SOAT

→ Digitalize society

Formation F♯ 5.0

Orienté-objet

Décembre 2021

SOAT.FR

About me

Romain DENEAU

- ➔ SOAT depuis 2009
- ➔ Senior Developer C♯ F♯ TypeScript
- ➔ Passionné de Craft
- ➔ Auteur sur le blog de SOAT

DeneauRomain

rdeneau

Introduction

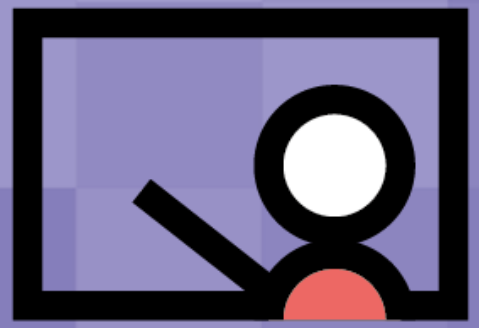
En F♯, orienté-objet parfois + pratique que style fonctionnel.

Briques permettant l'orienté-objet en F♯ :

1. Membres
    - ➔ Méthodes, propriétés, opérateurs
    - ➔ Consistent à attacher des fonctionnalités directement dans le type
    - ➔ Permettent d'encapsuler l'état (en particulier mutable) de l'objet
    - ➔ S'utilisent avec la notation "pointée" `my-object.my-member`
2. Interfaces et classes
    - ➔ Supports de l'abstraction par héritage

Sommaire

1. Membres : méthodes, propriétés, opérateurs
2. Extensions de type
3. Classe, structure
4. Interface
5. Expression objet



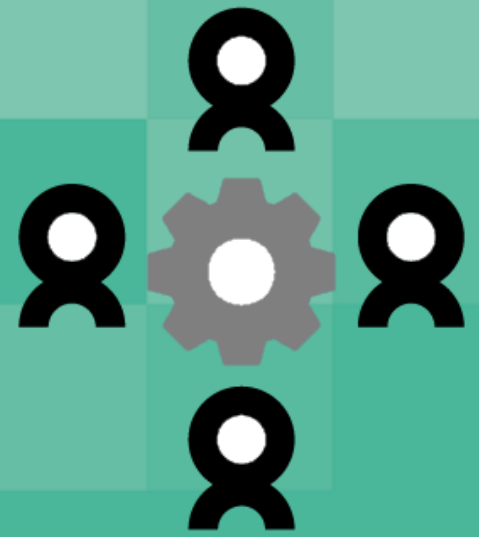
Polymorphisme

4e pilier de l'orienté-objet

En fait, il existe plusieurs polymorphismes. Les principaux :

1. Par sous-typage : celui évoqué classiquement avec l'orienté-objet
   → Type de base définissant membres abstraits ou virtuels
   → Sous-types en héritant et implémentant ces membres
2. Ad hoc/overloading → surcharge de membres de même nom
3. Paramétrique → génériques en C♯, Java, TypeScript
4. Structurel/duck-typing → SRTP en F♯, typage structurel en TypeScript
5. Higher-kinded → classes de type en Haskell

1. Les Membres



Membres

Éléments complémentaires dans définition d'un type *(classe, record, union)*

- ➔ *(Événement)*
- ➔ Méthode
- ➔ Propriété
- ➔ Propriété indexée
- ➔ Surcharge d'opérateur

Membres statiques et d'instance

Membre statique : `static member member-name ...`

Membre d'instance :

→ Membre concret : `member self-identifier.member-name ...`

→ Membre abstrait : `abstract member member-name : type-signature`

→ Membre virtuel = nécessite 2 déclarations

1. Membre abstrait
2. Implémentation par défaut : `default self-identifier.member-name ...`

→ Surcharge d'un membre virtuel : `override self-identifier.member-name ...`

☝ `member-name` en PascalCase *(convention .NET)*

☝ Pas de membre `protected` ou `private` !

Self identifier

En C♯, Java, TypeScript : `this`
En VB : `Me`
En F♯ : au choix → `this`, `self`, `me`, n'importe quel *identifier* valide…

Définissable de 3 manières complémentaires :

1. Pour le constructeur primaire : avec `as` → `type MyClass() as self = ...`
2. Pour un membre : `member me.Introduce() = printfn $"Hi, I'm {me.Name}"`
3. Pour un membre ne l'utilisant pas : avec `_` → `member _.Hi() = printfn "Hi!"`

Appeler un membre

💡 Mêmes règles quand C♯

Appeler un membre statique
→ Préfixer par le nom du type : `type-name.static-member-name`

Appeler un membre d'instance à l'intérieur du type
→ Préfixer avec *self-identifier* : `self-identifier.instance-member-name`

Appeler un membre d'instance depuis l'extérieur
→ Préfixer avec le nom de l'instance : `instance-name.instance-member-name`

Méthode

Méthode ≃ Fonction attachée directement à un type

2 formes de déclaration des paramètres :

1. Paramètres curryfiés = Style FP
2. Paramètres en tuple = Style OOP
    - ➔ Meilleure interop avec C♯
    - ➔ Seul mode autorisé pour les constructeurs
    - ➔ Support des paramètres nommés, optionnels, en tableau
    - ➔ Support des surcharges *(overloads)*

