Dawan
Training is our way

Langage C#

Christophe Fontaine
cfontaine@dawan.fr

05/10/2020

Objectifs

- Apprendre à développer avec C#
- Créer des interfaces de gestion de bases
- Manipuler les objets de la plate-forme .NET

Bibliographie

- **C# 8 et Visual Studio 2019**
  Les fondamentaux du langage
  Sébastien Putier
  Éditions ENI - janvier 2020

- **Documentation .Net**
  https://docs.microsoft.com/fr-fr/dotnet/

- **Documentation C#**
  https://docs.microsoft.com/fr-fr/dotnet/csharp/

- **Documentation Visual Studio**
  https://docs.microsoft.com/fr-fr/visualstudio/ide/?view=vs-2019

Plan

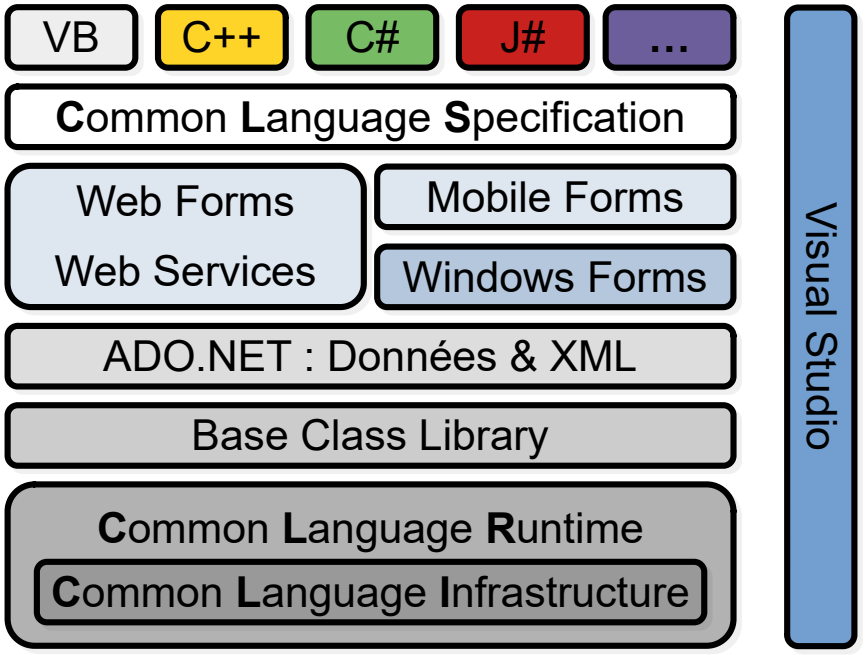
- Présentation
- Syntaxe de base
- Tableaux
- Méthodes et paramètres
- Exceptions
- Bibliothèque de classes .Net
- Interfaces Graphiques : WinForms
- Programmation Orienté Objet
- Accès à une base de données : ADO.NET
- Threads

Dawan
Training is our way

Présentation

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 Windows Update

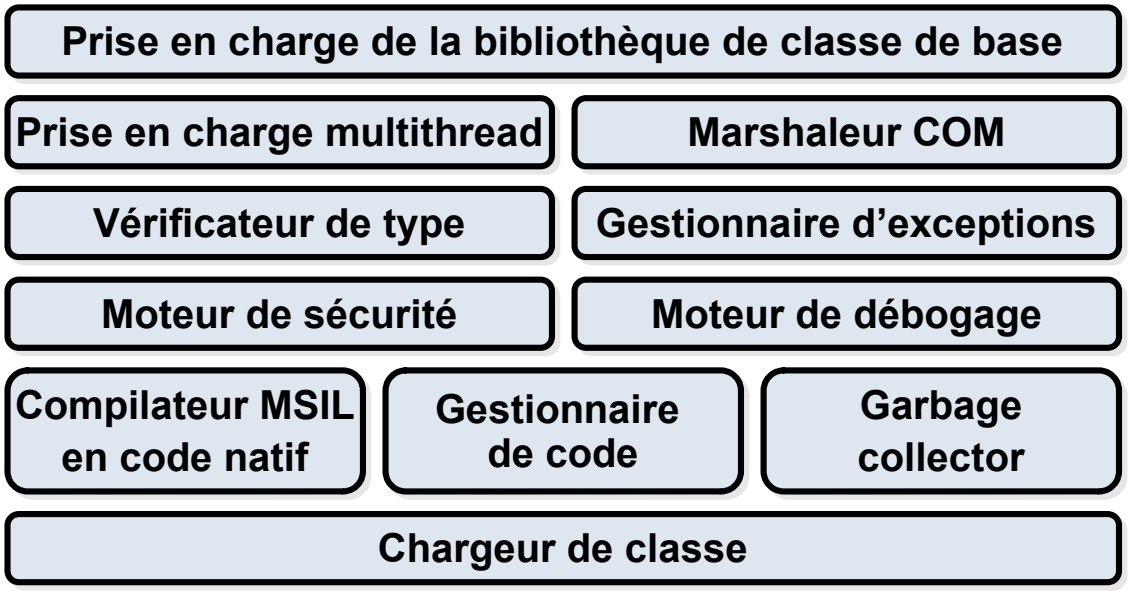


Historique

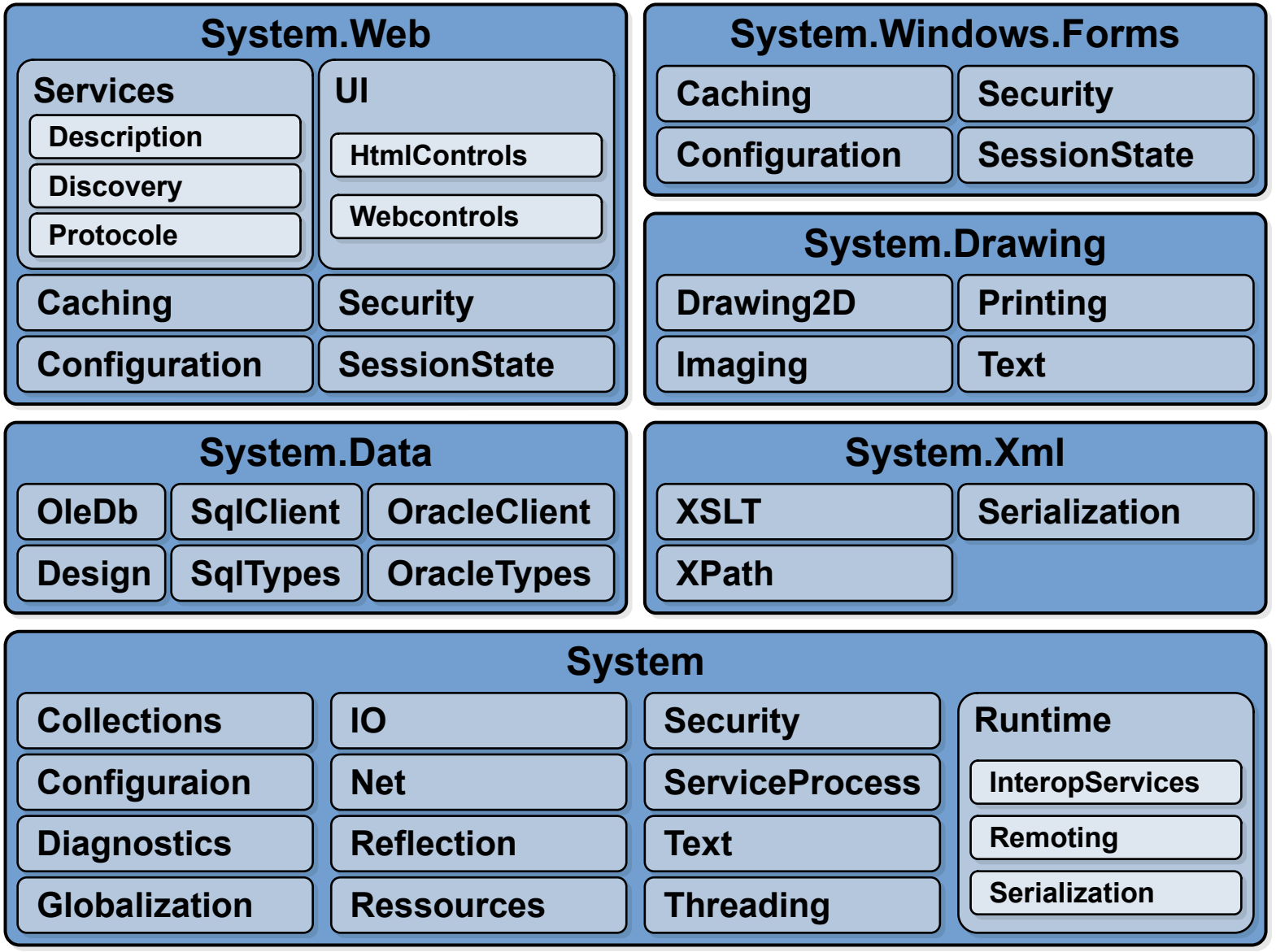
2000	Lancement du développement	**concepteur → Anders Hejlsberg**
2002	.NET Framework 1	**Winform, ASP.Net**
2003	.NET Framework 1.1	**Intégration ASP.Net, IPV6**
2005	.NET Framework 2.0	**64 bits, Type générique**
2006	.NET Framework 3.0	**WPF, WCF, WF, CardSpace**
2008	.NET Framework 3.5	**Linq, Entity Framework**
2010	.NET Framework 4.0	**Parallel Linq , DRL**
2012	.NET Framework 4.5	
2015	.NET Framework 4.6	**RyuJIT**
2017	.NET Framework 4.7	
2019	.NET Framework 4.8	
2020	.NET 5.0	**Fusion de .NET Framework et .NET Core**

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



Présentation C#

- Langage orienté objet de type sécurisé
- Très proche du C++ et du Java
- RAD
- Multi-plateformes (IL)
- Plusieurs versions successives (actuellement 8.0)

Développement C#

- Applications Windows
- Pages ASP.NET
- Services Web
- Services Windows
- C#.Net est Multi-plateformes

IDE :

- **Microsoft Visual Studio** payant ou version Community
- **Rider** payant et multi-plateforme
- **SharpDevelop** ou **MonoDevelop** open source mais moins performants

