

```
MethodOptParams(1, "One", 1);
MethodOptParams(2, "Two");
MethodOptParams(3);
MethodOptParams(4,optionalint:4);
```

Passage de paramètres

Tableau de paramètres (params)

```
void MaFonction(string s, params string[] ss)
{
}
```

Appel :

```
MaFonction("Test","Nb","De","Parametres","Inconnu");
string[] tableauParametres = new string[]
   {"Nb","De","Parametres","Inconnu"};
MaFonction("Test", tableauParametres);
```

Surcharge de méthode

Plusieurs méthodes peuvent avoir le même nom et des arguments différents. Pour une fonction, le type de retour doit être identique.

```
void MaFonction(int param1) {
...
}
void MaFonction(int param1, string param2) {
...
}
void MaFonction(int param1, int param2) {
...
}
```

Récursivité

Capacité d'une méthode à s'appeler elle-même.

```
int factorial(int n) {
    if (n <= 1) {
        return 1 ;
    } else {
        return factorial(n - 1) * n ;
    }
}
```

Règles d'écriture

- Conventions de codage :
  - Accolades isolées sur leur lignes
  - Laisser faire l'IDE
- Conventions de nommage : uneVariable, unArgument, uneMethode, unAttribut, UNE_CONSTANTE, UnType ...
- Commentaires interprétés :

```
/// <summary>...
```

Exceptions

Définition

Situations inattendues ou exceptionnelles qui surviennent pendant l'exécution d'un programme, interrompant le flux normal d'exécution

Le système déclenche ses propre exceptions. Notre code peut le faire aussi.

Les exception interrompent le processus normal. Elle sont lancées, puis attrapées par nous (ou l'OS).

Types d'Exceptions

- **Checked Exceptions**

  - Le développeur doit les anticiper et coder des lignes pour les traiter.

  - Exemple : on peut essayer de charger un fichier qui n'existe pas.

- **Errors**

  - On ne doit pas les identifier et le programme s'arrête en les rencontrant.

- **Runtime exceptions**

  - Ne peuvent être prévues (dans certains cas)

Gestion des exceptions

- *try / catch / finally* : récupération
- *throw* : lancement
- Classe System.Exception et ses dérivées
- Il est possible de définir des exceptions

```
try
   {
      throw new NullReferenceException();
   }
   catch(NullReferenceException e)
   {
      Console.WriteLine("Exception {0}.", e);

   }
   finally
      ...
```

Using pour les exceptions

- Équivalent d'un *try / finally* + Close()
- Seulement pour certaines classes
- Ajouter éventuellement *try / catch* autour

```
using(StreamReader sr = new StreamReader("a.txt"))
   {
      ...
   }
```

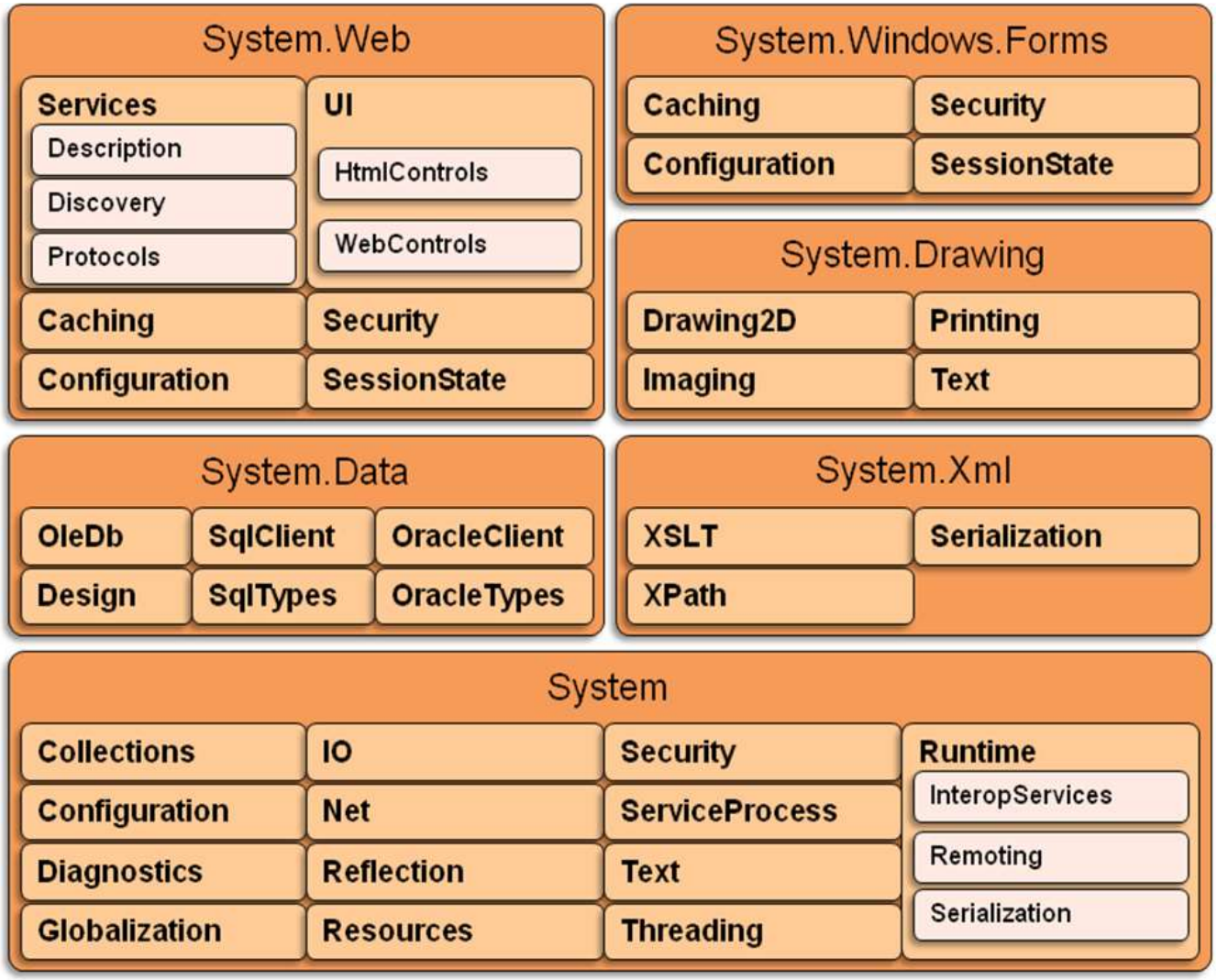
Bibliothèque de classes .Net

Définition

BCL : Base Class Library

Il s'agit des classes fondementales sur lequel les applications .NET sont construites.