Méthode (2)

```fsharp
// (1) Forme en tuple (la + classique)
type Product = { SKU: string; Price: float } with
    member this.TupleTotal(qty, discount) =
        (this.Price * float qty) - discount  // (A)

// (2) Forme currifiée
type Product' =
    { SKU: string; Price: float }
    member me.CurriedTotal qty discount =
        (me.Price * float qty) - discount  // (B)
```

☝ `with` nécessaire en ① mais pas en ② à cause de l'indentation
→ `end` peut terminer le bloc commencé avec `with`

☝ `this.Price` Ⓐ et `me.Price` Ⓑ
→ Accès à l'instance via le *self-identifier* défini par le membre

Arguments nommés

Permet d'appeler une méthode tuplifiée en spécifiant le nom des paramètres :

```F#
type SpeedingTicket() =
    member _.SpeedExcess(speed: int, limit: int) =
        speed - limit

    member x.CalculateFine() =
        if x.SpeedExcess(limit = 55, speed = 70) < 20 then 50.0 else 100.0
```

Pratique pour :

→ Clarifier un usage pour le lecteur ou le compilateur (en cas de surcharges)

→ Choisir l'ordre des arguments

→ Ne spécifier que certains arguments, les autres étant optionnels

☝️ Les arguments *après un argument nommé* sont forcément nommés eux-aussi

Paramètres optionnels

Permet d'appeler une méthode tuplifiée sans spécifier tous les paramètres.

Paramètre optionnel :
- Déclaré avec `?` devant son nom → `?arg1: int`
- Dans le corps de la méthode, wrappé dans une `Option` → `arg1: int option`
  → On peut utiliser `defaultArg` pour indiquer la **valeur par défaut**
  → Mais la valeur par défaut n'apparaît pas dans la signature !

Lors de l'appel de la méthode, l'argument est spécifiable au choix :
- Directement dans son type → `M(arg1 = 1)`
- Wrappé dans une `Option` si nommé avec préfixe `?` → `M(?arg1 = Some 1)`

☝️ Autre syntaxe pour interop .NET : `[<Optional; DefaultParameterValue( ... )>] arg`

Paramètres optionnels : exemples

```fsharp
type DuplexType = Full | Half

type Connection(?rate: int, ?duplex: DuplexType, ?parity: bool) =
    let duplex = defaultArg duplex Full
    let parity = defaultArg parity false
    let defaultRate = match duplex with Full → 9600 | Half → 4800
    let rate = defaultArg rate defaultRate
    do printfn "Baud Rate: %d • Duplex: %A • Parity: %b" rate duplex parity

let conn1 = Connection(duplex = Full)
let conn2 = Connection(?duplex = Some Half)
let conn3 = Connection(300, Half, true)
```

F#

☝ Noter le *shadowing* des paramètres par des variables de même nom

```fsharp
let parity (* bool *) = defaultArg parity (* bool option *) Full
```

Tableau de paramètres

Permet de spécifier un nombre variable de paramètres de même type
→ Via attribut `System.ParamArray` sur le **dernier** argument de la méthode

```fsharp
open System

type MathHelper() =
    static member Max([<ParamArray>] items) =
        items ▷ Array.max

let x = MathHelper.Max(1, 2, 4, 5)  // 5
```

F#

💡 Équivalent en C♯ de `public static T Max<T>(params T[] items)`

Appeler méthode C♯ *TryXxx()*

❓ Comment appeler en F♯ une méthode C♯ `bool TryXxx(args, out T outputArg)` ?
(Exemple : `int.TryParse`, `IDictionnary::TryGetValue`)

- ➔ 👎 Utiliser équivalent F♯ de `out outputArg` mais utilise mutation 🤮
- ➔ ✅ Ne pas spécifier l'argument `outputArg`
  - ➔ Change le type de retour en tuple `bool * T`
  - ➔ `outputArg` devient le 2e élément de ce tuple

```fsharp
match System.Int32.TryParse text with
| true, i  → printf $"It's the number {value}."
| false, _ → printf $"{text} is not a number."
```
F#

Appeler méthode *Xxx(tuple)*

❓ Comment appeler une méthode dont 1er param est lui-même un tuple ?!

Essayons :

```fsharp
let friendsLocation = Map.ofList [ (0,0),"Peter" ; (1,0),"Jane" ]
// Map<(int * int), string>
let peter = friendsLocation.TryGetValue (0,0)
// 💥 error FS0001: expression censée avoir le type `int * int`, pas `int`
```

💡 Explications : `TryGetValue(0,0)` = appel méthode en mode tuplifié
→ Spécifie 2 paramètres, `0` et `0`.
→ `0` est un `int` alors qu'on attend un tuple `int * int` !

