

CSharp

Perfectionnement

CSharp Perfectionnement

Sommaire

- [Rappels Essentiels](#)
- [Evolution du langage C# des versions 6 à 12](#)
- [Multithreading](#)
- [Asynchronisme](#)
- [Reflection et Attributes](#)
- [LINQ](#)
- [Interopérabilité](#)
- [Fonctionnement avancé du Runtime](#)
- [Garbage Collector](#)
- [Tests Unitaires en C#](#)
- [TDD](#)

Rappels Essentiels

Types .NET standards

- **System.Object** : Racine de tous les types
- **System.ValueType** : Base des types valeurs
- **System.String** : Type immuable pour les chaînes de caractères
- **System.Collections** et **System.Collections.Generic** : Collections standards

Les variables

Les **variables** ont pour but de **stocker** des informations dans la mémoire vive de l'ordinateur. Les **variables** peuvent être de plusieurs **types**, qui sont parmi les plus fréquents :

- Les variables de type **numériques** servant à stocker des nombres. On y retrouve différents types pour les **entiers** et les **réels**
- Les **caractères** et **chaines de caractères** pour stocker du texte
- Les **booléens**, pour les valeurs binaires (Vraie=True / Fausse=False)
- Le **Vide**, **null** en C#, est une valeur à part qui ne représente '**Rien**', il n'est ni un 0, ni un False, ni une chaine vide

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
bool	System.Bool	Booléen (vrai ou faux : true ou false)	true false
sbyte	System.SByte	Entier signé sur 8 bits (1 octet)	-128
byte	System.Byte	Entier non signé sur 8 bits (1 octet)	255
short	System.Int16	Entier signé sur 16 bits	-129
ushort	System.UInt16	Entier non signé sur 16 bits	1450
int	System.Int32	Entier signé sur 32 bits	-100000
uint	System.UInt32	Entier non signé sur 32 bits	8000000
long	System.Int64	Entier signé sur 64 bits	-2565018947302L
ulong	System.UInt64	Entier non signé sur 64 bits	80000000000000L

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
float	System.Single	Réel sur 32 bits	3.14F
double	System.Double	Réel sur 64 bits	3.14159
decimal	System.Decimal	Réel sur 128 bits	3.1415926M
char	System.Char	Caractère Unicode (16 bits)	'A' 'λ' 'ω'
string	System.String	Chaîne de caractères unicode	"C:\\windows\\system32"
Tuple	System.ValueTuple<>	Regroupement de données	(4.5, "test")
dynamic	NON TYPÉE	Variable dynamique (faiblement typée et "évaluée" à l'exécution)	2 "test" 3.14 true

Les différents types de variable

Type	Classe (BCL)	Description	Exemples
enum	System.Enum	Énumération de possibilités	<code>enum Season {...}</code>
delegate	System.Action<> System.Func<>	Fonction/Méthode anonyme	<code>x => x*2</code>
struct		Structure de données (par valeur)	<code>struct Person {...}</code>
object	System.Object	Tous types d' objets instanciés	<code>new Person(...){...}</code>
class		Classes pour instancier des objets	<code>class Person {...}</code>
record		Enregistrement (classe simplifiée)	<code>record Person(...)</code>
interface		Définition d'un contrat	<code>interface MyContract {...}</code>

Type	Exemples
sbyte	-128 à 127
byte	0 à 255
short	-32 768 à 32 767
ushort	0 à 65 535
int	-2 147 483 648 à 2 147 483 647
uint	0 à 4 294 967 295
long	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
ulong	0 à 18 446 744 073 709 551 615
float	$\pm 1,5 \times 10^{-45}$ à $\pm 3,4 \times 10^{38}$ soit ~6-9 chiffres
double	$\pm 5,0 \times 10^{-324}$ à $\pm 1,7 \times 10^{308}$ soit ~15-17 chiffres
decimal	$\pm 1,0 \times 10^{-28}$ to $\pm 7,9228 \times 10^{28}$ soit 28-29 chiffres

Quelques précisions

Le mot clé **var** s'utilise à la place du type d'une variable, il **ne rend pas cette variable dynamique** mais **la type implicitement**, ainsi son type dépendra de **l'instruction d'affectation**

Le type **decimal** est utilisé pour les opérations financières

- Il permet une **très grande précision**
- Les opérations avec ce type plus lentes que les types **double** ou **float**

Cas exceptionnels

- Les **programmes** doivent souvent gérer des **situations exceptionnelles**, rendant le code **complexe** et difficile à lire.
- Exemples : saisie utilisateur en int, division par zéro
- En algorithmie, on appelle ces **situations exceptionnelles** des **Exceptions**
- C# introduit un **mécanisme de gestion des exceptions** pour **séparer le code utile du traitement de ces cas exceptionnels**.

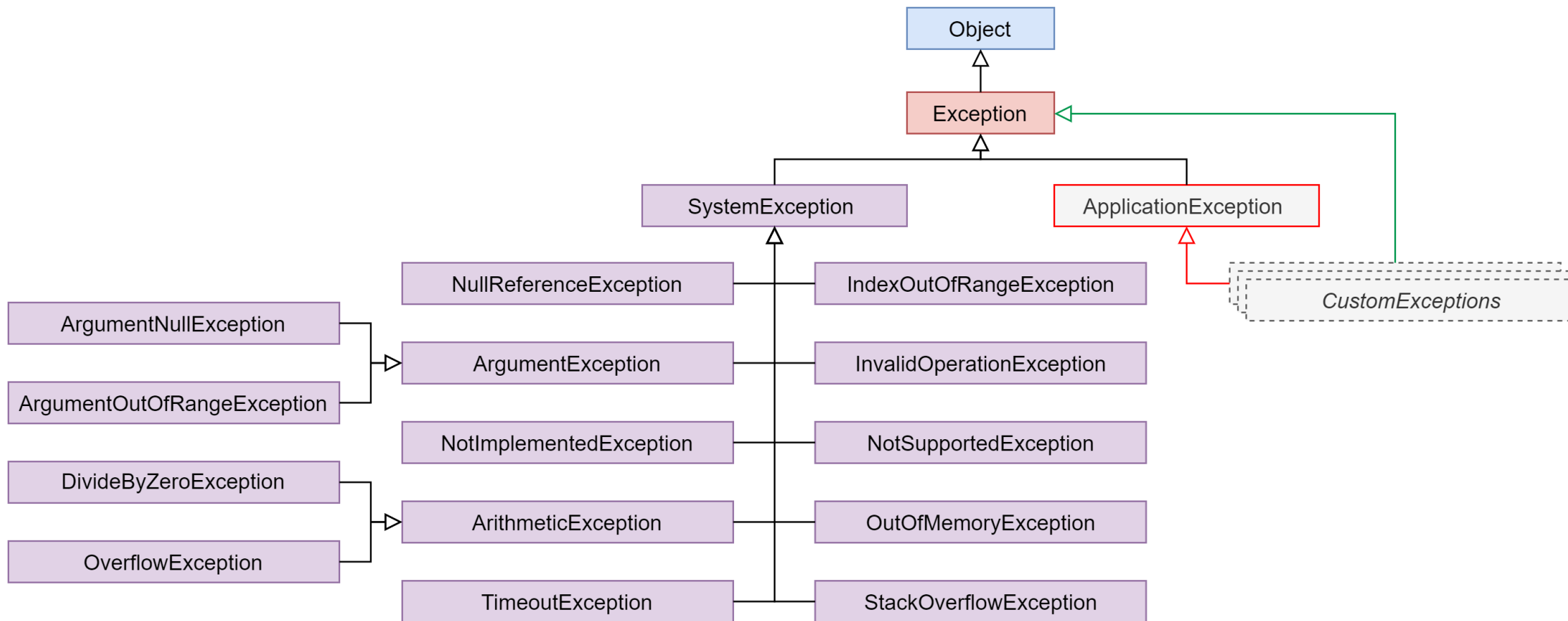
Pour des information plus détaillées sur les exceptions merci de vous référer à la partie dédiée du support "Csharp Avancé"

Gestion des exceptions

- **try-catch-finally** : Structure de base pour la gestion des exceptions
- **Exception** : Classe de base des exceptions
- Utilisation de `throw` pour déclencher des exceptions

```
try {  
    int result = 10 / 0;  
} catch (DivideByZeroException ex) {  
    Console.WriteLine("Division par zéro !");  
} finally {  
    Console.WriteLine("Opération terminée.");  
}
```

Architecture des exceptions les plus connues



ApplicationException n'est plus utilisée, on hérite d'Exception directement

Evolution du langage C# des versions 6 à 12

Evolution du langage C#

Version	Année	Version .NET	Principales fonctionnalités
C# 1	2002	.NET Framework 1.0	Classes, Struct, Delegates, Events
C# 2	2005	.NET Framework 2.0	Generics, Nullable types, Anonymous methods
C# 3	2007	.NET Framework 3.5	LINQ, Lambda expressions, Extension methods
C# 4	2010	.NET Framework 4.0	Dynamic typing, Optional parameters
C# 5	2012	.NET Framework 4.5	Async/await for asynchronous programming
C# 6	2015	.NET Framework 4.6	Null-conditional, String interpolation, Expression-bodied members
C# 7	2017	.NET Framework 4.6.2 / .NET Core 2.0	Pattern matching, Tuples, Local functions, <code>ref</code> returns

Evolution du langage C#

Version	Année	Version .NET	Principales fonctionnalités
C# 8	2019	.NET Core 3.0 / .NET 5	Nullable reference types, Default interface methods, Async streams
C# 9	2020	.NET 5	Records, Init-only properties, Improved pattern matching
C# 10	2021	.NET 6	Global using directives, File-scoped namespaces, synthetic program.cs
C# 11	2022	.NET 7 (STS)	Generic attributes, List patterns, nameof scope
C# 12	2023	.NET 8	Primary constructors, Inline arrays, Lambda default parameters
C# 13	2024	.NET 9 (STS)	Lock, params enhancement, \e escape sequence, Implicit index access, ...