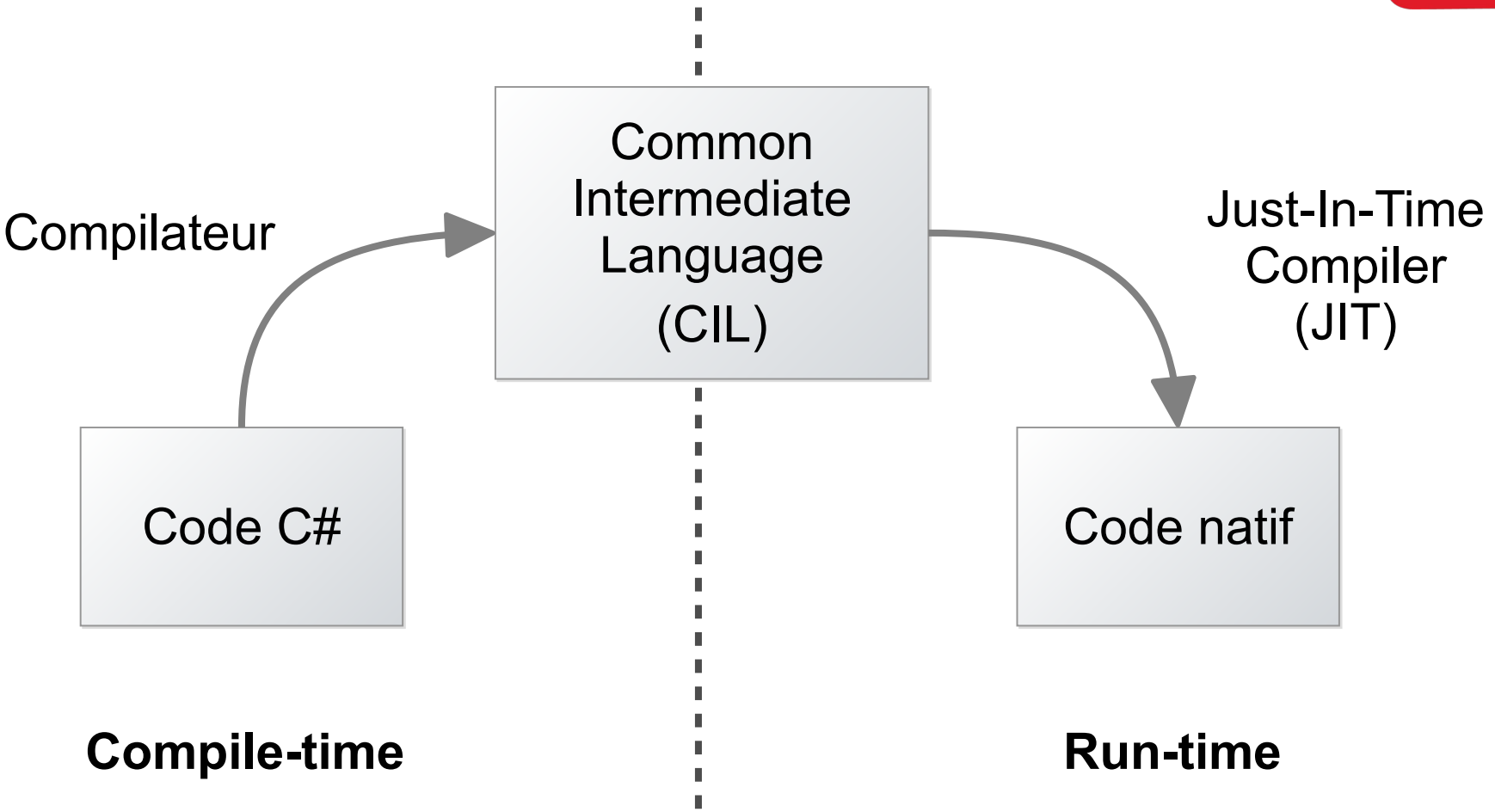



Programme C#

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

```
using System;

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

Étapes de compilation



En ligne de commande :

csc.exe msbuild.exe

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)

Dawan
Training is our way

Syntaxe de base

Base du langage

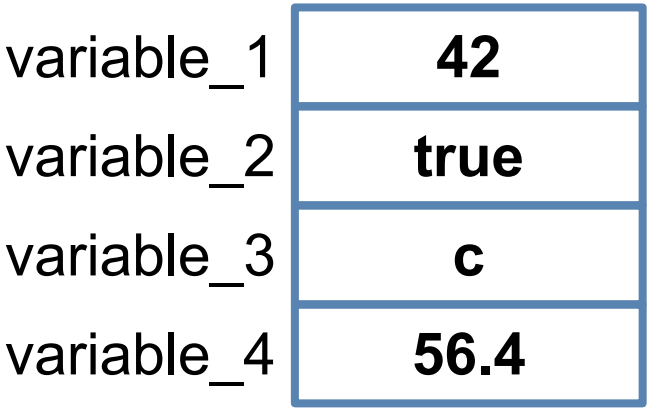
- Les instructions se terminent par un **;**
- Différences entre **minuscule** et **MAJUSCULE**
- Espaces / Tabulations / CR / LF sans conséquences
- Les fichiers sources sont encodés en **UTF-8**
- Bloc de code : suite d'instructions entre **{ }**
- Commentaire

```
// Commentaire fin de ligne

/* Commentaire
   sur plusieurs lignes
*/
```

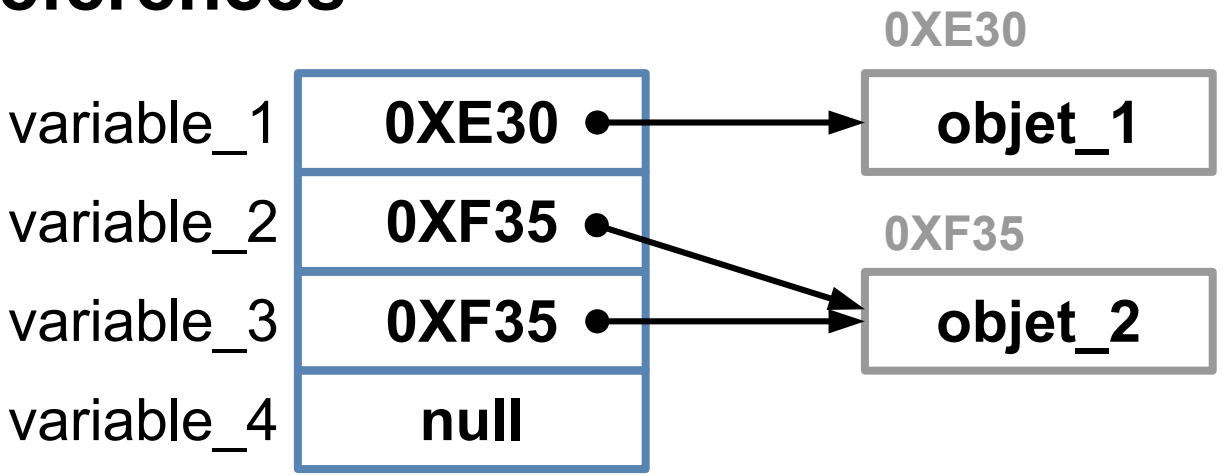
```
/// <summary>
///   Commentaires interprétés
/// </summary>
/// <param name="val"></param>
/// <returns></returns>
```

Types de données

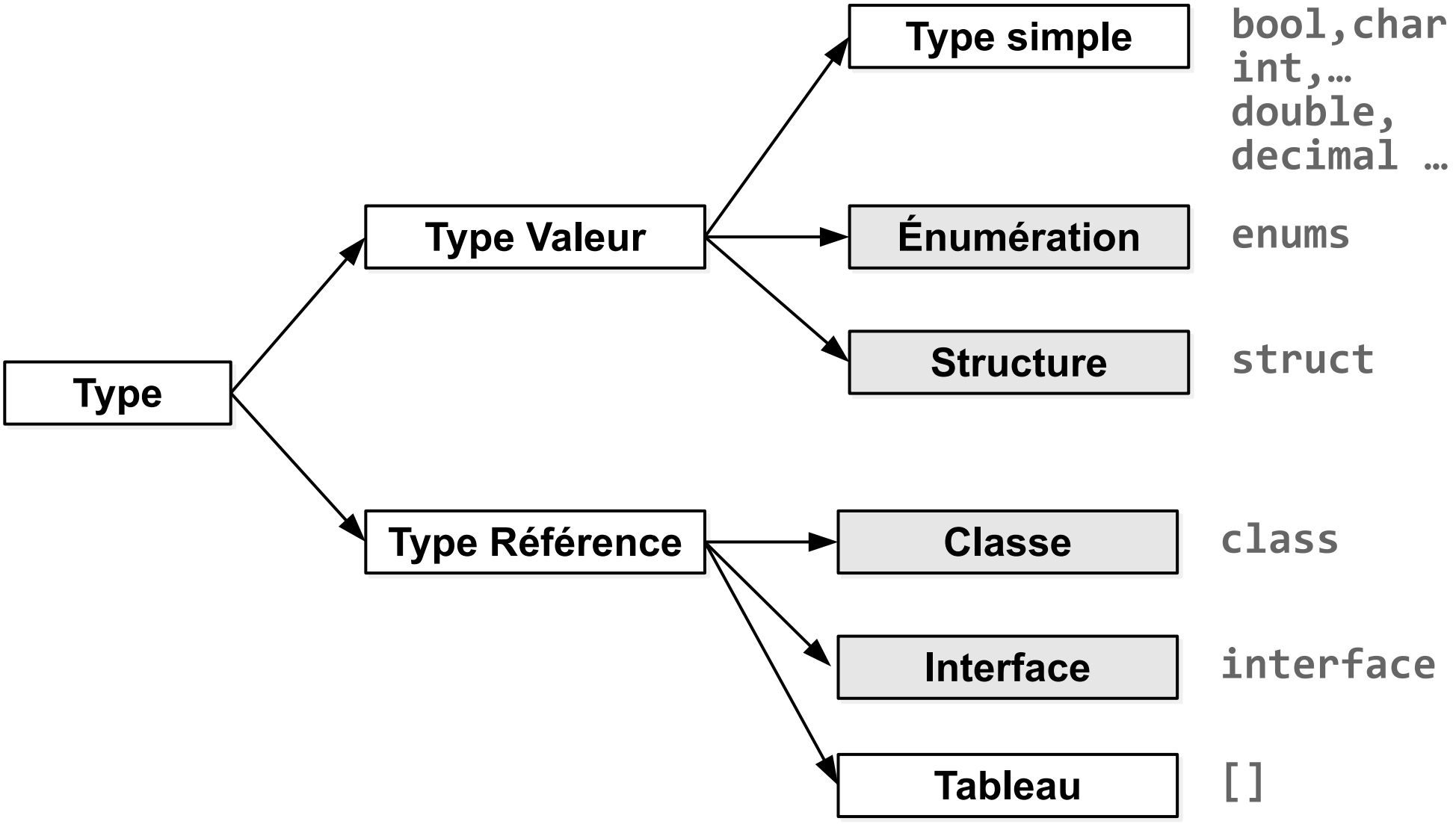
- **Types Valeurs**



- **Types Références**



Types de données



Types simples

Type	Type .NET	Description	Plage de valeurs
bool	System.Bool	booléen	true ou false
char	System.Char	caractère unicode sur **16** bits	'\u0000' à '\uFFFF'
sbyte	System.SByte	entier signé sur **8** bits	-128 à 127
byte	System.Byte	entier non signé sur **8** bits	0 à 255
short	System.Int16	entier signé sur **16** bits	-2^{15} à $2^{15}-1$
ushort	System.UInt16	entier non signé sur **16** bits	0 à $2^{16}-1$
int	System.Int32	entier signé sur **32** bits	-2^{31} à 2^{31}
uint	System.UInt32	entier non signé sur **32** bits	0 à $2^{32}-1$
long	System.Int64	entier signé sur **64** bits	-2^{63} à $2^{63}-1$
ulong	System.UInt64	entier non signé sur **64** bits	0 à $2^{64}-1$
float	System.Single	réel signé sur **32** bits	±1,5 x 10^{-45} à ±3,4 x 10^{38} **précision** → 6 à 9 chiffres
double	System.Double	réel signé sur **64** bits	±5,0 × 10^{-324} à ±1,7 × 10^{308} **précision** → 15 à 17 chiffres
decimal	System.Decimal	réel signé sur **128** bits	±1,0 x 10^{-28} to ±7,9228 x 10^{28} **précision** → 28 à 29 chiffres

Variables

- **Déclaration**