Appeler méthode *Xxx(tuple)* - Solutions

1. 😕 Doubles parenthèses, mais syntaxe confusante

   ➔ `friendsLocation.TryGetValue((0,0))`

2. 😕 *Backward pipe*, mais confusant aussi

   ➔ `friendsLocation.TryGetValue <| (0,0)`

3. ✅ Utiliser une fonction plutôt qu'une méthode

   ➔ `friendsLocation |> Map.tryFind (0,0)`

Méthode *vs* Fonction

Fonctionnalité	Fonction	Méthode currifiée	Méthode tuplifiée
Application partielle	✅ oui	✅ oui	❌ non
Arguments nommés	❌ non	❌ non	✅ oui
Paramètres optionnels	❌ non	❌ non	✅ oui
Tableau de paramètres	❌ non	❌ non	✅ oui
Surcharge / *overload*	❌ non	❌ non	✅ oui ①

① Si possible, préférer paramètres optionnels à surcharge

Méthode *vs* Fonction (2)

Fonctionnalité	Fonction	Méthode statique	Méthode d'instance
Nommage	camelCase	PascalCase	PascalCase
Support du `inline`	✅ oui	✅ oui	✅ oui
Récursive	✅ si `rec`	✅ oui	✅ oui
Inférence de `x` dans	`f x` → ✅ oui	—	`x.M()` → ❌ non
Passable en argument	✅ oui : `g f`	✅ oui : `g T.M`	❌ non : `g x.M` ①

① Alternative : wrappée dans lambda → `g (fun x → x.M())`

Propriétés

≃ Sucre syntaxique masquant un *getter* et/ou un *setter*
→ Permet d'utiliser la propriété comme s'il s'agissait d'un champ

2 façons de déclarer une propriété :

- Déclaration **explicite** : en relation avec un *backing field*
  - → *Getter* : `member this.Property = expression`
  - → Autres : verbeux (*détails*) 👉 Préférer méthodes explicites
- Déclaration **automatique** : *backing field* implicite
  - → *Read-only* : `member val Property = value`
  - → *Read/write* : `member val Property = value with get, set`

☝️ *Getter* évalué à chaque appel ≠ *Read-only* initialisé à la construction

Propriétés - exemple

```F#
type Person = { First: string; Last: string } with
    member this.FullName = // Getter
        $"{this.Last.ToUpper()} {this.First}"

let joe = { First = "Joe"; Last = "Dalton" }
let s = joe.FullName  // "DALTON Joe"
```

Propriétés et pattern matching

⚠️ Les propriétés ne sont pas déconstructibles.
→ Peuvent participer à un pattern matching que dans partie `when`

```fsharp
type Person = { First: string; Last: string } with
    member this.FullName = // Getter
        $"{this.Last.ToUpper()} {this.First}"

let joe = { First = "Joe"; Last = "Dalton" }
let { First = first } = joe  // val first : string = "Joe"
let { FullName = x } = joe
// 💥 ~~~~~~~~ Error FS0039: undefined record label 'FullName'

let salut =
    match joe with
    | _ when joe.FullName = "DALTON Joe" → "Salut, Joe !"
    | _ → "Bonjour !"
// val salut : string = "Salut, Joe !"
```

Propriétés indexées

Permet accès par indice, comme si la classe était un tableau : `instance.[index]`
→ Intéressant pour une collection ordonnée, pour masquer l'implémentation

Mise en place en déclarant membre `Item`

```fsharp
member self-identifier.Item
    with get(index) =
        get-member-body
    and set index value =
        set-member-body
```

F#

💡 Propriété *read-only* (*write-only*) → ne déclarer que le *getter* (*setter*)

☝️ Paramètre en tuple pour *getter* ≠ paramètres curryfiés *setter*

Propriétés indexées : exemple

F#

```fsharp
type Lang = En | Fr

type DigitLabel() =
    let labels = // Map<Lang, string[]>
        [| (En, [| "zero"; "one"; "two"; "three" |])
           (Fr, [| "zéro"; "un"; "deux"; "trois" |]) |] |> Map.ofArray

    member val Lang = En with get, set
    member me.Item with get(i) = labels.[me.Lang].[i]
    member _.En with get(i) = labels.[En].[i]

let digitLabel = DigitLabel()
let v1 = digitLabel.[1]     // "one"
digitLabel.Lang ← Fr
let v2 = digitLabel.[2]     // "deux"
let v3 = digitLabel.En(2)   // "two"
// 💡 Notez la différence de syntaxe de l'appel à la propriété `En`
```

Slice

> Idem propriété indexée mais renvoie plusieurs valeurs

Définition : via méthode *(normale ou d'extension)* `GetSlice(?start, ?end)`

Usage : via opérateur `..`

F#
```fsharp
type Range = { Min: int; Max: int } with
    member this.GetSlice(min, max) =
        { Min = System.Math.Max(defaultArg min this.Min, this.Min)
        ; Max = System.Math.Min(defaultArg max this.Max, this.Max) }

let range = { Min = 1; Max = 5 }
let slice1 = range.[0..3] // { Min = 1; Max = 3 }
let slice2 = range.[2..]  // { Min = 2; Max = 5 }
```

Surcharge d'opérateur

Opérateur surchargé à 2 niveaux possibles :

1. Dans un module, sous forme de fonction
   - ➔ `let [inline] (operator-symbols) parameter-list = ...`
   - ➔ 👉 Cf. session sur les fonctions
   - ➔ ☝️ Limité : 1 seule surcharge possible
2. Dans un type, sous forme de membre
   - ➔ `static member (operator-symbols) (parameter-list) =`
   - ➔ Mêmes règles que pour la forme de fonction
   - ➔ 👍 Plusieurs surcharges possibles (N types × P *overloads*)