Suivre l'évolution du .NET et C#

[Support Policy](#)

[Version History](#)

[Language Versioning](#)

Dépôts Github à Watch :

- [.NET Core Announcements](#)
- [ASP.NET Core & Entity Framework Core Announcements](#)

[Timeline](#)

[Language feature Status](#)

[Chaine YouTube](#)

C# 6 : Null-conditionnel

- Opérateur `?.` pour gérer les références nulles sans déclencher d'exception
- Permet de simplifier les vérifications de nullité, surtout dans les structures imbriquées

```
string? name = person?.Name; // Retourne null si person est null
```

C# 6 : Amélioration des propriétés automatiques

- Initialisation directe des propriétés automatiques dès leur déclaration
- Support des propriétés en lecture seule (`get` uniquement) sans définir un `set` manuel

```
public string Name { get; } = "John"; // Lecture seule, initialisée à "John"
```

C# 6 : Fonctions "Expression Bodied"

- Simplifie les méthodes courtes et les propriétés en les écrivant sous forme d'expressions

```
public int Square(int x) => x * x; // Retourne le carré de x
```

C# 7 : Lisibilité des constantes

- Permet l'ajout du séparateur  pour améliorer la lisibilité des grands nombres, sans changer leur valeur

```
const int largeNumber = 1_000_000; // Plus facile à lire
```

C# 7 : Variables "out"

- Déclaration directe des variables "out" dans les méthodes, rendant le code plus concis et lisible

```
if (int.TryParse("123", out int result)) {  
    Console.WriteLine(result); // Affiche 123  
}
```

C# 7 : Tuples

- Syntaxe simplifiée pour les tuples, permettant de retourner facilement plusieurs valeurs d'une méthode

```
(string, int) person = ("Alice", 30);  
Console.WriteLine(person.Item1); // Affiche "Alice"  
  
(string NomComplet, string NomMaj, string Prenom) NomComplet(string nom, string prenom)  
{  
    var nomMaj = nom.ToUpper();  
    var t = (nomMaj + " " + prenom, nomMaj, prenom);  
    return t;  
}  
  
var t3 = NomComplet("Guillaume", "Mairesse");  
  
Console.WriteLine(t3);  
Console.WriteLine(t3.NomComplet);  
Console.WriteLine(t3.Item1);
```


C# 7 : Pattern Matching

- Simplifie la vérification des types et des valeurs, permettant un code plus sûr et mieux structuré

```
if (obj is string s) {  
    Console.WriteLine($"String length: {s.Length}"); // Vérifie et cast en une seule étape  
}
```

C# 7 : Retour de référence

- Utilisation de `ref` pour renvoyer une référence plutôt qu'une copie, utile pour la performance et pour éviter une duplication de la donnée

```
public ref int Find(int[] array, int value) { ... }  
// la référence retournée correspondra à celle dans le tableau
```

C# 8 : Expressions Switch

- Nouvelle syntaxe pour `switch`, permettant des expressions plus concises et flexibles

```
var result = direction switch
{
    "North" => "↑",
    "South" => "↓",
    _ => "?" // default
};
```

```
var value = 25;

int example3 = value switch
{
    _ when value > 10 => 0,
    _ when value <= 10 => 1,
    _ => throw new Exception(),
};
```

C# 8 : Méthodes d'interface par défaut

- Implémentation de méthodes par défaut dans les interfaces, permettant des évolutions non destructives

```
public interface IShape {  
    double Perimeter() {return 0;}  
    double Area() => 0;  
    // Fournit une implémentation par défaut  
}
```

/!\ Il n'est pas recommandé de les utiliser (SOLID, ...)

C# 8 : Type référence Nullable

- Gestion des types nullable pour les références, aidant à réduire les erreurs de nullité

```
string? nullableString = null; // Peut accepter la valeur null
```

- Configurable dans le .csproj (Nullable enable/disable) et avec la directive `#nullable`

C# 9 : Records

- Type immuable pour les données, particulièrement utile pour les modèles de données et DTOs, peut avoir constructeurs, méthodes
- Immuable signifie qu'une entité, comme une variable ou un objet, ne peut pas être modifiée une fois qu'elle a été créée.
- La comparaison se fait avec le contenu plutôt que la référence

```
public record Person(string Name, int Age); // Crée une classe immuable

var person1 = new Person("Alice", 30);
var person2 = new Person("Alice", 30);
Console.WriteLine(person1 == person2); // Affiche "True"
var (name, age) = person1; // Déstructure un record en variables
var person3 = person1 with { Age = 31 }; // Crée une nouvelle instance avec l'âge modifié
```

C# 9 : Target-typed new

- Simplifie l'instanciation en utilisant le type de destination

```
Person person = new("Alice", 30); // Type déduit de la déclaration
```

- Incompatible avec `var` car opposé

C# 9 : Pattern Matching (Améliorations)

- Ajout de `is not`, `and`, `or` pour construire des expressions plus complexes
- Type patterns

```
if (shape is Circle or Square) { ... } // Combine les types
```

- Relational Patterns

```
public static LifeStage LifeStageAtAge(int age) => age switch  
{  
    < 0 => LifeStage.Prenatal,  
    < 6 => LifeStage.EarlyChild,  
    < 12 => LifeStage.MiddleChild,  
    < 18 => LifeStage.Adolescent,  
    < 40 => LifeStage.EarlyAdult,  
    _ => LifeStage.LateAdult,  
};
```


C# 10 : Global Namespace

- Déclaration de namespaces globales pour simplifier l'utilisation des namespaces

```
global using System.Text; // Accessible dans tout le projet
```

- Utilisé dans le `Program.cs` du .NET 6 pour les namespaces de base

C# 10 : File-scoped Namespaces

- Simplifie la déclaration de namespace pour réduire l'indentation

```
namespace MyApp;
```

- Anciennement :

```
namespace MyApp  
{  
    ...  
}
```

C# 10 : Record Struct

- Structure immuable avec les avantages des `records`, adaptée aux types de valeur

```
public record struct Point(int X, int Y); // Struct immuable
```

C# 11 : Attributs génériques

- Permet d'utiliser des attributs génériques pour simplifier et renforcer le typage

```
[Example<T>]
```

C# 11 : Amélioration `IntPtr` et `UIntPtr`

- `IntPtr` et `UIntPtr` représentent des pointeurs en mémoire (32/64 bits) signé et non signés (unsigned).

Améliorations de C# 11 :

- Introduction des **littéraux natifs** `nint` (pour `IntPtr`) et `nuint` (pour `UIntPtr`).
- Simplifie la déclaration et l'utilisation des pointeurs natifs.
- Meilleure **interopérabilité** avec du code natif tout en conservant la **sécurité** du langage.
- Meilleure gestion dans les génériques

C# 11 : Modèles de listes (Pattern Matching)

- Permet de vérifier des motifs dans des listes, facilitant la vérification de séquences

```
if (list is []) { /* vide */ }
if (list is [_, _, _]) { /* liste de trois éléments, valeurs quelconques */ }
if (list is [1, 2, 3]) { /* éléments exacts */ }
if (list is [1, 2, _]) { /* commence par 1 et 2, troisième ignoré */ }
if (list is [1, 2, ..]) { /* commence par 1 et 2, reste ignoré */ }
if (list is [1, .., 4]) { /* commence par 1, finit par 4 */ }
if (list is [var first, 2, var last])
{ /* first premier élément, last le dernier, et middle le reste */ }
if (list is [1, 2, .. var lasts]) { /* reste dans lasts */ }
if (list is [>=10, >=20, >=30]) { /* valeurs min */ }
```

C# 11 : Portée `nameof` étendue

- Utilisation de `nameof` dans des contextes supplémentaires, améliorant la maintenabilité

```
Console.WriteLine(nameof(MyClass.MyProperty));
```

C# 11 : Amélioration mathématiques génériques

- Permet des opérations mathématiques génériques, simplifiant les calculs avec les types numériques `System.Numerics`

```
using System.Numerics;

T Add<T>(T a, T b) where T : INumber<T> // types numériques de base
{
    return a + b;
}

T Addition<T>(T a, T b) where T : IAdditionOperators<T, T, T> // types supportant l'addition
{
    return a + b;
}
```

[Documentation](#)

C# 12 : Constructeurs primaires

- Simplifie les constructeurs avec des paramètres directement dans la déclaration de classe

```
public class Person(string name, int age) { ... }
```

- Les paramètres nom et age deviennent des propriétés en lecture seule

C# 12 : Tableaux inline

- Permet l'initialisation directe des tableaux, améliorant la lisibilité

```
int[] numbers = [1, 2, 3];  
// avant :  
int[] nombres = new int[] { 1, 2, 3 };
```

- Ne fonctionne pas avec `var`

C# 12 : Intercepteurs

- Méthodes d'interception permettant d'ajouter des logiques de validation ou transformation à des appels de méthodes
- Est actuellement une feature expérimentale
- [Feature Specification](#)

C# 12 : Paramètres **ref readonly**

- Améliore la performance des méthodes en limitant la copie de paramètres grands ou complexes tout en gardant leur immutabilité

```
public struct Vecteur3D
{
    public double X { get; }
    public double Y { get; }
    public double Z { get; }

    public Vecteur3D(double x, double y, double z) => (X, Y, Z) = (x, y, z);
}

public static double CalculerDistance(ref readonly Vecteur3D vecteur)
{
    return Math.Sqrt(vecteur.X * vecteur.X + vecteur.Y * vecteur.Y + vecteur.Z * vecteur.Z);
}

Vecteur3D point = new Vecteur3D(3, 4, 5);
double distance = CalculerDistance(ref point);
```

C# 12 : Paramètres Lambda par défaut

- Valeurs par défaut dans les expressions lambda, simplifiant leur utilisation dans les filtres et calculs

```
Func<int, int, int> addition = (a, b = 5) => a + b;
```

C# 12 : Alias any type

- Nouveau :

```
using IntPair = (int int1, int int2);  
using NullableInt = int?; // que pour les types valeur, ex : pas pour string
```

- Déjà présent

```
using Csl = System.Console;  
using Days = System.DayOfWeek;  
using IntStringMap = System.Collections.Generic.Dictionary<int, string>;  
using TupleList = System.Collections.Generic.List<(string Name, int Age)>;  
using NullableStringList = System.Collections.Generic.List<string?>;  
using Operation = System.Func<int, int, int>;
```

Multithreading

Introduction au Multithreading

- **Exécution parallèle** de plusieurs **threads** (fils d'exécution) pour **maximiser l'utilisation des ressources processeur**
- Permet d'améliorer la réactivité et les performances des applications en exécutant **plusieurs tâches simultanément** (**concurrency**)

Cas d'utilisation du Multi-threading

- **Traitement parallèle** : Utiliser plusieurs threads pour traiter de grandes quantités de données en parallèle (ex. traitement d'image, calculs scientifiques).
- **Réactivité de l'interface utilisateur** : Garder l'interface réactive tout en exécutant des tâches lourdes en arrière-plan (ex. téléchargement, traitement des fichiers).
- **Serveurs et applications réseau** : Gérer plusieurs connexions ou requêtes simultanément, chaque thread étant responsable d'une connexion client.

