Bienvenue au cours “C,C#,.NET Intermediaire”

Nom de l’enseignant : Fabrice Meynckens

Promo : 2025-2026

Classe : B3CDEV

Objectifs de la session

- mécanismes get;set
- héritage simple
- polymorphisme dynamique
- classes abstraites
- tableaux, collections, listes
- gestion des erreurs (try, catch)

C# : mécanismes get;set

Le mécanisme de déclaration de variables privées avec les accesseurs (`get; set`) en C# est appelé **Propriétés** (`Properties`). C'est la pierre angulaire de l'**Encapsulation** en C#.

Ce mécanisme permet de manipuler les données membres d'une classe de manière sécurisée et contrôlée.

1. Principe de l'Encapsulation

L'Encapsulation est un principe de la POO qui consiste à masquer les données internes d'une classe et à n'autoriser l'accès et la modification qu'à travers une interface contrôlée.

Champs (`Fields`) Privés : Les données brutes (variables membres) sont déclarées comme `private`. Elles sont appelées **champs**. Cela empêche toute lecture ou écriture directe depuis l'extérieur de la classe.
C#

```
private int _age; // Le champ privé stocke la donnée
```

- **Propriétés Publiques** : Pour exposer ces données, on utilise des **Propriétés** (`Properties`). Une propriété agit comme une "façade" publique qui utilise des méthodes spéciales appelées **accesseurs** (`get` et `set`) pour interagir avec le champ privé sous-jacent.

C# : mécanismes get;set

Propriété Implémentée Automatiquement (Auto-Implemented Property)

Si vous n'avez **aucune logique supplémentaire** à effectuer (c'est-à-dire que le `get` retourne simplement le champ et le `set` l'affecte directement), C# offre une syntaxe simplifiée.

Le compilateur crée automatiquement et gère un **champ privé anonyme** (le *backing field*) en arrière-plan.

```
public class Produit
{
    // Le compilateur crée automatiquement un champ privé caché pour 'Nom'
    public string Nom { get; set; }

    // Propriété en lecture seule (le 'set' est omis)
    public decimal Prix { get; }
}
```

C# : mécanismes get;set

En C#, il existe deux syntaxes principales pour les propriétés : la **Propriété Complète** et la **Propriété Automatique**.

A. Propriété Complète (avec Logique de Validation)

C'est la syntaxe utilisée lorsque vous avez besoin d'ajouter une **logique métier** (validation, conversion, calculs) lors de la lecture ou de l'écriture de la donnée.

Composant - Rôle

`get` - L'**accesseur de lecture**. Il est exécuté chaque fois que la propriété est lue. Il retourne généralement la valeur du champ privé.

`set` - L'**accesseur d'écriture**. Il est exécuté chaque fois que la propriété est modifiée. Il reçoit la nouvelle valeur à affecter via le mot-clé implicite `value`.

```
public class Personne
{
    // 1. Déclaration du champ privé
    private int _age;

    // 2. Déclaration de la propriété publique
    public int Age
    {
        // Accesseur de lecture
        get
        {
            return _age;
        }

        // Accesseur d'écriture avec validation
        set
        {
            if (value >= 0 && value <= 120) // Logique de validation
            {
                _age = value; // Assigne la valeur valide au champ privé
            }
            else
            {
                throw new ArgumentException("L'âge doit être entre 0 et 1
            }
        }
    }
}
```

C# : héritage simple

L'**Héritage** est le mécanisme par lequel une classe (**classe dérivée** ou **enfant**) acquiert les champs, propriétés et méthodes d'une autre classe (**classe de base** ou **parent**).

- **Syntaxe** : En C#, l'héritage est déclaré en utilisant les deux-points (`:`).
  Contrairement à C++, C# n'autorise que l'**héritage simple** (une classe ne peut hériter que d'une seule autre classe).
  C#

```
public class Animal { /* ... */ }
public class Chien : Animal // Chien hérite de Animal
{
    // ... hérite des membres de Animal
}
```

Mot-clé `base` : Permet à la classe dérivée d'accéder aux membres de la classe de base, notamment pour appeler le **constructeur** de la classe de base :

```
public Chien(string nom) : base(nom) { /* ... */ }
```

C# : polymorphisme dynamique

Le **Polymorphisme** (du grec "plusieurs formes") est la capacité d'un objet à prendre plusieurs formes. En C#, le **polymorphisme dynamique** permet à la classe dérivée de remplacer (redéfinir) le comportement d'une méthode définie dans la classe de base.