Surcharge d'opérateur : exemple

F#

```fsharp
type Vector(x: float, y: float) =
    member _.X = x
    member _.Y = y

    override me.ToString() =
        let format n = (sprintf "%+.1f" n)
        $"Vector (X: {format me.X}, Y: {format me.Y})"

    static member (*)(a, v: Vector) = Vector(a * v.X, a * v.Y)
    static member (*)(v: Vector, a) = a * v
    static member (~-)(v: Vector) = -1.0 * v
    static member (+) (v: Vector, w: Vector) = Vector(v.X + w.X, v.Y + w.Y)

let v1 = Vector(1.0, 2.0)   // Vector (X: +1.0, Y: +2.0)
let v2 = v1 * 2.0           // Vector (X: +2.0, Y: +4.0)
let v3 = 0.75 * v2          // Vector (X: +1.5, Y: +3.0)
let v4 = -v3                // Vector (X: -1.5, Y: -3.0)
let v5 = v1 + v4            // Vector (X: -0.5, Y: -1.0)
```

2. Extensions de type

Extension de type

Membres d'un type définis hors de son bloc `type` principal.

Chacun de ces membres est appelé une **augmentation**.

3 catégories d'extension :

- ➔ Extension intrinsèque
- ➔ Extension optionnelle
- ➔ Méthodes d'extension

Extension intrinsèque

Définie dans même fichier et même namespace que le type
→ Membres intégrés au type à la compilation, visibles par *Reflection*

💡 **Cas d'usage**

Déclarer successivement :
1. Type (ex : `type List`)
2. Module compagnon de ce type (ex : fonction `List.length list`)
3. Extension utilisant ce module compagnon (ex : membre `list.Length`)

👉 Façon + propre en FP de séparer les fonctionnalités des données
💡 Inférence de types marche mieux avec fonctions que membres

Extension intrinsèque - Exemple

```fsharp
namespace Example

type Variant =
    | Num of int
    | Str of string

module Variant =
    let print v =
        match v with
        | Num n → printf "Num %d" n
        | Str s → printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
    member x.Print() = Variant.print x
```

Extension optionnelle

Extension définie en-dehors du module/namespace/assembly du type étendu.

💡 Pratique pour les types dont la déclaration n'est pas modifiable directement, par exemple ceux issus d'une librairie.

```fsharp
module EnumerableExtensions

open System.Collections.Generic

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for _ in 1 .. n → x
        }
```

Extension optionnelle (2)

Compilation : en méthode statique → version simplifiée :

```C#
public static class Extensions
{
    public static IEnumerable<T> RepeatElements<T>(IEnumerable<T> xs, int n) { ... }
}
```

Usage : comme un vrai membre, après avoir importé son module :

```F#
open Extensions

let x = [1..3].RepeatElements(2) ▷ List.ofSeq
// [1; 1; 2; 2; 3; 3]
```

Extension optionnelle - Autre exemple

```F#
// File Person.fs
type Person = { First: string; Last: string }

// File PersonExtensions.fs
module PersonExtensions =
    type Person with
        member this.FullName =
            $"{this.Last.ToUpper()} {this.First}"

// Usage elsewhere
open PersonExtensions
let joe = { First = "Joe"; Last = "Dalton" }
let s = joe.FullName  // "DALTON Joe"
```

Extension optionnelle - Limites

- ➔ Doit être déclarée dans un module
- ➔ Pas compilée dans le type, pas visible par Reflection
- ➔ Membres visibles qu'en F#, invisibles en C#

Extension de type et surcharges

☝️ Implémenter des surcharges :
→ Recommandé dans la déclaration initiale du type ✅
→ Déconseillé dans une extension de type ⛔

```fsharp
type Variant = Num of int | Str of string with
    override this.ToString() = ... ✅

module Variant = ...

type Variant with
    override this.ToString() = ... ⚠️
    // Warning FS0060: Override implementations in augmentations are now deprecated ...
```

Extension de type et alias de type

Sont incompatibles :

```fsharp
type i32 = System.Int32

type i32 with
    member this.IsEven = this % 2 = 0
// 💥 Error FS0964: Les abréviations de type ne peuvent pas avoir d'augmentations
```

F#

💡 **Solution :** il faut utiliser le vrai nom du type

```fsharp
type System.Int32 with
    member this.IsEven = this % 2 = 0
```

F#

☝️ Les tuples F# tels que `int * int` ne peuvent pas être augmentés ainsi.
→ Mais on peut avec une méthode d'extension à la C# 📍

Extension de type - Limite

Extension autorisée sur type générique sauf quand contraintes diffèrent :

```fsharp
open System.Collections.Generic

type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
// 💥       ~~~~~~~~~~ Error FS0670
// Ce code n'est pas suffisamment générique. Impossible de généraliser la variable de type
// ^T when ^T: (static member get_Zero: → ^T) and ^T: (static member (+) : ^T * ^T → ^T)

// ☝️ Cette contrainte provient de `Seq.sum`
```

Solution : méthode d'extension à la C# 📌

Méthode d'extension

Méthode statique :