```
type nomVariable;
```

```
double valeur;
int i, j;
```

- **Initialisation**

```
nomVariable = valeur;
```

```
valeur = 134.8;
i = 42;
j = 0;
```

- **Initialisation pendant la déclaration**

```
char c = 'a';
double hauteur = 1.25, d = 1.26;
```

Variables

- **Typage déterminé par le compilateur**

  `var nomVariable = valeur`

  Pour les variables locales, avec le mot-clef **var** le compilateur peut déduire le type de la variable à partir du type de la valeur

  La variable doit être obligatoirement initialisée à la déclaration

```
int ie = 10;     // typé explicitement
var it = 10;     // typé implicitement
var x = ie;                      // x → int
var str = "type implicite";  // str → string
var hauteur = 123.65F;           // hauteur → float
```

Règle de nommage des identifiants

- Le nom doit commencer par : une **lettre** ou _
- Les **nombres** sont autorisés **sauf en tête**
- Ne doit pas être un mot réservé, sauf s'il est préfixé avec **@**

Correct	→	identifier	conv2Int	_test	@if
Faux	→	**3**dPoint	**public**	*$coffe	**while**

- Par convention on utilise le :
  - **P**ascal**C**ase pour les noms de membre, de type, d'espace de noms ou tous les éléments publics
  - **c**amel**C**ase pour les noms de paramètres ou tous les éléments privés

Porté d'une variable locale

- Sa portée se limite au bloc où elle est définie. Leur espace mémoire est libéré lorsque le bloc se termine (pile LIFO)

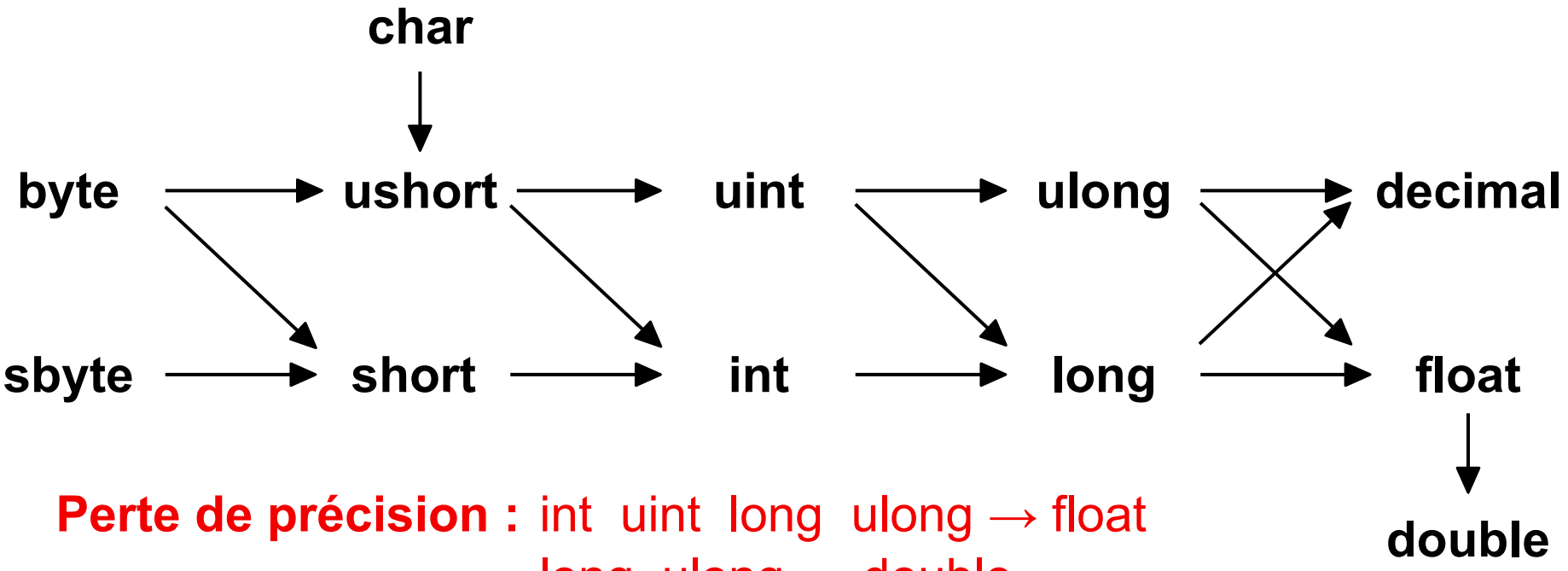
  ↳ **Chaque bloc de code a sa propre portée**

- Quand des blocs contiennent d'autre blocs
  Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse

```
int a = 10;
if (a > 0)
{
    int somme = a + 20;        // OK
}
Console.WriteLine(somme);      // erreur
```

Transtypage implicite (Automatique)

- Type inférieur vers un type supérieur
- Pour les types réel uniquement : float → double



Perte de précision : int uint long ulong → float
long ulong → double

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

Transtypage explicite (cast)

- Type supérieur vers un type inférieur
- Réel vers un entier
- double ou float → decimal

```
type variable = (type) variableToCast;
```

```
int i = 123;
short s = (short)i;

double d = 44.95;
decimal dec = (decimal)d;

float f = 3.45f;
sbyte b1 = (sbyte)f;

// Dépassement de capacité
int j = 130;
sbyte b2 = (sbyte)j; // b2 vaut -126
```

Fonctions de conversions

- La classe **Convert** contient des méthodes statiques permettant la conversion de type

  Les types de base pris en charge sont **Boolean**, **Char**, **SByte**, **Byte**, **Int16**, **Int32**, **Int64**, **UInt16**, **UInt32**, **UInt64**, **Double**, **Decimal**, **DateTime** et **String**

```
string s = "2.81";
double d = Convert.ToDouble(s);
int i = Convert.ToInt32(d);
int j = Convert.ToInt32("42");
```

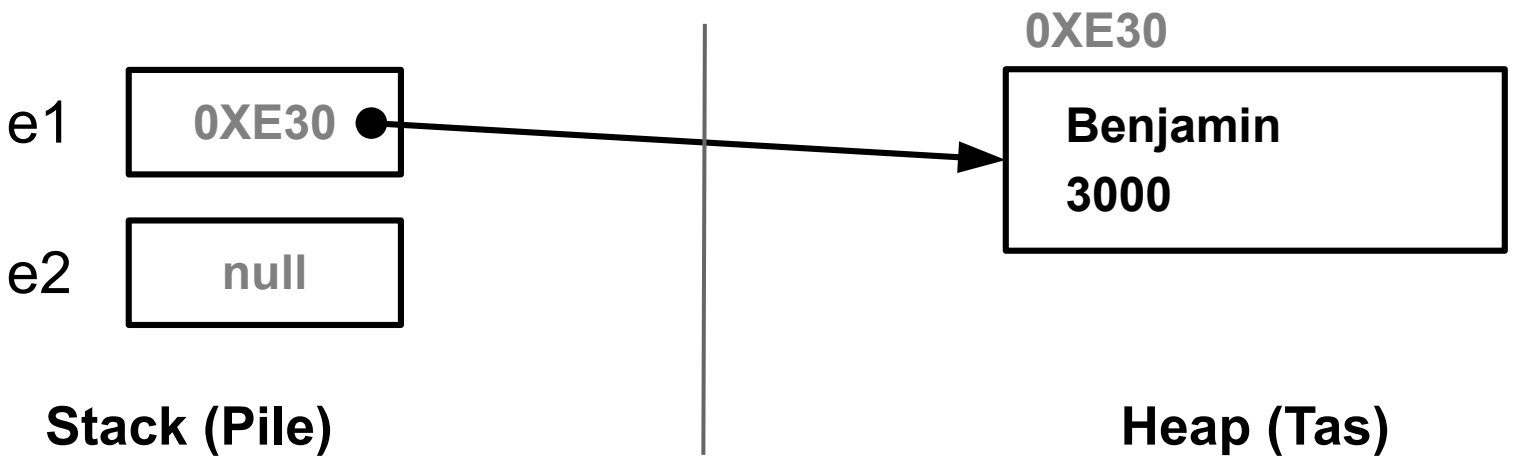
- Les classes **Int32**, **Double**... ont des méthodes **Parse** et **TryParse** qui permettent la conversion d'une chaîne de caractère vers ce type

```
int i = Int32.Parse("1258");
double d ;
bool b= Double.TryParse("1258", out d)
```

Type référence

- Il existe des types complexes :
  - string (chaîne de caractère)
  - tableau
  - …

```
Employe e1 = new Employe("Benjamin", 3000);
Employe e2 = null;
```



Nullable

- Un type **nullable** permet d'utiliser une valeur **null** dans un type valeur

  `type? nomVariable`

```
int? i = null;      // autorisé
```

- La propriété **HasValue** permet de tester si la variable contient une valeur. On peut tester aussi l'égalité à **null**

```
if (i.HasValue)     // équivaut à i!=null
{    // …
```

- La conversion d'un type vers un type nullable est implicite
- La conversion d'un type nullable vers un type est explicite ou obtenu avec la propriété **Value**

```
int? i = 1;
int j = (int)i;     // équivaut i.Value ;
```

Constantes

```
const type nom-constante = valeur;
```

- Une constante doit obligatoirement être initialisée à la déclaration
- Seul les **types valeurs** et les **string** peuvent être des constantes

```
const int A = 26;
const float Pi = 3.14159;
const string Constante = "constante";
const int X;// erreur, une constante doit être initialisée
A = 4;      // erreur, une constante ne peut être modifiée
```

Énumérations

- Une énumération est un type de données, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs qui sont des constantes prédéfinies

```
enum non : type { enumerator = constexpr,… }
```

```
enum Direction { NORD, EST, SUD, OUEST };
Direction dir = NORD;
```

- Par défaut, une énumération est de type **int** , on peut définir un autre type qui ne peut-être que de type **intégral**
- Par défaut, Les membres d'une énumération sont numéroté séquentiellement et commence à **0**
  On peut définir pour chaque membre une valeur à la déclaration

```
enum Direction : byte { NORD=1, EST=15, SUD=30, OUEST=45 }
```

Énumérations

- **Énumération comme indicateurs binaires**
  - Par convention marquée avec l'attribut **[Flags]**
  - Il faut assigner explicitement tous les membres, pour éviter toute ambiguïté (généralement puissance de 2)

