# Développement en C#

# C# & .NET : Présentation

# C# & .NET

C# est un langage de programmation multiparadigme développé par Microsoft au début des année 2000 sur la même idée que Java. Son fonctionnement est très similaire. Le compilateur C# produit du code dit "Intermediate Language" (IL) une espèce d'assembleur spécifique, destiné au "Common Language Runtime" (CLR), l'interpréteur de l'écosystème .NET. Il existe d'autres languages qui compilent en IL pour le CLR : Visual Basic .NET et F#.

Contrairement à certaines idées reçues, C# et .NET sont :

- Entièrement gratuits et open-source (licences MIT)
- Cross-platforme (comme Java, *write once, run anywhere*)

C# est principalement orienté objet, mais a su évolué de manière moins dogmatique que Java, adoptant au fil de son histoire des fonctionnalités telles que :

- Programmation orientée fonction avec LinQ, le pattern matching et les records
- Programmation événementielles avec les événements et les délégués
- Programmation asynchrone avec la Task Parallel Library

Le caractère multiparadigme du C# permet d'écrire du code tirant le meilleur de chaque paradigme afin de faire face à toute sorte de situation avec une code performance et expressif.

# .NET Core / .NET Framework / .NET 6

L'écosystème .NET a un historique de 20 ans qui a donné lieu a une situation un peu complexe qui peut préter à confusion. Historiquement, il y avait le .NET Framework, uniquement spécifique à Windows, qui est désormais déprécié. En 2016 a été lancé .NET Core, un inmplémentation cross-plateforme du runtime, qui est devenu la version de référence de l'écosystème lors de la sortie de .NET 5 (version faisant suite à .NET Core 3.1 mais sans le core) mais qui actait le remplacement du .NET Framework par .NET Core, en appelant maintenant .NET Core .NET "tout court". La version courante est la version .NET 7.

# Installation

## Windows

Plusieurs possibilités :

- Via l'installeur téléchargeable ici
- Via `winget` : `winget install -e --id Microsoft.DotNet.SDK.6`

# Linux

1. Télécharager le package x64 de la dernière version ici
2.

```
mkdir -p $HOME/dotnet && tar zxf dotnet-sdk-7.0.100-linux-x64.tar.gz -C $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH:$HOME/dotnet
```

# MacOS

Utiliser l'installeur téléchargeable ici

# Vérification

Une fois installé, vous pouvez vérifier l'installation en exécutant `dotnet --version`

# Tooling et .NET CLI

# Environnements de développement

Possibilités en matière d'outillage :

- JetBrains Rider (recommandé) : gratuit pour les étudiants et excellent intelisense et autres tooling
- Visual Studio Community (second choix) : gratuit et intellisense moyenne, mais meilleur car c'est un IDE
- Visual Studio Code avec le pack C# : gratuit et léger mais Intellisense très mauvaise + peu de tooling (pas un IDE)

# Projet et Solution

La base d'un projet .NET n'est pas le projet mais la solution. Une solution est une collection de projets. Pour créer une solution :

```
mkdir mon-nouveau-projet
cd mon-nouveau-projet
dotnet new sln
```

La solution est définie par le fichier `.sln` créé par cette commande. On peut ensuite créer des projets. Par convention de structure, on créer au niveau du fichier `.sln` deux dossier, `src` et `test` dans lesquels seront rangés les projets. Le premier pour le code de production, le second pour le code de test.

Les projets le créent à partir de templates. `dotnet new -l` permet de lister les templates installés. Par exemple pour créer un projet "Application Console" et le mettre dans le dossier `src` par la même occasion : `dotnet new console -o src/MonApp.Console`. On peut ensuite relier le projet à la solution avec `dotnet sln mon-nouveau-projet.sln add src/MonApp.Console/MonApp.Console.csproj`. Le fichier `.csproj` d'un projet est le fichier de configuration du projet au format XML.

Les autres templates de projet qui peuvent être intéressants sont les suivants :

- `classlib` : librairie de classe, pas de point d'entrée
- `xunit` : projet de tests avec le framework de test XUnit

# Exécution

Les commandes d'exécution sont les suivantes (à partir du répertoire d'un projet) :

- `dotnet run` : exécuter le projet à partir de son point d'entrée
- `dotnet test` : exécuter les tests d'un projet

> `dotnet test` peut être exécuté depuis le répertoire de la solution pour exécuter tous les tests de la solution

# Librairies

L'outil de gestion de dépendances de .NET (équivalent NPM / Maven / Gradle ...) est Nuget. Pour installer une librairie avec Nuget, la commande `dotnet add package` est disponible. Exemple : `dotnet add package Newtonsoft.Json`.

