



→ Digitalize society



Formation F# 5.0

Types composites



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



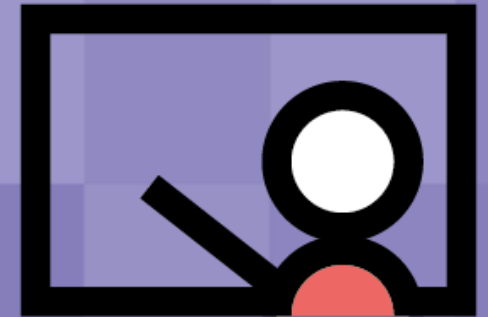
DeneauRomain



rdeneau

Sommaire

1. Généralités
2. Tuples
3. Records
4. Unions
5. Enums
6. Records anonymes
7. Types valeur



1. Généralités sur les types



Vue d'ensemble

Classifications des types .NET :

1. Types valeur vs types référence -- abrégés *TVal* et *TRef*
2. Types primitifs vs types composites
3. Types génériques
4. Types créés à partir de valeurs littérales
5. Types algébriques : somme vs produit

Types composites

Créés par combinaison d'autres types

Catégorie	Version	Nom	TRef	TVal
Types .NET		<code>class</code>	✓	✗
		<code>struct</code> , <code>enum</code>	✗	✓
Spécifiques C#	C# 3.0	Type anonyme	✓	✗
	C# 7.0	<i>Value tuple</i>	✗	✓
	C# 9.0	<code>record (class)</code>	✓	✗
	C# 10.0	<code>record struct</code>	✗	✓
Spécifiques F#		<i>Tuple, Record, Union</i>	<i>Opt-in</i>	<i>Opt-out</i>
	F# 4.6	<i>Record anonyme</i>	<i>Opt-in</i>	<i>Opt-out</i>

Types composites (2)

Peuvent être génériques (sauf `enum`)

Localisation :

- *Top-level* : `namespace`, *top-level* module F#
- *Nested* : `class` (C#), `module` (F#)
- Non définissables dans méthode (C#) ou valeur simple / fonction (F#) !

En F# toutes les définitions de type se font avec mot clé `type`

- y compris les classes, les enums et les interfaces !
- mais les tuples n'ont pas besoin d'une définition de type

Particularité des types F# / types .NET

Tuple, Record, Union sont :

- Immuables
- Non nullables
- Égalité et comparaison structurelles (*sauf avec champ fonction*)
- `sealed` : ne peuvent pas être hérités
- Déconstruction, avec même syntaxe que construction !

Reflète approches différentes selon paradigme :

- FP : focus sur les données organisées en types
- OOP : focus sur les comportements, possiblement polymorphiques

Types à valeurs littérales

Valeurs littérales = instances dont le type est inféré

- Types primitifs : `true (bool) · "abc" (string) · 1.0m (decimal)`
- Tuples C# / F# : `(1, true)`
- Types anonymes C# : `new { Name = "Joe", Age = 18 }`
- Records F# : `{ Name = "Joe"; Age = 18 }`

👉 Note :

- Les types doivent avoir été définis au préalable !
- Sauf tuples et types anonymes C#

Types algébriques

“ Types composites, combinant d'autres types par produit ou par somme. ”

Soit les types **A** et **B**, alors on peut créer :

- Le type produit **A × B** :
 - Contient 1 composante de type **A** ET 1 de type **B**
 - Composantes anonymes ou nommées
- Le type somme **A + B** :
 - Contient 1 composante de type **A** OU 1 de type **B**

Idem par extension les types produit / somme de N types

Pourquoi les termes "Somme" et "Produit" ?

Soit $N(T)$ le nombre de valeurs dans le type T , par exemples :

- `bool` → 2 valeurs : `true` et `false`
- `unit` → 1 valeur `()`

Alors :

- Le nombre de valeurs dans le type somme $A + B$ est $N(A) + N(B)$.
- Le nombre de valeurs dans le type produit $A \times B$ est $N(A) * N(B)$.