Bibliothèque de classes



Chaines de caractères

System.String :

- Comparaison : Compare, Equals
- Concaténation : Concat, Join
- Découpage : Split, Substring
- Recherche : StartsWith, EndsWith, IndexOf, ...
- Mise en forme : Format, PadLeft/Right, TrimStart/End, …
- Longueur : Length
- Opérateurs : = <>

Date

System.DateTime :

- Comparaison : Compare, Equals
- Opération : Add, Substract
- Conversion : Parse, ToString
- Recherche : Date, Day, Hour, Month, …
- Opérateurs : + - = <> > >= < <=

Collections

System.Collections :
Regroupe des classes pour gérer des ensembles d'objets :

- Faiblement typée : ArrayList, Hashtable, Queue, Stack, ...
- Fortement typée (System.Collections.Generic) : List, Dictionnary, HashSet, Queue, Stack, ...

Génerics

Utilisés pour typer :

- une classe :

```
public class classHolder<T> {
    public void processNewItem(T newItem)
        // code qui traite un objet de type T.
    }
}
```

- un objet :

```
List<string> stagiaires = New List<string>()
```

- un paramètre d'une méthode

```
void Saisie(out List<string> elements) { ...}
```

Contraintes sur les Génerics

Sécuriser la vérification du typage : where

Options de contraintes :

where T : class

T doit être un type référence

where T : struct

T doit être un type valeur

where T : new()

T doit avoir un constructeur sans paramètres

where T : <ClassType>

T doit hériter de la classe « ClassType »

Parcours

```
List<string> maCollection = new List<string>() ;
foreach(string s in maCollection)
{
    Console.WriteLine(s);
}
' Equivalent via un enumérateur.
IEnumerator monEnum = maCollection.GetEnumerator();
monEnum.Reset();
string monElement;
while monEnum.MoveNext(){
     monElement = monEnum.Current();
     Console.WriteLine(monElement);
}
```

Itérateurs

Permet un parcours personnalisé d'une collection

```
foreach (var letter in Letters)
    Console.Write(letter) ;

…
private iterator IEnumerable<Char> Letters(){
    char currentCharacter  = 'a';
    do{
        yield currentCharacter ;
    }while (currentCharacter++ < 'z')
}
```

Entrées/Sorties

System.IO :
Regroupe des classes pour lire et écrire des données dans des fichiers ou des flux de données

Entrées/Sorties

- **Stream** : transfert de données
- **Principe d'utilisation d'un flux**:
  - Ouverture du flux
  - Identification de l'information (lecture/écriture)
  - Fermeture du flux

```
StreamReader sr =new StreamReader(filename);
   string s = sr.ReadToEnd();
   sr.Close();
```

```
using(StreamReader sr = new
   StreamReader("a.txt"))
   {
      string s = sr.ReadToEnd();
   }
```

Interfaces Graphiques (Windows Forms)

Interfaces graphiques

- Création d'un projet & Architecture de l'application
- Les références
- Le point d'entrée
- Les objets ApplicationContext et Application
- Lancement et arrêt de l'application
- Le fichier AssemblyInfo.cs

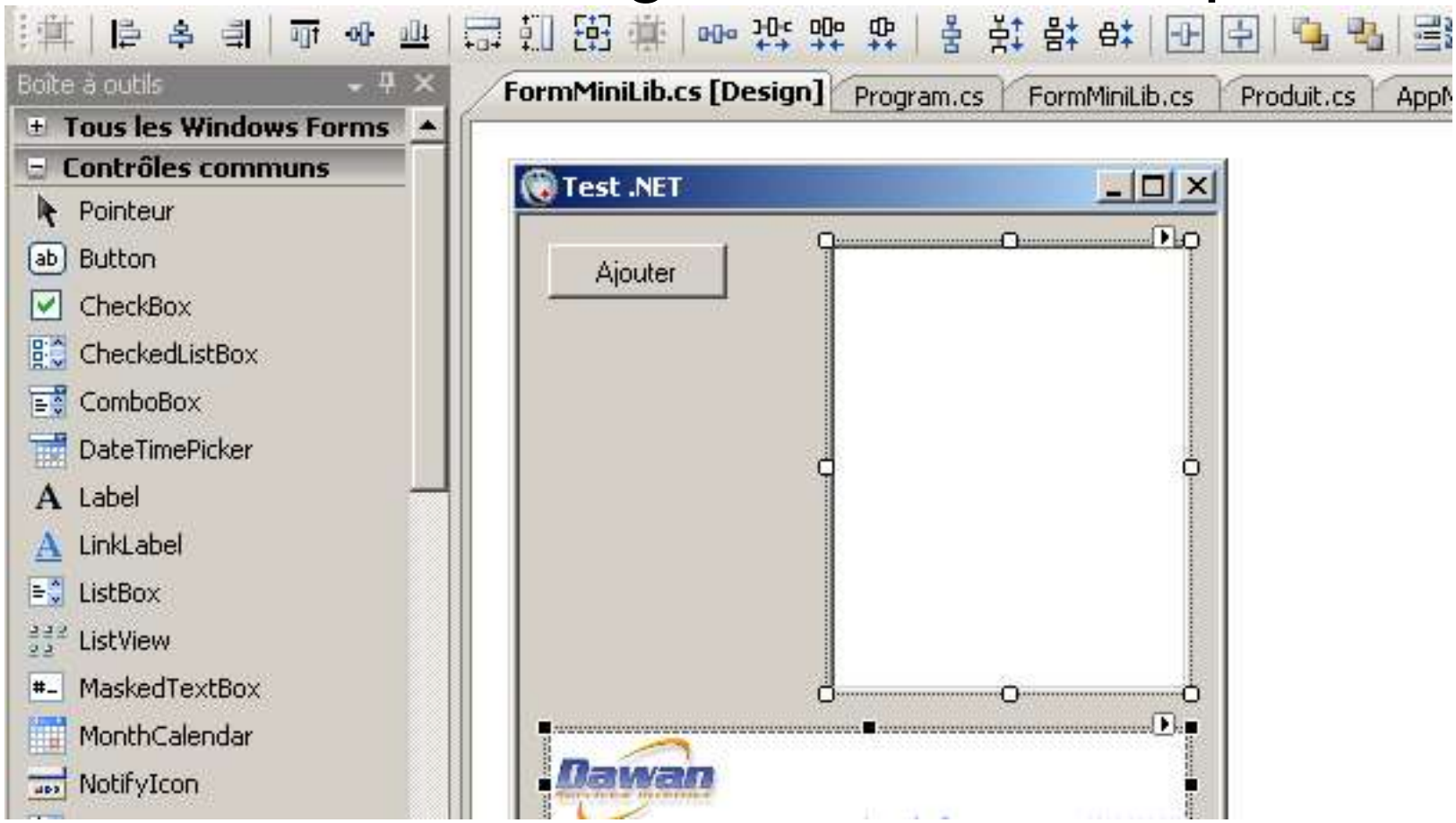
Types d'interfaces

Deux approches :