Thread et Task

- **Thread** : **Unité d'exécution de base** (bas niveau) dans un programme, gérée par le **système d'exploitation**.
Chaque thread a **sa propre pile d'exécution** (Stack) et peut être **exécuté simultanément**. Leur gestion se fait **manuellement**.
- **Task** : **Abstraction de niveau supérieur** utilisée pour la **gestion de la concurrence** en C#.
Simplifie le travail avec le multithreading, gère **automatiquement** le thread de manière **optimisée**.
Permet de gérer les **tâches asynchrones** de façon **plus simple** que l'utilisation directe de **Thread**, nous y reviendrons ensuite.

Exemple Thread

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        // Création d'un thread et assignation d'une méthode
        Thread thread = new Thread(DoWork);

        // Démarrage du thread
        thread.Start();

        // Attendre que le thread se termine avant de continuer
        thread.Join();

        Console.WriteLine("Travail terminé !");
    }

    static void DoWork()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"Exécution du thread : {i}");
            Thread.Sleep(500); // Pause de 500 ms pour simuler du travail
        }
    }
}
```

Exemple Task

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        // Appel d'une méthode asynchrone
        Task tache = DoWorkAsync();

        // Attendre que la tache se termine avant de continuer
        await tache;

        Console.WriteLine("Travail asynchrone terminé !");
    }

    static async Task DoWorkAsync()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"Exécution de la tâche : {i}");
            await Task.Delay(500); // Attendre 500 ms pour simuler un travail asynchrone
        }
    }
}
```

Race-Conditions

Problèmes de Synchronisation :

- Accès **concurrent** aux **ressources partagées** peut provoquer des conditions de course (**Race-Conditions**).
- Il n'est alors pas possible de savoir à l'avance quelle instruction de quel Thread va s'exécuter avant une autre
- **Deux exécution d'un même programme** peuvent donc donner des **résultats divergeants**

Exemple Race-Conditions

- Deux threads accèdent à la même variable partagée.
- Chaque thread **incrémente** la valeur dans une **boucle**.
- Cependant, comme l'opération `counter++` n'est **pas atomique** (lire, incrémenter, écrire), les deux threads peuvent **modifier** `counter` **en même temps**.
- Le **résultat** est **indéfini**.

```
class Program
{
    // Variable partagée entre les threads
    private static int counter = 0;

    static void Main()
    {
        // Création de deux threads qui accèdent à la même ressource
        Thread thread1 = new Thread(IncrementCounter);
        Thread thread2 = new Thread(IncrementCounter);


        // Démarrage des threads
        thread1.Start();
        thread2.Start();

        // Attente de la fin de l'exécution des threads
        thread1.Join();
        thread2.Join();

        // Affichage du résultat
        Console.WriteLine($"Valeur finale du compteur : {counter}");
    }

    // Méthode qui incrémente la variable partagée
    static void IncrementCounter()
    {
        for (int i = 0; i < 100000; i++)
        {
            counter++; // Potentielle condition de concurrence ici
        }
    }
}
```

Synchronisation des Threads

- Il existe des **solutions** au **problème de Race-Condition**
- **Verrous (lock)** : Utilisation d'un **mécanisme de verrouillage** pour **éviter l'accès simultané** aux **ressources partagées**.
- En C#, on utilise souvent une variable `static readonly` de type `object` (ou autre) comme verrou et un bloc de code [lock](#).
-  A partir de .NET 9 / C# 13, il sera recommandé d'utiliser le type `System.Threading.Lock` pour de meilleures performances.

Exemple lock

```
class Program
{
    // Solde partagé entre les threads
    private static int balance = 0;
    // Objet de verrouillage pour synchroniser l'accès à la balance
    private static readonly object balanceLock = new object();

    static void Main()
    {
        // Création de threads pour simuler des dépôts et des retraits
        Thread thread1 = new Thread(Deposit);
        Thread thread2 = new Thread(Withdraw);

        // Démarrage des threads
        thread1.Start();
        thread2.Start();

        // Attente de la fin des threads
        thread1.Join();
        thread2.Join();

        // Affichage du solde final
        Console.WriteLine($"Solde final du compte : {balance}");
    }
}
```

```
// Méthode pour simuler un dépôt d'argent
static void Deposit()
{
    for (int i = 0; i < 1000; i++)
    {
        lock (balanceLock) // Blocage de l'accès à 'balance'
        {
            balance += 100; // Ajouter 100 au solde
            Console.WriteLine($"Déposé 100, solde actuel : {balance}");
        }
        Thread.Sleep(10); // Pause pour simuler le délai de l'opération
    }
}

// Méthode pour simuler un retrait d'argent
static void Withdraw()
{
    for (int i = 0; i < 1000; i++)
    {
        lock (balanceLock) // Blocage de l'accès à 'balance'
        {
            if (balance >= 50)
            {
                balance -= 50; // Retirer 50 du solde
                Console.WriteLine($"Retiré 50, solde actuel : {balance}");
            }
        }
        Thread.Sleep(10); // Pause pour simuler le délai de l'opération
    }
}
```


Dead Locks

Un **deadlock** se produit lorsque deux threads s'attendent mutuellement à libérer des objets, créant ainsi un **blocage infini**.

```
class Program
{
    // Objets Locks
    static readonly object l1 = new object();
    static readonly object l2 = new object();

    static void Main()
    {
        Thread t1 = new Thread(Thread1);
        Thread t2 = new Thread(Thread2);
        t1.Start();
        t2.Start();
        //...
        t1.Join();
        t2.Join();
        Console.WriteLine("Fin du programme");
    }
}
```

```
static void Thread1()
{
    lock (l1) // Le premier thread verrouille l1
    {
        Console.WriteLine("Thread1 locked l1");

        // Attendre un moment pour simuler un délai
        Thread.Sleep(100);

        lock (l2) // Puis il essaie de verrouiller l2
        {
            Console.WriteLine("Thread1 locked l2");
        }
    }
}
```

```
static void Thread2()
{
    lock (l2) // Le second thread verrouille d'abord l2
    {
        Console.WriteLine("Thread2 locked l2");

        // Attendre un moment pour simuler un délai
        Thread.Sleep(100);

        lock (l1) // Puis il essaie de verrouiller l1
        {
            Console.WriteLine("Thread2 locked l1");
        }
    }
}
```

Moniteurs

- **Monitor** : Une classe plus flexible que **lock**, car elle offre des méthodes comme **Monitor.Enter**, **Monitor.Exit**, et **Monitor.Wait/**
Monitor.Pulse, permettant une gestion plus détaillée de la synchronisation.

```
private static readonly object monitorLock = new object();

void ThreadSafeMethod()
{
    Monitor.Enter(monitorLock);
    try
    {
        // Code à exécuter de manière thread-safe
        // possible d'utiliser Monitor.Wait/.Pulse/.PulseAll pour l'attente et la notification
    }
    finally
    {
        Monitor.Exit(monitorLock);
    }
}
```

Classes de synchronisation avancée

- **SemaphoreSlim** : Gère un **nombre limité de threads** pouvant accéder à **une ressource partagée** simultanément. Par rapport à **Semaphore**, **SemaphoreSlim** est plus léger et souvent préféré.
- **Mutex** : Permet une synchronisation entre **plusieurs processus** (au-delà des threads), mais est généralement **plus lent** que **lock** ou **Monitor** pour les cas intra-processus.
- **ReaderWriterLockSlim** : Optimisé pour les scénarios où plusieurs threads doivent lire des données en parallèle, mais l'écriture est faite par un seul thread à la fois.

Communication Inter-Threads

- Pour **échanger des messages/données** entre les threads, on peut utiliser utiliser une `Queue<T>` (**First-In/First-Out**) ou, pour une gestion de la concurrence plus robuste, une `BlockingCollection<T>`.
- On aura alors des **producteurs** et des **consommateurs**.
- La `BlockingCollection<T>` offre des méthodes de blocage comme `Take` et `Add` pour **éviter les conditions de course** et gérer l'**attente** quand la collection est **vide** ou **pleine**.

L'équivalent pour les Task correspond aux Channels

Exemple `BlockingCollection<T>`

```
class Program
{
    // Création de la BlockingCollection pour la communication
    static BlockingCollection<string> messageQueue = new();

    static void Main()
    {
        // Crée et lance les threads producteurs et consommateurs
        Thread producerThread = new Thread(Producer);
        Thread consumerThread = new Thread(Consumer);

        producerThread.Start();
        consumerThread.Start();

        // Attendez que les threads se terminent
        producerThread.Join();
        consumerThread.Join();

        Console.WriteLine("Communication terminée.");
    }
}
```

```
// Thread producteur
static void Producer()
{
    string[] messages = { "Message 1", "Message 2",
                          "Message 3", "Message 4" };

    foreach (var message in messages)
    {
        Console.WriteLine($"Producteur envoie : {message}");
        messageQueue.Add(message); // Ajoute un message à la queue
        Thread.Sleep(1000); // Simule un délai entre les envois
    }

    Console.WriteLine("Fin de l'envoi");
    // Signale la fin de la production
    messageQueue.CompleteAdding();
}

// Thread consommateur
static void Consumer()
{
    foreach (var message in messageQueue.GetConsumingEnumerable())
    // Bloque et attend de nouveaux messages
    {
        Console.WriteLine($"Consommateur a reçu : {message}");
        Thread.Sleep(1500); // Simule un délai de traitement
    }

    Console.WriteLine("Fin de la réception");
}
```

ThreadPool et Efficacité

- Le **ThreadPool** permet de **gérer un ensemble de threads réutilisables, améliorant ainsi l'efficacité en réduisant la surcharge liée à la création et à la destruction** de threads.
- Il est utilisé pour exécuter des tâches de manière parallèle **sans avoir à créer de nouveaux threads à chaque fois** et en **limitant leur nombres** (géré par le runtime .NET).
- Les threads sont **réutilisés** à la fin de leur tâche en cours.