- Décorée de `[<Extension>]`
- Définie dans classe `[<Extension>]`
- Type du 1er argument = type étendu *(`IEnumerable<'T>` ci-dessous)*

```fsharp
namespace Extensions

open System.Collections.Generic
open System.Runtime.CompilerServices

[<Extension>]
type EnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs

// 💡 `inline` est nécessaire
```

Méthode d'extension - Exemple simplifié

```fsharp
open System.Runtime.CompilerServices

[<Extension>]
type EnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: seq<_>) = Seq.sum xs

let x = [1..3].Sum()
//-------------------------------
// Output en console FSI (syntaxe verbeuse) :
type EnumerableExtensions =
  class
    static member
      Sum : xs:seq<^a> -> ^a
              when ^a : (static member ( + ) : ^a * ^a -> ^a)
              and  ^a : (static member get_Zero : -> ^a)
  end
val x : int = 6
```

F#

Méthode d'extension - Décompil' en C#

Pseudo-équivalent en C# :

```C#
using System.Collections.Generic;

namespace Extensions
{
    public static class EnumerableExtensions
    {
        public static TSum Sum<TItem, TSum>(this IEnumerable<TItem> source) { ... }
    }
}
```

☝ **Note :** en vrai, il y a plein de `Sum()` dans LINQ pour chaque type : `int`, `float`…
→ *Code source*

Méthode d'extension - Tuples

On peut ajouter une méthode d'extension à tout tuple F# :

```fsharp
open System.Runtime.CompilerServices

[<Extension>]
type EnumerableExtensions =
    [<Extension>]
    // static member IsDuplicate : ('a * 'a) → bool when 'a : equality
    static member inline IsDuplicate((x, y)) =
        x = y

let b1 = (1, 1).IsDuplicate()  // true
let b2 = ("a", "b").IsDuplicate()  // false
```

Extensions - Comparatif

Fonctionnalité	Extension de type	Méthode d'extension
Méthodes	✅ instance, ✅ statique	✅ instance, ❌ statique
Propriétés	✅ instance, ✅ statique	❌ *Non supporté*
Constructeurs	✅ intrinsèque, ❌ optionnelle	❌ *Non supporté*
Étendre contraintes	❌ *Non supporté*	✅ *Supporte SRTP*

Extensions - Limites

Ne participent pas au polymorphisme :

- ➔ Pas dans table virtuelle
- ➔ Pas de membre `virtual`, `abstract`
- ➔ Pas de membre `override` *(mais surcharges 👌)*

Extensions *vs* classe partielle C♯

Fonctionnalité	Multi-fichiers	Compilé dans type	Tout type
Classe partielle C♯	✅ Oui	✅ Oui	Que `partial class`
Extens° intrinsèque	❌ Non	✅ Oui	✅ Oui
Extens° optionnelle	✅ Oui	❌ Non	✅ Oui

3. Classe & Structure

Classe

Classe en F♯ ≡ classe en C♯
→ Brique de base pour l'orienté-objet
→ Constructeur d'objets contenant des données de type défini et des méthodes

Définition d'une classe
→ Commence par `type` *(comme tout type en F♯)*
→ Nom de la classe généralement suivi du **constructeur primaire**

```F#
type CustomerName(firstName: string, lastName: string) =
    // Corps du constructeur primaire
    // Membres ...
```

☝️ Paramètres `firstName` et `lastName` visibles dans tout le corps de la classe

Classe générique

Paramètres génériques à spécifier car non inférés

```F#
type Tuple2_KO(item1, item2) = // ⚠️ 'item1' et 'item2': type 'obj' !
    // ...

type Tuple2<'T1, 'T2>(item1: 'T1, item2: 'T2) =  // 👌
    // ...
```

Classe : constructeur secondaire

Syntaxe pour définir un autre constructeur :

`new(argument-list) = constructor-body`

☝ Doit appeler le constructeur primaire !

```fsharp
type Point(x: float, y: float) =
    new() = Point(0, 0)
    // Membres ...
```

F#

☝ Paramètres des constructeurs : que en tuple, pas curryfiés !

Instanciation

Appel d'un des constructeurs, avec arguments en tuple
→ Ne pas oublier `()` si aucun argument, sinon on obtient une fonction !

Dans un `let` binding : `new` optionnel et non recommandé
→ `let v = Vector(1.0, 2.0)` 👌
→ `let v = new Vector(1.0, 2.0)` ❌

Dans un `use` binding : `new` obligatoire
→ `use d = new Disposable()`

Initialisation des propriétés

On peut initialiser des propriétés avec setter à l'instanciation
→ Les spécifier en tant que **arguments nommés** dans l'appel au constructeur
→ Les placer après les éventuels arguments du constructeur :

```F#
type PersonName(first: string) =
    member val First = first with get, set
    member val Last = "" with get, set

let p1 = PersonName("John")
let p2 = PersonName("John", Last="Paul")
let p3 = PersonName(first="John", Last="Paul")
```

💡 Équivalent de la syntaxe C# `new PersonName("John") { Last = "Paul" }`

Classe abstraite

Annotée avec `[<AbstractClass>]`

Un des membres est **abstrait** :

1. Déclaré avec mot clé `abstract`
2. Pas d'implémentation par défaut avec mot clé `default`
   (Sinon le membre est virtuel)