```
[Flags]
enum Options
{
    ToitOuvrant = 1, Climatisation = 2,
    FeuxAntiBrouillard = 4, JantesAlu = 8
}

static void Main()
{
    Options opts = Options.ToitOuvrant |
    Options.JantesAlu;
    Console.WriteLine(opts);
}
```

Structures

- Une structure est un type de données de type valeur
- Elle permet de créer une seule variable contenant des données liées à différents types de données
- Le mot-clé **struct** est utilisé pour créer une structure

```
struct Automobile
{
    public int puissance;
    public string couleur;
    public string marque;
};
static void Main(string[] args)
{
   Automobile maVoiture;
   maVoiture.puissance = 6;
   // …
}
```

Opérateurs

- Arithmétiques : **- + * / %**
- Incrémentation : **++**
  Décrémentation : **--**
  } pré : ++var et post : var++
- Affectations : **= += -= *= /= %= &= |= ^= ~= <<= >>=**
- Relationnels : **== != < > <= >=**
- Logiques : **! || &&**
- Binaires (bit à bit) : **~** (complément) **|** (ou) **&** (et) **^** (ou exclusif)
  (décalage) : **<< >>**
- Autre : **??** (fusion de valeur nulle)

Promotion numérique

La promotion numérique rend compatible le type des opérandes avant qu'une opération arithmétique ne soit effectuée

- **Opérateur unaire** (+, - , ~)
  - **sbyte**, **byte**, **short**, **ushort**, **char** sont promus en **int**
  - **uint** est promu **long**
- **Opérateur Binaire**
  - Le type le **+ petit** est promu vers le **+ grand** type des deux sauf pour les cas suivants, où l'on obtiendra une **erreur** :
    - **decimal** si l'autre opérande est de type : **float** ou **double**
    - **ulong** si l'autre opérande est de type : **sbyte**, **short**, **int** ou **long**
  - **uint** avec **sbyte**, **short** ou **int** : les 2 sont promus en **long**
  - **sbyte**, **byte**, **short**, **ushort**, **char** sont promus en **int**

Condition: if

```
if (condition) {
    // bloc d'instructions 1 (condition vrai)
}
else {
    // bloc d'instructions 2 (condition fausse)
}
```

- **else** n'est pas obligatoire
- On peut **imbriquer** les **if / else**

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

Condition: switch / case

```
switch (variable)
{
    case valeur1:
        // si variable à pour valeur valeur1
        break;
    case valeur2:
    case valeur3:
        // si variable à pour valeur valeur2 ou valeur3
        break;
    default:
        // si aucune valeur des cases ne correspond
        break;
}
```

- La variable peut être de type:
  char, **bool**, valeur intégrale(**int**, **long** …), **string** ou **enum**
- La valeur de case doit avoir une valeur constante
- **default** n'est pas obligatoire

Condition: switch / case

- **Exemple**

```
int jours = 7;
switch (jours)
{
    case 1:
        Console.WriteLine("Lundi");
        break;
    case 6:
    case 7:
        Console.WriteLine("Week end !");
        break;
    default:
        Console.WriteLine("Un autre jour");
        break;
}
```

Condition: opérateur ternaire

condition? condition_vraie **:** condition_fausse**;**

Utilisation : affectation conditionnelle

```
int i = 25;
string resultatTest = (i < 25) ? "Inf à 25":"Sup à 25";
```

équivaut à

```
int i = 25;
string resultat;
if (i < 25)
{
    resultat = "Inf à 25";
}
else
{
    resultat = "Sup à 25";
}
```

Boucle: while

```
while (condition)
{
    // instructions à exécuter
}
```

Tant que la condition est vérifiée, le bloc d'instructions est exécuté

```
int i = 0;
int somme = 0;
while (i <= 10)
{
    somme = somme + i;
    i = i + 1;
}
Console.WriteLine("Somme= " + somme);
```

Boucle: do while

```
do
{
   // instructions à exécuter
} while (condition);
```

Identique à **while**, sauf que le test est réalisé après l'exécution du bloc

```
int i = 0;
int somme = 0;
do
{
    somme = somme + i;
    i = i + 1;
} while (i <= 10);
Console.WriteLine("Somme= " + somme);
```

Boucle: for

```
for (initialisation; condition; itérateur)
{
    // instructions à exécuter
}
```

1. **initialisation** est exécutée
2. si **la condition** est fausse ⟶ on sort de la boucle
3. le bloc d'instruction est exécuté
4. **itérateur** est exécuté

```
for (int i = 0; i < 10; i = i + 1)
{
    Console.WriteLine("i= " + i);
}
```

Instructions de saut

- **break**
  termine le traitement de la boucle ou du switch courant
- **continue**
  passe à l'itération suivante dans un traitement de boucle
- **goto**
  permet de se brancher sur une instruction étiquetée
  utilisé pour :
  - transférer le contrôle à un case ou à l'étiquette par défaut d'une instruction **switch**

```
goto case 1;
```

  - quitter des boucles fortement imbriquées

```
goto label;
//..
label:
```

Dawan
Training is our way

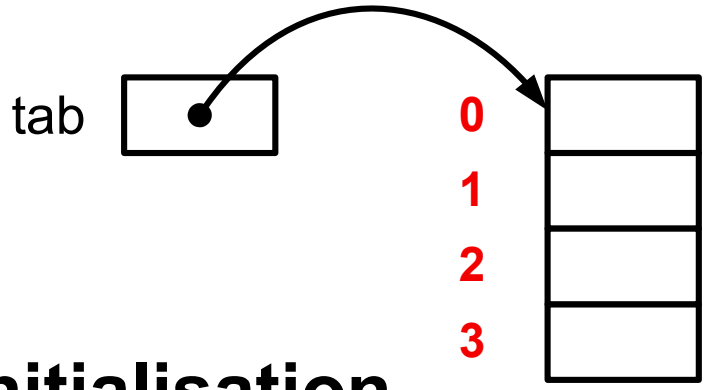
Tableaux

Tableaux

- **Déclaration d'un tableau**

```
type[] nom_tableau = new type[taille_tableau];
```

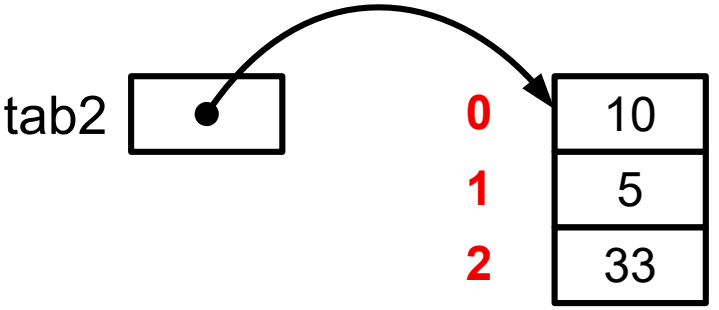
```
int[] tab = new int[4];
```



- **Déclaration d'un tableau avec initialisation**

```
type[] nom_tableau = { valeur1, valeur2, … };
```

```
int[] tab2 = { 10, 5, 33 };
```



Tableaux

- **Accès à un élément d'un tableau**

  `nom_tableau[indice]`

  L'indice d'un tableau commence à 0

```
int[] tab = { 10, 30, 40 };
Console.WriteLine(tab[0]);   // affiche 10
```

- **Taille d'un tableau**

  `nom_tableau.Length`

```
int[] tab = new int[20];
int n = tab.Length;        // n a pour valeur 20
```

Tableaux à 2 dimensions

```
type[,] nom_tableau = new type[ligne,colonne];

type[,] nom_tableau = {  {val00,val01,… },
                         {val10,val11,… },… }
```

```
int[,] tab2d = new int[4, 2];
int[,] tab2dInit = { { 10, 3 }, { 12, -9 },
                     { 1, -1 }, { 4, 1 } };
Console.WriteLine(tab2dInit[1, 1]);  // affiche -9
```

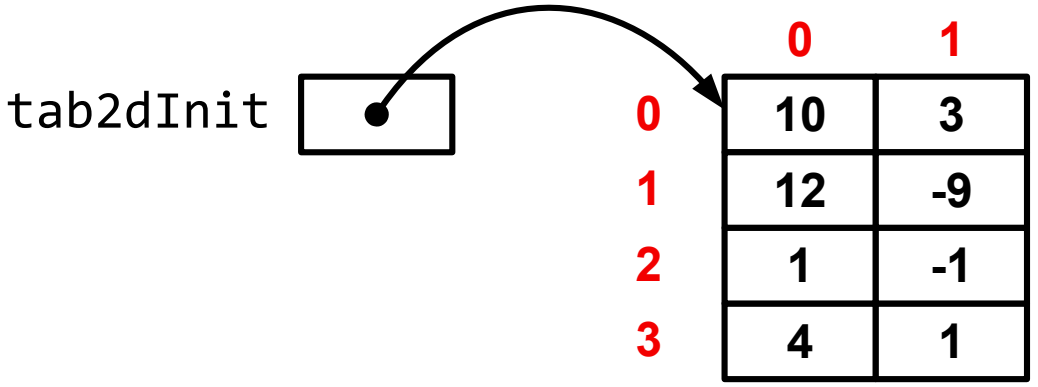


Tableau multidimensionnels


```
type[, , …,] nom_tableau = new type[d1,d2 … ,dn];
```

```
int[,,] cube = new int[5, 10, 5];
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
array3D[1, 0, 2] = 5; // remplace 9 par 5
```

- **Taille d'un tableau multidimensionnel**

  `int GetLength(int dimension);`

  retourne le nombre d'éléments en fonction de la dimension (commence à 0)

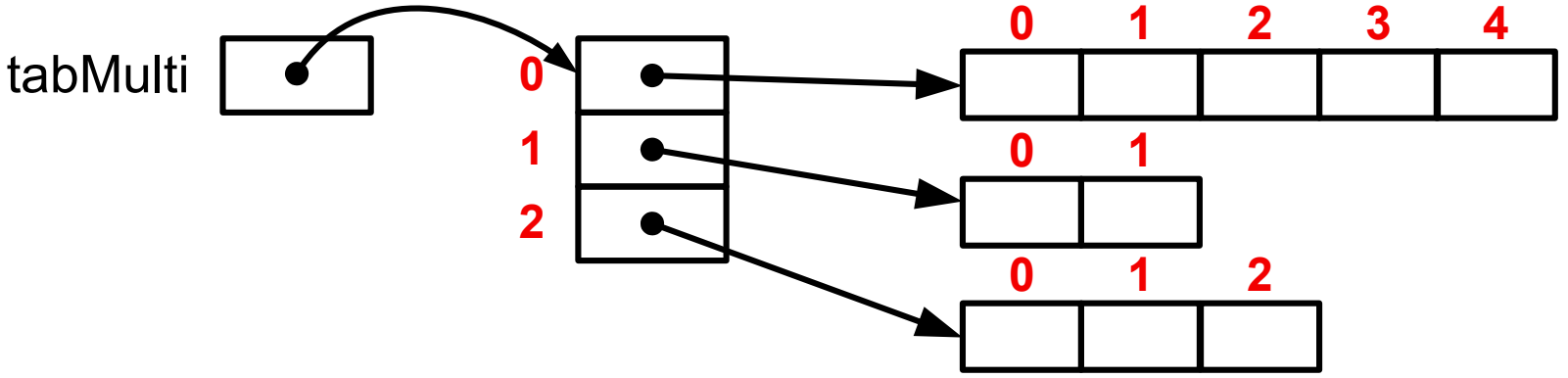
  La propriété **Rank** contient le nombre dimension