- WinForms : encapsulation des objets Win32
- WPF (Windows Presentation Foundation) : abstraction totale des contraintes de l'OS

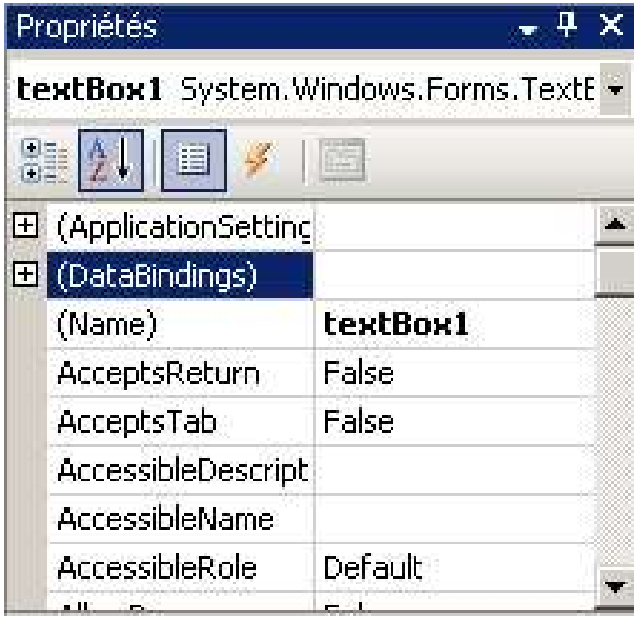
Manipulation

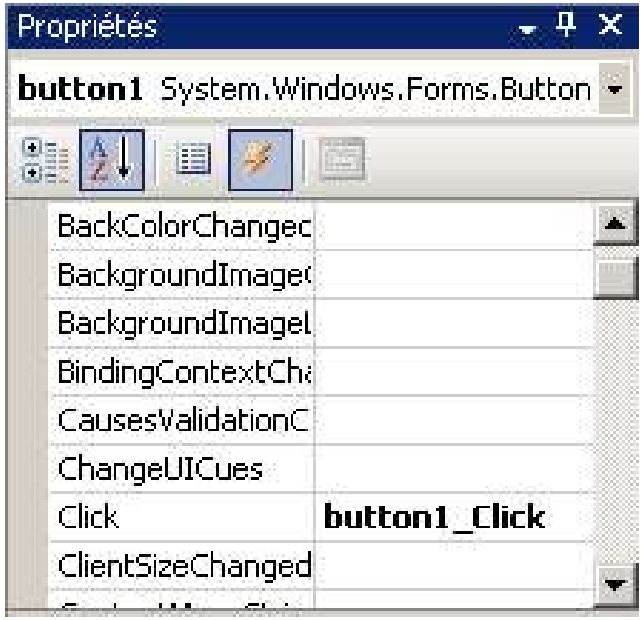
- Un éditeur complet est utile
- Le code est alors généré automatiquement



Manipulation : propriétés, événements

- Chaque composant a des propriétés et événements à disposition
- Par défaut les propriétés ont des valeurs standards (sauf : Name et Location), et les événements aucun délégué

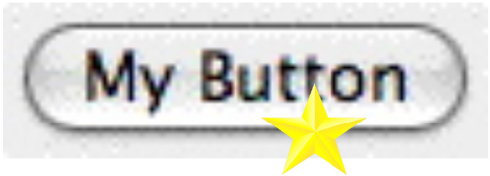




Manipulation : propriétés, événements

- Pour intercepter un événement, un listener doit être associé au composant
- Le listener est appelé lors de l'évènement
- Il peut y avoir plusieurs listener pour un même événement sur un même composant

Listener 1



Créations

- Base : les forms (fiche / fenêtre), associées à des fichiers de code
- Contrôles simples : Label, TextBox, Button etc...
- Gestions des événements (standards et spécifiques)
- Listes (ex : ListBox : Items, SelectedIndex)
- Menus (habituel et contextuel)
- OpenFileDialog et SaveFileDialog
- FontDialog, ColorDialog, FolderBrowserDialog, etc...

Configuration et Déploiement

- Fichier de configuration

```
<configuration>
  <appSettings>
    <add key="myLabel" value="Label prédéfini" />
  </appSettings>
</configuration>
```

```
string label =
   ConfigurationManager.AppSettings("myLabel")
```

- Déploiement WindowsInstaller
- ClickOnce

Assemblages

- Assemblage (de classes)
  - Exécutable ou bibliothèque dynamique
  - Créer une bibliothèque :
    - Projet de bibliothèque de classe
    - Contient des classe
    - Nom fort : mettable dans le GAC
  - Utiliser une bibliothèque :
    - Ajouter comme « référence » du projet
    - « Copie locale » : copiée à côté de l'utilisateur

Programmation Orientée Objet

Définition

L'orienté-objet = approche de résolution algorithmique de problèmes permettant de produire des programmes modulaires de qualité..

Objectifs :

- développer une partie d'un programme sans qu'il soit nécessaire de connaître les détails internes aux autres parties;

- Apporter des modifications locales à un module, sans que cela n'affecte le reste du programme;

- Réutiliser des fragments de code développés dans un cadre différent.

Qu'est-ce qu'un Objet ?

Objet = élément identifiable du monde réel, soit concret (voiture, stylo,...), soit abstrait (entreprise, temps,...)

Un objet est caractérisé par :

- son état (les données de l'objet)
- son comportement (opérations)

Qu'est-ce qu'une Classe ?

- Une classe est un type de structure ayant des attributs et des méthodes.
- On peut construire plusieurs **instances** d'une classe.

Différences entre classes et structures :

- Une structure est de type valeur, une classe est de type référence.
- Par défaut tout membre d'une structure est public alors qu'il est privé pour une classe.
- Une structure ne peut être héritée, une classe oui.
- Une structure ne nécessite pas de constructeur
- Tout champ d'une structure ne peut être initialisé que s'il est déclaré statique ou constant

Unified Modeling Language

UML = Language pour la modélisation des classes, des objets, des intéractions etc...