Héritage via mot clé `inherit`
→ Suivi de l'appel au constructeur de la classe de base

Classe abstraite : exemple

```F#
[<AbstractClass>]
type Shape2D() =
    member val Center = (0.0, 0.0) with get, set
    member this.Move(?deltaX: float, ?deltaY: float) =
        let x, y = this.Center
        this.Center ← (x + defaultArg deltaX 0.0,
                       y + defaultArg deltaY 0.0)
    abstract GetArea : unit → float
    abstract Perimeter : float  with get

type Square(side) =
    inherit Shape2D()
    member val Side = side
    override _.GetArea () = side * side
    override _.Perimeter = 4.0 * side

let o = Square(side = 2.0, Center = (1.0, 1.0))
printfn $"S={o.Side}, A={o.GetArea()}, P={o.Perimeter}"  // S=2, A=4, P=8
o.Move(deltaY = -2.0)
printfn $"Center {o.Center}"  // Center (1, -1)
```

Champs

Convention de nommage : camelCase

2 types de champs : implicite ou explicite

- ➔ Implicite ≃ Variable à l'intérieur du constructeur primaire
- ➔ Explicite ≡ Champ classique d'une classe en C♯ / Java

Champ implicite

Syntaxe :

- Variable : `[static] let [ mutable ] variable-name = expression`
- Fonction : `[static] let [ rec ] function-name function-args = expression`

☝ **Notes**

➔ Déclaré avant les membres de la classe

➔ Valeur initiale obligatoire

➔ Privé

➔ S'utilise sans devoir préfixer par le `self-identifier`

Champ implicite d'instance : exemple

```fsharp
type Person(firstName: string, lastName: string) =
    let fullName = $"{firstName} {lastName}"
    member _.Hi() = printfn $"Hi, I'm {fullName}!"

let p = Person("John", "Doe")
p.Hi()  // Hi, I'm John Doe!
```

Champ implicite statique : exemple

```fsharp
type K() =
    static let mutable count = 0

    // do binding exécuté à chaque construction
    do
        count ← count + 1

    member _.CreatedCount = count

let k1 = K()
let count1 = k1.CreatedCount  // 1
let k2 = K()
let count2 = k2.CreatedCount  // 2
```

Champ explicite

Déclaration du type, sans valeur initiale :

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

- ➔ `val mutable a: int` → champ publique
- ➔ `val a: int` → champ interne `a@` + propriété `a ⇒ a@`

Champ *vs* propriété

```fsharp
// Champs explicites readonly
type C1 =
    val a: int
    val b: int
    val mutable c: int
    new(a, b) = { a = a; b = b; c = 0 } // 💡 Constructeur 2ndaire "compacte"

// VS propriétés readonly ⇒ ordre inversé dans SharpLab : b avant a
type C2(a: int, b: int) =
    member _.A = a
    member _.B = b
    member _.C = 0

// VS propriétés auto-implémentées
type C3(a: int, b: int) =
    member val A = a
    member val B = b with get
    member val C = 0 with get, set
```

F#

Champ explicite ou implicite ou propriété

Champ explicite **peu utilisé** :
→ Ne concerne que les classes et structures
→ Utile avec fonction native manipulant la mémoire directement
(Car ordre des champs préservés - cf. SharpLab)
→ Besoin d'une variable `[<ThreadStatic>]`
→ Interaction avec classe F♯ de code généré sans constructeur primaire

Champ implicite - `let` binding
→ Variable intermédiaire lors de la construction

Autres cas d'usages → propriété auto-implémentée
→ Exposer une valeur → `member val`
→ Exposer un "champ" mutable → `member val ... with get, set`

Structures

Alternatives aux classes mais + limités / héritage et récursivité

Même syntaxe que pour les classes mais avec en plus :

- ➔ Soit attribut `[<Struct>]`
- ➔ Soit bloc `struct ... end` *(fréquent)*

```fsharp
type Point =
    struct
        val mutable X: float
        val mutable Y: float
        new(x, y) = { X = x; Y = y }
    end

let x = Point(1.0, 2.0)
```

F#

4. Les Interfaces

Interface - Syntaxe

Idem classe abstraite avec :

- Que des membres abstraits, définis par signature
- Sans l'attribut `[<AbstractClass>]`

```F#
type [accessibility-modifier] interface-name =
    abstract memberN : [ argument-typesN → ] return-typeN
```

- Nom d'une interface commence par `I` pour suivre convention .NET
- Les arguments peuvent être nommés *(sans parenthèses sinon* 💥*)*

```F#
type IPrintable =
    abstract member Print : format:string → unit
```

Interface - Implémentation

2 manières d'implémenter une interface :

1. Dans un type *(comme en C♯)*
2. Dans une expression objet 📍

Implémentation dans un type

```fsharp
type IPrintable =
    abstract member Print : unit → unit

type Range = { Min: int; Max: int } with
    interface IPrintable with
        member this.Print() = printfn $"[{this.Min}..{this.Max}]"
```

⚠️ **Piège :** mot clé `interface` en F♯
≠ mot clé `interface` en C♯, Java, TS
≡ mot clé `implements` Java, TS

Implémentation dans une expression objet

```fsharp
type IConsole =
    abstract ReadLine : unit → string
    abstract WriteLine : string → unit

let console =
    { new IConsole with
        member this.ReadLine () = Console.ReadLine ()
        member this.WriteLine line = printfn "%s" line }
```

Interface - Implémentation par défaut

F♯ 5.0 supporte les interfaces définissant des méthodes avec implémentations par défaut écrites en C♯ 8+ mais ne permet pas de les définir.