# Base de la Programmation Objet avec C#

# Définition

## Membres

La programmation orientée objet consiste à rapprocher les traitements (fonctions) des données (variables). Cela permet de modéliser des situations de façon plus logique et naturelle. La POO s'articule donc autour de structures appelées **classes** qui possède un état (des attributs) et des comportements (les méthodes). L'ensemble des attributs et méthodes d'une classe sont appelées les membres de la classe.

En C#, on peut définir une classe à l'aide du mot clé `class`. Une classe doit être définie dans un fichier qui porte son nom. Pas convention, les noms des classes sont en PascalCase. Les noms des membres camelCase.

Définition d'une classe :

```
class Cat {


}
```

Pour définir un attribut, on précise d'abord son type, puis son nom :

```
class Cat {
    string name;
}
```

Pour définir une méthode, on précise d'abord son type de retour, puis son nom, puis ses arguments :

```
class Cat {
    string name;


    void Pet(int duration) {
```

```
        // Corps de la méthode
    }
}
```

# Constructeur

Le constructeur est un membre (plus spécifiquement une méthode) particulier de la classe, qui est appelé à son instanciation et qui sert à initialiser l'état de l'objet.

Pour définir un constructeur on définit une méthode qui a le nom de la classe :

```
class Cat {
    string name;

    Cat(string catName){
        name = catName;
    }
}
```

Une classe peut avoir plusieurs constructeurs (avec différents prototypes). Un constructeur peut faire appel à un autre par un appel à `this()`.

```
class Cat {
    string name;

    Cat() : this("Felix"){
    }

    Cat(string catName){
        name = catName;
    }
}
```

# Classes et Instances

## Instances

Une classe est un type qui définit de quels membres seront dotés ses instances. Une instance d'une classe (aussi appelée objet), est une variable du type de la classe et possède tous les

membres définis par cette dernière, avec des valeurs qui lui sont propres. On peut instancier un objet avec le mot clé `new`, suivi d'un appel au constructeur de la classe :

```
Cat cat = new Cat("Félix");
```

On peut accèder aux membres (attributs ou méthodes) de la classe avec l'opérateur `.` sur la référence de l'objet :

```
Console.WriteLine(cat.name);
cat.Pet(10);
```

# Membres de classe

On peut aussi déclarer dans une classe des membres (attributs ou méthodes) qui seront communs à toutes les instances d'une classe (une valeur pour toutes les instances) grâce au mot clé `static` :

```
class Cat {
    static numberOfCats = 0;

    Cat(string catName){
        name = catName;
        numberOfCats++;
    }
}
```

Les membres de classe peuvent être utilisés au sein de la classe, ou alors à l'extérieur avec l'opérateur `.` sur le nom de la classe :

```
Console.WriteLine(Cat.numberOfCats);
```

# Référence `this`

La référence `this` est accessible dans toute classe dans les contextes non `static`. Elle pointe vers l'instance courante. Elle est utilisée pour lever des ambiguités de nommage :

```
class Cat {
    Cat(string name){
        this.name = name;
    }
}
```

Elle est également utilisée pour des raisons de visibilité afin de permettre de dissocier rapidement une variable locale d'un attribut d'instance.

# Références & Garbage Collection

En C#, toutes les classes sont de types référence. Les `struct` ansi que les primitifs :

- int
- char
- long
- double
- float
- bool
- short
- byte

sont des types valeur.

Tous les classes sont des types références, c'est-à-dire que les variables contiennent les adresses mémoires des objets. Tous les objets sont alloués en mémoire sur le tas, et c'est le runtime .NET qui s'occupe de les désallouer lorsqu'ils ne sont plus référencés via un mécanisme appelé la **Garbage Collection**.

# Encapsulation

## Namespaces

Les namespaces sont les espaces de noms pour organiser le code. Par convention on les fait correspondre à la structure de dossiers. Leur nomage est par convention en PascalCase.