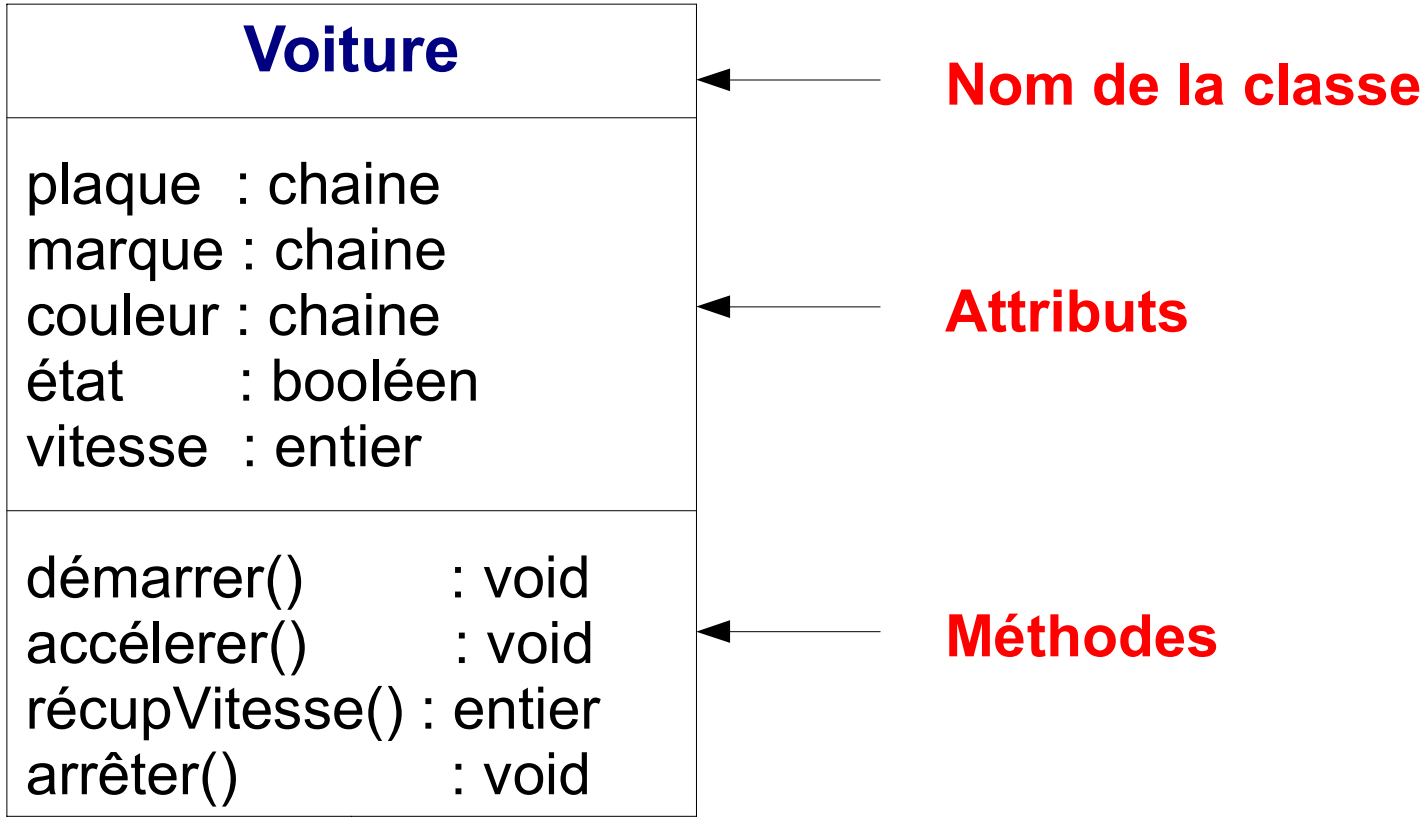
UML 2.0 comporte ainsi treize types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information :

- Diagramme fonctionnel

- Diagrammes structurels (statiques)

- Diagrammes comportementaux (dynamiques)

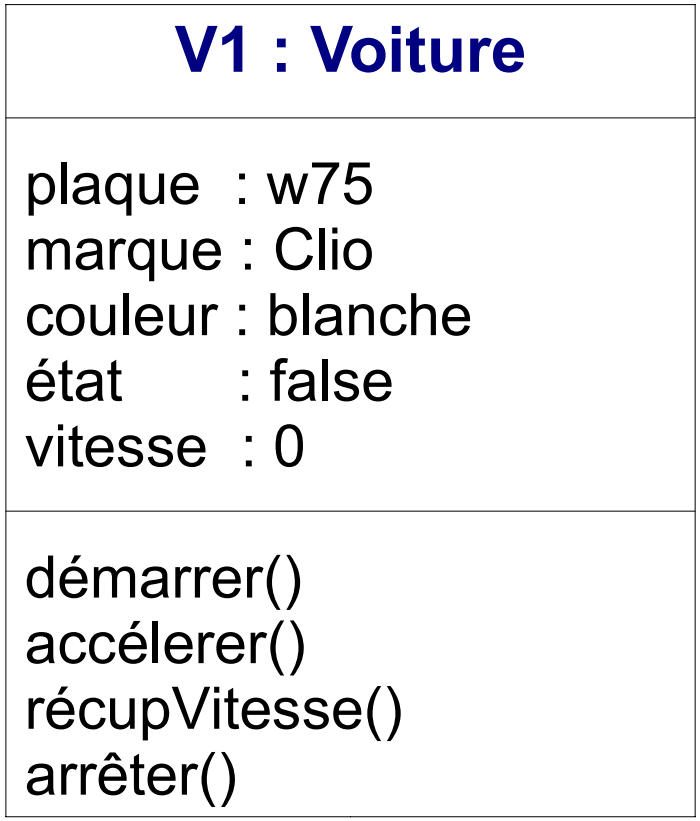
Représentation UML d'une Classe



Les attributs ou les méthodes peuvent être précédés par un opérateur (+, #, -) pour indiquer le niveau de visibilité

Représentation UML d'un Objet

Un objet est représenté graphiquement par un rectangle à coins divisé en 3 parties pour inscrire l'identité, les champs et les opérations.



L'accès aux membres d'un objet s'effectue grâce au point (.) :

v1.plaque
v1.démarrer()

Déclaration d'une Classe

```
public class NomClasse {

   // Attributs
      …

   // Méthodes
      …

   // Propriétés
      …
}
```

Instanciation d'objets

Un constructeur est une méthode particulière appelée automatiquement à chaque "création" d'un objet (soit par déclaration, soit par allocation dynamique dans le tas).

Cette méthode permet d'initialiser l'objet créé avec des valeurs valides.

