C# : introduction

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

C#

C# : création

- Présenté officiellement par **Microsoft** en 2002
- Conçu sous la direction d'**Anders Hejlsberg**, l'architecte de **.NET Framework** et le créateur de :
  - **Turbo Pascal**
  - **TypeScript** utilisé par **Angular** de **Google**
- Son design est inspiré par :
  - **C++** (syntaxe),
  - **Java** (organisation du code et API).

C#

C# : langage de programmation

- multi paradigme : orienté objet, fonctionnel...,
- fortement typé,
- doté d'un typage statique, dynamique et générique,
- syntaxiquement proche de **C++** et **Java**.

C#

C# permet de développer

- Applications Web (**ASP.NET Core**)
- Applications desktop (**WPF**, **WinForms**)
- Applications mobiles (**MAUI**)
- API REST
- Jeux (**Unity**)
- Scripts et outils.
- ...

C#

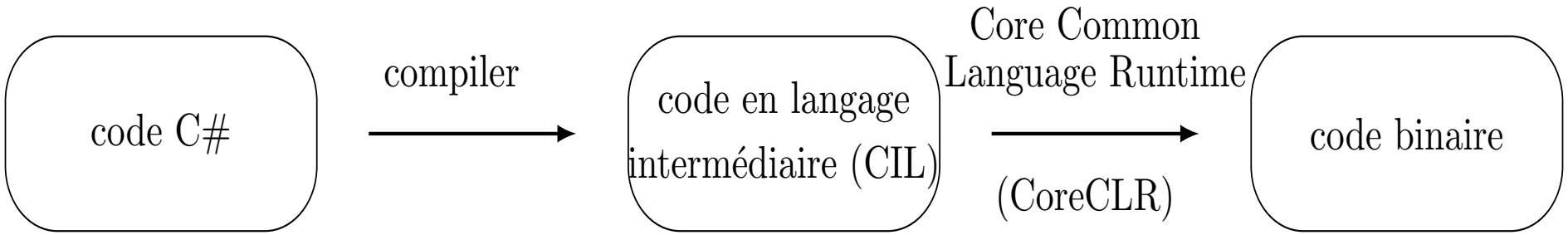
Comment ça fonctionne ?

- On écrit un programme **C#**.
- Le code **C#** (fichiers `.cs`) est compilé, par **Roslyn**, en langage intermédiaire (appelé **CIL** pour **C**ommon **I**ntermediate **L**anguage) ou **MSIL** pour **M**icro**S**oft **I**ntermediate **L**anguage.
- Cette étape a lieu avant l'exécution de l'application.
- Le **CIL** est stocké dans un `.dll`.
- Le **CIL** est exécuté par le **JIT**, qui fait partie du **CLR** / **CoreCLR** et qui prend le **CIL** et le traduit en code machine.

C#

Comment ça fonctionne ?

- On écrit un programme **C#**.
- Le code **C#** (fichiers `.cs`) est compilé, par **Roslyn**, en langage intermédiaire (appelé **CIL** pour **C**ommon **I**ntermediate **L**anguage) ou **MSIL** pour **M**icro**S**oft **I**ntermediate **L**anguage.
- Cette étape a lieu avant l'exécution de l'application.
- Le **CIL** est stocké dans un `.dll`.
- Le **CIL** est exécuté par le **JIT**, qui fait partie du **CLR** / **CoreCLR** et qui prend le **CIL** et le traduit en code machine.