Pour déclarer le namespace dans un fichier (en haut par convention) :

```
namespace MonNamespace;
```

## Visibilité

La notion de visibilité permet de restreindre l'accès aux membres d'une classe. Il existe 5 modificateurs d'accès :

| Modificateur | Classe | Projet | Classe Enfant | Tout le reste |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| `public` | Oui | Oui | Oui | Oui |
| `internal` | Oui | Oui | Non | Non |
| `protected` | Oui | Non | Oui | Non |
| `protected internal` | Oui | Oui | Oui | Non |
| `private` | Oui | Non | Non | Non |

La visibilité par défaut (pas de modification de visibilité) est la visibilité `private`.

# Principe d'encapsulation

Le principe d'encapsulation dicte que seul les informations que l'interface publique (l'ensemble des membres publics) d'une classe doit être la plus restreinte possible. C'est pourquoi tous les attributs doivent être privés. Les méthodes sont publiques que si elles ont besoin de l'être.

Pour exposer avec une granularité fine les attributs, on utilise des méthodes un peu spéciales, les propriétés. Les propriétés peuvent contenir :

- Un **accesseur**, pour lire un champs. Avec le mot clé `get`
- Un **mutateur**, pour écrire un champs. Avec le mot clé `set`

Exemple de champs correctement encapsulé :

```
namespace Animals;

public class Cat {
    private string name;

public string Name {
      get => this.name;
      set => this.name = value;
    }

    public Cat(String name){
        this.name = name;
    }
}
```

On peut simplifier cette syntaxe, avec les propriétés automatiques, elles déclarent automatiquement leur champ privé :

```
namespace Animals;

public class Cat {
    public string Name {get;set;}

    public Cat(String name){
        this.Name = name;
    }
}
```

Il ne faut déclarer un accesseurs ou un mutateur sur une propriétés que si besoin, par exemple :

```
namespace Animals;

public class Cat {
    public string Name {get;}

    public Cat(String name){
        this.Name = name;
    }
}
```

On peut aussi jouer sur la visibilité :

```
namespace Animals;

public class Cat {
    public string Name {get;private set;}

    public Cat(String name){
        this.name = name;
    }
}
```

# Fonctionnalités syntaxiques

## Inférence de type

C# supporte l'inférence de type pour les variables locales avec le mot clé `var`, exemple :

```
var hello = "Hello world";
```

A utiliser uniquement lorsque le type est évident (constructor ou littéral).

## Interpolation de chaine

C# supporte l'interpolation de chaine avec `$`, exemple :

```
var firstName = "John";
var lastName = "Shepard"
var fullName = $"{firstName} {lastName}";
Console.WriteLine(fullName); // Prints John Shepard
```

## Opérateurs `?` et `??`

### Nullabilité

Un type suivi de `?` signfie que le type est nullable. En cas d'assignation de `null` à un type qui n'est pas déclaré comme nullable, le compilateur produit un warning. Exemple :

```

int nonNullableNumber= 42;
nonNullableNumber = null; // warning de compilation

int? nullableNumber = 42;
nullableNumber = null; // pas de warning de compilation
```

### Ternaires

```
decimal ticketPrice = age >= 18 ? 10 : 6;
```

Equivalent :

```
decimal ticketPrice;
if(age >= 18){
  ticketPrice = 10;
} else {
  ticketPrice = 6
}
```

# Accès conditionné par `null`

Permet de conditionner l'accès à un champs ou l'appel d'une méthode par la non présence d'une valeur `null` sur la référence sur laquelle on effecture l'appel. Exemple :

```
User? user = GetUser();
string? name = user?.Name;
```

Equivalent :

```
User? user = GetUser();
string? name;

if(user is not null){
name = User.Name;
}
```

# `null`-coallescing

Valeur de repli en cas de valeur `null` :

```
User user = GetUser() ?? new User("John Shepard");
```

Equivalent :

```
User? temp = GetUser();

User user;
```

```
if(temp is not null){
    user = temp;
} else {
    user = new User("John Shepard");
}
```

# Méthodes "Expression Body"

```
public class Point2D {
    private int x;
    private int y;

    public Point2D(int x, int y){
    this.x = x;
        this.y = y;
    }

    public Point Move(int distanceX, int distanceY) => new Point(x + distanceX, y +
distanceY);
}
```

# Intialisation

## Propriétés d'objets

```
var employee = new Employee{ FirstName = "John", LastName = "Shepard"};
```

Equivalent :

```
var employee = new Employee();
employee.FirstName = "John";
employeeLastName = "Shepard";
```