⚠️ Mot clé `default` : supporté que dans les classes, pas dans les interfaces !

Une interface F♯ est explicite

Implémentation d'une interface en F♯
≡ Implémentation explicite d'une interface en C♯

→ Les méthodes de l'interface ne sont consommables que par *upcasting* :

```fsharp
type IPrintable =
    abstract member Print : unit → unit

type Range = { Min: int; Max: int } with
    interface IPrintable with
        member this.Print() = printfn $"[{this.Min}..{this.Max}]"

let range = { Min = 1; Max = 5 }
(range :> IPrintable).Print()  // Opérateur `:>` de upcast 📌
// [1..5]
```

Implémentation d'une interface générique

```F#
type IValue<'T> =
    abstract member Get : unit → 'T

type BiValue() =
    interface IValue<int> with
        member _.Get() = 1
    interface IValue<string> with
        member _.Get() = "hello"

let o = BiValue()
let i = (o :> IValue<int>).Get() // 1
let s = (o :> IValue<string>).Get() // "hello"
```

Héritage

Défini avec mot clé `inherit`

```F#
type Base(x: int) =
    do
        printf "Base: "
        for i in 1..x do printf "%d " i
        printfn ""

type Child(y: int) =
    inherit Base(y * 2)
    do
        printf "Child: "
        for i in 1..y do printf "%d " i
        printfn ""

let child = Child(1)

// Base: 1 2 3 4
// Child: 1
```

5. Expression Objet

Expression objet

Expression permettant d'implémenter à la volée un type abstrait
→ Similaire à une classe anonyme en Java

```F#
let makeResource (resourceName: string) =
    printfn $"create {resourceName}"
    { new System.IDisposable with
        member _.Dispose() =
            printfn $"dispose {resourceName}" }
```

☝️ La signature de `makeResource` est `string → System.IDisposable`.

Implémenter 2 interfaces

Possible mais 2e interface non consommable facilement et sûrement

```fsharp
let makeDelimiter (delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member _.ToString(format: string, _: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value
      interface System.IComparable with
        member _.CompareTo(_) = -1 }

let o = makeDelimiter("<", ">", "abc")
// val o : System.IFormattable
let s = o.ToString("D", System.Globalization.CultureInfo.CurrentCulture)
// val s : string = "<abc>"
let i = (d :?> System.IComparable).CompareTo("cde")  // ! Dangereux
// val i : int = -1
```

6. Recommandations pour l'orienté-objet

Pas d'orienté-objet là où F♯ est bon

Inférence marche mieux avec fonction(objet) que objet.membre

Hiérarchie simple d'objets

❌ Éviter héritage

✅ Préférer type *Union* et *pattern matching* exhaustif, + simple en général

→ En particulier les types récursifs comme les arbres, épaulés par fonction `fold`

→ https://fsharpforfunandprofit.com/series/recursive-types-and-folds/

Égalité structurelle

❌ Éviter classe *(égalité par référence par défaut)*

✅ Préférer un *Record* ou une *Union*

❓ Envisager égalité structurelle custom / performance

→ https://www.compositional-it.com/news-blog/custom-equality-and-comparison-in-f/

Orienté-objet recommandé

1. Encapsuler état mutable → dans une classe
2. Grouper fonctionnalités → dans une interface
3. API expressive et user-friendly → méthodes tuplifiées
4. API F♯ consommée en C♯ → membres d'extension
5. Gestion des dépendances → injection dans constructeur
6. Dépasser limites des fonctions d'ordre supérieur

Classe pour encapsuler état mutable

```fsharp
// 😐 Encapsuler état mutable dans une closure → fonction impure contre-intuitif ⚠️
let counter =
    let mutable count = 0
    fun () →
        count ← count + 1
        count

let x = counter ()  // 1
let y = counter ()  // 2

// ✅ Encapsuler état mutable dans une classe
type Counter() =
    let mutable count = 0    // Champ privé
    member _.Next() =
        count ← count + 1
        count
```

Interface pour grouper fonctionnalités

```fsharp
let checkRoundTrip serialize deserialize value =
    value = (value ▷ serialize ▷ deserialize)
// val checkRoundTrip :
//   serialize:('a → 'b) → deserialize:('b → 'a) → value:'a → bool
//     when 'a : equality
```

`serialize` et `deserialize` forment un groupe cohérent
→ Les grouper dans un objet

```fsharp
let checkRoundTrip serializer data =
    data = (data ▷ serializer.Serialize ▷ serializer.Deserialize)
```

Interface pour grouper fonctionnalités (2)

💡 Préférer une interface à un *Record*

```fsharp
// ❌ Éviter : ce n'est pas un bon usage d'un Record
type Serializer<'T> = {
    Serialize: 'T → string
    Deserialize: string → 'T
}

// ✅ Recommandé
type Serializer =
    abstract Serialize<'T> : value: 'T → string
    abstract Deserialize<'T> : data: string → 'T
```