- `virtual` **(Classe de base)** : Ce mot-clé doit être ajouté à la méthode dans la classe de base pour signaler qu'elle **peut** être redéfinie dans les classes dérivées.
- `override` **(Classe dérivée)** : Ce mot-clé doit être ajouté à la méthode dans la classe dérivée pour signaler qu'elle **redéfinit** (remplace) la méthode virtuelle de la classe de base.

```
public class Vehicule
{
    public virtual void Demarrer() // Méthode virtuelle
    {
        Console.WriteLine("Le véhicule démarre.");
    }
}

public class Moto : Vehicule
{
    public override void Demarrer() // Redéfinit le comportement de Demarrer()
    {
        Console.WriteLine("La moto démarre en vrombissant.");
    }
}
```

C# : classes abstraites

L'**Abstraction** est le processus de simplification où l'on masque les détails d'implémentation pour ne montrer que les fonctionnalités pertinentes. Une classe abstraite est une classe **incomplète** qui ne peut pas être instanciée directement.

- **`abstract class`** : Une classe déclarée avec le mot-clé `abstract`. Elle peut contenir :
  - Des méthodes concrètes (avec une implémentation).
  - Des **méthodes abstraites** (sans implémentation, seulement une signature).
- **`abstract method`** : Une méthode sans corps (pas d'accolades `{}`). Toute classe non abstraite qui hérite d'une classe abstraite **doit** obligatoirement fournir une implémentation pour toutes ses méthodes abstraites en utilisant le mot-clé **`override`**.

```
public abstract class Forme
{
    // Méthode abstraite : doit être implémentée par les enfants
    public abstract double CalculerAire();

    // Méthode concrète : peut être utilisée ou redéfinie
    public void AfficherNom() { Console.WriteLine("Ceci est une forme."); }
}
```

C# : abstraction avec interface

Une **Interface** est un contrat pur. Elle ne contient que des **signatures de membres** (méthodes, propriétés, événements) mais **aucune implémentation**.

- **Syntaxe** : Déclarée avec le mot-clé `interface`. Le nom des interfaces commence souvent par un `I` majuscule (convention).
- **Héritage** : Une classe implémente une interface en utilisant le même symbole `:` que l'héritage. Une classe peut implémenter **plusieurs interfaces** (contrairement à l'héritage de classes).
- **Contrat** : Toute classe qui implémente une interface **doit** fournir une implémentation pour **tous** ses membres.

```
public interface IDeplacable
{
    // Propriété sans implémentation
    int Vitesse { get; set; }

    // Méthode sans implémentation
    void SeDeplacer();
}

public class Voiture : IDeplacable // Implémente l'interface
{
    public int Vitesse { get; set; } // Doit implémenter les membres

    public void SeDeplacer()
    {
        Console.WriteLine($"La voiture roule à {Vitesse} km/h.");
    }
}
```

C# : exercice pratique

Créez une hiérarchie de classes pour gérer différents types de notifications dans une application.

1. **Interface `INotifiable`** : Définissez une interface qui garantit que toute notification a une méthode `Envoyer()` et une propriété `Destinataire` (string).

2. **Classe Abstraite `Notification`** : Créez une classe abstraite qui implémente `INotifiable`.
   - Implémentez la propriété `Destinataire` comme propriété automatique.
   - Déclarez une méthode abstraite `void FormaterMessage()`.
   - Implémentez la méthode `Envoyer()` (du `INotifiable`) de manière concrète : elle doit appeler `FormaterMessage()` puis afficher un message générique.

3. **Classe `NotificationEmail`** : Hérite de `Notification`.
   - Ajoutez une propriété `Sujet` (string).
   - Implémentez `FormaterMessage()` pour afficher : "Email formaté pour [Destinataire] avec Sujet : [Sujet]".

4. **Classe `NotificationSMS`** : Hérite de `Notification`.
   - Ajoutez une propriété `NumeroExpediteur` (string).
   - Implémentez `FormaterMessage()` pour afficher : "SMS formaté pour [Destinataire] depuis le numéro : [NumeroExpediteur]".

5. **Polymorphisme** : Dans le `Main`, créez une liste de type `INotifiable` (ou `Notification`) et ajoutez-y une instance de `NotificationEmail` et de `NotificationSMS`. Parcourez la liste et appelez la méthode `Envoyer()` pour chaque élément.