## Collections

## Listes / Tableaux

```
var list = new List<string> { "Item1", "Item2", "Item3" }
```

## Dictionnaires

```
var map = new Dictionnary<int,string> {
  [42] = "Item1",
  [41] = "Item2",
  [123] = "Item3"
}
```

# Méthodes d'extensions

Les méthodes d'extensions permettent de rajouter des méthodes aux types qu'on ne controle pas (ex : classes de la Base Classes Library (BCL) ou des librairies qu'on utilise). Cela requiert de créer une classe `static`, et les méthodes d'exension doivent prendre en premier paramètre une argument du type étendu, avec le mot clé `this`. Exemple :

```
public static class Extensions {
    public static void Multiply(this string stringToMultiply, int times){
        return String.Concat(Enumerable.Repeat(stringToMultiply, times));
    }
}
```

# Surcharge d'opérateurs

En C#, les opérateurs du langage (`+`, `-`, `*`, etc...) peuvent être redéfinis :

```
    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);
```

---

# Héritage & Polymorphisme

# Héritage

## Principe de base

Dire qu'une classe hérite d'une autre, c'est établir une relation **EST UN** entre deux classes.

Soit une classe `Mamal` telle que :

```
public class Mamal {
    ...
}
```

La classe `Human` peut hériter de la classe `Mamal` avec la syntaxe suivante :

```
public class Human : Mamal {
    ...
}
```

On dit que la classe `Mamal` est la **super classe** de `Human` et que `Human` est une **sous classe** de `Mamal`. Cela implique la logique suivante, un humain **est un** mamifère.

Quand une classe hérite d'une autre, cela implique que la classe fille récupère tous les membres (champs et méthodes) de la classe mère, quelle que soit leur visibilité. En C#, une classe peut hériter que d'une seule super classe.

> Attention, une sous classe possède tous les champs privés de sa super classe mais n'y a pas accès.

# Constructeur

Si une classe n'a qu'un constructeur pas défaut, les constructeurs des classes filles appellent implicitement ce constructeur.

Si une classe a un constructeur définit autre qu'un constructeur par défaut, ses sous classes doivent impérativement appeler ce constructeur via `base()`,dans la signature du constructeur.

Par exemple si on a une classe telle que :

```
public class Mamal {
    private string name;

    public Mamal(string name){
        this.name = name;
    }
}
```

Ses sous classes doivent implémenter ce constructeur, et faire appel à la logique ainsi :

```
public class Human : Mamal {
    public Human(string name) : base(name) {
    }
}
```

# Redéfinition de méthodes

Une sous classe peut redéfinir les méthodes de la super classe, afin de l'adapter à ce quelle est. Pour autorisée une méthode à être redéfinie, on ajoute le mot clé `virtual` à sa signature dans la classe parente, et on la réécrit dans la classe fille avec le mot clé `override` dans la signature. Par exemple une classe telle que :

```
public class Mamal {
    public virtual void Eat() {
        Console.WriteLine("I eat");
    }
}
```

Ses sous classes peuvent redéfinir la méthode `Eat()` pour qu'elle corresponde à ce qu'elles sont.

```
public class Human : Mamal {
    public override void Eat() {
        Console.WriteLine("I eat with a fork and a knife");
    }
}
```

Une méthode redéfinie peut faire appelle à la méthode originale de la super classe grâce à la référence `base` :

```
public class Human : Mamal {
    public override void Eat() {
        Console.WriteLine("I take a fork and a knife");
        base.Eat();
    }
}
```

La trace de ce code sera :

```
I take a fork and a knife
I eat
```

# Abstraction

La notion d'abstraction permet de définir de classes dites abstraites, qui ne peuvent être instanciées, mais établissent des **contrats de service** avec leur sous classes, c'est à dire qu'elles définissent le prototype de méthodes que les sous classes seront obligées de définir.

Pour définir une classe abstraite :

```
public abstract class Mamal {
    ...
}
```

Une classe abstraite peut définir tous les membres qu'une classe concrète peut définir, mais elle peut définir des méthodes abstraites. Ces méthodes sont les méthodes qui doivent être définies par les sous classes, et ne sont pas définies par la classe bastraite.

```
public abstract class Mamal {
    public abstract void Eat();
}
```

La classe abstraite ne fournit pas d'implémentation de la méthode, mais oblige par **contrat de service** ses sous classe à la définir. Une classe qui étend une classe abstraite doit définir ses méthodes abstraites ou alors être abstraite également, sans quoi elle ne compilera pas. Une méthode `abstract` est implicitement `virtual`.