Types algébriques vs Types composites

Type <i>custom</i>	Somme	Produit	Composantes nommées
<code>enum</code>	✓	✗	—
<i>Union F#</i>	✓	✗	—
<code>class</code> ★, <code>interface</code> , <code>struct</code>	✗	✓	✓
<i>Record F#</i>	✗	✓	✓
<i>Tuple F#</i>	✗	✓	✗

★ Classes + variations C# : type anonyme, *Value tuple* et `record`

👉 En C#, pas de type somme sauf enum, très limité / type union 📌

Abréviation de type

Alias d'un autre type : `type [name] = [existingType]`

Différents usages :

```
// Documenter le code voire éviter répétitions
type ComplexNumber = float * float
type Addition<'num> = 'num -> 'num -> 'num // ➡ Marche aussi avec les génériques

// Découpler (en partie) usages / implémentation pour faciliter son changement
type ProductCode = string
type CustomerId = int
```

F#

⚠ Effacée à la compilation → ~~type safety~~

→ Compilateur autorise de passer `int` à la place de `CustomerId` !

2. ■ Les Tuples



Tuples : points clés

- Types à valeurs littérales
- Types "anonymes" mais on peut leur définir des alias
- Types produit par excellence
 - Signe `*` dans signature `A * B`
 - **Produit cartésien** des ensembles de valeurs de A et de B
- Nombre d'éléments:
 - 🙌 2 ou 3 (`A * B * C`)
 - ⚠️ > 3 : possible mais préférer *Record*
- Ordre des éléments est important
 - Si `A` \neq `B`, alors `A * B` \neq `B * A`

Tuples : construction

- Syntaxe des littéraux : `a,b` ou `a, b` ou `(a, b)`
 - Virgule `,` caractéristique des tuples
 - Espaces optionnels
 - Parenthèses `()` peuvent être nécessaires
- ⚠ Piège : séparateur différent entre un littéral et sa signature
 - `,` pour littéral
 - `*` pour signature
 - Ex : `true, 1.2` → `bool * float`

Tuples : déconstruction

- Même syntaxe que construction
 - mais « de l'autre côté du `=` »
- Tous les éléments doivent apparaître dans la déconstruction
 - Utiliser la discard `_` pour ignorer l'un des éléments

```
let point = 1.0, 2.5
let x, y = point

let x, y = 1, 2, 3 // ✨ Erreur FS0001: Incompatibilité de type...
                  // ... Les tuples ont des longueurs différentes de 2 et 3

let result = System.Int32.TryParse("123") // (bool * int)
let _, value = result // Ignore le "bool"
```

F#

Tuples en pratique

Utiliser un tuple pour une structure de données :

- Petite : 2 à 3 éléments
- Légère : pas besoin de nom pour les éléments
- Locale : échange local de données qui n'intéresse pas toute la *codebase*
- Renvoyer plusieurs valeurs - cf. `Int32.TryParse`

Tuple immuable : les modifications se font en créant un nouveau tuple

```
let addOneToTuple (x,y,z) = (x+1,y+1,z+1)
```

F#

Tuples en pratique (2)

Égalité structurelle, mais uniquement entre 2 tuples de même signature !

```
(1,2) = (1,2)           // true
(1,2) = (0,0)           // false
(1,2) = (1,2,3)         // ✨ Erreur FS0001: Incompatibilité de type ...
                        // ... Les tuples ont des longueurs différentes de 2 et 3
(1,2) = (1,(2,3))        // ✨ Erreur FS0001: Cette expression était censée avoir le type `int`
                        // ... mais elle a ici le type `a * b`
```

F#

Imbrication de tuples grâce aux `()`

```
let doublet = (true,1), (false,"a") // (bool * int) * (bool * string) → pair de pairs
let quadruplet = true, 1, false, "a" // bool * int * bool * string → quadruplet
doublet = quadruplet                 // ✨ Erreur FS0001: Incompatibilité de type ...
```