C# : tableaux

Les Tableaux (`Arrays`) 📦

Un **tableau** est la structure de données la plus simple. C'est une collection d'éléments du **même type**, stockés dans un bloc de mémoire contigu.

- **Taille Fixe** : La taille d'un tableau est définie lors de sa création et ne peut plus être modifiée.
- **Indexation** : Les éléments sont accédés par leur **index**, qui commence toujours à **zéro (0)**.

```
// Déclaration et initialisation (taille de 3)
string[] jours = new string[3];
jours[0] = "Lundi";

// Initialisation simplifiée
int[] nombres = { 10, 20, 30, 40 };
```

C# : listes

Les Listes Génériques (`List<T>`) 📝

La **`List<T>`** est la collection la plus courante et la plus utilisée en C#. Elle est définie dans l'espace de noms `System.Collections.Generic`.

- **Taille Dynamique** : Contrairement aux tableaux, une liste peut **grandir** et **rétrécir** à l'exécution.
- **Généricité (`<T>`)** : Le `T` signifie "Type". Cela assure que la liste ne peut contenir que des éléments du type spécifié (`int`, `string`, `Personne`, etc.), garantissant la sécurité de type.

```
// T est remplacé par 'int'
List<int> scores = new List<int>();
scores.Add(95); // Ajout d'un élément
scores.Remove(95); // Suppression par valeur
scores.Insert(0, 100); // Ajout à un index spécifique
```

C# : foreach

Le Parcours avec `foreach` 🔄

Le mot-clé **`foreach`** permet d'itérer à travers tous les éléments d'une collection. C'est la façon **préférée** en C# pour parcourir la plupart des collections, car elle est plus lisible et moins sujette aux erreurs d'index.

- **`IEnumerable`** : `foreach` fonctionne sur toute collection qui implémente l'interface `System.Collections.IEnumerable` (ce qui est le cas de `List`, `Array`, `Dictionary`, `HashSet`, etc.).

```
foreach (var element in maCollection)
{
    // 'element' est l'élément courant de la collection
    Console.WriteLine(element);
}
```

C# : dictionnaire

Dictionnaires (`Dictionary<TKey, TValue>`) 🔑

Un **dictionnaire** stocke des données sous forme de paires **Clé-Valeur**. Il est optimisé pour la recherche rapide.

- **Clé (`TKey`)** : Doit être **unique** dans le dictionnaire. Elle sert d'identifiant pour récupérer la valeur associée.
- **Valeur (`TValue`)** : La donnée associée à la clé.

Parcours : Lors du parcours avec `foreach`, on itère sur des objets de type `KeyValuePair<TKey, TValue>`.

```
// Clé: string (Nom de la ville), Valeur: int (Population)
Dictionary<string, int> populations = new Dictionary<string, int>();
populations.Add("Paris", 2141000); // Ajout

int pop = populations["Paris"]; // Accès par clé

// Vérifier l'existence d'une clé avant d'y accéder
if (populations.ContainsKey("Lyon")) { /* ... */ }
```

```
foreach (KeyValuePair<string, int> paire in populations)
{
    Console.WriteLine($"Ville: {paire.Key}, Pop: {paire.Value}");
}
```

C# : hashset

HashSet (`HashSet<T>`) 🚫

Un `HashSet<T>` est une collection non ordonnée qui garantit que tous les éléments sont **uniques**. Il est très rapide pour l'ajout et la vérification de l'existence.

- **Unicité** : Si vous essayez d'ajouter un élément qui existe déjà, l'opération est ignorée sans erreur.
- **Pas d'Index** : Les éléments ne sont pas stockés avec un index et ne sont pas garantis d'être dans un ordre particulier.
- **Utilité** : Idéal pour des opérations d'ensembles (union, intersection, etc.) ou pour stocker une liste d'éléments uniques.

```
HashSet<string> tags = new HashSet<string>();
tags.Add("C#");
tags.Add("POO");
tags.Add("C#"); // Ignoré car déjà présent

bool existe = tags.Contains("C#"); // Très rapide
```

C# : exercice: collections

Vous devez créer un programme de gestion de tâches qui utilise plusieurs types de collections pour différents besoins.