# Polymorphisme

Le polymorphisme est l'un des principes fondamentaux de la programmation orientée objet, qui consiste à utiliser les contrats de service pour manipuler des classes qui partagent une super classe dont ils redéfinissent certains comportements. En effet, une référence du type d'une certaine classe, peut recevoir une référence de n'importe quelle sous classe de la classe en question. Avec les classes suivantes :

```
public abstract class Animal {
    public abstract void Shout(){
    }
}
```

```
public class Dog : Animal {
    public override void Shout(){
        Console.WriteLine("wof wof !")
    }
}
```

```
public class Cat : Animal {
    public override void Shout(){
        Console.WriteLine("mew mew !")
    }
}
```

Le code suivante est possible :

```
Animal animal1 = new Dog();
Animal animal2 = new Cat();
```

Si on appelle `Shout()`, la définition de la méthode correspondant à chaque sous classe sera appelée :

```
animal1.Shout();
animal2.Shout();
```

La trace de ce code sera :

```
wof wof !
mew mew !
```

Ce principe est très puissant, car il permet au code appelant d'ignorer les sous classes et leur implémentation, et d'utiliser juste ce dont il a besoin pour effectuer son travail, permettant de **séparer les responsabilités**. Par exemple, étant donné la classe `Human` suivante :

```
public class Human {
    private List<Animal> pets = new List<Animal>();

    public void Adopt(Animal animal){
        animal.Shout();
        this.pets.Add(animal);
    }
}
```

La classe `Human` ainsi que le méthode `Adopt()` n'ont à connaître des animaux que le fait qu'ils peuvent crier, car savoir de quel type sont les animaux, ou de savoir comment ils crient ne sont sont pas de sa responsabilité. Ainsi, si on change la façon de crier des animaux, si on rajoute des nouvelles classes d'animaux, pas besoin de modifier la classe `Human` ni la méthode `Adopt()`.

# Interfaces

Les interfaces sont des classes purement abstraites. Elles ne contiennent pas d'attributs et seulement des méthodes `public` et `abstract` (si bien qu'il n'y a pas besoin de le préciser, c'est implicite). Les interfaces permettent de définir des contrats de service en s'affranchissant des contraintes de l'héritage. En effet, une classe peut implémenter plusieurs interfaces. Ainsi, on va privilégier cette stratégie, et utiliser l'héritage uniquement lorsque l'on a besoin de factoriser des membres communs à plusieurs sous classes.

Définir une interface :

```
public interface Engine {
    void Start();
    void Accelerate();
    void Stop();
}
```

Implémenter une interface :

```
public class ElectricEngine : Engine {
    public void Start() {

    }

    public void Accelerate(){

    }
```

```
    public void Stop(){

    }
}
```

```
public class CombustionEngine implements Engine {

    public void Start(){

    }

    public void Accelerate(){

    }

    public void Stop(){

    }
}
```

L'utilisation des interfaces permet, de séparer au maximum les contrats de service de l'implémentation, afin de séparer les responsabilité.

> Attention à ne pas centraliser trop de comportements dans une seule interface. Une interface doit disposer uniquement des comportements qui doivent être exposés au code appelant, tout en ayant une valeur sémantique.

# Inversion de dépendance

Il est très important de réduire les dépendances au maximum en utilisant les interfaces. Il vaut toujours mieux dépendre d'une interface que d'une implémentation. Cela permet de rendre le code plus facile à changer mais aussi plus facile à tester.

Par exemple :

```
public class Car {
    private CombustionEngine engine;
```

```
    public Car(){
        engine = new CombustionEngine();
    }
}
```

Ici, la classe `Car` dépend de la classe `CombustionEngine`. Il vaudrait utiliser notre interface `Engine` pour cacher son implémentation à la classe `Car`. Cela permet de changer d'`Engine` sans modifier `Car` ainsi que de tester la classe `Car` indépendement de son `Engine`.

```
public class Car {
    private Engine engine;

    public Car(Engine engine){
        this.engine = engine;
    }
}
```

Et dans le code utilisant `Car` :

```
Car car = new Car(new CombustionEngine());
```

# Programmation Fonctionnelle

# Fonction de première classe

C# supporte les fonctions de première classes (first-class functions) car il permet de manipuler des fonctions comme des variables, ceci notamment grace aux délégués.