```
int[,,] cube = new int[5, 2, 3];
cube.GetLength(2); // retourne 3
cube.Length; // retourne 30, le nombre d'éléments du tableau
```

Tableaux de tableaux (Tableaux en escalier)

```
type[][] nom_tableau = new type[nb_ligne][];
nom_tableau[0] = new type[nb_colonne1];
        :              :              :
nom_tableau[nb_ligne - 1] = new type[nb_colonneN];
```

```
int[][] tabMulti = new int[3][];
tabMulti[0] = new int[5];
tabMulti[1] = new int[2];
tabMulti[2] = new int[3];
int[][] tabMulti2 = new int[][] { new int[] { 10, 3, 4, 1, 5 },
                  new int[] { 12, -9 }, new int[] { 1, -1, 12 }};
```



Tableaux: itération complète (foreach)


```
foreach (type variable in tab)
{
    // instructions à exécuter
}
```

```
int[] tab = { 2, 4, 6 };
foreach (int val in tab)      // → affiche :  2
{                             //              4
   Console.WriteLine(val);    //              6
}

int[,] tab2dInit = { { 10, 3 }, { 12, -9 },
                     { 1, -1 }, { 4, 1 } };
foreach( int a in tab2dInit)
{
   Console.Write(a + ",");  // → affiche : 10,3,12,
                                          -9,1,-1,4,1,
}
```

Dawan
Training is our way

Méthodes et paramètres

Méthodes

- **Une méthode permet de**
  - diviser le code en morceaux (réutilisabilité, clarté)
  - factoriser le code
- **Déclaration**

```
typeDeRetour nom(type param1, type param2, …) {
    // instructions à exécuter
}
```

```
int Somme(int a, int b)
{
    return a + b;
}
```

- **Appel**

```
nomMéthode(paramètres);
```

```
int num = 20;
int res = somme(num, 22);
```

Méthodes

- **Type de retour**
  - Il est obligatoire. S'il n'y en a pas → **void**
- **Corps de la méthode**
  - au minimum **{ }**
  - doit contenir au moins une instruction **return**
  - pour **void**: **return;** ou il **peut être omis**
- **Arguments**
  - Ils sont séparés par **,**
  - Pour leurs portés, ils sont considérés comme des variables locales à la méthode

Passage de paramètres

- **Passage par valeur (en entrée)**

```
void Inc(int x) {
    x ++;
}

void f() {
    int val = 3;
    Inc(val);          // val == 3
}
```

- **Passage par référence avec ref (en entrée, en sortie)**

```
void Inc(ref int x) {
    x ++;
}

void f()      {
    int val = 3;
    Inc(ref val);    // val == 4
}
```

Passage de paramètres

- **Passage par référence avec out (en sortie)**

```
void Method(int val, out int next)
{
    next = val * 2;
}

void f()
{
    int i = 1, next;
    Method(i, out next);
}
```

  - On peut de déclarer la variable de retour dans les arguments pendant l'appel de la fonction

```
Method(i, out int next);
```

  - On peut ignorer un paramètre **out** en le nommant _

```
Method(i, out _);
```

Passage de paramètres

- **Paramètre optionnel**
  - Ils ne peuvent pas être **ref** ou **out**
  - Ils doivent se trouver en fin de liste des arguments

```
void MethodOptParams(int required, string optStr =
                                "default", int optInt = 10)
{
   Console.Write("{0} {1} {2}", required,optStr,optInt);
}
MethodOptParams(4) ;              // affiche → 4 default 10
MethodOptParams(6, "other") ; // affiche → 6 other 10
```

- **Paramètre nommées**
  On identifie les arguments par leur noms

```
MethodOptParams(4, optInt: 4);
MethodOptParams(4, optStr: "param");
MethodOptParams(optStr: "param", required: 4);
```

Nombre d'arguments variable (params)

- Un paramètre avec le mot clef **params**
  - accepte un nombre de paramètre variable d'un même type
  - doit toujours être le dernier paramètre
  - doit être déclaré en tant que tableau

```
TypeRetour nomMethode(params type[] nomParam)
```

```
static void MaFonction(string sts, params string[] strs) {
        // …
 }

MaFonction("Test", "Nb", "De", "Parametres", "Inconnu");

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

Surcharge de méthode

- Plusieurs fonctions peuvent avoir le même nom et des arguments différents en nombre ou/et de type
- Le type de retour n'est pas pris en compte

```
void MaFonction(int param1)
{
    // ...
}
int 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)      // factoriel= 1* 2* … n
{
    if (n <= 1)           // condition de sortie
    {
        return 1;
    }
    else
    {
        return Factorial(n - 1) * n;
    }
}
-------------------------------------------------------------------------
int f = Factorial(3);     // f vaut 6
```

Appels successifs

```
Factorial(3)= Factorial(2) * 3
    Factorial(2)= Factorial(1) * 2
        Factorial(1)
```

Remontée des résultats

```
Factorial(3) = (2) * 3 = 6
    Factorial(2) = (1) * 2 = 2
        Factorial(1) = 1
```

Condition de sortie n=1

Méthode main

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

- point d'entrée du programme
- doit être statique
- ne doit pas être public
- peut avoir comme type de retour **void** ou **int**
- peut être déclarée avec ou sans paramètre **string[]**, qui contient des arguments de ligne de commande
- dans un programme, il ne peut y avoir qu'une seule classe contenant un main. Dans le cas contraire, il faut compiler avec l'option **-main** pour préciser le main utilisé

Dawan
Training is our way

Programmation Orienté 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 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

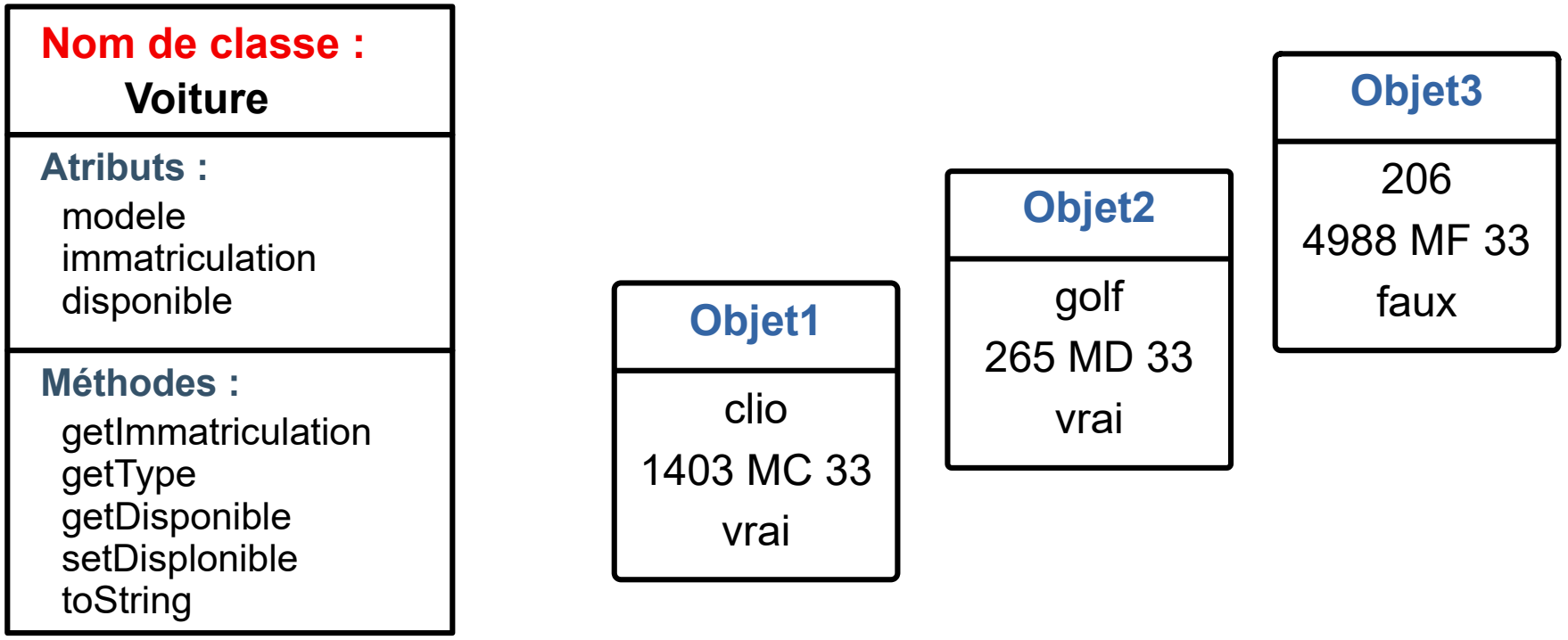
- concret (voiture, stylo,…)
- abstrait (entreprise, temps,…)

Un objet est caractérisé par :

- son **identité**
- son **état** → les données de l'objet
- son **comportement** → ce qu'il sait faire

Qu'est-ce qu'une Classe ?