1. **Liste (`List<string>`)** : Créez une liste pour stocker les noms des tâches à faire. Ajoutez-y au moins trois tâches (ex: "Faire les courses", "Coder l'exercice C#", "Répondre aux emails").

2. **Dictionnaire (`Dictionary<string, bool>`)** : Créez un dictionnaire pour suivre l'état d'achèvement de chaque tâche. La clé sera le nom de la tâche (string) et la valeur sera l'état (`bool`, `true` pour complété).

3. **HashSet (`HashSet<string>`)** : Créez un HashSet pour stocker les noms des catégories uniques associées aux tâches (ex: "Personnel", "Travail", "Coding").

4. **Parcours et Mise à Jour :**
   - Parcourez la **liste** des tâches à faire avec `foreach` et ajoutez chaque tâche au **dictionnaire** avec l'état `false`.
   - Simulez l'achèvement d'une tâche ("Coder l'exercice C#") en modifiant son état dans le dictionnaire.
   - Parcourez le dictionnaire avec `foreach` et affichez chaque tâche ainsi que son état (Complété/En cours).

C# : gestion des erreurs

La gestion des erreurs en C# est principalement réalisée à l'aide des blocs **`try-catch-finally`**. Ce mécanisme permet de gérer les exceptions, qui sont des événements anormaux survenant lors de l'exécution d'un programme, afin d'éviter qu'il ne plante et de lui permettre de se rétablir ou de se fermer proprement.

1. Le Mécanisme `try-catch-finally`

Le Bloc `try`

Le bloc **`try`** délimite la section de code qui est **susceptible de générer une exception**. Si le code dans ce bloc s'exécute sans erreur, le programme ignore les blocs `catch`.

Le Bloc `catch`

Le bloc **`catch`** est conçu pour **attraper et gérer** l'exception si elle se produit dans le bloc `try`.

- Il peut spécifier le type d'exception qu'il est capable de gérer (par exemple, `FormatException`, `DivideByZeroException`, ou le type générique `Exception`).
- Lorsque le bloc `try` lance une exception, le Common Language Runtime (CLR) recherche un bloc `catch` correspondant. Si une correspondance est trouvée, le code dans le bloc `catch` s'exécute, permettant de journaliser l'erreur, d'informer l'utilisateur, ou de tenter une récupération.

C# : gestion des erreurs

Le Bloc `finally`

Le bloc `finally` contient du code qui doit être exécuté **dans tous les cas**, que l'exception ait été levée ou non, et qu'elle ait été gérée ou non.

- Il est généralement utilisé pour effectuer des tâches de **nettoyage**, comme fermer des fichiers, des connexions à la base de données, ou libérer des ressources externes.
- L'exécution du code dans `finally` est garantie.

```
try
{
    // 1. Code à surveiller pour les exceptions
    // Exemple : Tenter de convertir une chaîne invalide en nombre
}
catch (ExceptionType variableNom) // 2. Si une exception se produit et correspond
{
    // Code de gestion de l'erreur
    // Exemple : Afficher un message d'erreur ou journaliser
}
finally // 3. Le code s'exécute toujours, peu importe le résultat
{
    // Code de nettoyage des ressources
    // Exemple : Fermer une connexion
}
```

C# : gestion des erreurs

. Gestion de Multiples Exceptions

Vous pouvez chaîner plusieurs blocs `catch` pour gérer différents types d'exceptions de manière spécifique. Le CLR essaie de faire correspondre l'exception levée avec les blocs `catch` dans l'ordre de leur déclaration. Il est préférable de placer les exceptions les plus spécifiques en premier, et l'exception générique `Exception` en dernier.

```
try
{
    string s = "abc";
    int nombre = int.Parse(s); // Lance une FormatException
    int resultat = 10 / nombre; // Peut lancer une DivideByZeroException

    // ... code ...
}
catch (FormatException ex) // Gère les erreurs de conversion (plus spécifique)
{
    Console.WriteLine($"Erreur de format : {ex.Message}. Veuillez entrer un nombre
}
catch (DivideByZeroException ex) // Gère la division par zéro (plus spécifique)
{
    Console.WriteLine("Erreur mathématique : Division par zéro impossible.");
}
catch (Exception ex) // Gère toutes les autres exceptions (le plus générique)
{
    Console.WriteLine($"Une erreur inattendue s'est produite : {ex.Message}");
}
finally
{
    Console.WriteLine("Fin de la tentative de traitement.");
}
```

C# : try/catch, bonnes pratiques

L'utilisation de `try/catch` n'est pas seulement une bonne pratique, c'est une nécessité pour garantir la **stabilité, la fiabilité et la robustesse** de votre application C# :

1. **Prévention du plantage (Crash Prevention)** 🛡️ : La raison principale est d'empêcher que des erreurs inattendues (comme la division par zéro, un accès à un fichier inexistant ou des problèmes de connexion) ne provoquent l'arrêt brutal et non maîtrisé de l'application, offrant une expérience utilisateur désastreuse.