## Délégués

Les délégués (delegates) sont le spport de première classe pour les fonctions en C#. Les délégués sont des pointeurs de fonction typés, qui permettent de manipuler les méthodes comme des variables et de les appeler. Le délégué se définit avec le mot-clé `delegate` et contraint la signature de fonction correspondante :

```
public delegate int MathOperation(int leftOperand, int rightOperand);
public class Addition{
public int Add(int leftOperand, int rightOperand) => leftOperand + rightOperand;
}

MathOperation addition = new Addition().Add;
```

On peut ensuite exécuter la méthode à partir de cette variable :

```
var result = addition.Invoke(4,2);
var result = addition(4,2);
```

## Lambdas

On peut aussi utiliser la syntaxe lambda pour définir des fonctions anonymes, et les assigner à des délégués :

Lambda "expression body" :

```
MathOperation addition = (int left, int right) => left + right;
```

Lambda normale :

```
MathOperation addition = (int left, int right) => {
  return left + right;
};
```

# LinQ

LinQ, pour Language Integrated Queries est une fonctionnalités de C# qui permet d'utiliser les opérateurs fonctionnels sur les collections.

## `IEnumerable`

L'interface `IEnumerable` est une abstraction de toute collection sur laquel fonctionne LinQ. Pour générer une `IEnumerable` dans une méthode, on utilise le mot clé `yield` :

```
public static IEnumerable<int> Power(int number, int exponent)
{
  int result = 1;

  for (int i = 0; i < exponent; i++)
  {
    result = result * number;
    yield return result;
  }
}
```

On peut ainsi l'énumérer comme une collection avec une boucle `foreach` :

```
foreach (int i in Power(2, 8))
{
  Console.Write("{0} ", i);
}
```

# Opérateurs fonctionels sur collections

LinQ fournit des opérateur fonctionels pour écrire du code très expressif de traitement des collections (toutes les collections de la Base Class Library (BCL) implémentent `IEnumerable`). Le nommage des méthodes rappellent le SQL. LinQ peut s'exécuter sur des données en mémoire ou en dehors du système par le biais des arbres d'expressions. Les méthodes LinQ ont en argument des délégués, on les utilise souvent en passant des lambdas.

```
public class Employee {
    public string Name{get;set;}
    public int Age {get;set;}
}

List<Employee> employees = new List<Employee> {
    new Employee{ Name = "Shepard", Age = 28},
    new Employee{ Name = "Liara", Age = 106}
};

List<string> seniorNames = employees
    .Where(employee => employee.Age >= 50) // Opérateur de filtrage
    .Select(employee => employee.Name) // Opérateur de transformation
    .ToList(); // opérateur terminal

// Resultat : { "Liara" }
```

Les opérateurs de filtrage et de transformation sont "lazy" ; ils ne font rien tant que l'on pas appliqué un opérateur terminal qui va, lui, procéder à l'énumération.

# Opérateurs de filtrage

## `Where`

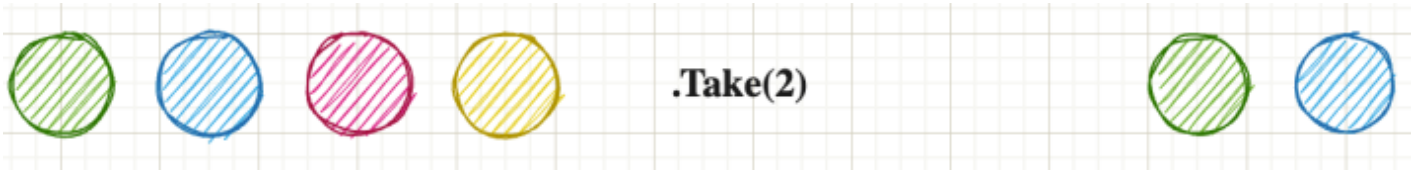
Filtre les éléments à partir d'un prédicat, il conserve les éléments qui valident le prédicat.


```
List<Employee> employees = new List<Employee> {
    new Employee{ Name = "Shepard", Age = 28},
    new Employee{ Name = "Liara", Age = 106}
};

IEnumerable<Employee> seniors = employees
    .Where(user => user.Age >= 18);
```

```
// Resultat : [ Employee {Name = "Liara", Age = 106} ]
```

## Take

Prends seulement les n premiers éléments.




```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};


IEnumerable<Employee> adults = users
.Take(2);

// Resultat : [ Employee {Name = "Shepard", Age = 28}, Employee {Name = "Liara", Age = 106} ]
```