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



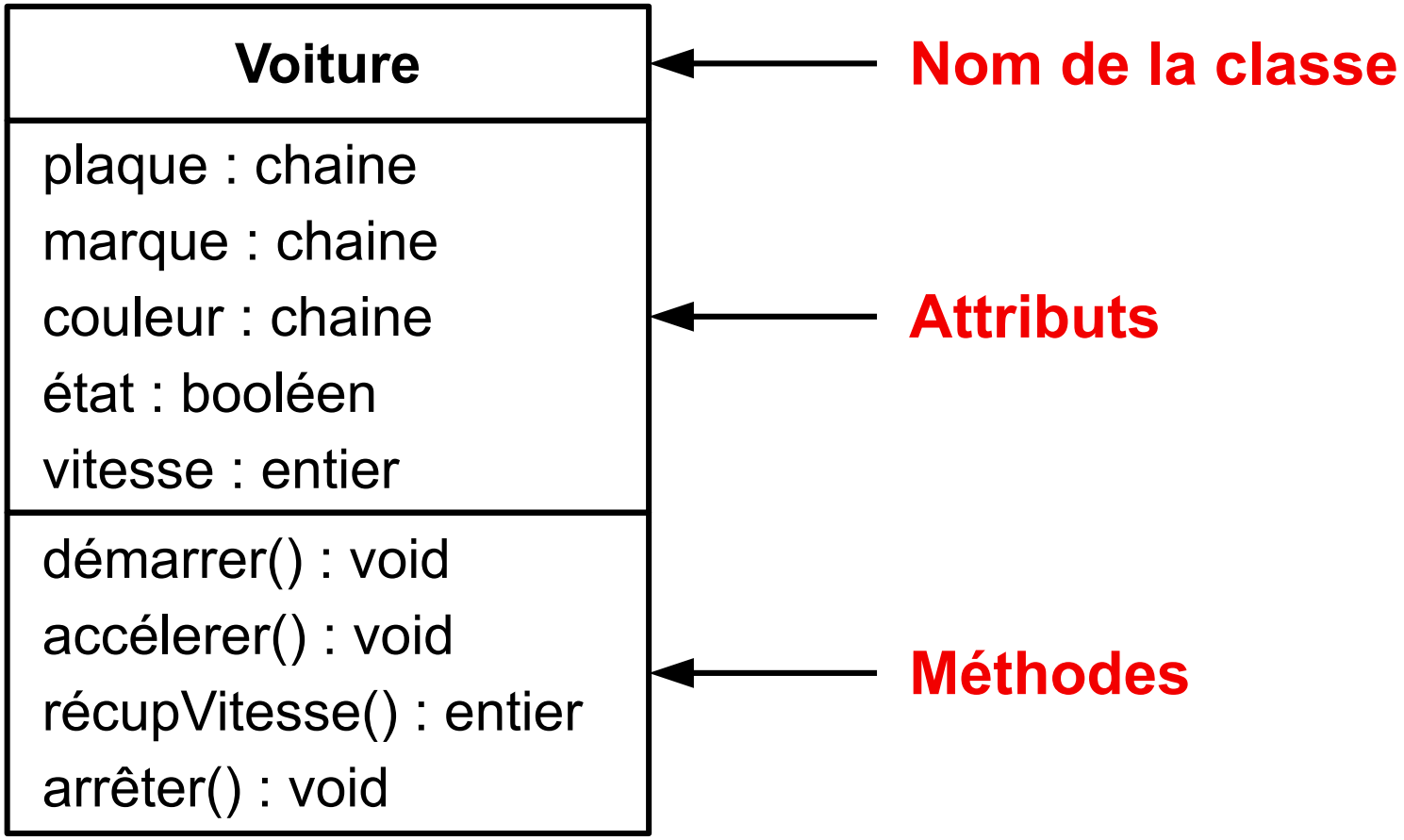
Unified Modeling Language

UML = Langage pour la modélisation des classes, des objets, des interactions etc...

UML 2.0 comporte ainsi 13 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 (**+** → public, **#** → protected, **-** → private) pour indiquer le niveau de visibilité

Déclaration d'une classe

```
public class NomClasse
{
      // Attributs
         …
      // Méthodes
         …
      // Propriétés
         …
}
```

ou

```
public struct Nom
{
      // Attributs
         …
      // Méthodes
         …
      // Propriétés
         …
}
```

Différences entre classes et structures

Une structure :

- est de type valeur, une classe est de type référence
- n'a pas de constructeur par défaut et n'a pas de destructeur
- peut être instancier sans opérateur new
- ne peut être héritée
- peut implémenter un interface

Variables d'instances

- Les variables d'instance définissent **l'état de l'objet**
- Elles sont également appelées **attributs**
- La valeur d'un attribut est **propre à chaque** instance

```
public class Voiture
{
    string marque;
    string plaque;
    string couleur;
    // …
}
```

- **Accès à un attribut**

  instance.attribut

```
clio.couleur
```

Méthodes d'instances

- Méthodes qui définissent **un comportement** d'une instance
- Elles sont déclarées dans la classe
- Elles peuvent être surchargées

```
public class MaClasse
{
    public void maMethode()
    {
        // ...
    }
}
```

- **Appel d'une méthode d'instance**

instance.methode();

```
clio.deplacer();
```

Variables locales

- Variables temporaires qui existent seulement pendant l'exécution de la méthode

```
public class MaClasse
{
    // …

    void maMethode()
    {
        int monNombre = 10;
    }

    void maMethode2()
    {
        Console.WriteLine(monNombre);
        // Erreur de compilation
    }
}
```

Initialisation des variables

- **Variables d'instance**

  Si une variable d'instance n'est pas initialisée, elle prend une valeur par défaut

bool	false
Entier (sbyte, short...)	0
float, double, decimal	0
char	\0000
référence	null

```
public class Voiture
{
    string marque;     // null
    int nbKilometre = 1000;
    // ...
}
```

- **Variables locales**

  Les variables locales n'ont pas de valeur par défaut et **doivent obligatoirement être initialisées**

Constructeur

Le constructeur est une méthode spéciale dans la classe appelée à la création d'instances

- Un constructeur:
  - porte le nom de la classe
  - n'a pas de type de retour

```
public class Voiture
{
    public Voiture()    // Pas de type de retour
    {
        // Corps du constructeur
    }
}
```

- On peut surcharger le constructeur

Constructeur par défaut

- Lorsqu'une classe ne comporte pas de constructeur, un constructeur par défaut est automatiquement

```
public class MaClasse
{
    // public MaClasse() {  }  est implicitement généré
}
```

- Si un constructeur est ajouté à la classe, il n'y a plus de constructeur par défaut ajouté automatiquement
  Il devra être ajouté explicitement

```
public class MaClasse
{
    public MaClasse() { }   // doit être ajouté
    public MaClasse(String str) { }
}
```

Destructeur

Un destructeur est une méthode qui est appelée automatiquement lorsque l'objet est supprimé

- Un destructeur
  - porte le même nom que la classe,précédé d'un **~**
  - n'a de type de retour
  - n'a pas d'argument
- Une classe ne peut avoir qu'un seul destructeur

```
public class Voiture
{
    public ~Voiture()
    {
        // corps du destructeur
    }
}
```

Initialiseur d'objet

- Un initialiseur d'objet permet d'affecter des valeurs aux attribut ou propriétés accessibles d'un objet, au moment de sa création, sans devoir appeler un constructeur

```
public class Chat
{
    public int Age { get; set; }
    public string Nom { get; set; }

    public Chat() {  }

    public Chat(string nom)
    {
        this.Nom = nom;
    }
}

Chat chat1 = new Chat { Age = 4, Nom = "Gordon" };
Chat chat2 = new Chat("Gordon") { Age = 4 };
```

Le mot clé this

- Le mot clé **this** fait référence à l'objet en cours
- On peut l'utiliser pour :
  - manipuler l'objet en cours

```
maMethode(this);
```

  - faire référence à une variable d'instance

```
public class MaClasse
{
    private int nombre;
    public MaClasse(int nombre)
    {
        this.nombre = nombre;
    }
}
```

  - déclarer des indexeurs

Cycle de vie d'un objet

- Un objet est instancié avec **new**

```
string str = new String("hello world");
```

- Un objet peut être collecté par le garbage collector lorsqu'il n'y a plus de référence qui pointe sur lui

- Garbage collector
  - Travail en arrière plan
  - Intervient lorsque le système a besoin de mémoire ou de temps en temps (priorité faible)

Variables de classes

- Variables partagées par toutes les instances de classe
- Elles sont déclarées avec le mot clé **static**
- **Pas besoin d'instancier** la classe pour les utiliser
- Chaque objet détient la **même** valeur de cette variable

```
public class Voiture
{
    String type;
    static int nbVoitures;
    // …
}
```

- **L'appel de ses variables :**
  `Classe.variableDeClasse;`

```
Voiture.nbVoiture;
```

Méthodes de classes

- Méthodes définissant un comportement global ou un service particulier
- Déclarées avec le mot clé **static**
- Peuvent être surchargées (même nom, ≠ paramètres)
- N'utilisent pas de variables d'instance parce qu'elles doivent être appelées depuis la classe

```
public class MaClasse
{
    public static void maMethode()
    {
        // ...
    }
}
```

- **Appel des méthodes de classes**

```
MaClasse.maMethode();
```

Constructeur static

- Un constructeur statique est automatiquement appelé avant la création de la première instance ou le référencement d'un membre statique

```
class MaClasse
{
    static MaClasse()
    {
        //...
    }
}
```

- Un constructeur statique
  - n'a pas de modificateur d'accès et n'a pas de paramètre
  - ne peut pas être hérité ou surchargé
  - ne peut pas être appelé directement

Portée des variables

- Chaque bloc de code à sa propre portée
- Quand des blocs contiennent d'autre bloc. Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse

```
int a = 10;
if (a > 0)
{
    int somme = a + 20;      // OK
}
Console.WriteLine(somme);  // erreur
```

- **variable locale :** de sa déclaration → à la fin du bloc
- **variable d'instance** : de sa déclaration → jusqu'à la destruction de l'objet par le garbage collector
- **variable de classe** : de sa déclaration → à la fin du programme

Modificateurs d'accès

Les modificateurs d'accès permettent de spécifier les niveaux d'accessibilité :

• **public**	L'accès n'est pas limité
• **protected**	L'accès se limite à la classe ou aux types contenants dérivés de la classe contenante
• **internal**	L'accès est limité à l'assemblage actuel
• **protected internal**	L'accès est limité à l'assemblage actuel ou aux types dérivés de la classe contenante
• **private**	accessible seulement dans la classe elle-même

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

- 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

```
Type NomPropriété
{
    get
    {
        // retourne une valeur du Type spécifié
    }
    set
    {
        // utilise le paramètre prédéfini "value"
    }
}
```

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

Propriétés

- **En C# 7.0**

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

- **Accesseurs auto-implémentés**
  La variable d'instance est générée par le compilateur

```
public class Test
{
    public int X { get; set; }
}
```

Indexeur

- Les indexeurs sont similaires aux propriétés
- Ils permettent l'utilisation de propriétés indexées, qui sont référencées à l'aide d'un ou plusieurs arguments

```
Type_élément this[Type_index index]
{
    get
   {
      // Retourne une valeur du Type_éléments
      // dont l'index est dans le paramètre index
   }
    set
   {
      // Stocker à l'index spécifié par le paramètre
         index
      // le paramètre prédéfini "value"
   }
}
```

Classes Imbriquées

- Définir une classe à l'intérieur d'une classe
- Un type **imbriqué** (interne) a accès à tous les membres de son type **conteneur (externe)**

```
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
{
    // ...
}
```

Classe statique

- Une classe statique ne peut pas être instanciée
- Une classe statique :
  - contient uniquement des membres statiques
  - est verrouillée (sealed)
  - ne peut pas contenir de constructeurs d'instances

```
public static class TemperatureConvertisseur
{
    public static double CelsiusToFahrenheit(string tmpC)
    {
        return  (Double.Parse(tmpC) * 9 / 5) + 32;
    }

    public static double FahrenheitToCelsius(string tmpF)
    {
        return (Double.Parse(tmpF) - 32) * 5 / 9;
    }
}
```