Exemple ThreadPool

```
class Program
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            ThreadPool.QueueUserWorkItem(ExecuteTask, i);
        }

        Console.WriteLine("Toutes les tâches ont été soumises.");
        Console.ReadLine();
    }

    static void ExecuteTask(object taskId)
    {
        int id = (int)taskId;
        Console.WriteLine($"Tâche {id} sur le thread {Thread.CurrentThread.ManagedThreadId}.");
        Thread.Sleep(2000); // Simule un délai
    }
}
```

Daemon Threads

- Pour créer un **Daemon Thread**, qui s'arrête automatiquement à la fin du programme principal, définissez `IsBackground` à `true`.
- Ils sont souvent utilisés pour des tâches en arrière-plan (surveillance, gestion de la mémoire ou mise à jour des données).

```
Thread daemonThread = new Thread(() =>
{
    while (true)
    {
        Console.WriteLine("Daemon thread en arrière-plan...");
        Thread.Sleep(1000);
    }
});

daemonThread.IsBackground = true; // Définit comme daemon
daemonThread.Start();

Console.WriteLine("Programme principal en cours...");
Thread.Sleep(3000);
Console.WriteLine("Fin du programme principal.");
```


Asynchronisme

Introduction à l'asynchronisme

- L'asynchronisme permet de **ne pas bloquer l'exécution** d'un programme **pendant qu'il attend une opération**, comme une lecture de fichier ou une requête réseau.
- En C#, la gestion de l'asynchronisme se fait principalement avec les mots-clés **async** et **await** et l'utilisation de **Task<>**.

```
public async Task<string> DownloadDataAsync(string url) {  
    HttpClient client = new HttpClient();  
    return await client.GetStringAsync(url); // Ne bloque pas le thread principal  
}
```

Différence entre Appels Synchrones et Asynchrones

Appels synchrones :

- Le programme **attend** que l'opération soit **terminée**.
- Peut entraîner des **blocages** si une opération **prend du temps** (téléchargement d'un fichier, appel à une API).

Appels asynchrones :

- Le programme **continue** son exécution **sans attendre** que l'opération soit **terminée**, permettant ainsi de **ne pas bloquer** l'interface ou le thread principal.

Async Opérations

- Le mot-clé **async** permet de **marquer une méthode** comme étant **asynchrone**.
- **await** permet d'**attendre le résultat** d'une opération asynchrone **sans bloquer le thread** courant.
- **Exécution non-bloquante** : Pendant l'attente d'une opération asynchrone d'autres tâches peuvent être exécutées.

```
public async Task ProcessDataAsync() {  
    Task<TypeDeData> tache = await GetDataFromDatabaseAsync(); // Ne bloque pas le thread principal  
    // ... Autres taches  
    TypeDeData data = await tache; // Attente de la fin de la tache et récupération du résultat  
    Console.WriteLine(data);  
}
```

Précisions

- La syntaxe `NomMéthodeAsync()` est utilisé pour indiquer qu'une méthode est **asynchrone**, c'est une **norme à toujours utiliser**.
- Lorsqu'une méthode asynchrone ne retourne **rien** (`void`), vous pouvez utiliser `Task` comme type de retour.
- Si la méthode asynchrone **doit retourner** un résultat d'un **type particulier**, vous utilisez `Task<Type>` comme type de retour.

Gestion de la progression

- La classe `IProgress<T>` permet de **suivre l'avancement** d'une opération asynchrone.
- **Utilisation** pour afficher des barres de progression ou mettre à jour l'interface utilisateur **pendant des tâches longues**.

```
class Program
{
    static async Task Main(string[] args)
    {
        // Créez un objet IProgress pour suivre l'avancement
        IProgress<double> progress = new Progress<double>(percent =>
        {
            Console.WriteLine($"Progression : {percent}%");
        });

        // Appelez la méthode asynchrone avec gestion de la progression
        await LongRunningOperationAsync(500, progress);
    }
}
```

```
public static async Task LongRunningOperationAsync(int nbTaches,
                                                    IProgress<double> progress)
{
    for (int i = 1; i <= nbTaches; i++)
    {
        // Simule un travail avec un délai aléatoire par étape
        await Task.Delay(new Random().Next(100));

        double avancementPct = (double) i * 100 / nbTaches;

        if (avancementPct % 5 == 0)
            progress.Report(avancementPct);
        // Signal l'avancement à l'objet IProgress
        // et Reporte la progression en pourcentage
    }

    Console.WriteLine("Opération terminée !");
}
```

Abandon d'une opération asynchrone

- Utilisation d'un **token d'annulation** (CancellationToken) pour **annuler** une tâche asynchrone **si nécessaire**.

```
public class CancellationExample
{
    public static async Task Main(string[] args)
    {
        // Création d'un CancellationTokenSource
        using CancellationTokenSource cts = new();

        // On lance la tâche avec le token d'annulation
        Task longRunningTask = LongRunningOperationAsync(cts.Token);

        // On simule un délai avant d'annuler la tâche
        await Task.Delay(2000);

        // Annuler la tâche
        cts.Cancel();

        // Attendre que la tâche termine (ou soit annulée)
        try { await longRunningTask; }
        catch (OperationCanceledException)
        { Console.WriteLine("L'opération a été annulée."); }
    }
}
```

```
public static async Task LongRunningOperationAsync(
    CancellationToken cancellationToken)
{
    Console.WriteLine("L'opération longue a commencé.");

    for (int i = 0; i < 10; i++)
    {
        // Vérifier si l'annulation a été demandée
        cancellationToken.ThrowIfCancellationRequested();

        // Simuler un travail de longue durée
        Console.WriteLine($"Traitement en cours... {i + 1}");
        await Task.Delay(100); // Délai entre chaque étape
    }

    Console.WriteLine("L'opération longue est terminée.");
}
```

Reflection et Attributes

Introduction à la Reflection

- La **Reflection** en C# permet d'inspecter et de **manipuler** les **métadonnées** des **types**, des **objets** et des **assemblies** à l'**exécution**.
- Elle est utilisée pour **obtenir des informations** sur les types, les méthodes, les propriétés, etc., et pour **invoquer dynamiquement du code**.
- Très utile dans des scénarios comme la création d'outils de test, la sérialisation, ou la gestion dynamique des objets.

Reflection vs Introspection

- **Réflexion** : Concept **général**, inclut l'inspection et l'**interaction** avec des types et objets **à l'exécution**.
- **Introspection** : **Sous-ensemble** de la réflexion, se limite à l'**examen passif de la structure** sans intervention active.

Introspection des Assemblies et Classes

Utilisation de la Reflection pour explorer des Assemblies

- La classe `Assembly` permet d'obtenir des **informations sur les assemblies chargées** dans l'application.
- Elle peut aussi être utilisée pour charger des assemblies dynamiquement.

```
using System.Reflection;

Assembly assembly = Assembly.LoadFrom("MyLibrary.dll");
Console.WriteLine(assembly.FullName); // Affiche le nom de l'assembly
Assembly exAssembly = Assembly.GetExecutingAssembly();
Console.WriteLine(exAssembly.FullName);
```

Introspection des Assemblies et Classes

Accéder aux types d'une assembly

- La Reflection permet d'accéder à tous les types définis dans une assembly et d'explorer leurs membres (méthodes, propriétés, etc.).

```
Type myType = assembly.GetType("MyNamespace.MyClass");  
MethodInfo method = myType.GetMethod("MyMethod");  
Console.WriteLine(method.Name); // Affiche le nom de la méthode  
foreach (Type type in assembly.GetTypes())  
{  
    Console.WriteLine($"Type: {type.FullName}");  
}
```

Exemples d'Introspection d'une classe

```
class Class { }

internal class Program
{
    public int MyProperty { get; set; }
    public static int Add(int a, int b) { return a + b; }
    private static void Main(string[] args)
    {
        var assembly = Assembly.GetExecutingAssembly();
        foreach (Type type in assembly.GetTypes())
        {
            Console.WriteLine($"Type: {type.FullName}");
            foreach (MethodInfo method in type.GetMethods())
            {
                Console.WriteLine($"- {method.Name} : {method.ReturnType} <= ( "
                    + string.Concat(method.GetParameters().Select(p => $"{p.ParameterType} {p.Name}, ") + ")");
            }
        }

        Type type1 = assembly.GetType("Program");
        object instance = Activator.CreateInstance(type1);
        MethodInfo method1 = type1.GetMethod("Add");
        object result = method1.Invoke(instance, new object[] { 1, 2 });
        Console.WriteLine("Résultat : " + result);
    }
}
```

Qu'est-ce qu'un **Attribute** ?

- Un **attribute** (attribut en français) permet de définir des **métadonnées** associées à des **éléments de programme** (classes, méthodes, propriétés, etc.).
- Les attributes **ne modifient pas le comportement** du programme mais servent à **fournir des informations supplémentaires** ou à **interagir** avec le code **de manière déclarative**.

Exemples d'Attributes en C#

- **[Obsolete]** : Marque un élément comme obsolète.
- **[Serializable]** : Indique qu'un type peut être sérialisé.
- **[Table("ma_table")]** : Utilisé dans Entity Framework code.
- **[HttpGet]** : Dans ASP.NET Core pour définir les actions HTTP.
- **[Required]** : Pour la **Validation de données (Data Annotation)**

```
[Obsolete("Cette méthode est obsolète.")]  
public void OldMethod() {  
    // Code...  
}
```

Définir des Attributes personnalisés

Vous pouvez définir vos propres attributes en créant des classes dérivées de `Attribute`.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class MyCustomAttribute : Attribute {
    public string Description { get; }
    public MyCustomAttribute(string description) {
        Description = description;
    }
}

[MyCustom("This is a custom attribute")]
public class MyClass {
    // Code...
}
```


Paramètres des Attributes

- Les attributes peuvent **accepter des arguments** pour fournir des **informations supplémentaires** lors de leur utilisation et pour **configurer un comportement spécifique** dans votre code.

```
[AttributeUsage(AttributeTargets.Method)]
public class LogExecutionTimeAttribute : Attribute {
    public bool Log { get; }
    public LogExecutionTimeAttribute(bool log) {
        Log = log;
    }
}

// Application de l'attribut avec un paramètre
[LogExecutionTime(true)]
public void SomeMethod() {
    // Code à exécuter
}
```

Accéder aux paramètres d'un Attribute via Reflection

- La Reflection peut être utilisée pour lire les valeurs des attributs appliqués à des éléments de code à l'exécution.