## Skip

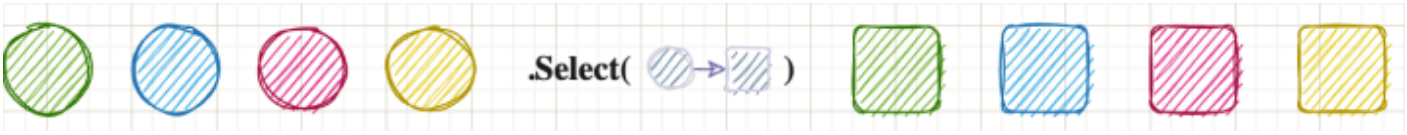
Prends seulement les éléments après en avoir supprimé n.


```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};

IEnumerable<Employee> adults = users
.Skip(2);

// Resultat : [ Employee {Name = "Tali", Age = 23} ]
```

# Opérateurs de transformation

## `Select`

Mappe chaque élément à autre chose en utilisant une fonction.




```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};

IEnumerable<string> employeesNames = employees
  .Select(employee => employee.Name)

// Resultat : [ "Shepard","Liara","Tali" ]
```

## `SelectMany`

Mappe chaque élément à une collection d'autre chose, puis applatis le résultat.


```
public class Team {
public string Name {get;set;}
    public List<Employees> Members {get;set;}
}

List<Team> teams = new List<Team> {
  new Team {
      Name = "First Team",
      Members =  new List<Employee> {
          new Employee{ Name = "Shepard", Age = 28},
          new Employee{ Name = "Liara", Age = 106},
          new Employee{ Name = "Tali", Age = 23}
      };
```

```
    },
    new Team {
      Name = "Second Team",
      Members =  new List<Employee> {
          new Employee{ Name = "Garrus", Age = 27},
          new Employee{ Name = "Kaidan", Age = 34},
          new Employee{ Name = "Joker", Age = 30}
      };
    },
}

IEnumerable<string> employeesName = teams
  .SelectMany(team => team.Members) // Retourne IEnumerable<Employee>
  .Select(employee => employee.Name)
```

# Opérateurs Terminaux

## `ToList` / `ToArray`

Enumère l'expression LinQ pour construire une liste / un tableau.

## `First` / `FirstOrDefault`

Enumère l'expression LinQ, jusqu'à trouver un élément qui correspond au prédicat. `First` jette une exception si pas d'élément correspondant, `FirstOrDefault` retourne `null`.


```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
    new Employee{ Name = "Tali", Age = 22}
};

Employee tali = employees
.First(user => user.Name == "Tali");
```

# Pour aller plus loin

LinQ possède d'autres opérateurs intéressants :

- Aggrégation : `Count`, `Max`, `Aggregate`, `GroupBy` (Terminaux)
- Quantification : `All`, `Any` (Terminaux)
- Fusion : `Join`, `Zip` (Transformation)
- Ensemblistes : `Union`, `Intersect` (Transformation)

# Pattern Matching

Le pattern matching consiste à tester une expression, pour vérifier si elle a certaines caractéristiques. On peut l'utiliser en C# dans les `if` (avec le mot-clé `is`) et dans les `switch`-expression.

## Vérification de `null`

En utilisant un pattern de type non-nullable :

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

En utilisant le pattern `not` avec `null` :

```
string? message = "This is not the null string";

if (message is not null)
{
    Console.WriteLine(message);
}
```

## Vérification de type

```
Animal animal = ...;
if(animal is Human human){
  ...
}
```

# Valeur discrètes

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command", nameof(command)),
   };
```

# Pattern logique

```
public string GetGreeting(string name, TimeOnly time) => time.Hour switch
{
    >=0 and <6 or >=20 => $"{GOOD_NIGHT_MESSAGE} {name}!",
    >=6 and <12 => $"{GOOD_DAY_MESSAGE} {name}!",
    >=12 and <20 => $"{GOOD_AFTERNOON_MESSAGE} {name}!",
    _ => throw new ArgumentException($"Unsupported hour :
{this.currentTimeProvider.CurrentTime.Hour}")
}
```

# Pattern de propriétés

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate discount on
null order"),
```

```
        var someObject => 0m,
    };
```

# Tests Unitaires avec XUnit

Pour créer un nouveau projet de test avec le framework de tests XUnit :

```
dotnet new xunit -o "MonProjet.Test"
```