F#

Tuples : pattern matching

Patterns reconnus avec les tuples :

```
let print move =  
    match move with  
    | 0, 0 → "No move" // Constante 0  
    | 0, y → $"Vertical {y}" // Variable y (≠ 0)  
    | x, 0 → $"Horizontal {x}"  
    | x, y when x = y → $"Diagonal {x}" // Condition x et y égaux  
    // `x, x` n'est pas un pattern reconnu !  
    | x, y → $"Other ({x}, {y})"
```

F#

👉 Notes :

- Les patterns sont à ordonner du + spécifique au + générique
- Le dernier pattern `(x, y)` correspond au pattern par défaut (obligatoire)

Paires

- Tuples à 2 éléments
- Tellement courant que 2 helpers leur sont associés :
 - `fst` comme *first* pour extraire le 1^{er} élément de la paire
 - `snd` comme *second* pour extraire le 2^{ème} élément de la paire
 - ⚠ Ne marche que pour les paires

```
let pair = ('a', "b")  
fst pair // 'a' (char)  
snd pair // "b" (string)
```

F#

Pair Quiz

1. Comment implémenter soi-même `fst` et `snd` ?

```
let fst ... ?  
let snd ... ?
```

F#

2. Quelle est la signature de cette fonction ?

```
let toList (x, y) = [x; y]
```

F#



Pair Quiz

1. Implémenter soi-même `fst` et `snd` ?

```
let inline fst (x, _) = x // Signature : 'a * 'b → 'a  
let inline snd (_, y) = y // Signature : 'a * 'b → 'b
```

F#

- Déconstruction avec *discard*, le tout entre `()`
- Fonctions peuvent être `inline`



Pair Quiz

2. Signature de `toList` ?

```
let inline toList (x, y) = [x; y]
```

F#

- Renvoie une liste avec les 2 éléments de la paire
- Les éléments sont donc du même type
- Ce type est quelconque → générique `'a`

Réponse : `x: 'a * y: 'a → 'a list`

→ Soit le type `'a * 'a → 'a list`



Tuple **struct**

- Littéral : `struct(1, 'b', "trois")`
- Signature : `struct (int * char * string)`
- Usage : optimiser performance

 <https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/conventions#performance>

3 ■ Les *Records*



Record : points clés

Type produit, alternative au Tuple quand type imprécis sous forme de tuple

- Exemple : `float * float`
 - Point dans un plan ?
 - Coordonnées géographiques ?
 - Parties réelle et imaginaire d'un nombre complexe ?
- *Record* permet de lever le doute en nommant le type et ses éléments

```
type Point = { X: float; Y: float }  
type Coordinate = { Latitude: float; Longitude: float }  
type ComplexNumber = { Real: float; Imaginary: float }
```

F#

Record : déclaration

- Membres nommés en PascalCase, pas en camelCase
- Membres séparés `;` ou retours à la ligne
- Saut de ligne après `{` qu'en cas de membre additionnels

```
type PostalAddress =  
    { Address: string  
      City: string  
      Zip: string }  
:  
type PostalAddress =  
    {  
        Address: string  
        City: string  
        Zip: string  
    }  
    member x.ZipAndCity = $"{x.Zip} {x.City}"
```

F#

<https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/formatting>

- [#use-pascalcase-for-type-declarations-members-and-labels](#)
- [#formatting-record-declarations](#)

Record : instantiation

- Même syntaxe qu'un objet anonyme C# sans `new`
- Mais le record doit avoir été déclaré au-dessus !
- Membres peuvent être renseignés dans n'importe quel ordre
- Mais doivent tous être renseignés → pas de membres optionnels !

```
type Point = { X: float; Y: float }  
let point1 = { X = 1.0; Y = 2.0 }  
let point2 = { Y = 2.0; X = 1.0 } // ➡ Possible mais confusant ici  
let pointKo = { Y = 2.0 }         // ✨ Error FS0764  
// ~~~~~ Aucune assignation spécifiée pour le champ 'X' de type 'Point'
```