2. **Rétablissement Contrôlé (Graceful Recovery)** 🩹 : Au lieu de crasher, `try/catch` vous donne l'opportunité d'intercepter l'erreur et d'exécuter un plan de secours : réessayer l'opération, charger des données par défaut, ou informer l'utilisateur de manière polie.

3. **Intégrité des Données et des Ressources** 💾 : Le bloc `finally` garantit que les ressources critiques (connexions de base de données, flux de fichiers) sont correctement fermées et libérées, même en cas d'échec. Cela prévient la corruption des données et les fuites de ressources.

4. **Diagnostic et Audit (Logging)** 📝 : Le bloc `catch` est l'endroit idéal pour enregistrer les détails de l'exception (le type, le message, et surtout la pile d'appels ou *stack trace*) dans des journaux. Ces informations sont cruciales pour le débogage et la maintenance future.

Pour que le mécanisme `try/catch` soit réellement efficace, il doit être utilisé de manière ciblée et intentionnelle.

1. Attrapez les Exceptions Spécifiques 🎯

- **Évitez le "Catch All" (Antipattern)** : N'utilisez **jamais** un simple `catch (Exception)` sans raison impérative, car cela masquerait des erreurs que vous n'aviez pas prévues et rendrait le débogage impossible.
- **Soyez précis** : Attrapez les types d'exceptions précis que vous anticipez et savez gérer (`FileNotFoundException`, `FormatException`, `ArgumentNullException`, etc.). Cela rend le code plus clair et vous permet de gérer chaque cas de manière appropriée.

2. Gérez ou Relancez l'Exception (Do Something!) ✍️

- **Ne pas ignorer** : Le `catch` vide (`catch {}`) est un **antipattern critique**. Si vous interceptez une exception, vous devez soit la gérer (par exemple, fournir une valeur par défaut, réessayer), soit la journaliser.
- **Journalisation** : Dans le bloc `catch`, utilisez un système de journalisation (comme Serilog ou NLog) pour enregistrer l'exception complète, y compris la propriété **`ex.StackTrace`**. Le message simple ne suffit pas pour diagnostiquer un problème en production.
- **Relancer (Re-throwing) Appropriément** : Si vous ne pouvez pas gérer l'exception dans le bloc `catch` actuel, vous devriez la relancer pour qu'un niveau supérieur puisse la gérer. Utilisez simplement `throw;` (sans spécifier l'objet exception) pour **préserver la trace de la pile d'appels** originale.

C# : try/catch, bonnes pratiques

3. Utilisez le Bloc `finally` pour le Nettoyage 🧹

- **Ressources externes** : Placez toujours le code de nettoyage des ressources (fermeture de connexions, libération de handles) dans le bloc `finally`.
- **Principe de Garantie** : Le code du `finally` est **garanti** d'être exécuté, que le bloc `try` ait réussi, ait échoué, ou que la méthode soit quittée prématurément.

4. `using` pour les Objets Jetables (IDisposable) 🗑️

- **Éviter `try/finally` pour les `IDisposable`** : Pour les objets qui implémentent l'interface `IDisposable` (fichiers, connexions), utilisez l'instruction **`using`** à la place de `try/finally`.
- **Simplicité et Sécurité** : Le bloc `using` garantit que la méthode `Dispose()` de l'objet est appelée automatiquement lorsque le bloc se termine (même en cas d'exception), simplifiant ainsi grandement le code de nettoyage.