# Test unitaire

## Cas de test

Pour créer un test unitaire, créez une classe dans ce projet de test. Une classe correspond à une collection de test, chaque méthode est un cas de test. Pour créer un cas de test, ajoutez une méthode avec l'attribut `[Fact]` :

```
public class MaCollectionDeTests {

  [Fact]
  public void MonCasDeTest(){
  
  }

}
```

## Organisation et nommage

Les tests se déroulent en 3 étapes :

- "Etant donné" (*Given* en anglais) : mette en place la situation (souvent les données) requises par le cas de test
- "Lorsque" (*When* en anglais) : appel au code que l'on veut tester dans ce cas de test
- "Ainsi" (*Then* en anglais) : vérifications (assertions) sur les résultats du tests

Il convient donc, pour avoir des tests lisible des les diviser entre ces trois étapes, ainsi que de nommer les méthode correspondant au cas de tests de façon à expliciter le contenu de ces étapes. Le pattern suivant est utilisé de manière général pour la nommage : `Action_Given_Then`. "Action" étant souvent le nom de méthode testée.

Exemple :

```
public class CalculatorTests {

  [Fact]
  public void Multiply_ValidOperands_ReturnsCorrectResult(){
  // Given
    int leftOperand = 3;
    int rightOperand = 4;

    var calculator = new Calculator();

    // When
    int result = calculator.Multiply(leftOperand, rightOperand);

    // Then
    Assert.Equal(12, result);
  }
}
```

Et avec un cas négatif, avec une vérification d'exception :

```
  [Fact]
  public void Divide_By0_Throws(){
  // Given
    int leftOperand = 3;
    int rightOperand = 0;

    var calculator = new Calculator();

    // When
    Action act = () => calculator.Divide(leftOperand, rightOperand);



    // Then
    Assert.Throws<DivideBy0Exception>(act);
  }
```

# Setup

Pour faire un setup commun à tous les cas de tests d'une collection, il suffit d'utilise le constructeur de la classe qui correspond à cette collection, il est appelé pour chaque cas de test.

# Tests paramétrisés

Il est possible de faire des tests paramétrisés, avec l'attribut `[Theory]`. On peut passer les paramètres de plusieurs manières.

## Avec l'attribut `[InlineData]`

L'attribut `[InlineData]` permet de passer des données à un test paramètrisé mais ne supporte que les constantes à la compilation (littéraux de types primitif) :

```
public class CalculatorTests {

  [Theory]
  [InlineData(3,4, 12)]
  [InlineData(2,3, 6)]
  [InlineData(3,5, 15)]
  [InlineData(4,4, 16)]
  public void Multiply_ValidOperands_ReturnsCorrectResult(int leftOperand, int rightOperand,
int expectedResult){
  // Given

    var calculator = new Calculator();

    // When
    int result = calculator.Multiply(leftOperand, rightOperand);

    // Then
    Assert.Equal(expectedResult, result);
  }
}
```

Cela exécutera autant d'instant de cas de tests que de `[InlineData]`, ce cas de test en est en fait quatres !

## Avec `TheoryData`

`TheoryData` Permet de passer des données à un test paramètrisé qui ne sont pas des constantes de compilation, et ce de façon fortement typée.

```
public class CalculatorTests {

  public static TheoryData<int, int, int> Data =>
    new TheoryData<int, int, int>
        {
            { 1, 2, 3 },
            { -4, -6, -10 },
            { -2, 2, 0 },
            { int.MinValue, -1, int.MaxValue }
        };


  [Theory]
  [MemberData(nameof(Data))]
  public void Add_ValidOperands_ReturnsCorrectResult(int leftOperand, int rightOperand, int
expectedResult){
  // Given

    var calculator = new Calculator();

    // When
    int result = calculator.Add(leftOperand, rightOperand);

    // Then
    Assert.Equal(expectedResult, result);
  }
}
```

# Assertions

XUnit fourni un bon nombre de fonctions d'assertion, comme `Assert.Equal` ou `Assert.Throws`, mais certains librairies comme FluentAssertions existent pour faire des assertions un petit peu plus expressives :

```
string actual = "ABCDEFGHI";
actual.Should().StartWith("AB").And.EndWith("HI").And.Contain("EF").And.HaveLength(9);
```

```
IEnumerable<int> numbers = new[] { 1, 2, 3 };
numbers.Should().OnlyContain(n => n > 0);
numbers.Should().HaveCount(4, "because we thought we put four items in the collection");
```