F#

⚠ **Piège** : Syntaxe similaire mais pas identique à celle de la déclaration

- `:` pour définir le type du membre
- `=` pour définir la valeur du membre

Record : instantiation (2)

- Les instances "longues" devraient être écrites sur plusieurs lignes
 - On peut aligner les `=` pour aider la lecture
- Les `{}` peuvent apparaître sur leur propre ligne
 - + facile à ré-indenter et à ré-ordonner (`Lackeys` avant `Boss`)

```
let rainbow =  
  { Boss = "Jeffrey"  
    Lackeys = ["Zippy"; "George"; "Bungle"] }  
  
let rainbow =  
  {  
    Boss      = "Jeffrey"  
    Lackeys   = ["Zippy"; "George"; "Bungle"]  
  }
```

F#

Record : déconstruction

Même syntaxe pour déconstruire un *Record* que pour l'instancier

💡 On peut ignorer certains champs
→ Rend explicite les champs utilisés 👍

```
let { X = x1 } = point1
let { X = x2; Y = y2 } = point1

// On peut aussi accéder aux champs via le point '.'
let x3 = point1.X
let y3 = point1.Y
```

F#

Record : déconstruction (2)

⚠ On ne peut pas déconstruire les membres additionnels (*propriétés*) !

```
type PostalAddress =  
    {  
        Address: string  
        City: string  
        Zip: string  
    }  
    member x.CityLine = $"{x.Zip} {x.City}"  
  
let address = { Address = ""; City = "Paris"; Zip = "75001" }  
  
let { CityLine = cityLine } = address    // ✨ Error FS0039  
//      ~~~~~ L'étiquette d'enregistrement 'CityLine' n'est pas définie  
let cityLine = address.CityLine         // 🔥 OK
```

F#

Record : inférence

- L'inférence de type ne marche pas quand on "dot" une `string`
- ... mais elle marche avec un *Record* ?!

```
type PostalAddress =  
    { Address: string  
      City   : string  
      Zip    : string }  
  
let department address =  
    address.Zip.Substring(0, 2) ▷ int  
    //      ^^^^ 💡 Permet d'inférer que address est de type `PostalAddress`  
  
let departmentKo zip =  
    zip.Substring(0, 2) ▷ int  
    // ~~~~~ Error FS0072
```

F#

Record : pattern matching

Fonction `inhabitantOf` donnant le nom des habitants à une adresse :

```
type Address = { Street: string; City: string; Zip: string }

let department { Zip = zip } = zip.Substring(0, 2) ▷ int

let inIleDeFrance departmentNumber =
    [ 75; 77; 78 ] @ [ 91..95 ] ▷ List.contains departmentNumber

let inhabitantOf address =
    match address with
    | { Street = "Pôle"; City = "Nord" } → "Père Noël"
    | { City = "Paris" } → "Parisien"
    | _ when department address = 78 → "Yvelinois"
    | _ when department address ▷ inIleDeFrance → "Francilien"
    | _ → "Français" // Le discard '_' sert de pattern par défaut (obligatoire)
```

F#

Record : conflit de noms

En F#, typage est nominal, pas structurel comme en TypeScript

→ Les mêmes étiquettes `First` et `Last` ci-dessous donnent 2 Records ≠

→ Mieux vaut écrire des types distincts ou les séparer dans ≠ modules

```
type Person1 = { First: string; Last: string }
type Person2 = { First: string; Last: string }
let alice = { First = "Alice"; Last = "Jones" } // val alice: Person2 ...
// (car Person2 est le type le + proche qui correspond aux étiquettes First et Last)

// ⚠ Déconstruction
let { First = firstName } = alice // Warning FS0667
// ~~~~~ Les étiquettes et le type attendu du champ de ce Record
// ~~~~~ ne déterminent pas de manière unique un type Record correspondant

let { Person2.Last = lastName } = alice // 🔥 OK avec type en préfixe
let { Person1.Last = lastName } = alice // ⚡ Error FS0001
// ~~~~~ Type 'Person1' attendu, 'Person2' reçu
```