```
var method = typeof(MyClass).GetMethod("SomeMethod");  
var attribute = (LogExecutionTimeAttribute)Attribute.GetCustomAttribute(method, typeof(LogExecutionTimeAttribute));  
Console.WriteLine(attribute.Log); // Affiche la valeur du paramètre
```

Génération de Code : Emitters

- Les **emitters** permettent de **générer du code dynamique à l'exécution**, ce qui peut être utile pour des situations comme la création de code à la volée pour des expressions ou des classes spécifiques.

Utilisation de `System.Reflection.Emit`

- Le namespace `System.Reflection.Emit` fournit des outils pour **générer** des **assemblies**, des **types** et des **méthodes dynamiquement**.
- Cela peut être utilisé pour des optimisations de performances ou pour la création de code générique dans des applications avancées.
- **Création dynamique d'assemblies, types et méthodes en utilisant `AssemblyBuilder`**
- Dans .NET Core et les versions récentes de .NET, pour créer des **assembly dynamiques**, on utilise `AssemblyBuilder` directement, sans passer par `AppDomain`.

Exemple : Génération de HelloWorld.SayHello()

```
// Créer un assembly dynamique
var assemblyName = new AssemblyName("DynamicAssembly");
var assemblyBuilder = AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
var moduleBuilder = assemblyBuilder.DefineDynamicModule("MainModule");
var typeBuilder = moduleBuilder.DefineType("HelloWorld", TypeAttributes.Public);

// Créer une méthode
var methodBuilder = typeBuilder.DefineMethod("SayHello", MethodAttributes.Public, typeof(void), Type.EmptyTypes);
var ilGenerator = methodBuilder.GetILGenerator(); // générateur de CIL
ilGenerator.EmitWriteLine("Hello, World!");
ilGenerator.Emit(OpCodes.Ret);

// Générer le type et invoquer la méthode
var helloWorldType = typeBuilder.CreateType();
var helloWorldInstance = Activator.CreateInstance(helloWorldType);
var sayHelloMethod = helloWorldType.GetMethod("SayHello")!;
sayHelloMethod.Invoke(helloWorldInstance, null);

foreach (Type type in assemblyBuilder.GetTypes())
{
    Console.WriteLine($"Type: {type.FullName}");
    foreach (MethodInfo method in type.GetMethods())
    {
        Console.WriteLine($"- {method.Name} : {method.ReturnType} <= ( "
            + string.Concat(method.GetParameters().Select(p => $"{p.ParameterType} {p.Name}, ") + ")");
    }
}
```

Scénarios d'utilisation

- Création d'applications à la volée ou générées dynamiquement.
- Optimisation de performance en générant des types spécifiques dans des situations avancées.

LINQ (Language Integrated Query)

Rappels LINQ

- **LINQ (Language Integrated Query)** est un langage de requête intégré au C# qui permet de **manipuler les collections de données** de manière **expressive** et **déclarative**.
- Il est composé de **méthodes d'extension** qui offrent des **opérations** de **filtrage**, de **projection**, de **tri** et de **regroupement**.

Syntaxe de LINQ

LINQ fournit deux syntaxes :

- la **syntaxe de requête** (type SQL-like)

```
// Syntaxe de requête  
var evenNumbers = from num in numbers where num % 2 == 0 select num;
```

- la **syntaxe des méthodes** (utilisation de lambda expressions).

```
// Syntaxe des méthodes  
var evenNumbers = numbers.Where(num => num % 2 == 0);
```

Types de LINQ

- **LINQ to Objects** : Travaille sur les **collections en mémoire** comme les listes et les tableaux.
- **LINQ to SQL / LINQ to Entities (EF Core)** : Travaille sur des **sources de données externes**, comme des **bases de données**, en traduisant les requêtes en expressions SQL.
- **LINQ to XML** : Pour interroger les **documents XML**.

Les interfaces principales de LINQ

- **IEnumerable** : Utilisé pour **LINQ to Objects**. Les opérations sont réalisées en mémoire.
- **IQueryable** : Utilisé pour **LINQ to SQL/Entities**. Permet d'exécuter des requêtes à distance et de les transformer dynamiquement.

Travailler toujours avec `IEnumerable<>`

Il est conseillé de **garder** `IEnumerable<>` pour les **variables résultats** le **plus longtemps possible** avant de transformer en `List<>`.

1. **Exécution différée** : la requête n'est exécutée qu'au moment où les données sont vraiment nécessaires.
2. **Optimisation** : avec `IQueryable`, les filtres sont appliqués en base de données, évitant de charger des données inutiles.
3. **Économie de mémoire** : limite le chargement en mémoire jusqu'au dernier moment.
4. **Composition dynamique** : facilite l'ajout de conditions sans exécuter la requête.

Expression Trees

- Les requêtes `IQueryable` sont converties en **arbres d'expressions** (Expression Trees), qui peuvent être analysés, modifiés, ou traduits en SQL pour une exécution en base de données.

```
IQueryable<int> query = dbContext.Numbers.Where(n => n % 2 == 0);  
Expression expressionTree = query.Expression; // Arbre d'expression
```

Fonctionnalités avancées de LINQ

- **Opérations de projection avancées**

`SelectMany` pour aplatir des collections imbriquées.

Utilisé pour les jointures et pour extraire des informations de collections complexes.

```
var result = authors.SelectMany(author => author.Books);
```

- **Méthodes de regroupement et d'agrégation**

`GroupBy`, `Sum`, `Average`, `Count` pour des opérations de regroupement et de calcul.

```
var averagePrice = products.GroupBy(p => p.Category)
    .Select(g => new { Category = g.Key, AvgPrice = g.Average(p => p.Price) });
```

Fonctionnalités avancées de LINQ

- **Gestion des jointures**

Les méthodes `Join` et `GroupJoin` permettent de réaliser des jointures de tables et d'assembler les données.

```
var query = customers.Join(orders,  
    customer => customer.Id,  
    order => order.CustomerId,  
    (customer, order) => new { customer.Name, order.OrderDate });
```

Créer son propre provider **IQueryable**

Pourquoi créer un provider personnalisé ?

- Un provider **IQueryable** personnalisé est utile pour intégrer LINQ à des **sources de données spécifiques non supportées nativement**, comme une API, des fichiers, ou des systèmes externes.

Créer son propre provider `IQueryable`

Étapes pour implémenter un provider `IQueryable` personnalisé

1. Définir un provider personnalisé (`CustomQueryProvider`) qui implémente `IQueryProvider`.
2. Créer une classe pour représenter la requête qui implémente `IQueryable`.
3. Analyser et traduire l'arbre d'expression : Interpréter les expressions pour appliquer la logique de requête souhaitée.
4. Exécuter la requête et obtenir les résultats.

Exemple : Provider IQueryable pour API RESTful

```
public class ApiQueryProvider : IQueryProvider
{
    private readonly HttpClient _httpClient;
    private readonly string _baseUrl;

    public ApiQueryProvider(HttpClient httpClient, string baseUrl)
    {
        _httpClient = httpClient ?? throw new ArgumentNullException(nameof(httpClient));
        _baseUrl = baseUrl ?? throw new ArgumentNullException(nameof(baseUrl));
    }

    public IQueryable CreateQuery(Expression expression)
    {
        Type elementType = expression.Type.GetGenericArguments().First();
        var queryableType = typeof(ApiQueryable<>).MakeGenericType(elementType);
        return (IQueryable)Activator.CreateInstance(queryableType, this, expression);
    }

    public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
    {
        return new ApiQueryable<TElement>(this, expression);
    }

    public object Execute(Expression expression)
    {
        return ExecuteAsync<object>(expression).GetAwaiter().GetResult();
    }

    public TResult Execute<TResult>(Expression expression)
    {
        return ExecuteAsync<TResult>(expression).GetAwaiter().GetResult();
    }

    private async Task<TResult> ExecuteAsync<TResult>(Expression expression)
    {
        // Convertit l'expression en URI pour une requête RESTful
        string requestUri = ExpressionToUriConverter.Convert(expression, _baseUrl);

        // Effectuer la requête HTTP
        HttpResponseMessage response = await _httpClient.GetAsync(requestUri);
        response.EnsureSuccessStatusCode();

        // Analyser la réponse JSON
        string json = await response.Content.ReadAsStringAsync();
        return System.Text.Json.JsonSerializer.Deserialize<TResult>(json);
    }
}
```

```
public class ApiQueryable<T> : IQueryable<T>
{
    private readonly Expression _expression;
    private readonly IQueryProvider _provider;

    public ApiQueryable(IQueryProvider provider, Expression expression)
    {
        _provider = provider ?? throw new ArgumentNullException(nameof(provider));
        _expression = expression ?? throw new ArgumentNullException(nameof(expression));
    }

    public Type ElementType => typeof(T);

    public Expression Expression => _expression;

    public IQueryProvider Provider => _provider;

    public IEnumerator<T> GetEnumerator()
    {
        // Appelle le provider pour exécuter la requête
        var result = _provider.Execute<IEnumerable<T>>(_expression);
        return result.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

```
var httpClient = new HttpClient();
var provider = new ApiQueryProvider(httpClient, "https://api.example.com/items");
var queryable = new ApiQueryable<MyEntity>(provider, Expression.Constant(""));

var results = queryable.Where(x => x.Name == "example").ToList();
```

```
public static class ExpressionToUriConverter
{
    public static string Convert(Expression expression, string baseUrl)
    {
        var visitor = new QueryExpressionVisitor();
        visitor.Visit(expression);
        return $"{baseUrl}?{visitor.QueryString}";
    }

    private class QueryExpressionVisitor : ExpressionVisitor
    {
        private readonly StringBuilder _queryString = new();
        public string QueryString => _queryString.ToString();

        protected override Expression VisitMethodCall(MethodCallExpression node)
        {
            if (node.Method.Name == "Where")
            {
                // Traite l'expression Where
                var lambda = (LambdaExpression)StripQuotes(node.Arguments[1]);
                var binaryExpression = (BinaryExpression)lambda.Body;
                if (binaryExpression != null)
                {
                    string propertyName = ((MemberExpression)binaryExpression.Left).Member.Name;
                    string value = ((ConstantExpression)binaryExpression.Right).Value.ToString();
                    _queryString.Append($"{propertyName}={value}");
                }
            }
            return base.VisitMethodCall(node);
        }

        private static Expression StripQuotes(Expression e)
        {
            while (e.NodeType == ExpressionType.Quote)
            {
                e = ((UnaryExpression)e).Operand;
            }
            return e;
        }
    }
}
```

Cet exemple illustre la complexité d'implémentation de cette notion avancée...

Performances sur l'utilisation de LINQ

Opérations différées et immédiates

- Les requêtes LINQ sont **différées** par défaut : elles ne sont **évaluées que lorsque les résultats sont nécessaires** (`ToList()`, `Count()`, etc.).
- **Attention** : L'**utilisation excessive** de `ToList()` peut entraîner des évaluations prématurées et **impacter les performances**.

Performances sur l'utilisation de LINQ

- **Limiter les opérations en mémoire** : Préférer les méthodes `IQueryable` comme `Where` avant d'appeler `ToList`.
- **Éviter les sous-requêtes inutiles** : Réduire le nombre d'appels `Select` imbriqués ou `GroupBy` qui peuvent alourdir les requêtes.