C#

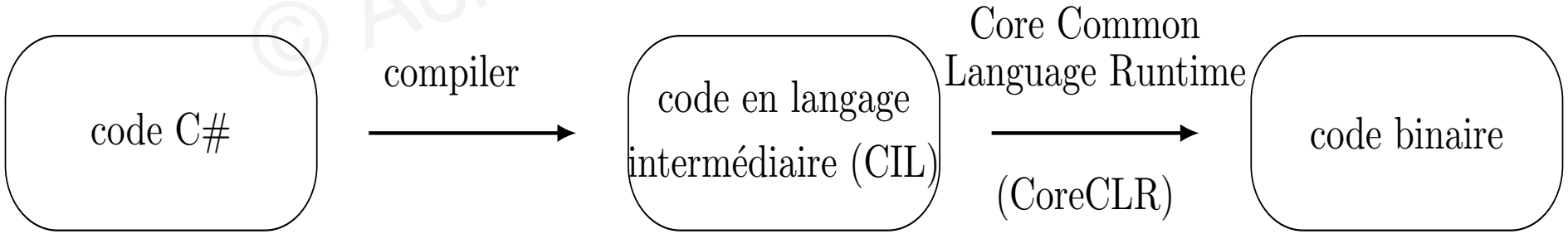
Code **CIL** vs Code binaire

- **CIL** : code intermédiaire indépendant de la plateforme.
- Code machine : code machine : généré par le **JIT** pour la plateforme d'exécution.

C#

Code **CIL** vs Code binaire

- **CIL** : code intermédiaire indépendant de la plateforme.
- Code machine : code machine : généré par le **JIT** pour la plateforme d'exécution.



C#

CLR vs CoreCLR

- **CLR** : machine virtuelle du **.NET Framework**.
- **CoreCLR** : version open-source et multi-plateforme utilisée par **.NET** moderne.

C#

CLR vs CoreCLR

- **CLR** : machine virtuelle du **.NET Framework**.
- **CoreCLR** : version open-source et multi-plateforme utilisée par **.NET** moderne.

Tous deux incluent (**CLR** et **CoreCLR**)

- Le chargeur d'assemblage (Assembly Loader)
- Le **JIT** (**J**ust **I**n **T**ime)
- Le ramasse-miettes **Garbage Collector**
- Le système de gestion des exceptions
- ...

C#

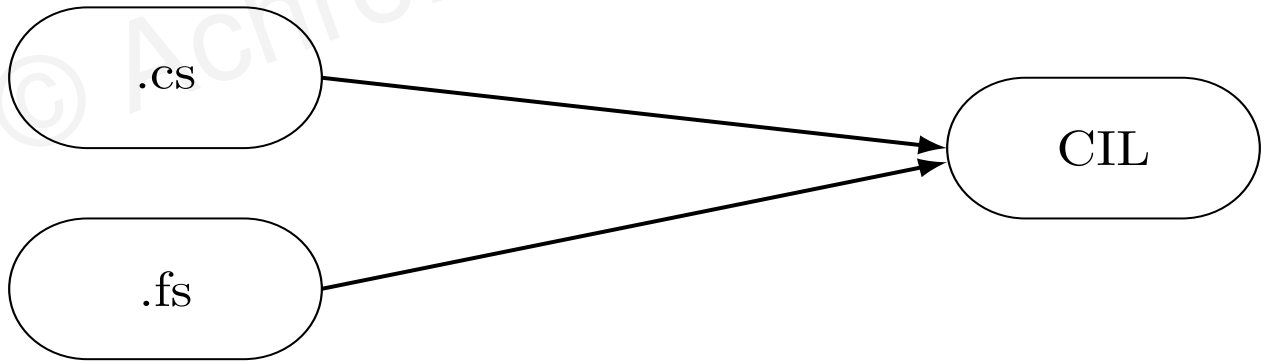
CoreCLR ?

- Initialement une copie de **CLR** pour **.NET Core**
- Modifié par la suite pour supporté **Linux** et **Mac**
- **CLR** et **CoreCLR** maintenus séparément et en parallèle

C#

Dans **.NET Core**

- On peut écrire un code **C#** et aussi **F#**.
- Tous ces langages seront compilés en code **CIL**.



C#

À partir d'un programme **C#**, il est possible de créer

- soit un fichier `.exe`
- soit une bibliothèque de classe sous la forme d'un fichier `.dll`

C#

À partir d'un programme **C#**, il est possible de créer

- soit un fichier `.exe`
- soit une bibliothèque de classe sous la forme d'un fichier `.dll`

C'est quoi la différence ?

- `.exe` (Executable) permet de lancer un programme
- `.dll` (**D**ynamic-**L**ink **L**ibrary) peut être utilisée par plusieurs programmes `.exe`

C#

À partir d'un programme **C#**, il est possible de créer

- soit un fichier `.exe`
- soit une bibliothèque de classe sous la forme d'un fichier `.dll`

C'est quoi la différence ?