F#

Record : modification

Record immuable mais facile de créer nouvelle instance ou copie modifiée

- Expression de ***copy and update*** d'un *Record*
- Syntaxe spéciale pour ne modifier que certains champs
- Multi-lignes si expression longue

```
let address2 = { address with Street = "Rue Vivienne" }

let { City = city; Zip = zip } = address
let address2' = { Street = "Rue Vivienne"; City = city; Zip = zip }
// address2 = address2'

let address3 =
    { address with
      City = "Lyon"
      Zip  = "69001" }
// address3 = address3'
```

F#

Record *copy-update* : C# / F# / JS

```
// Record C# 9.0  
address with { Street = "Rue Vivienne" }
```

C#

```
// F# copy and update  
{ address with Street = "Rue Vivienne" }
```

F#

```
// Object destructuring with spread operator  
{ ...address, street: "Rue Vivienne" }
```

JS

Record *copy-update* : limites

Lisibilité réduite quand plusieurs niveaux imbriqués

```
type Street = { Num: string; Label: string }
type Address = { Street: Street }
type Person = { Address: Address }

let person = { Address = { Street = { Num = "15"; Label = "rue Neuf" } } }

let person' =
    { person with
      Address =
        { person.Address with
          Street =
            { person.Address.Street with
              Num = person.Address.Street.Num + " bis" } } }
```

F#



Record **struct**

Attribut `[<Struct>]` permet de passer d'un type référence à un type valeur :

```
[<Struct>]  
type Point = { X: float; Y: float; Z: float }
```

F#

Pros/Cons d'une **struct** :

-  Performant car ne nécessite pas de *garbage collection*
-  Passée par valeur → pression sur la mémoire
- 👉 Adapté à un type "petit" en mémoire (~2-3 champs)

4. ■ Les Unions



Unions : points clés

- Terme exacte : « Union discriminée », *Discriminated Union (DU)*
- Types Somme : représente un **OU**, un **choix** entre plusieurs *Cases*
 - Même principe que pour une `enum` mais généralisé
- Chaque *case* doit avoir un *Tag (a.k.a Label)* -- en PascalCase **!**
 - C'est le **discriminant** de l'union pour identifier le *case*
- Chaque *case* **peut** contenir des données
 - Si Tuple, ses éléments peuvent être nommés -- en camelCase

```
type Billet =  
  | Adulte           // aucune donnée → ≈ singleton stateless  
  | Senior of int    // contient un 'int' (mais on ne sait pas ce que c'est)  
  | Enfant of age: int // contient un 'int' de nom 'age'  
  | Famille of Billet list // contient une liste de billet  
                          // type récursif -- pas besoin de 'rec'
```

F#

Unions : déclaration

Sur plusieurs lignes : 1 ligne / case

→ 🙌 Ligne indentée et commençant par `|`

Sur une seule ligne -- si déclaration reste **courte** !

→ 💡 Pas besoin du 1er `|`

```
open System

type IntOrBool =
    | Int32 of Int32           // 💡 Tag de même nom que ses données
    | Boolean of Boolean

type OrderId = OrderId of int // 🙌 Single-case union
                                // 💡 Tag de même nom que l'union parent
type Found<'T> = Found of 'T | NotFound // 💡 Type générique
```

F#

Unions : instantiation

Tag \simeq **constructeur**

→ Fonction appelée avec les éventuelles données du *case*

```
type Shape =  
    | Circle of radius: int  
    | Rectangle of width: int * height: int  
  
let circle = Circle 12           // Type: 'Shape', Valeur: 'Circle 12'  
let rect = Rectangle (4, 3)      // Type: 'Shape', Valeur: 'Rectangle (4, 3)'  
  
let circles = [1..4] > List.map Circle    // ➡ Tag employé comme fonction
```