Performances sur l'utilisation de LINQ

- Utiliser **AsEnumerable** pour basculer entre **IQueryable** et **IEnumerable** : Cela peut aider à optimiser des parties spécifiques de la requête en mémoire (**Linq to SQL -> Linq to Object**).
 - Quand vous utilisez des méthodes **non traduisibles en SQL**, trop **complexes** ou **non supporté par le serveur SQL** dans une requête LINQ.

```
var optimizedQuery = dbContext.Products
    .Where(p => p.Price > 100)    // IQueryable
    .AsEnumerable()
    .Where(p => p.IsAvailable);  // IEnumerable
```

Interopérabilité

Interopérabilité en C#

- Permet d'**appeler** du **code natif** (C, C++) ou **COM** (Component Object Model) **depuis une application .NET**.
- Utilisée pour **intégrer** des **fonctionnalités non gérées** ou **accéder** à des **bibliothèques systèmes**.

Méthodes d'interopérabilité

- **P/Invoke (Platform Invocation)** : Appel direct de fonctions natives.
- **Interop COM** : Communication avec des composants COM (Component Object Model).

P/Invoke (Platform Invocation)

- P/Invoke permet d'appeler des **fonctions natives** exportées par des **bibliothèques (DLLs)**.
- Typiquement utilisé pour **accéder** aux **API Windows** ou aux **bibliothèques natives écrites en C/C++**.

P/Invoke (Platform Invocation)

- Utilisation de l'attribut `[DllImport]` pour indiquer la bibliothèque externe et la méthode à appeler.
- Il est crucial d'utiliser des **types compatibles** entre C# et le code natif.

```
using System.Runtime.InteropServices;

public class NativeMethods
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr hWnd, string text, string caption, uint type);
}

// Appel de la fonction native
NativeMethods.MessageBox(nint.Zero, "Hello, world!", "New Window", 0);
```

Component Object Model

- Le **Component Object Model (COM)** est un modèle permettant la création de **composants logiciels interopérables**, indépendants du langage de programmation.
- Il repose sur des **interfaces** pour définir les interactions et utilise un système de **références comptées** pour la gestion de la mémoire (compteur de références permettant la suppression après utilisation des objets).

Interopérabilité COM (COM Interop)

- Permet d'utiliser des **composants COM** depuis .NET.
- Utilisé pour exploiter des applications et services COM, comme Microsoft Office.
- Pour l'utiliser : **Ajouter une référence COM** dans le projet C# (génère des **Wrappers COM**).

COM Wrappers

Les **wrappers COM** sont des classes de proxy générées pour encapsuler les objets COM et les adapter à .NET.

- **RCW (Runtime Callable Wrapper)** : Généré automatiquement pour interagir avec COM à partir de .NET.
- **CCW (COM Callable Wrapper)** : Pour exposer des objets .NET comme objets COM.

COM Wrappers

Exemple de RCW

En référence à une bibliothèque COM dans .NET, Visual Studio génère un RCW.

```
// Exemple d'utilisation d'un RCW généré pour un composant COM  
var comObject = new SomeCOMLibrary.SomeCOMClass();  
comObject.SomeMethod();
```

Gestion de la mémoire

Utiliser `Marshal.ReleaseComObject` pour libérer manuellement les objets COM et éviter les fuites mémoire.

```
Marshal.ReleaseComObject(comObject);
```

Exemple COM Interop avec Excel

```
// Ajouter la référence COM : Microsoft Excel 16.0 Object Library 1.9
using Microsoft.Office.Interop.Excel;

var excelApp = new Application(); // COM Wrapper
excelApp.Visible = true;
var workbook = excelApp.Workbooks.Add(); // COM Wrapper
var sheet = (Worksheet)workbook.Sheets[1]; // COM Wrapper
sheet.Cells[1, 1] = "Hello Excel from .NET 8!";

Thread.Sleep(3000);

// /\ Libérer les ressources COM (pas de Garbage Collector)
Marshal.ReleaseComObject(sheet);
Marshal.ReleaseComObject(workbook);

// Fermer Excel et libérer l'objet Excel
excelApp.Quit();
Marshal.ReleaseComObject(excelApp);
```

Passage de paramètres

Appels vers du code non géré

- Les appels P/Invoke et COM nécessitent une **gestion spécifique des types** pour assurer la **compatibilité**.
- **Types standards** : Types primitifs comme `int`, `float`, `bool` sont généralement compatibles.
- **Types complexes** : Struct, tableaux, chaînes de caractères nécessitent une attention particulière.

Passage de paramètres

Marshalling des paramètres

- Le **marshalling** transforme les types de données entre le code géré (C#) et le code non géré.
- **Attribut [MarshalAs]** : Contrôle la manière dont les types sont marshallés.

```
[DllImport("user32.dll", CharSet = CharSet.Unicode)]  
public static extern int MessageBox(IntPtr hWnd,  
                                   [MarshalAs(UnmanagedType.LPWStr)] string text,  
                                   string caption, uint type);
```

LPWStr = 32-bit pointer to a string of 16-bit Unicode characters

https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/50e9ef83-d6fd-4e22-a34a-2c6b4e3c24f3

Passage de paramètres

Exemple de structures

- Les structures doivent être explicitement marshallées pour garantir la correspondance des champs.

```
[StructLayout(LayoutKind.Sequential)]  
public struct MyStruct {  
    public int intValue;  
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)]  
    public string stringValue;  
}
```

Fonctionnement avancé du Runtime

Dynamic Language Runtime (DLR)

- Le **DLR** est **une couche au-dessus** du **CLR** (Common Language Runtime) qui permet l'**exécution de langages dynamiques**.
- Supporte des langages comme Python ([IronPython](#)) ou Ruby ([IronRuby](#)) **via des types dynamiques**.
- **Objet `dynamic`** : Permet d'**utiliser des types dynamiques en C#** où le **type** d'un objet est **résolu au moment de l'exécution**.

```
using System.Dynamic;

dynamic expando = new ExpandoObject();
expando.Name = "Alice";
Console.WriteLine(expando);
Console.WriteLine(expando.Name);
```

Avantages du DLR

- **Interopérabilité** avec des langages dynamiques.
- **Flexibilité** pour les opérations où le type n'est pas connu au moment de la compilation.
- Utilisation d'**arbres d'expressions** et de **caches de sites d'appels** pour optimiser l'accès aux types dynamiques.

Task Parallel Library (TPL)

- La **TPL (Task Parallel Library)** est une **bibliothèque** qui **simplifie** la **parallélisation** et le **multithreading** en C#.
- Basée sur le concept de **Task** pour représenter des opérations asynchrones et parallèles.

Principales méthodes de TPL

- **Task.Run** : Exécute une tâche en parallèle.
- **Task.Wait** et **Task.WaitAll** : Synchronise les tâches.
- **async / await** : Simplifie l'utilisation des tâches asynchrones.

```
Task task = Task.Run(() => { /* Code à exécuter en parallèle */ });  
task.Wait(); // Attend que la tâche soit terminée
```

Fonctionnalités avancées de TPL

- **Parallel.For** et **Parallel.ForEach** : Exécution de boucles en parallèle.

```
Parallel.For(1, 10000, x => Console.WriteLine(x));
```

- **CancellationToken** : Permet d'annuler des tâches en cours.
- **Task Continuations** : Chaînage des tâches pour créer des séquences d'opérations dépendantes.

```
await Task.Run(() => 42)  
    .ContinueWith(t => Console.WriteLine($"Résultat : {t.Result}"));
```


Garbage Collector

Algorithme de nettoyage du Garbage Collector

Le Garbage Collector (GC) est un **gestionnaire de mémoire automatique** en .NET.

- Il utilise un **algorithme de génération** pour optimiser la collecte des objets, il les sépare en 3:
 - **Génération 0** : objets de courte durée
 - **Génération 1** : objets de moyenne durée
 - **Génération 2** : objets de longue durée
- Dans .NET 8, le GC améliore cette approche en optimisant les collectes de fond et la gestion concurrente, réduisant ainsi les pauses et améliorant la performance des applications.

Algorithme de nettoyage du Garbage Collector

- **Processus de collecte**

- Le GC libère la mémoire des objets non référencés.
- La collecte est déclenchée automatiquement en cas de besoin, mais peut aussi être forcée via `GC.Collect()` (pas recommandé).

```
GC.Collect(); // Forcer la collecte  
GC.WaitForPendingFinalizers(); // Attend la fin de la collecte
```

Ressources managées vs non managées

- **Ressources managées**

- Ressources **gérées par le CLR** (ex. objets .NET).
- **Libérées automatiquement** par le Garbage Collector.

- **Ressources non managées**

- Ressources **externes au CLR** (ex. connexions de base de données, handles de fichiers, objets COM).
- Nécessitent une **libération manuelle** pour éviter les fuites de mémoire.

Bien libérer les ressources

- **Pourquoi libérer les ressources ?**
 - **Libérer** les ressources non managées permet d'**optimiser la mémoire** et d'**éviter des fuites potentielles**.
 - Important pour les **applications** qui **manipulent** des **ressources externes** ou du **code natif**.

Bien libérer les ressources

- La méthode `Dispose()` permet de **libérer explicitement** les **ressources non managées**.
- Appelée directement ou via une instruction `using`.

```
var resource = new SomeResource();  
// Utiliser la ressource  
resource.Dispose();  
// Libérée manuellement
```

```
using (var resource = new SomeResource())  
{  
    // Utiliser la ressource  
}  
// Libérée automatiquement
```

```
{  
    // ...  
    using var resource = new SomeResource();  
    // Utiliser la ressource  
} // Libérée automatiquement
```

Pattern IDisposable

- Le pattern IDisposable est utilisé pour **libérer correctement les ressources**.
- Il convient d'utiliser un **finalizer** (méthode `~ClassName`) pour **garantir la libération** des ressources non managées **en cas d'oubli** de `Dispose`.

```
public class ResourceHolder : IDisposable
{
    private bool disposed = false; // Indiquer si l'objet a déjà été disposé

    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this); // empêche l'appel du finalizer
    }

    // true = appel manuel, false = appel par finalizer
    protected virtual void Dispose(bool disposing) {
        if (!_disposed)
            return;
        if (disposing)
        {
            // Libérer les ressources managées
        }
        // Libérer les ressources non managées
        _disposed = true;
    }

    // Appel du finalizer pour libérer les ressources en cas d'oubli
    ~FileHandler()
    {
        Dispose(disposing: false);
    }
}
```

Exemple complet IDisposable

```
public class FileHandler : IDisposable, IAsyncDisposable
{
    private FileStream? _fileStream;
    private bool _disposed = false;

    public FileHandler(string filePath)
    {
        _fileStream = new FileStream(filePath, FileMode.OpenOrCreate,
                                     FileAccess.ReadWrite);
    }

    public void WriteData(string data)
    {
        if (_fileStream == null)
            throw new ObjectDisposedException(nameof(FileHandler));
        using var writer = new StreamWriter(_fileStream, leaveOpen: true);
        writer.WriteLine(data);
        writer.Flush();
    }

    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }

    public async ValueTask DisposeAsync()
    {
        await DisposeAsync(disposing: true);
        GC.SuppressFinalize(this);
    }
}
```