On parle ainsi de processus d'instanciation.

```
Classe nomObjet =
   new Constructeur([paramètres]);
```

Le mot clé « this »

- Référence à l'instance actuelle de la classe
- Utilisations :
  - faire référence à un attribut
  - manipuler l'objet en cours
  - déclarer des indexeurs

Encapsulation

- **Rassembler des attributs et méthodes propres à un type donné afin d'en restreindre l'accès et/ou d'en faciliter l'utilisation et la maintenance.**

- **Modificateurs d'accès :**
  public, protected, internal, protected internal, private
- **Propriétés (get et set) :**
  - Récupérer/Définir la valeur d'un champ
  - Associé ou non à un attribut (propriété publique, méthode privée)

Propriétés

- Intermédiaires entre les attributs et l'extérieur de la classe :

```
private int x;
public int X
{
   get
   {
      return _x;
   }
   set
   {
      _x = value;
   }
}
```

Déclaration d'attributs et de méthodes

- Variables / Méthodes d'instance : une valeur de la variable par instance, seules les méthodes d'instances peuvent y accéder
- Variables / Méthodes de classes (static) : une valeur de la variable en tout (pour le processus), les méthodes de classes peuvent y accéder. Appel à l'aide du nom de la classe :

```
MaClasse.MethodeStatique();
```

Méthode principale

- La méthode **Main** représente le point d'entrée d'une application en exécution.

- Elle peut être intégrée dans une classe existante ou écrite dans une classe séparée.

```
public class Voiture
   // Méthode principale
   public static void Main(){
   // Création d'une instance de la classe
Voiture
      Voiture Clio = new Voiture();
      ...
    }
}
```

Constructeur

- Le constructeur est une méthode spéciale dans la classe permettant la création d'instances.
(Si on ne définit pas de constructeur, le compilateur en créera un par défaut).

- Un constructeur doit obligatoirement porter le nom de la classe (destructeur : nom précédé de « ~ »).

```
class Voiture {

   public Voiture() {
      // Corps du contructeur
    }
   …
}
```

Constructeurs multiples

Il est possible de déclarer plusieurs constructeurs différents pour une même classe, afin de permettre plusieurs manières d'initialiser un objet.
Les constructeurs diffèrent alors par leur signature.

```
class Voiture {

    // Contructeur par défaut sans paramètres
    public Voiture() {
        // Corps du contructeur
     }

    // Contructeur avec un paramètre
    public Voiture(String couleur):this() {
        // Corps du contructeur
     }
    …
}
```

Classes Imbriquées

- Définir une classe à l'intérieur d'une classe
- Seules classes ayant des modificateurs protected ou private

```
public class MonDataSet
{
    protected class MaDataTable {
        ...
    }
}
```

Classes Partielles

Fractionner la définition d'une classe en plusieurs fichiers sources combinés lors de la compilation

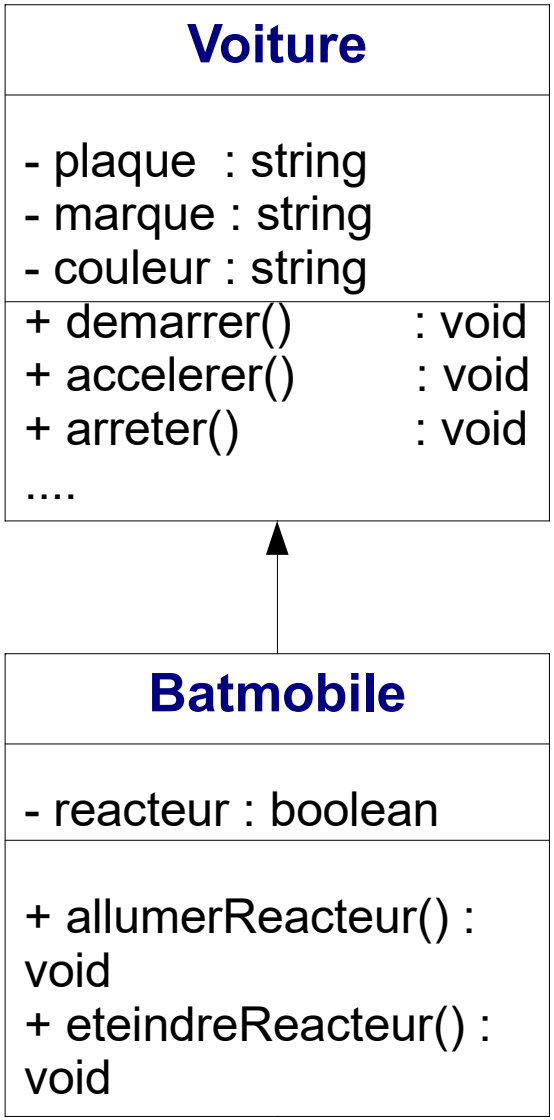
```
//fichier1.cs
public partial class Form1
{
   ...
}

//fichier2.cs
public partial class Form1
{
  ...
}
```

Héritage

- L'héritage permet de créer la structure d'une classe à partir des membres d'une autre classe.

- La sous-classe hérite de tous les attributs et méthodes de sa classe mère (selon la visibilité de ceux-ci).

- Pas d'héritage multiple : une classe ne peut hériter que d'une classe.



Héritage

- Appel du constructeur de la classe mère : **base**
- Redéfinition de méthodes : virtual, override

```
class Voiture {
    public Voiture() {

    }
}
class Batmobile:Voiture {
    public Batmobile():base() {
        ...
    }
}
```

Redéfinition

La **redéfinition** (***overriding***) consiste à réimplémenter une version spécialisée d'une méthode héritée d'une classe mère (les signatures des méthodes dans la classe mère et la classe fille doivent être identiques).

```
class Voiture {
    public void description() {
        ...
        details();
    }
    public void details() {
        ...
    }
}
class Batmobile:Voiture {
    public void override details() {
        ...
    }
}
```

Classes scellées

On peut interdire l'héritage d'une classe grâce au mot-clé : sealed

```
class sealed Voiture {
    public void description() {
        ...
        details();
    }
    public void details() {
        ...
    }
}
```

Objet Avancé en C#

Paquetages ou Espaces de noms

Espace de noms = regroupement de classes qui traitent un même problème pour former des « bibliothèques de classes ».