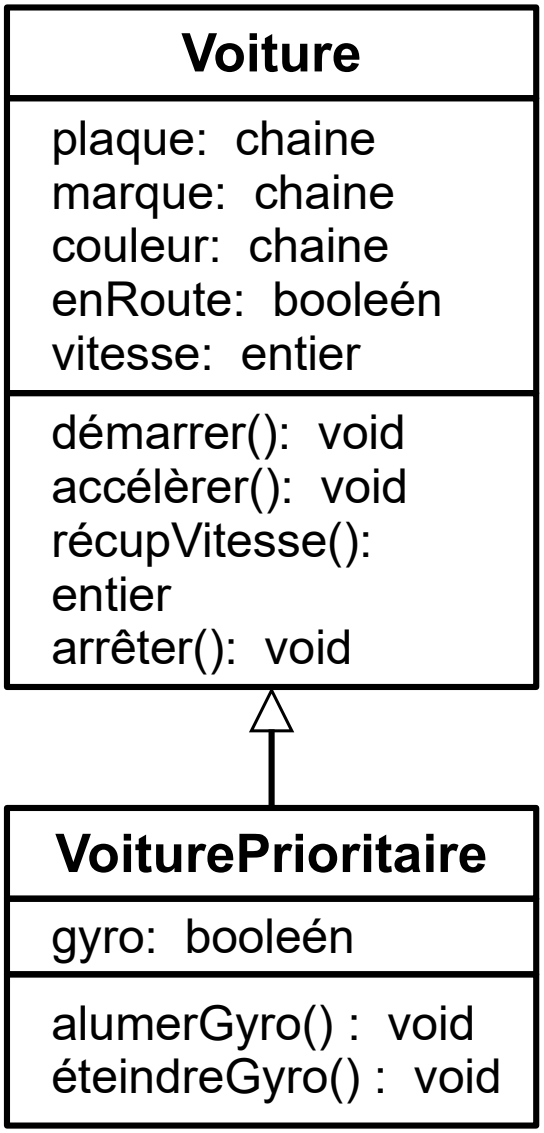
Agrégation

- **Agrégation = associer un objet avec un autre**
  ex : Objet Propriétaire à l'intérieur de la classe Voiture


```
public class Proprietaire
{
    // …
}

public class Voiture {
    Proprietaire owner;
    // …
}
```

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 en C#

- **En C#, L'héritage est simple**
  Une classe ne peut hériter que d'une seule classe mère

```
class VoiturePrioritaire : Voiture
{
    public VoiturePrioritaire() : base()
    {
    }
    private bool gyrode;
    // ...
}
```

- Le mot clé **base** sert à accéder aux membres de la classe de base à partir d'une classe dérivée, On peut :
  - appeler une méthode de la classe de base
  - spécifier quel constructeur de classe de base doit être appelé lors de la création d'instances de la classe dérivée

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 details()
    {
        //...
    }
}
```

Classes scellées

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

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

System.Object

Toute classe hérite directement ou indirectement de la classe System.Object, de façon implicite

- **Méthodes** :
  - **ToString()** retourne une chaîne qui représente l'objet
  - **Equals**(Object) détermine si l'objet spécifié est identique à l'objet actuel
  - **GetType**() retourne le **Type** de l'instance
  - **Finalize**() appelé avant que l'objet soit détruit par le garbage collector
- **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 déclarés leur type exact

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

Redéfinition avec polymorphisme

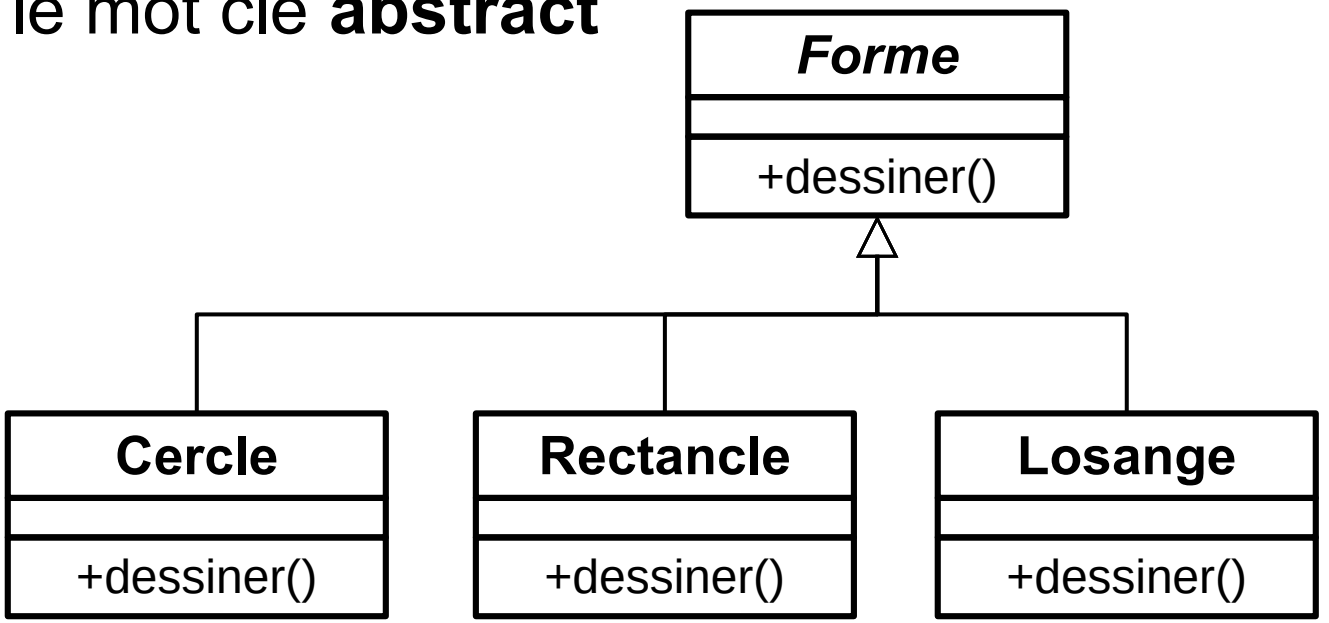
- La virtualisation consiste à définir un comportement par défaut à une méthode qui sera redéfinie

```
class Voiture
{
    public void description()
    {
        //...
    }
    public virtual void demarrer()
    {
        //...
    }
}

class Batmobile : Voiture
{
    public override void  demarrer()
    {
        //...
    }
}
```

Classe Abstraite

- Une classe **qui ne peut être instanciée**
- Définit un type de squelette pour les sous-classes
  - Si elle contient des méthodes abstraites,

    les sous-classes doivent implémenter le corps des méthodes abstraites
- Déclarée avec le mot clé **abstract**

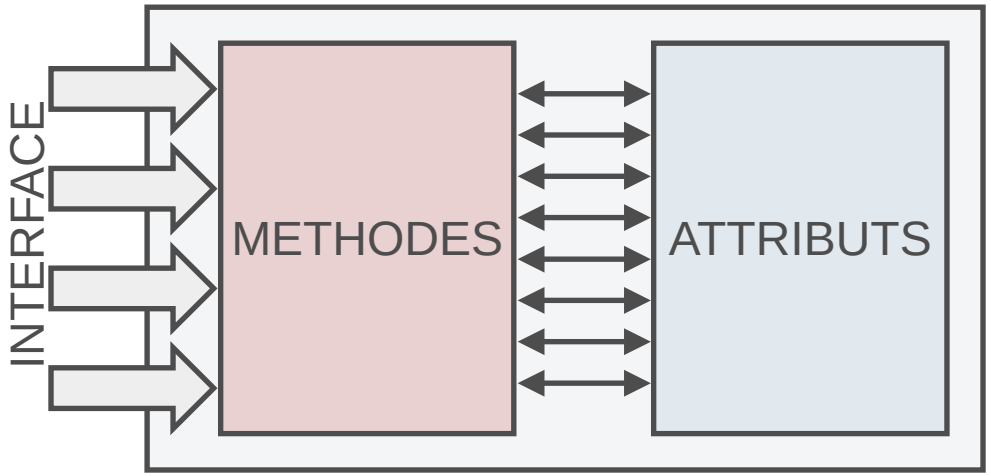


Classe Abstraite

```
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
{
    //.....
    // Méthodes sans corps et sans modificateurs d'accès
}
```

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 new void details()
    {
        //…
    }
}
```

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 Formation
{
    public class myClass
    {
        //…
    }
}
```

Surcharge d'opérateurs

Gestion du comportement d'un objet face à un opérateur

- On utilise le mot clé **operator** suivit de l'opérateur à surcharger
- Opérateurs qui peuvent être surchargés :
  - unaires : **+ - ! ~ ++ -- true false**
  - binaires :**+ - * / % & | ^ << >>**
    == != < > <= >=
- Les opérateurs d'assignation composée sont surchargés implicitement en même temps que l'opérateur binaire associé
- Les opérateurs de comparaison doivent être surchargés par paires **==** et **!=** **<** et **>** **<=** et **>=**

Surcharge d'opérateurs

- Une déclaration d'opérateur doit
  - être **public static**
  - un opérateur **unaire** → **un** paramètre d'entrée
  - un opérateur **binaire** → **2** paramètres d'entrée
    - ↳ au moins un paramètre doit être de type T , le type qui contient la déclaration d'opérateur

```
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);
```

Expression lambda

- Une expression lambda est une fonction anonyme qui peut être utiliser où une valeur est attendue pour un délégué

```
( paramètre1, paramètre2, ...) => { instructions }
```

- S'il y a un paramètre, les parenthèses sont facultatives
- S'il n'y a pas de paramètre, les parenthèses sont obligatoires
- Le bloc d'instruction peut retourner une valeur ou pas
- S'il n'y a qu'une seule instruction, les accolades sont facultatives

```
public delegate int Operation(int n1, int n2);
Operation multiplication = (opt1, opt2) => { return op1 +  op2; };
```

Expressions Rationnelles

- 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+" ";
    // ...
}
```

Dawan
Training is our way

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)

Bloc try et catch

- Utilisé pour encadrer un bloc susceptible de déclencher une exception

```
try
{
    // des lignes de code susceptibles
    // de lever une exception
}
catch (FileNotFoundException e)
{
    // capture et traitement de l'exception de type
    // FileNotFoundException
}
finally
{
    // toujours exécuté
    // même sans exception ou une exception imprévue
}
```

Bloc try et catch

- En C# 6.0, on peut avec le mot-clé **when** ajouter une condition pour attraper une exception
  La condition n'a uniquement accès à la variable du catch

  `catch (Exception variable) when (condition)`
- **Exemple**

```
int x = 0;
try
{
    int y = 100 / x;
}
catch (ArithmeticException e)
{
   Console.WriteLine($"ArithmeticException Handler: {e}");
}
catch (Exception e)
{
   Console.WriteLine($"Generic Exception Handler: {e}");
}
```

Lever une exception

- Le mot clé **throw** est utilisé pour déclencher une exception à n'importe quel moment

```
if(age < 0)
{
    throw new Exception("Impossible: age négatif");
}
```

- **Exception levée plusieurs fois**

  Une fois que l'exception a été traitée dans un catch, on peut :

  - la relancé avec **throw**
  - ou relancé une autre exception et conserver les informations de l'exception d'origine dans une exception interne

```
throw new ArgumentNullException("message", e);
```