- `.exe` (Executable) permet de lancer un programme
- `.dll` (**D**ynamic-**L**ink **L**ibrary) peut être utilisée par plusieurs programmes `.exe`

Dans les deux cas

On parle d'un `assembly`

C#

Que contient un assembly `.dll` ou `.exe` ?

- Le **CIL** : instructions de bas niveau traduites à partir du **C#**.
- Les métadonnées : informations descriptives qui expliquent la structure du **CIL**.
  - Les informations sur les types (classes, structures, interfaces)
  - Les signatures des méthodes
  - Les dépendances de l'assemblage

Quelques versions de **C#** (les plus importantes)

- **C#** 13 (sortie en 2025) : ˆ operator, field
- **C#** 12 (sortie en 2023) : spread operator
- **C#** 11 (sortie en 2022) : propriétés obligatoires
- **C#** 10 (sortie en 2021) : affectation et déclaration dans la dé-construction
- **C#** 9 (sortie en 2020) : Records, init-only setters.
- **C#** 8 (sortie en 2019) : méthodes d'interface par défaut, `ReadOnly`...
- **C#** 7 (sortie en 2016) : tuples, `out`, `ref`...
- **C#** 6 (sortie en 2015) : interpolation de chaîne de caractères, indexeur...
- **C#** 5 (sortie en 2012) : programmation asynchrone...
- **C#** 4 (sortie en 2010) : typage dynamique...
- **C#** 3 (sortie en 2007) : **LINQ**...
- **C#** 2 (sortie en 2005) : généricité, nullable, itérateurs, types partiels...
- **C#** 1 (sortie en 2002) : classes, interfaces, opérateurs, structures, instructions...

Principales versions d'**.NET Core**

- **.NET** 8.0 (LTS) sorti en novembre 2025 (supportant **C#** 13)
- **.NET** 9.0 sorti en novembre 2024
- **.NET** 8.0 (LTS) sorti en novembre 2023 (supportant **C#** 12)
- **.NET** 7.0 sorti en novembre 2022 (supportant **C#** 11)
- **.NET** 6.0 (LTS) sorti en novembre 2021 (supportant **C#** 10)
- **.NET** 5.0 sorti en novembre 2020 (supportant **C#** 9)
- **.NET Core** 3.1 (LTS) sorti en décembre 2019
- **.NET Core** 3.0 sorti en novembre 2019 (supportant **C#** 8)
- **.NET Core** 2.2 sorti en décembre 2018 (supportant **C#** 7.3)
- **.NET Core** 2.1 (LTS) sorti en main 2018
- **.NET Core** 2.0 sorti en août 2017
- **.NET Core** 1.1 sorti en novembre 2016
- **.NET Core** 1.0 sorti en juin 2016

C#

Remarque

- À partir de **.NET 5.0**, **.NET Core** a été rebaptisé **.NET**
- À partir de **.NET 5.0**, plus de nouvelles versions pour **.Net Framework**
- **ASP.NET Core** et **Entity Framework Core** suivent le même cycle de vie que **.NET Core**

C#

Règles de nommage en **C#**

- Projets et solutions : **PascalCase**
- Classes et fichiers : **PascalCase**
- Variables et objets : **camelCase**
- Méthodes : **PascalCase**

C#

Règles de nommage en **C#**

- Projets et solutions : **PascalCase**
- Classes et fichiers : **PascalCase**
- Variables et objets : **camelCase**
- Méthodes : **PascalCase**

Pour plus de détails

`https://wprock.fr/blog/conventions-nommage-programmation/`

C#

Instructions

- Chaque instruction se termine par ;
- Une instruction par ligne pour une meilleure lisibilité.