Une classe appartient à un espace de noms s'il existe une ligne au début renseignant cette option :

```
Namespace Initiation
{
   public class myClass {
      …
   }
}
```

System.Object

Toute classe hérite directement ou indirectement de la classe System.Object

- Méthodes :
  - ToString()
  - Equals()
  - GetType()
  - Finalize()

- Opérateurs :
  - TypeOf : `Type t = typeof(MaClasse)`
  - Is : `if (x is MaClasse)`
  - As : `y = x as MaClasse;`

Polymorphisme

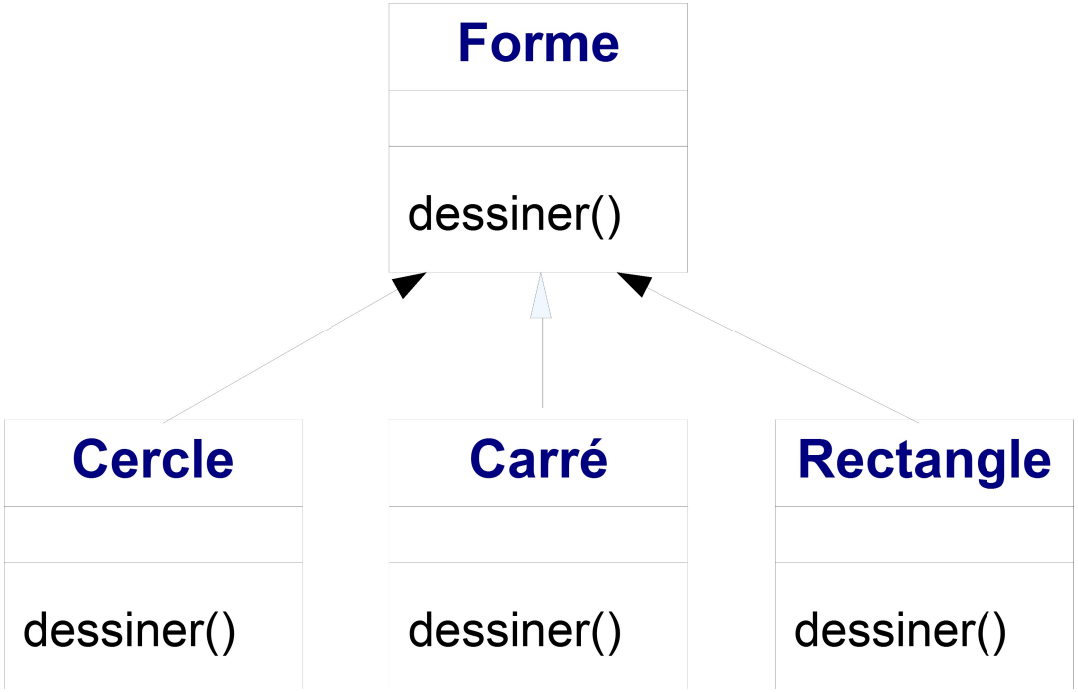
Le polymorphisme est la propriété d'une entité de pouvoir se présenter sous diverses formes. Ce mécanisme permet de faire collaborer des objets entre eux sans que ces derniers aient à donner leur type.

Exemples :
- On peut avoir une voiture prioritaire avec le type Voiture.

- On peut créer un tableau de Voitures et placer à l'intérieur des objets de type Voiture et d'autres de type VoiturePrioritaire

Classe Abstraite

Définit un type de squelette (Stéréotype) pour les sous-classes.Si elle contient des méthodes abstraites, les sous-classes doivent redéfinir le corps des méthodes abstraites.

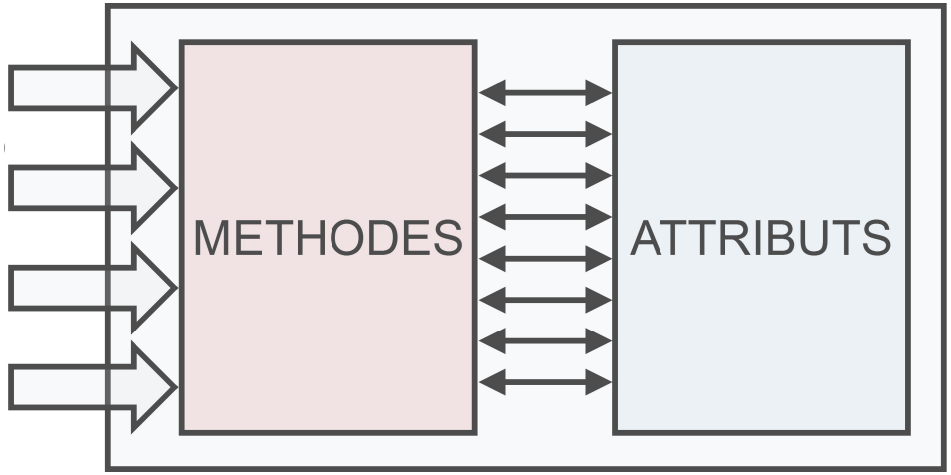


Classe Abstraite

```
public abstract class Forme
{
    ...
    // méthode abstraite
    public abstract void dessiner();
}

public class Cercle : Forme
{
    ...
    public override void dessiner()
    {
        ...
    }
}
```

Interface

- Une classe abstraite marquée par le mot clé interface contenant juste des signatures de méthodes dans le but de forcer la redéfinition.




```
public interface INomInterface
{
        .....    // NB : méthodes sans corps et sans
modificateurs d'accès
}
```

Processus

C# permet de démarrer et d'arrêter des processus locaux comme ouvrir un navigateur ou un éditeur de texte.

```
Process p2 = Process.Start("Notepad.exe")
```

La classe Process fournit les propriétés et les méthodes pour gérer l'état des processus

Virtualisation

La virtualisation consiste à définir un comportement par défaut à une méthode qui sera redéfinie

```
class Voiture {
    public void description() {
        ...
        details();
    }
    public virtual void demarrer() {
        ...
    }
}
class Batmobile:Voiture {
    public void new demarrer() {
        ...
    }
}
```

Occultation

L'**occultation** (***shadowing***) consiste à redéfinir une méthode d'une classe mère et à « casser » le lien vers la classe mère

```
class Voiture {
    public void description() {
        ...
        details();
    }
    public void details() {
        ...
    }
}
class Batmobile:Voiture {
    public void new details() {
        ...
    }
}
```

Surcharge d'opérateurs
Délégations

Surcharge d'opérateurs

- Gestion du comportement d'un objet face à un opérateur
- Opérateurs :
  - Unaires : + - Not IsTrue IsFalse CType
  - Binaires : + - * / \ & Like Mod And Or Xor ^ << >> = <> > < >= <=

```
public static Type1 Operator +(op1 As Type1,
op2 As Type1) {
   Type1 Result = ...
   Return Result;
}
```

Délégations

- Définition de prototypes de fonctions

```
// définition du délégué
public delegate int Operation(int n1, int n2);

// méthode correspondant au prototype
public static int Ajouter(int n1, int n2) {
    return n1+n2;
}

// Appel d'un délégué
Operation add1 = new Operation(Ajouter);
// Appel d'un délégué C# 2.0
Operation add2 = delegate(int n1, int n2) { return n1+n2;}
// Appel d'un délégué C# 3.0
Operation add3 = (n1, n2)=> n1+n2;
int res1 = add1(5,7);
int res2 = add2(5,7);
int res3 = add3(5,7);
```

- Delegates ou abstract ?

Expressions Rationnelles

Présentation

- Définir des patterns (modèles) pour des chaînes de caractères ou autre.