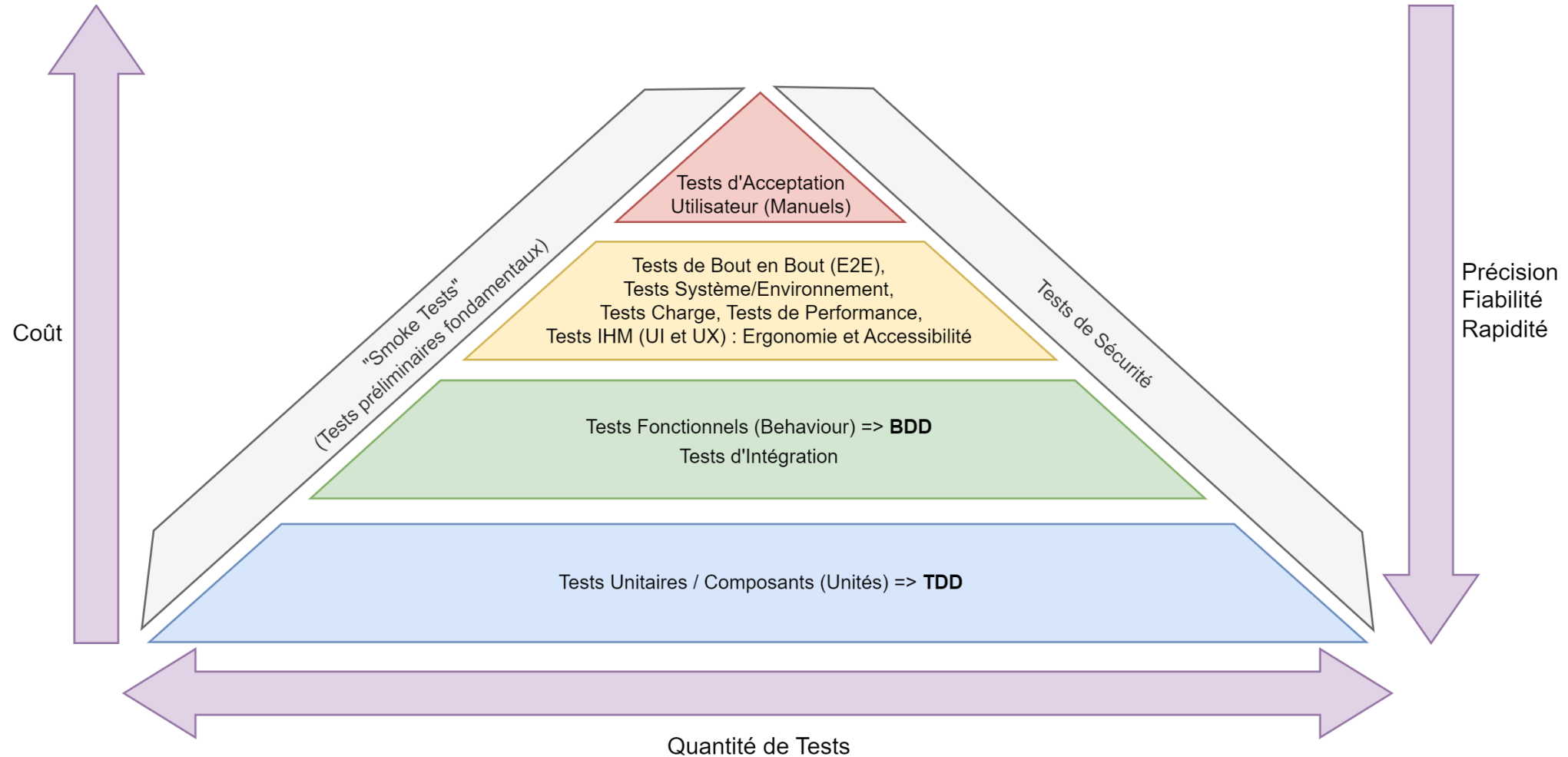
```
protected virtual void Dispose(bool disposing)
{
    if (_disposed)
        return;
    if (disposing)
        _fileStream?.Dispose();
    _disposed = true;
}

protected virtual async ValueTask DisposeAsync(bool disposing)
{
    if (_disposed)
        return;
    if (disposing)
        if (_fileStream != null)
            await _fileStream.DisposeAsync();
    _disposed = true;
}

~FileHandler()
{
    Dispose(disposing: false);
}
}
```


Tests Unitaires en C#

Différents types de Tests / Pyramide des tests



Pourquoi les tests unitaires ?

- Garantissent la **fiabilité** et la **maintenabilité** du code.
- Détectent rapidement les **régressions** et **erreurs**.
- Favorisent une **meilleure compréhension** et **documentation** du code.

Frameworks de tests .NET

- Il existe en .NET 3 frameworks principaux pour les tests Unitaires :
 - **MSTest**
 - **NUnit**
 - **XUnit**
- Pour faire des tests unitaires, on **créera un projet** de type "**Bibliothèque de Tests**", il fera **référence au projet C#** que l'on veut tester.

MSTest

```
[TestClass]
public class MyTests {
    [TestMethod]
    public void TestMethod1() {
        Assert.AreEqual(4, 2 + 2);
    }
}
```

NUnit

```
[TestFixture]
public class MyTests {
    [Test]
    public void TestMethod1() {
        Assert.That(4, Is.EqualTo(2 + 2));
    }
}
```

XUnit

```
public class MyTests {  
    [Fact]  
    public void TestMethod1() {  
        Assert.Equal(4, 2 + 2);  
    }  
}
```

Comparatif syntaxique

Aspect	xUnit	NUnit	MSTest
Test simple	[Fact]	[Test]	[TestMethod]
Test paramétré	[InlineData]	[TestCase]	[DataRow]
Initialisation globale	Non disponible	[OneTimeSetUp]	[ClassInitialize]
Nettoyage global	Non disponible	[OneTimeTearDown]	[ClassCleanup]
Initialisation avant chaque test	Non disponible	[SetUp]	[TestInitialize]
Nettoyage après chaque test	Non disponible	[TearDown]	[TestCleanup]

Comparatif syntaxique

Aspect	xUnit	NUnit	MSTest
Ignorer un test	<code>[Fact(Skip="raison")]</code>	<code>[Ignore("raison")]</code>	<code>[Ignore]</code>
Tests paramétrés avancés	<code>[MemberData]</code>	<code>[TestCaseSource]</code>	<code>[DynamicData]</code>
Ordre d'exécution	<code>[TestCaseOrderer]</code>	<code>[Order]</code>	Non disponible
Catégories de tests	Non disponible	<code>[Category("nom")]</code>	<code>[TestCategory("nom")]</code>
Timeout pour un test	Non disponible directement	<code>[Timeout(ms)]</code>	<code>[Timeout(ms)]</code>

Comment bien écrire un test unitaire ?

- **Caractéristiques d'un bon test unitaire**
 - **Indépendant** : Un test ne doit **pas dépendre des autres**.
 - **Précis** : Chaque test **vérifie un cas spécifique**.
 - **Automatisé** : **Exécuté facilement** dans une suite de tests.
 - **Rapide** : Pour permettre des **itérations rapides**.
 - **Nom explicite** : Le **nom** du test doit **indiquer clairement son objectif**. => [Conventions de nommage](#)
- [Recommandations Microsoft](#)

AAA (Arrange, Act, Assert)

Il convient de décomposer un test en 3 phases :

- **Arrange** : **Prépare** les données nécessaires au test.
- **Act** : **Exécute** la logique à tester.
- **Assert** : **Vérifie** le résultat attendu.

```
[TestMethod]
public void CalculateTotal_ShouldReturnCorrectValue() {
    // Arrange
    var calculator = new Calculator();
    int expected = 4;

    // Act
    int result = calculator.CalculateTotal(2, 2);

    // Assert
    Assert.AreEqual(expected, result);
}
```

Convaincre les développeurs de l'utilité des tests unitaires

Arguments en faveur des tests unitaires

- **Détection précoce des erreurs** : Moins de bugs en production, ce qui réduit les coûts, surtout avec l'utilisation de CI/CD.
- **Confiance dans les modifications** : Les tests assurent la non-régression.
- **Documentation vivante** : Les tests montrent des exemples d'utilisation du code.
- **Maintenance facilitée** : Les tests facilitent les refactorisations et améliorent la qualité du code.

Convaincre les développeurs de l'utilité des tests unitaires

Démontrer l'efficacité

- Partager des exemples concrets de bugs évités grâce aux tests.
- Montrer comment les tests peuvent accélérer le développement sur le long terme.
- Encourager une culture de tests via des révisions de code et des sessions de pair programming.

Mocking

- Permet de créer des **objets simulés** pour **tester sans les dépendances**.
- Facilite les tests en isolant **la logique métier des dépendances externes** (autre classe, base de données, API, etc.).
- **Essentiels** pour **se concentrer** sur **un aspect** (une unité) du code à **la fois** et ainsi **respecter** le principe de Test **Unitaire**.

Les Frameworks de Mocking

- **Moq** : Simple et flexible, souvent utilisé pour les interfaces et les services.
- **NSubstitute** : Syntaxe intuitive, permet de créer des substituts dynamiques.
- **FakeItEasy** : Design minimaliste et syntaxe expressive.

Exemple de Moq

```
var mockRepository = new Mock<IRepository>();  
mockRepository.Setup(repo => repo.GetData()).Returns("Mocked Data");  
  
var service = new MyService(mockRepository.Object);  
string result = service.GetData();  
  
Assert.AreEqual("Mocked Data", result);
```


Principes de Mocking

- **Stubbing** : Créer des comportements spécifiques pour des méthodes.
- **Verification** : Valider que certaines méthodes ont été appelées.
- **Isolation des tests** : Permet de tester une unité de code en se concentrant uniquement sur sa logique.