C#

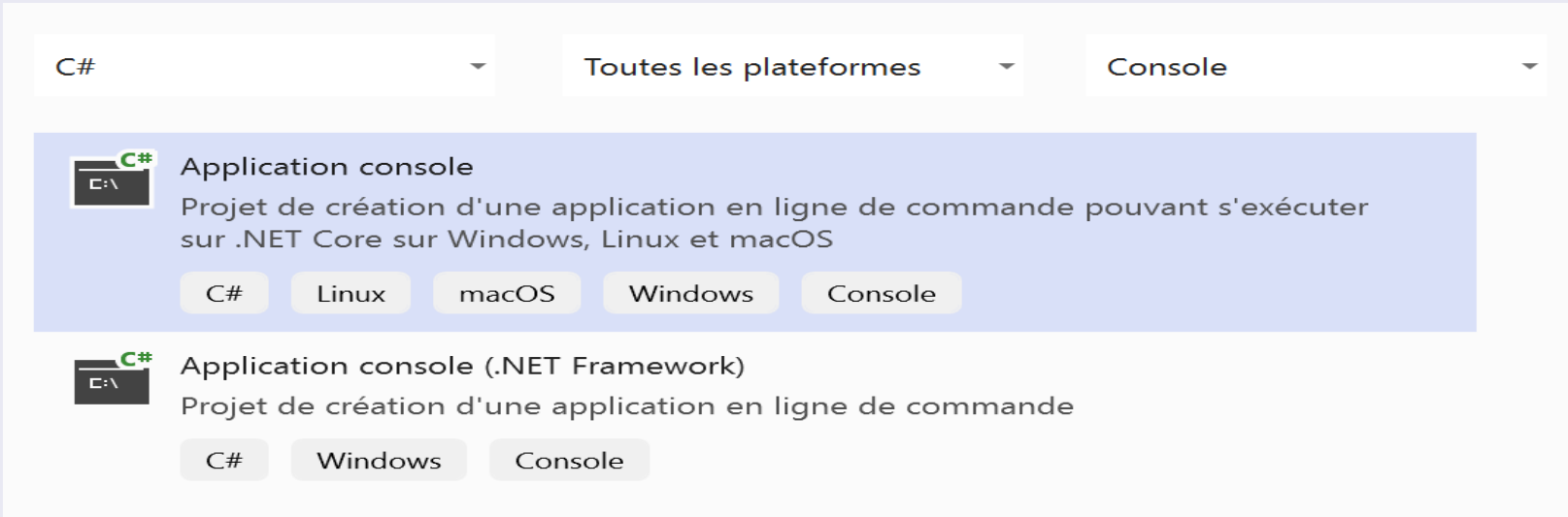
Comment organiser un projet **C#** ?

- Une classe par fichier.
- Possibilité de définir une classe dans plusieurs fichiers différents (classes partielles).
- Possibilité de regrouper plusieurs projets par solution.

C#

Étapes

- Créer un nouveau projet `Fichier > Nouveau > Projet`
- Choisir `C#` dans `Tous les langages`
- Sélectionner `Application console` (à ne pas confondre avec `Application console (.NET Framework)`)



C#

Étapes

- Cliquer sur `Suivant`
- Remplir les champs
  - `Nom :` avec `ConsoleCore`
  - `Solution` avec `MaSolution`
- Dans `Emplacement`, **Visual Studio** nous indique l'emplacement physique de notre projet (Par défaut dans `c:/utilisateurs/utilisateur/source/repos`).
- Valider

C#

Pour connaître la version de **C#** utilisée dans le projet

- Aller dans `Outils > Ligne de commande > Invite de commandes développeur`
- Dans la console, saisir la commande `csc -langversion:?`

C#

Code obtenu

```
namespace ConsoleCore
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Syntaxe et concept assez proches de celles de **C++** et **Java**.

C#

Explication

- Le mot-clé `namespace` permet d'indiquer l'espace de nom courant (comme un package **Java**).
- `static void Main(string[] args)` : point d'entrée de notre application console.

C#

À partir de .NET 6, le modèle de projet pour les nouvelles applications de console C# génère le code suivant dans le fichier `Program.cs`