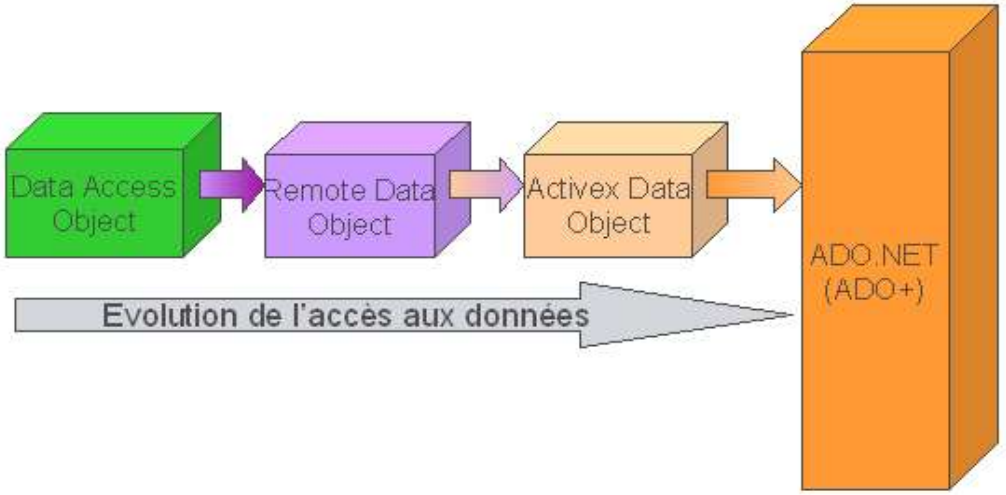
- Utilisation de la classe :
  System.Text.RegularExpressions.Regex

```
MatchCollection mc =
   Regex.Matches("abracadabra",
      "(a|b|r)+");
for (int i = 0; i < mc.Count; i++)
{
   s += mc[i].Value+" ";
}
```

Connexions BDD (ADO.NET)

Présentation

- ADO (ActiveX Data Objects)



- Accès en mode déconnecté à la base de données.
- Indépendant du SGBD
- Inter-opérable avec d'autres systèmes (XML, AD, Index Server)
- Performant et utilisable sur Internet

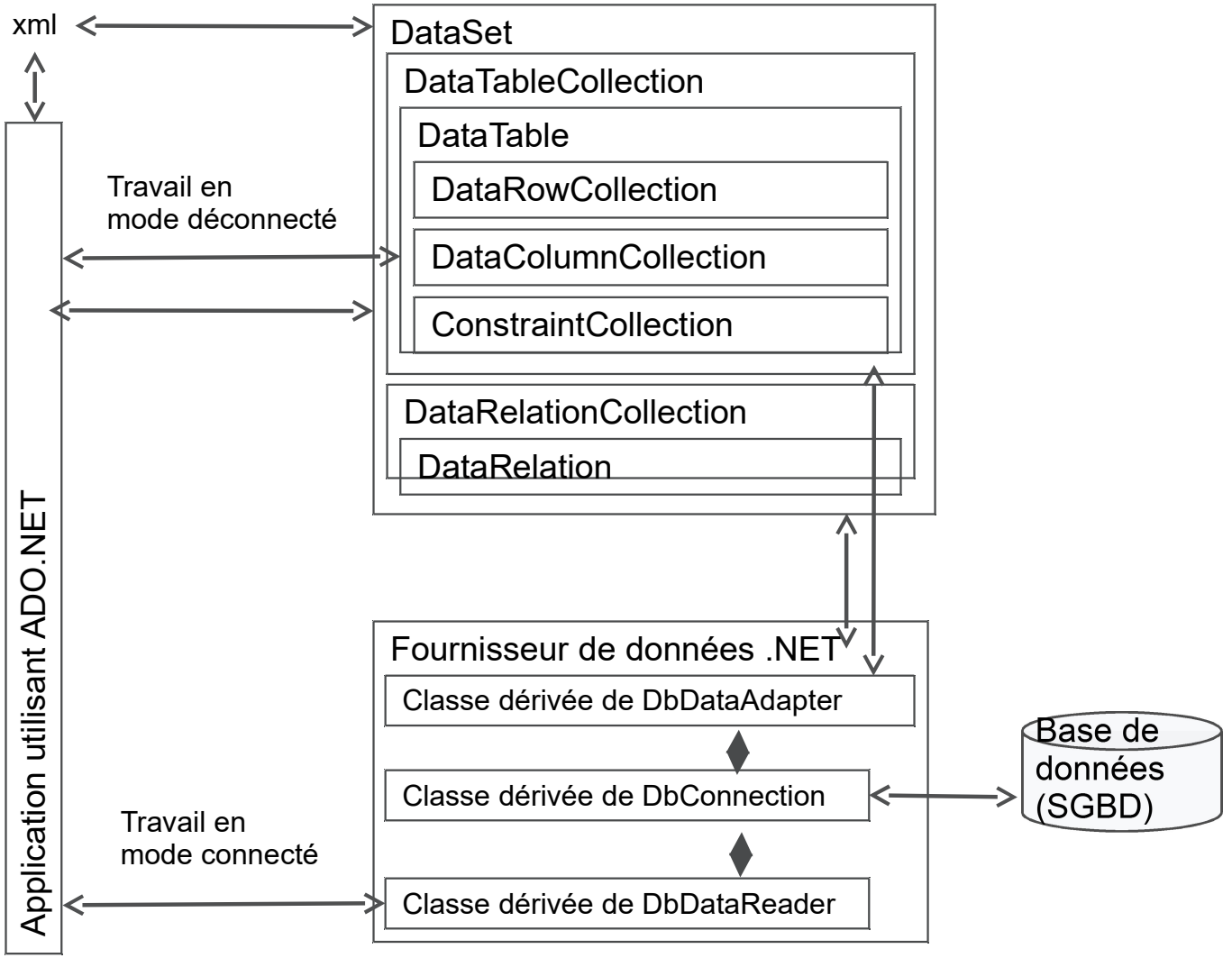
Avantages

- Pool de connexion
- Création générique du fournisseur d'accès
- Objet ConnectionStringBuilder