5. Lancer des Exceptions Adéquates (Throwing) 📢

- **Utilisez des exceptions standard** : Lorsque vous codez, lancez des exceptions dont le type correspond à l'erreur (ex: `throw new ArgumentNullException();` si un paramètre est nul).
- **Exceptions personnalisées** : Si les exceptions standard ne suffisent pas, créez vos propres classes d'exception (qui héritent de `Exception`) pour encapsuler des informations métier spécifiques.

```
// Bonne pratique : utiliser 'using' au lieu de try/finally pour le nettoyage
using (FileStream fs = new FileStream("chemin.txt", FileMode.Open))
{
    // Utilisation de fs
} // fs.Dispose() est automatiquement appelé ici, même s'il y a une erreur.
```

C# : try/catch, bonnes pratiques

Le mot-clé **`throw`** en C# est un mécanisme fondamental de gestion des erreurs. Il est utilisé pour **déclencher (ou "lever") explicitement une exception**.

Lorsqu'une instruction `throw` est exécutée, elle arrête le déroulement normal du programme et transmet le contrôle au premier bloc `catch` approprié qu'elle trouve dans la pile d'appels.

Rôle et Utilité de `throw`

1. Signaler une condition d'erreur 📢

Le rôle principal de `throw` est de signaler qu'une situation anormale ou inacceptable s'est produite dans le code.

- **Exemple** : Si une méthode attend un nombre positif et reçoit un nombre négatif, elle devrait lever une exception `ArgumentOutOfRangeException`.

2. Créer des exceptions

Pour utiliser `throw`, vous devez lui fournir une **instance d'un objet exception** (une classe qui hérite de `System.Exception`).**3. Propager l'erreur (Relance)** 🔄

Le mot-clé `throw` est également utilisé dans un bloc `catch` pour **relancer** (re-throw) l'exception. Cela se produit lorsque vous interceptez une exception pour la journaliser ou y ajouter du contexte, mais que vous ne pouvez pas la résoudre à ce niveau du code.

Il y a deux façons de relancer, mais seule la première est recommandée :

- **Bonne Pratique (Préserve le Stack Trace) :** Utilisez simplement `throw;`. Cette syntaxe est cruciale car elle préserve la **trace de la pile d'appels** originale (le lieu exact où l'erreur s'est produite), ce qui est vital pour le débogage.

```
public void DefinirAge(int age)
{
    if (age < 0)
    {
        // On lève explicitement une exception standard
        throw new ArgumentOutOfRangeException("L'âge ne peut pas être négatif.");
    }
    this._age = age;
}
```

```
// Syntaxe pour lever une nouvelle exception
throw new TypeException(messageOptionnel);

// Exemple avec une exception personnalisée
throw new SoldeInsuffisantException("Le retrait dépasse le solde disponible.");
```

Mauvaise Pratique (Détruit le Stack Trace) : Si vous utilisez `throw ex;` (en spécifiant l'objet), la trace de la pile d'appels est réinitialisée à l'emplacement du `throw` actuel, ce qui masque l'origine réelle de l'erreur. **Évitez cette forme lors de la relance.**

Différence clé avec `return`

Il est important de noter que `throw` n'est **pas** un simple `return`.

- **`return`** : Quitte la méthode et reprend le contrôle du programme au point d'appel, en renvoyant une valeur (sauf pour `void`).
- **`throw`** : Quitte la méthode et interrompt le flux normal du programme. Le contrôle est transféré à un **gestionnaire d'exceptions** (`catch`) dans la pile d'appels. Si aucun gestionnaire n'est trouvé, le programme s'arrête.

Exercice Pratique : Calculatrice Sûre

Écrivez un programme C# simple qui demande à l'utilisateur d'entrer deux nombres, puis effectue une division. Utilisez le mécanisme `try-catch` pour gérer deux erreurs potentielles spécifiques :

1. **`FormatException`** : Si l'utilisateur entre une chaîne de caractères non numérique.
2. **`DivideByZeroException`** : Si l'utilisateur essaie de diviser par zéro.

Assurez-vous d'utiliser un bloc `finally` pour toujours afficher un message de confirmation de fin de processus.

C# : initiation à Linq

LINQ (Language-Integrated Query) est un ensemble de fonctionnalités introduites dans C# et le .NET Framework qui permet d'écrire des **requêtes** directement dans le langage C#, offrant une manière uniforme d'accéder aux données provenant de diverses sources.

C'est comme avoir le langage SQL intégré à C#, mais applicable non seulement aux bases de données, mais aussi aux collections d'objets, au XML, et bien d'autres sources.

Historique de LINQ 🕰️

LINQ a été introduit pour la première fois avec **C# 3.0** et le **.NET Framework 3.5** en 2007.