```
Console.WriteLine("Hello, World!");
```

C#

Pour exécuter, cliquez sur

- ▶ `ConsoleCore`
- `Ctrl` + `F5`
- `fn` + `Ctrl` + `F5`

C#

Dans la console

- Le résultat du programme est affiché.
- Le titre de la console indique l'emplacement de l'exécutable `c:\utilisateurs\utilisateur\source\repos\MaSolution\ConsoleCore\bin\Debug\net10.0\ConsoleCore.exe`

C#

Pour accéder à l'exécutable (ConsoleCore.exe)

- Aller dans le menu Affichage et cliquer sur Explorateur de solutions
- Faire un clic droit sur MaSolution qui apparaît dans le panneau Explorateur de solutions et choisir Ouvrir le dossier dans l'Explorateur de fichiers
- Aller à ConsoleCore\bin\Debug\net10.0
- Faire un double-clic sur ConsoleCore.exe

Constat

La console apparaît et disparaît très rapidement avant de voir `Hello world`.

C#

Constat

La console apparaît et disparaît très rapidement avant de voir `Hello world`.

Solution

Ajouter une instruction bloquante

C#

Ajoutons l'instruction suivante puis exécutons le projet avant de vérifier l'exécutable

```
namespace ConsoleCore
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
            Console.ReadKey();
        }
    }
}
```

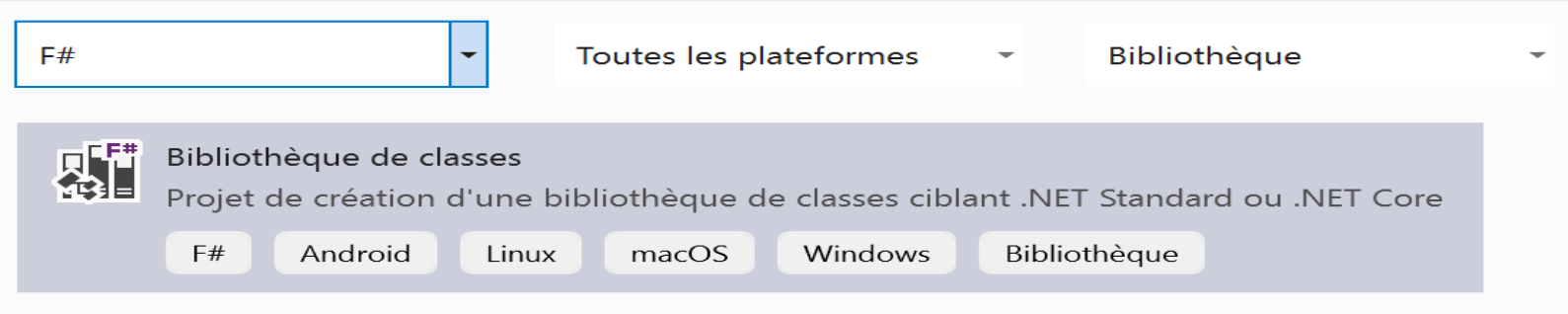
C#

Objectif

Écrire une solution **.NET** avec plusieurs langages de programmation.

Étape 1 : créer un projet **F#** dans `MaSolution`

- Dans l'`Explorateur de solutions`, faire clic droit sur `MaSolution`
- Aller à `Ajouter > Nouveau projet`
- Chercher `Bibliothèque de classes`



- Saisir `ProjetFSharp` dans `Nom`
- Valider
- Vérifier la présence de deux projets dans l'`Explorateur de solutions`

C#

Code obtenu

```
namespace ProjetFSharp

module Say =
    let hello name =
        printfn "Hello %s" name
```

C#

Code obtenu

```
namespace ProjetFSharp

module Say =
    let hello name =
        printfn "Hello %s" name
```


Si on vérifie le répertoire `bin/Debug` du projet **ProjetFSharp**, on verra qu'il est vide (pas d'exécutable), donc inexploitable.

C#

Étape 3 : générer le dll

- Aller dans l'Explorateur de solution
- Faire un clic droit sur le projet ProjetFSharp
- Choisir Générer

C#

Étape 3 : générer le dll

- Aller dans l'Explorateur de solution
- Faire un clic droit sur le projet ProjetFSharp
- Choisir Générer

Si on vérifie le répertoire bin/Debug, un fichier .dll a été généré.

C#

Étape 4 : connecter les deux projets

- Aller dans l'`Explorateur de solution`
- Dans le projet `ConsoleCore`, Faire un clic droit sur `Dépendances` et choisir `Ajouter une référence de projet`
- Cliquer sur `Projets` et cocher la case `ProjetFSharp`
- Valider

C#

Étape 4 : connecter les deux projets

- Aller dans l'`Explorateur de solution`
- Dans le projet `ConsoleCore`, Faire un clic droit sur `Dépendances` et choisir `Ajouter une référence de projet`
- Cliquer sur `Projets` et cocher la case `ProjetFSharp`
- Valider

Vérifier que `ProjetFSharp` figure dans la liste de dépendances de `ConsoleCore`.

C#

Étape 5 : utilisons le namespace `ProjetFSharp` du projet *F#*