Créer ses propres exceptions

- Il faut seulement hériter de la classe **Exception** (la classe de base de toutes les exceptions) ou une de ses sous-classe
- On doit définir les constructeurs de la classe d'exception
  La classe exception possèdent plusieurs constructeurs :
  - **Exception**() → constructeur par défaut
  - **Exception**(string) → avec un message d'erreur
  - **Exception**(string, Exception) → avec un message d'erreur et une exception interne

```
public class AgeException : Exception
{
    public AgeException() : base()
    {
    }
    // …
```

Using pour les exceptions

- Équivalent d'un **try / finally** + **Close**()
- Seulement pour les classes implémentant l'interface **IDisposable**
- On peut ajouter éventuellement un **try / catch** autour

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

Dawan
Training is our way

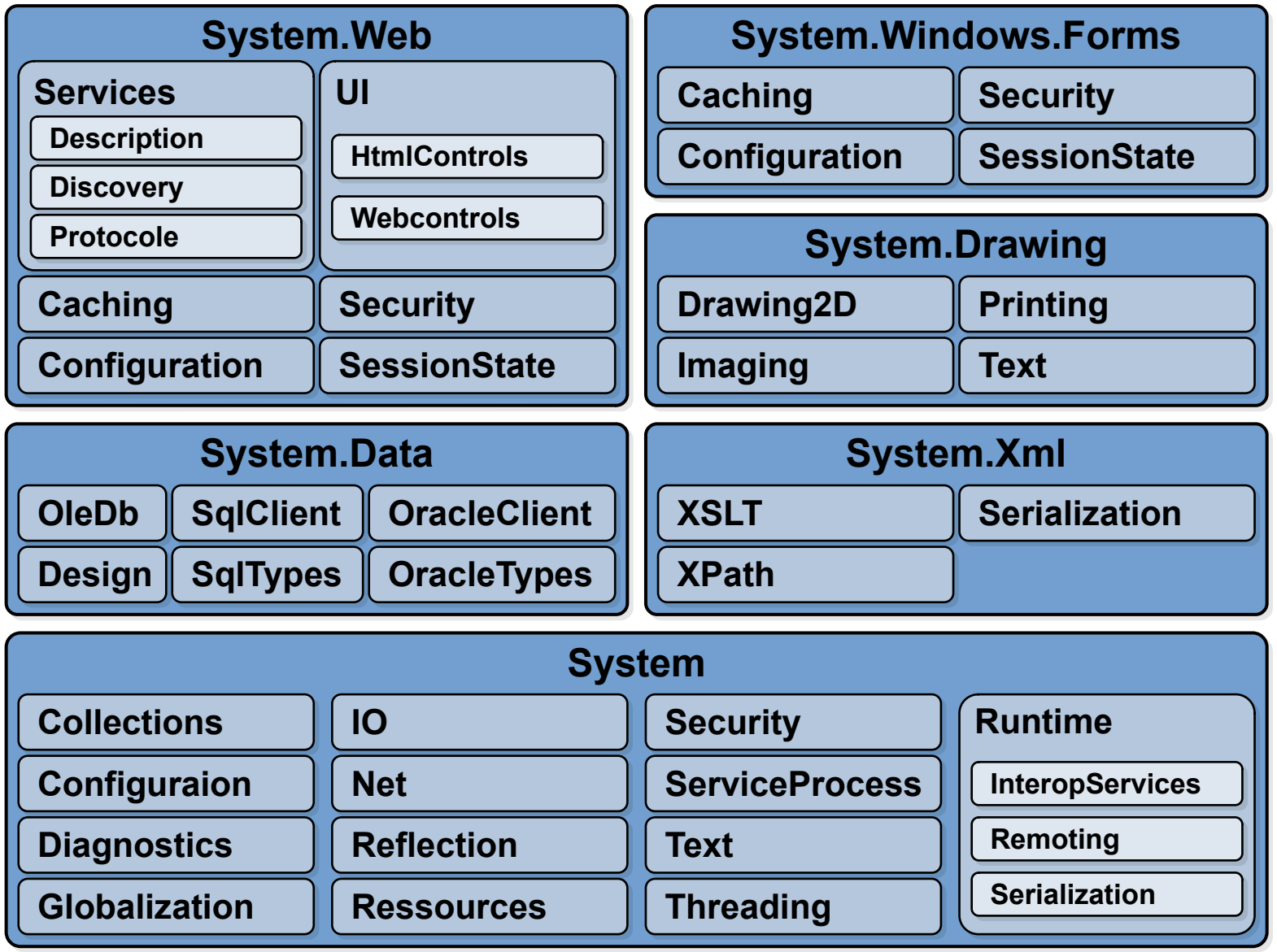
Bibliothèque de classes .Net

Définition

BCL : Base Class Library

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

Bibliothèque de classes



Chaînes de caractères

System.String

Les objets String sont immuables

↳ Une fois créé, ils ne peuvent plus être modifiés

- **Comparaison** : Compare, CompareTo, Equals
- **Concaténation** : Concat, Join
- **Découpage** : Split, Substring, Insert, Remove
- **Recherche** : StartsWith, EndsWith, IndexOf, LastIndexOf, Replace, Contains,
- **Mise en forme** : Format, PadLeft/Right, TrimStart/End, ToLower, ToUpper
- **Longueur** : Length
- **Opérateurs** : == !=

StringBuilder

La chaîne peut être modifiée → pas de création de chaîne intermédiaire

- **Append** Ajouter une chaîne à la fin
- **AppendFormat** Formater et Ajouter une chaîne en fin
- **Insert** Insère une chaîne ou un objet à l'index spécifié
- **Remove** Supprime un nombre de caractères spécifié
- **Remplace** Remplace toutes les occurrences d'un caractère ou d'une chaîne
- **ToString** Convertie un StringBuilder en String

Date

System.DateTime

Représente une date et une heure

- **Propriétés**
  - **date et heure actuelle** : Now, UtcNow, Today
  - Day, DayOfWeek, DayOfYear, Month, Year, Hour, Milisecond, Second, Minute
- **Méthodes**
  - **Comparaison :** Compare, Equals
  - **Opération :** Add, Substract
  - **Conversion :** Parse, ToString
- **Opérateurs**

  + - == != > >= < <=

Math

- La classe **Math** fournit des constantes (**E** et **PI**) et des méthodes statiques pour les fonctions mathématiques courantes (trigonométriques, logarithmiques…)

```
// valeur du + grand entier inférieur
double df = Math.Floor(5.99 / 2);      // 2
// valeur du + petit entier supérieur
double dc = Math.Ceiling(5.99 / 2);    // 3
// arrondit au nombre entier le + proche
double dr = Math.Round(0.51);          // 1
double dt = Math.Truncate(3.4);     // 3 partie entière
double d1 = Math.Sin(Math.PI * 0.75); // 0.707106781
double e1 = Math.Exp(2);          // 7,38905609893065
double d2 = Math.Pow(2.0, 3.0); // 8.0
double d3 = Math.Sqrt(9.0);      // 3.0
double d4 = Math.Abs(-4.0);      // 4.0
int min = Math.Min(3, 67);       // 3
```

Collections

System.Collections

Regroupe des classes pour gérer des ensembles d'objets

- **Faiblement typée**
  - ArrayList tableau à taille variable
  - Hashtable stocke des paires clé et valeur
  - Queue stocke les valeurs dans une "pile" LIFO
  - Stack stocke les valeurs dans une "pile" FIFO
  - …
- **Fortement typée** (System.Collections.Generic)
  - List tableau à taille variable
  - Dictionnary contient des paires clé et valeur
  - HashSet contient des éléments unique
  - Queue stocke les valeurs dans une "pile" LIFO
  - Stack stocke les valeurs dans une "pile" FIFO
  - ...

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 Générics

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 IEnumerable<Char> Letters(){
     char currentCharacter = 'a';
     do
     {
        yield return 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

DriveInfo

DriveInfo fournit des informations sur les lecteurs d'une machine

- **DriveInfo.GetDrives()** → retourne un tableau de DriveInfo correspondant aux lecteurs de la machine
- **DriveInfo("LettreLecteur")** → Constructeur qui prend en paramètre la lettre du lecteur (chaîne)

Membres	Fonctionnalités
Name	Nom du Lecteur
VolumeLabel	Nom du volume du lecteur
RootDirectory	Chemin du lecteur
DriveType	Énumération contenant le type du lecteur
DriveFormat	Le format du système de fichier du lecteur
TotalSize	Espace total du lecteur (en octet)
AvailableFreeSpace	Espace libre sur le lecteur (en octet)
isReady	Indique si le lecteur est prés à être utiliser

Directory

Directory permet de manipuler des répertoire. Si l'on a plusieurs opérations à effectuer, on peut utiliser **DirectoryInfo**

- **CreateDirectory** Créer un répertoire
- **Exists** Tester si le dossier existe
- **Delete** Effacer un répertoire vide
- **Move** Déplacer dossier (**ToMove** avec Directoryinfo)
- **GetDirectories** Obtenir un tableau de string contenant les sous-dossiers du répertoire
- **GetFiles** Obtenir un tableau de string contenant les fichiers du répertoire
- **GetRootDirectory** Obtenir le répertoire racine

```
if (Directory.Exists("C:\\EmptyFolder")) {
    Directory.Delete("C:\\EmptyFolder");
}
```

File

File permet de manipuler des fichiers. Si l'on a plusieurs opérations à effectuer, on peut utiliser **FileInfo**

- **Copy** Copier un fichier
- **Delete** Supprimer un fichier
- **Move** Déplacer un fichier
- **Exists** Indiquer si le fichier existe
- **Create** Créer un fichier un fichier binaire
- **CreateText** Créer ou ouvre un fichier texte UTF-8
- **Encypt/ Decrypt** Encrypter / Décrypter un fichier

```
if (File.Exists("C:\\Test.txt")) {
    Directory.Move("C:\\Test.txt",C:\\Move.txt");
}
```

Stream

Stream → transfert de données
(MemoryStream, NetWorkStream, FileStream...)

- Principe d'utilisation d'un flux :
  - Ouverture du flux
  - Identification de l'information (lecture/écriture)
  - Fermeture du flux
- **FileStream** → pour lire ou écrire dans un fichier binaire
  - int **ReadByte** () → Lire un octet dans le flux
  - int **Read** (byte[] array, int offset, int count) → Lire un bloc de ***count*** octets dans le flux et le place dans le tableau ***array*** à partir de ***offset***
  - **WriteByte** (byte value) → Écrit un octet dans le flux

Stream

- **Write** (byte[] array, int offset, int count) → Ecrit un bloc de ***count*** octet contenu dans le tableau ***array*** à partir d'***offset***

- **StreamReader** → pour lire des caractères à partir d'un flux
  - int **Read** () → lire un caractère dans le flux
  - string **ReadLine** () → lire un ligne de caractère dans le flux
  - string **ReadToEnd** () → lire tous les caractères de la position actuelle jusqu'à la fin du flux
  - bool **EndOfStream** → indique que la fin du flux est atteinte
- **StreamWriter** → pour écrire des caractères dans un flux
  - **Write** et **WriteLine** → méthodes surchargées pour écrire dans le flux