F#

→ Paramètres sont nommés dans les méthodes
→ Objet facilement instanciable avec une expression objet

API expressive

```fsharp
// ❌ Éviter

module Utilities =
    let name = "Bob"
    let add2 x y = x + y
    let add3 x y z = x + y + z
    let log x = ...
    let log' x retryPolicy = ...
```

```fsharp
// ✅ Préférer
[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
    static member Log(x, ?retryPolicy) = ...
```

F#

→ Méthode `Add` surchargée *vs* `add2`, `add3`

→ Une seule méthode `Log` avec paramètre optionnel `retryPolicy`

🔗 F♯ component design guidelines - Libraries used in C♯

API F♯ consommée en C♯ - Type

Ne pas exposer ce type tel quel :

```fsharp
type RadialPoint = { Angle: float; Radius: float }

module RadialPoint =
    let origin = { Angle = 0.0; Radius = 0.0 }
    let stretch factor point = { point with Radius = point.Radius * factor }
    let angle (i: int) (n: int) = (float i) * 2.0 * System.Math.PI / (float n)
    let circle radius count =
        [ for i in 0..count-1 → { Angle = angle i count; Radius = radius } ]
```

API F♯ consommée en C♯ - Type (2)

💡 Pour faciliter la découverte du type et l'usage de ses fonctionnalités en C♯

➔ Mettre le tout dans un namespace

➔ Augmenter le type avec fonctionnalités du module compagnon

```fsharp
namespace Fabrikam

type RadialPoint = { ... }
module RadialPoint =  ...

type RadialPoint with
    static member Origin = RadialPoint.origin
    static member Circle(radius, count) = RadialPoint.circle radius count ▷ List.toSeq
    member this.Stretch(factor) = RadialPoint.stretch factor this
```

API F♯ consommée en C♯ - Type (3)

👉 L'API consommée en C♯ est +/- équivalente à :

```C#
namespace Fabrikam
{
    public static class RadialPointModule { ... }

    public sealed record RadialPoint(double Angle, double Radius)
    {
        public static RadialPoint Origin ⇒ RadialPointModule.origin;

        public static IEnumerable<RadialPoint> Circle(double radius, int count) ⇒
            RadialPointModule.circle(radius, count);

        public RadialPoint Stretch(double factor) ⇒
            new RadialPoint(Angle@, Radius@ * factor);
    }
}
```

Gestion des dépendances - Technique FP

Paramétrisation des dépendances + application partielle

→ Marche à petite dose : peu de dépendances, peu de fonctions concernées

→ Sinon, vite pénible à coder et à utiliser 🤦

```fsharp
module MyApi =
    let function1 dep1 dep2 dep3 arg1 = doStuffWith dep1 dep2 dep3 arg1
    let function2 dep1 dep2 dep3 arg2 = doStuffWith' dep1 dep2 dep3 arg2
```

Gestion des dépendances - Technique OO

Injection de dépendances

→ Injecter les dépendances dans le constructeur de la classe

→ Utiliser ces dépendances dans les méthodes

👉 Offre une API + user-friendly 👍

```fsharp
type MyParametricApi(dep1, dep2, dep3) =
    member _.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1
    member _.Function2 arg2 = doStuffWith' dep1 dep2 dep3 arg2
```

F#

✅ Particulièrement recommandé pour encapsuler des **effets de bord** :

→ Connexion à une BDD, lecture de settings...

Gestion des dépendances - Techniques FP++

Dependency rejection = pattern sandwich
→ Rejeter dépendances dans couche Application, hors de couche Domaine
→ Puissant et simple 👍
→ … quand c'est adapté ❗

Monade *Reader*
→ Pour fans de Haskell, sinon trop disruptif 😱

Etc. https://fsharpforfunandprofit.com/posts/dependencies/

Limites des fonctions d'ordre supérieur

Mieux vaut passer un objet plutôt qu'une lambda
en paramètre d'une fonction d'ordre supérieure quand :

1. Lambda est une **commande** `'T → unit`
   ✅ Préférer déclencher un effet de bord via un objet
   → `type ICommand = abstract Execute : 'T → unit`
2. Arguments de la lambda pas explicites
   ❌ `let test (f: float → float → string) = ...`
   ✅ Solution 1 : type wrappant les 2 args `float`
   → `f: Point → string` avec `type Point = { X: float; Y: float }`
   ✅ Solution 2 : interface + méthode pour avoir paramètres nommés
   → `type IXxx = abstract Execute : x:float → y:float → string`

Limites des fonctions d'ordre supérieur (2)

3. Lambda "vraiment" générique

```F#
let test42 (f: 'T → 'U) =
    f 42 = f "42"
// ❌ ^^     ~~~~
// ^^ Cette construction rend le code moins générique :
//    paramètre de type 'T contraint de représenter le type `int`
// ~~ Type `int` attendu ≠ type `string` actuel
```

✅ Solution : wrapper la fonction dans un objet

```F#
type Func2<'U> =
    abstract Invoke<'T> : 'T → 'U

let test42 (f: Func2<'U>) =
    f.Invoke 42 = f.Invoke "42"
```

Merci 🙏

SOAT

→ Digitalize society

SOAT.FR