```
using ProjetFSharp;

namespace ConsoleCore
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

C#

Étape 6 : appelons la méthode `hello` la classe `Say`

```
using ProjetFSharp;

namespace ConsoleCore
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Say.hello("Wick");
            Console.WriteLine("Hello, World!");
        }
    }
}
```

C#

En exécutant, le résultat est

```
Hello Wick
Hello World!
```

C#

Pour écrire dans la console

```
Console.WriteLine("Un message et un retour à la ligne");
```

Pour écrire sans retourner à la ligne

```
Console.Write("Un message sans retour à la ligne");
```

C#

Commentaire sur une seule ligne

```
// commentaire
```

C#

Commentaire sur une seule ligne

```
// commentaire
```


Raccourcis **Visual Studio 2026**

- Pour commenter : `Ctrl` + `k` puis `Ctrl` + `c`
- Pour décommenter : `Ctrl` + `k` puis `Ctrl` + `u`
- Ou pour commenter/décommenter : `Ctrl` + `:`

C#

Commentaire sur une plusieurs lignes

```
/* le commentaire
   la suite
   et encore la suite
*/
```

C#

Commentaire sur une plusieurs lignes

```
/* le commentaire
   la suite
   et encore la suite
*/
```


Raccourci **Visual Studio 2026**

Pour commenter ou décommenter : `Ctrl` + `Shift` + `:`

C#

Commentaire pour la documentation

```
/// un commentaire qui sera inclus dans la documentation
```

C#

Commentaire pour la documentation

```
/// un commentaire qui sera inclus dans la documentation
```

Pour le générer avec Visual Studio 2026

Aller dans `Edition > IntelliSense > Insérer un commentaire`

C#

Modifier la console

- `Console.BackgroundColor = ConsoleColor.Red;` pour mettre la couleur de fond en rouge
- `Console.ForegroundColor = ConsoleColor.Yellow;` pour mettre la couleur de caractères en jaune
- `Console.ResetColor();` pour réinitialiser les couleurs
- `Console.Clear();` pour effacer le contenu de la console
- `Console.SetCursorPosition(50, 50);` pour positionner la console
- ...

C#

Remarque

Pour consulter la configuration d'un projet **.NET**, faites double-clic sur le nom du projet dans l'`Explorateur de solutions`.

C#

Exemple de code

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

C#

Exemple de code

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Explication

- `<OutputType>Exe</OutputType>` : spécifie que la sortie de la compilation doit être une application console exécutable (EXE) et non une librairie (**DLL**).
- `<TargetFramework>net10.0</TargetFramework>` : indique que le projet cible la version 10.0 du **.NET**.
- `<ImplicitUsings>enable</ImplicitUsings>` : ajoute automatiquement les directives `using` communes.
- `<Nullable>enable</Nullable>` : active les types de référence nullable (minimise les erreurs de référence nulle).

C#

Remarque

Les **Implicit Usings** existent depuis **.NET.6** et visent à alléger le code.

C#

Remarque

Les **Implicit Usings** existent depuis **.NET.6** et visent à alléger le code.

Les Implicit Usings d'un Projet Console / Class Library

```
System
System.Collections.Generic
System.IO
System.Linq
System.Net.Http
System.Threading
System.Threading.Tasks
```

C#

Les Implicit Usings d'un Projet ASP.NET Core (Web API / MVC / Minimal API)

```
Microsoft.AspNetCore.Builder
Microsoft.AspNetCore.Hosting
Microsoft.AspNetCore.Http
Microsoft.AspNetCore.Routing
Microsoft.Extensions.Configuration
Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Hosting
Microsoft.Extensions.Logging
```

C#

La documentation officielle (en français)

`https://docs.microsoft.com/fr-fr/dotnet/csharp/index`