TDD

Les paradigmes du TDD

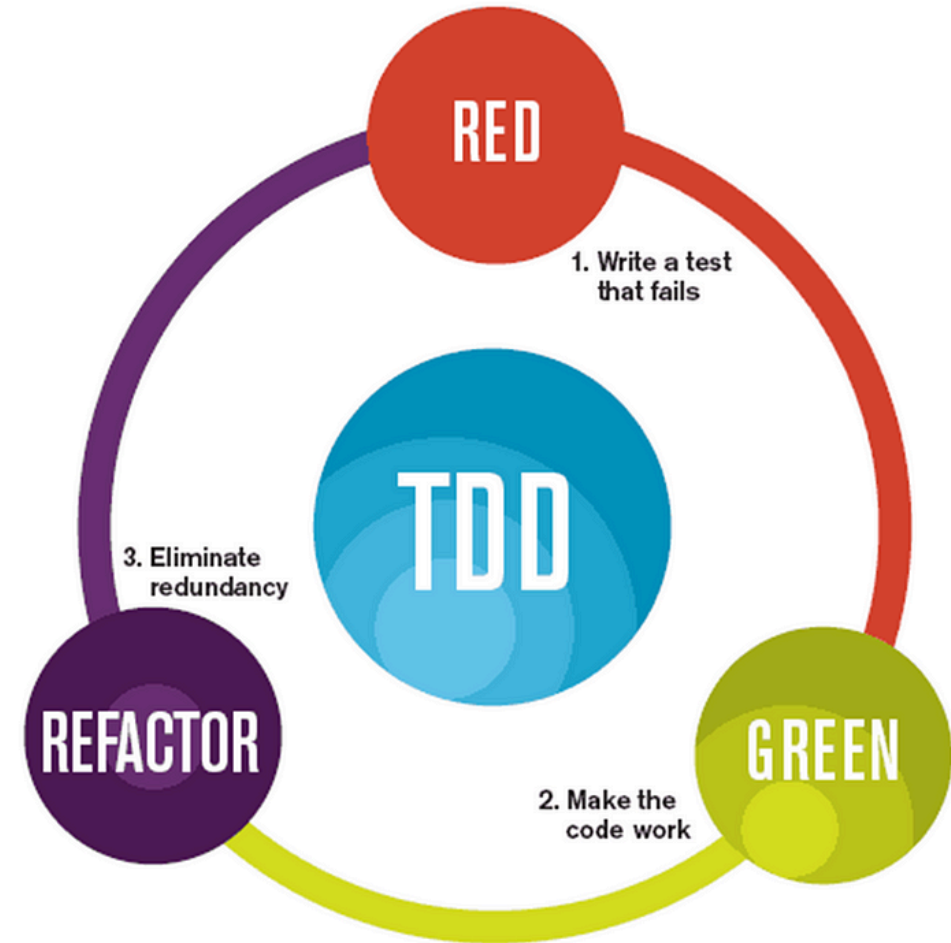
Le **Test Driven Development (TDD)**, ou Développement Dirigé par les Tests, consiste à écrire des tests avant le code de production.

En .NET, le processus suit généralement ces étapes :

- Red
- Green
- Refactor

Red Green Refactor

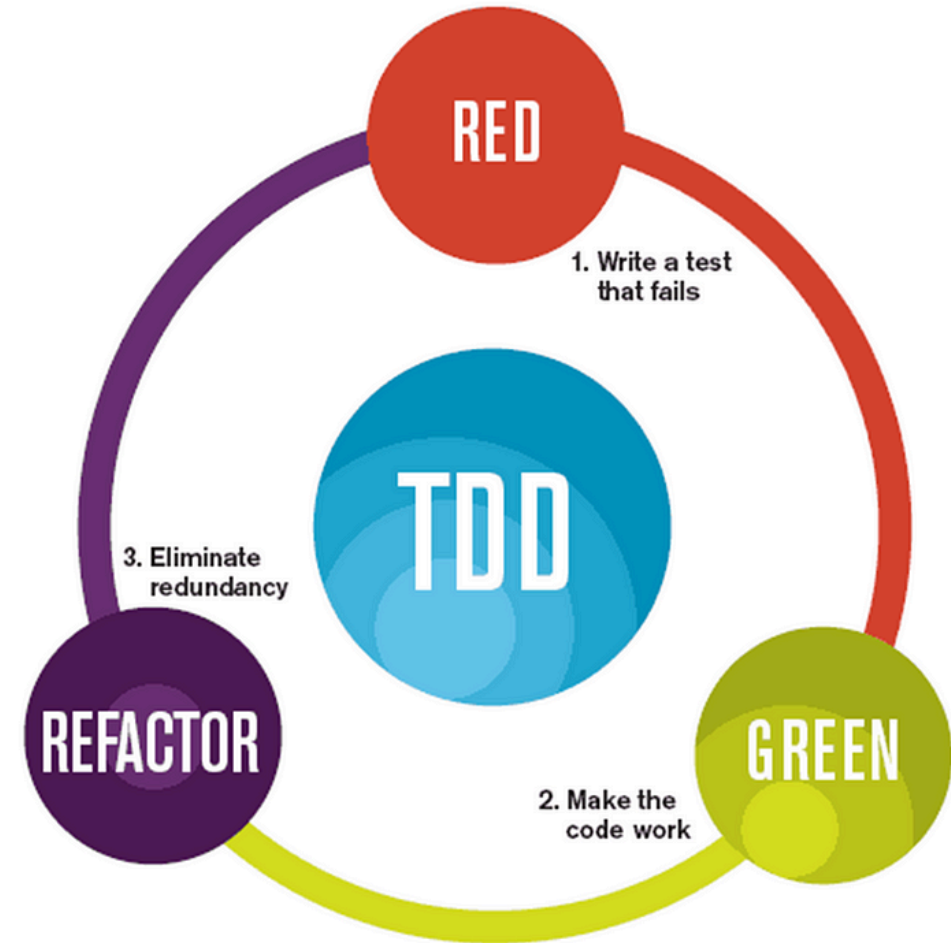
- **Écrire un test** : Rédigez un test pour la fonctionnalité cible. Ce test échoue initialement, car le code de production est inexistant.
- **Exécuter le test pour vérifier son échec** : Cela confirme que le test est valide et qu'il vérifie la bonne condition.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Red Green Refactor

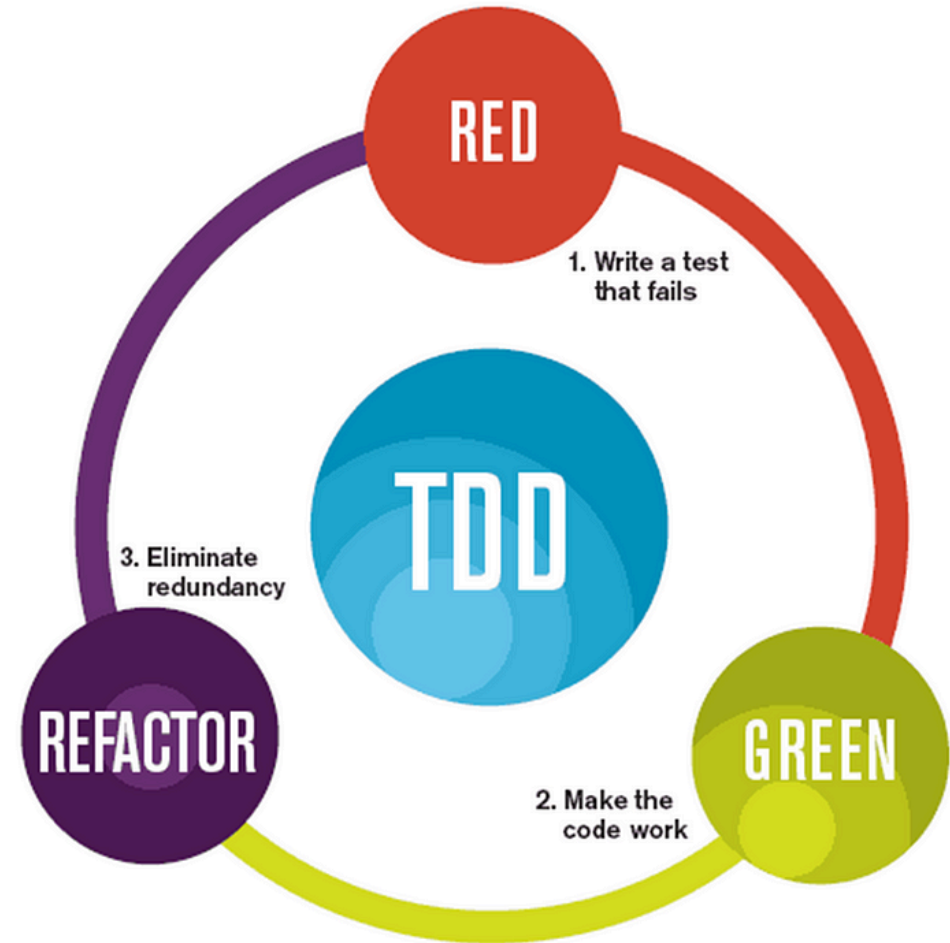
- **Écrire le code de production** : Écrivez juste assez de code pour que le test réussisse, sans chercher à optimiser.
- **Exécuter les tests** : Vérifiez que le nouveau test passe et que les anciens tests restent valides.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Red Green Refactor

- **Refactoriser le code** :
Améliorez la structure du code tout en conservant ses fonctionnalités. Après chaque refactorisation, exécutez les tests.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les bonnes pratiques du TDD

- **Comprendre les exigences** : Assurez-vous de bien comprendre ce que le code doit accomplir avant de commencer.
- **Tests simples** : Chaque test doit être indépendant et vérifier une seule fonctionnalité.
- **Éviter les anticipations** : Restez concentré sur les besoins actuels, sans coder pour des besoins futurs (principe **YAGNI**).
- **Code minimum** : Écrivez uniquement le code nécessaire pour que le test passe.

Les bonnes pratiques du TDD

- **Refactoriser régulièrement** : Simplifiez et améliorez le code une fois les tests validés, puis réexécutez-les.
- **Automatiser les tests** : Exécutez-les automatiquement à chaque modification pour détecter rapidement les problèmes.
- **Utiliser des doubles de test** : Simulez les dépendances avec des mocks, stubs ou objets factices pour isoler le code testé.
- **Exécution régulière** : Lancez les tests fréquemment pour maintenir leur efficacité.

Les bonnes pratiques du TDD

- **Bonne couverture** : Testez autant de parties du code que possible, sans viser systématiquement 100%.
- **Tester à différents niveaux** : Incluez tests unitaires, d'intégration, système, et d'acceptation.

Principe FIRST

F - Fast (Rapide) : Les tests doivent s'exécuter rapidement pour être utilisés fréquemment.

I - Independent (Indépendant) : Aucun test ne doit dépendre d'un autre.

R - Repeatable (Reproductible) : Les tests doivent fournir les mêmes résultats, quel que soit l'environnement.

S - Self-validating (Auto-vérifiable) : Les résultats doivent être clairs (succès/échec), sans analyse manuelle.

T - Timely (Opportun) : Les tests doivent être écrits en parallèle ou avant le code de production.

Merci pour votre attention