Avant LINQ, les développeurs devaient utiliser des syntaxes différentes pour interroger différentes sources de données :

- **Collections d'objets** : boucles `foreach`, structures `if/else` manuelles.
- **Bases de données relationnelles** : chaînes de requêtes SQL brutes (`string` non typées).
- **XML** : API complexes comme XPath.

Ce manque d'uniformité entraînait une perte de productivité, un manque de sécurité de type (les erreurs SQL n'étaient détectées qu'à l'exécution), et un code difficile à lire. LINQ a résolu ce problème en intégrant le concept de requête directement au langage.

Utilité et Avantages de LINQ ✨

1. **Uniformité (Unifying Data Access)** : Que vous interrogiez une `List<T>`, un tableau, un document XML, ou une base de données (via Entity Framework), la syntaxe de la requête reste pratiquement la même.
2. **Sécurité de Type (Type Safety)** : Les requêtes LINQ sont vérifiées par le compilateur C# lors de la compilation. Cela signifie que les erreurs de syntaxe, de noms de colonnes ou de types sont détectées immédiatement, et non en production comme c'était souvent le cas avec le SQL brut.
3. **Lisibilité et Maintenabilité** : La syntaxe déclarative de LINQ est souvent plus concise et plus facile à comprendre que l'écriture de boucles et de logique conditionnelle manuelles.
4. **IntelliSense et Débogage** : L'intégration complète à C# signifie que vous bénéficiez de l'autocomplétion (IntelliSense) et que vous pouvez déboguer les requêtes pas à pas.

C# : initiation à Linq

Comment Utiliser LINQ : Les Deux Syntaxes

Linq est implémenté dans “System.Link” (using System.Link)

LINQ peut être écrit de deux manières principales. Elles sont interchangeables, mais la syntaxe de méthode est généralement plus courante et plus flexible.

1. Syntaxe de Requête (Query Syntax)

Elle est plus proche du SQL et est souvent privilégiée pour les requêtes complexes impliquant des jointures ou des groupes.

2. Syntaxe de Méthode (Method Syntax)

Elle utilise des **méthodes d'extension** sur les collections (comme `Where`, `OrderBy`, `Select`) et des expressions lambda.

Note Importante : Le compilateur C# traduit la **Syntaxe de Requête** en **Syntaxe de Méthode** avant la compilation.

```
// Interrogation d'une liste de nombres
List<int> nombres = new List<int> { 1, 5, 12, 8, 25 };

var petitsNombres = from n in nombres // De la source 'nombres'
                    where n < 10      // Où 'n' est inférieur à 10
                    orderby n         // Trié par 'n'
                    select n;         // Sélectionne 'n'

// Résultat: 1, 5, 8
```

```
List<int> nombres = new List<int> { 1, 5, 12, 8, 25 };

// Utilisation des méthodes Where, OrderBy et ToList() pour obtenir le résultat
var petitsNombres = nombres.Where(n => n < 10)
                           .OrderBy(n => n)
                           .ToList();
```

C# Link : exercice simple

Exercice : Analyse des Ventes 📊

Vous êtes chargé d'analyser une liste d'enregistrements de ventes pour une petite boutique. Chaque enregistrement est un entier représentant le montant de la vente.

1. Le Contexte

Voici la liste des montants de vente (en euros) :

C#

```
List<int> ventes = new List<int> { 150, 45, 230, 80, 500, 15, 95, 300, 120, 60 };
```

2. Les Questions (Objectifs LINQ)

Utilisez la **Syntaxe de Méthode LINQ** (méthodes d'extension et expressions lambda) pour répondre aux questions suivantes :

1. **Filtrage (`Where`)** : Trouver toutes les ventes dont le montant est strictement **supérieur à 100** euros.
2. **Tri (`OrderBy`)** : Trier cette liste de grosses ventes (celles du point 1) par montant **croissant**.
3. **Comptage (`Count`)** : Compter combien de ventes sont considérées comme des "petites ventes" (montant **inférieur ou égal à 50** euros).
4. **Projection (`Select`)** : Créer une nouvelle liste qui contient le montant de chaque vente, mais **majoré de 10 %** (simulant une commission).
5. **Exécution Immédiate** : Assurez-vous que le résultat final de l'opération 2 est stocké dans une `List<int>` pour être affiché.