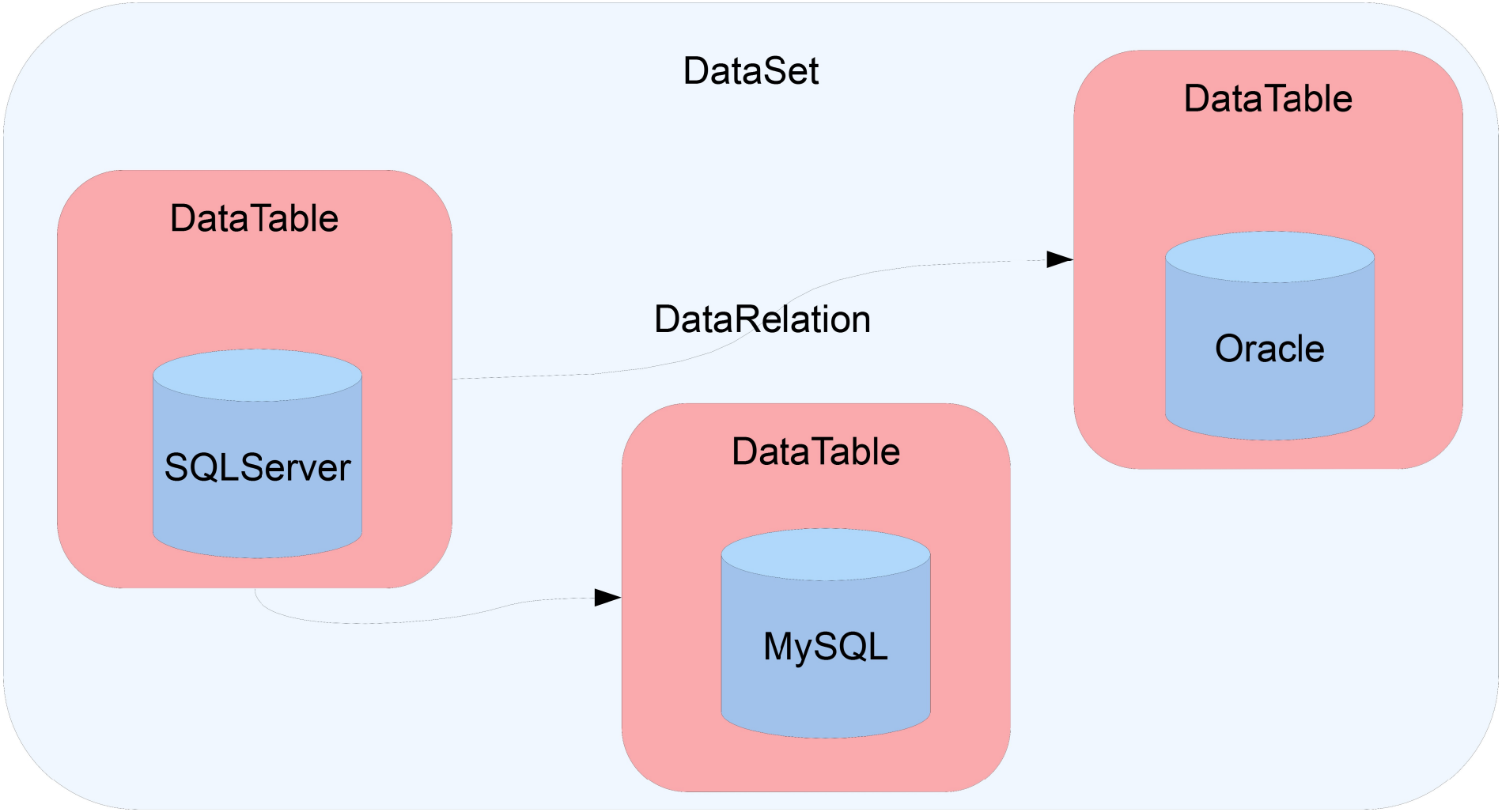
Espaces de noms :

- System.Data
- System.Data.Common
- System.Data.SqlTypes
- System.Data.SqlClient
- System.Data.OleDB

Schéma


xml
DataSet
DataTableCollection
DataTable
DataRowCollection
DataColumnCollection
ConstraintCollection
DataRelationCollection
DataRelation
Travail en
mode déconnecté
Application utilisant ADO.NET
Fournisseur de données .NET
Classe dérivée de DbDataAdapter
Classe dérivée de DbConnection
Classe dérivée de DbDataReader
Travail en
mode connecté
Base de
données
(SGBD)

Architecture



Fournisseurs d'accès

- Types et fonctionnalités spécifiques à une source de données

- Exemples :
  - SQLServer : System.Data.SqlClient
  - OLEDB : System.Data.OleDb
  - MySQL : MySql.Data.MySqlCient

Utilisation

- DbConnection
- DbCommand
- DbDataReader
- DbDataParameter
- DbTransaction

Exemples :

- Connexion à une BDD SQL Server
- Connexion à une BDD MySQL

Exemple

Définition du fournisseur de connexion dans la configuration :

```
<system.data>
   <DbProviderFactories>
      <add name="FournisseurSql1"
         type="System.Data.SqlClient.
         SqlClientFactory, System.Data,
         Version=2.0.0.0, Culture=neutral,
         PublicKeyToken=b77a5c561934e089"
      />
   </DbProviderFactories>
</system.data>
```

Exemple

Utilisation du fournisseur de connexion :

```
DbProviderFactory factory =
     DbProviderFactories.
     GetFactory("FournisseurSql1");
DbConnection conn = factory.CreateConnection();
conn.ConnectionString = "...";

conn.Open();
DbCommand command = conn.CreateCommand();
command.queryString = "SELECT * FROM os";
DbDataReader reader = command.ExecuteReader();
while (reader.Read())
     Console.Write(reader[1]);

reader.Close();
conn.Close();
```

Threads

Utilisation

- Lancement de plusieurs tâches simultanées
- La classe System.Threading.Thread
- Création de threads
- Communication entre threads
- Gestion de la concurrence
- La classe BackgroundWorker

Les timers

- Déclenchement (un délégué) plusieurs fois après un délai
- Besoin d'une méthode appelée de signature :

```
static void xyz(object state,
   ElapsedEventArgs e)
...


Timer t = new Timer( xyz, "Message",
   0, //appel tout de suite
   1000); //et toutes les secondes
t.Stop();
```

Utilisation

- Lancement de plusieurs tâches simultanées
- La classe *System.Threading.Thread*
  - Création de threads

```
public static void tMeth()
…
t = new Thread(new ThreadStart(tMeth));
t.Start();
```

  - Communication entre threads
  - Gestion de la concurrence
- La classe *BackgroundWorker*

Threads : la concurrence

- Plusieurs processus simultanés accèdent à la même donnée
- Verrouiller à un code : lock()
- lock a un paramètre : instance quelconque concernée par un seul lock en même temps :

```
Xyz xyz;
...
lock(xyz)
{
   x = x + 18;
}
```

DAWAN Paris, Tour CIT Montparnasse - 3,rue de l'Arrivée, 75015 PARIS
DAWAN Nantes, Le Sillon de Bretagne - 26e étage - 8, avenue des Thébaudières, 44800 ST-HERBLAIN
DAWAN Lyon, Le Britannia, 4ème étage - 20, boulevard Eugène Deruelle, 69003 LYON
formation@dawan.fr