Linq : exécution différée

L'exécution différée et l'exécution immédiate sont des concepts fondamentaux de **LINQ** qui définissent **quand** une requête est réellement traitée. Comprendre cette distinction est crucial pour maîtriser les performances et le comportement de votre code C#.

1. Exécution Différée (Deferred Execution) 💡

L'exécution différée signifie que la requête LINQ est uniquement **définie**, mais **pas encore exécutée**, au moment où vous l'écrivez. C'est comme créer une recette : vous avez la liste des étapes, mais vous n'avez pas encore commencé à cuisiner.

Comment ça marche ?

1. **Définition** : Lorsque vous utilisez des opérateurs comme `Where()`, `Select()`, ou `OrderBy()`, vous construisez un objet de type `IEnumerable<T>` ou `IQueryable<T>`.
2. **Exécution** : Le code de la requête n'est exécuté qu'au moment où vous tentez d'accéder aux résultats, c'est-à-dire lors de l'**itération**.
    - Ceci se produit typiquement lorsque vous utilisez une boucle `foreach` pour parcourir la collection.
    - Ou lorsque vous appelez une méthode qui nécessite de parcourir les éléments.
3. **Données à jour** : Puisque la requête est exécutée au dernier moment, elle interroge l'état actuel de la source de données. Si la source de données change entre la définition et l'itération, la requête tiendra compte de ces changements.

```
List<int> nombres = new List<int> { 1, 2, 3 };

// Définition de la requête (exécution différée)
var resultats = nombres.Where(n => n > 1);

// Changement de la source APRÈS la définition
nombres.Add(4);

Console.WriteLine("Affichage des résultats :");
foreach (var n in resultats) // C'est ici que la requête est EXÉCUTÉE
{
    Console.WriteLine(n); // Affiche 2, 3, et 4
}
```

Linq : exécution immédiate

Exécution Immédiate (Immediate Execution) 🚀

L'exécution immédiate signifie que la requête est **exécutée instantanément** au moment même où l'instruction est rencontrée.

Comment ça marche ?

1. **Déclenchement** : Certaines méthodes LINQ sont conçues pour forcer l'exécution de la requête à ce moment précis, car elles ont besoin de l'ensemble des résultats pour fonctionner.
2. **Opérateurs Forçant l'Exécution** :
   - **Conversion** : `ToList()`, `ToArray()`, `ToDictionary()` forcent l'exécution et stockent tous les résultats en mémoire.
   - **Agrégation** : `Count()`, `Sum()`, `Max()`, `Average()` doivent parcourir tous les éléments pour produire un seul résultat.
   - **Éléments Simples** : `First()`, `Single()`, `Any()` forcent l'exécution pour trouver le premier (ou unique) élément correspondant.

```
List<int> nombres = new List<int> { 1, 2, 3 };

// Exécution immédiate : La requête est exécutée MAINTENANT, et les résultats stocl
List<int> resultats = nombres.Where(n => n > 1).ToList();

// Changement de la source APRÈS l'exécution
nombres.Add(4);

Console.WriteLine("Affichage des résultats :");
foreach (var n in resultats) // Itération sur la liste STATIQUE et STOCKÉE
{
    Console.WriteLine(n); // Affiche 2 et 3. Le 4 n'est pas inclus car il a été ajo
}
```

Exécution immédiate VS différée

Caractéristique	Exécution Différée (e.g., `Where()`)	Exécution Immédiate (e.g., `ToList()`)
Quand Exécuté ?	Lors de l'itération (`foreach`, `Count`, etc.)	Immédiatement à l'appel de la méthode
Mémoire	Faible, les résultats ne sont pas stockés	Potentiellement élevée, tous les résultats sont stockés en mémoire
Utile pour...	Requêtes complexes, LINQ to Entities (SQL)	Stockage en cache des résultats, affichage
Piège à Éviter	**Exécution multiple** si la collection est itérée plusieurs fois sans `ToList()`.	Stocker des **résultats volumineux**

Le Piège de l'Exécution Multiple

Si vous utilisez l'exécution différée mais que vous itérez sur la même variable `IEnumerable` plusieurs fois (par exemple, dans deux boucles `foreach` différentes), la requête sera **réexécutée à chaque itération**.

C'est là que l'exécution immédiate est nécessaire : utiliser `ToList()` permet d'exécuter la requête une seule fois, de stocker le résultat, et d'éviter les coûteuses réexécutions, surtout si la source est une base de données.