F#

Unions : conflit de noms

Quand 2 unions ont des tags de même nom
→ Qualifier le tag avec le nom de l'union

```
type Shape =  
    | Circle of radius: int  
    | Rectangle of width: int * height: int  
  
type Draw = Line | Circle      // 'Circle' sera en conflit avec le tag de 'Shape'  
  
let draw = Circle              // Type='Draw' (type le + proche) -- ⚠ à éviter car ambigu  
  
// Tags qualifiés par leur type union  
let shape = Shape.Circle 12  
let draw' = Draw.Circle
```

F#

Unions : accès aux données

Uniquement via *pattern matching*

Matching d'un type Union est **exhaustif**

```
type Shape =  
    | Circle of radius: float  
    | Rectangle of width: float * height: float  
  
let area shape =  
    match shape with  
    | Circle r → Math.PI * r * r    // 💡 Même syntaxe que instantiation  
    | Rectangle (w, h) → w * h  
  
let isFlat = function  
    | Circle 0.                    // 💡 Constant pattern  
    | Rectangle (0., _)            // 💡 OR pattern  
    | Rectangle (_, 0.) → true  
    | Circle _                     //  
    | Rectangle _ → false
```

F#

Unions : *single-case*

Union avec un seul cas encapsulant un type (généralement primitif)

```
type CustomerId = CustomerId of int
type OrderId = OrderId of int

let fetchOrder (OrderId orderId) = // 💡 Déconstruction directe sans 'match'
    ...
```

F#

Assure *type safety* contrairement au simple type alias

→ Impossible de passer un `CustomerId` à une fonction attendant un `OrderId` 👍

Permet d'éviter *Primitive Obsession* à coût minime

Unions : style "enum"

Tous les cases sont vides = dépourvus de données

→ `enum` .NET 

L'instanciation et le pattern matching se font juste avec le *tag*

→ Le *tag* n'est plus une fonction mais une valeur ([singleton](#))

```
type Answer = Yes | No | Maybe
let answer = Yes

let print answer =
    match answer with
    | Yes    → printfn "Oui"
    | No     → printfn "Non"
    | Maybe  → printfn "Peut-être"
```

F#

5. ■ Les Enums



Vraies `enum` .NET

Enum : déclaration

- Ensemble de constantes de type entier (`byte`, `int`...)
- Syntaxe \neq union

```
type Color = Red | Green | Blue // Union
type ColorN = Red=1 | Green=2 | Blue=3 // Enum

type AnswerChar = Yes='Y' | No='N' // 💡 enum basée sur 'char'
type AnswerChar = Yes="Y" | No="N" // ⚡ Error FS0951
// ~~~~~ Littéraux énumérés doivent être de type 'int' ...

type File = a='a' | b='b' | c='c' // 💡 Membres d'enum peuvent être en camelCase
```

F#

Enum : usage

⚠ Contrairement aux unions, l'emploi d'un littéral d'enum est forcément qualifié

```
let answerKo = Yes // ✨ Error FS0039
//           ~~~ La valeur ou le constructeur 'Yes' n'est pas défini.
let answer = AnswerChar.Yes // 🟡 OK
```

F#

Cast via helpers `int` et `enum` (mais pas `char`) :

```
let redValue = int ColorN.Red // enum → int
let redAgain = enum<ColorN> redValue // int → enum via type générique
let red: ColorN = enum redValue // int → enum via annotation

// ⚠ Ne marche pas avec char enum
let ko = char AnswerChar.No // ✨ Error FS0001
let no: AnswerChar = enum 'N' // ✨ Error FS0001
```

F#

Enum : matching

⚠ Contrairement aux unions, le *pattern matching* n'est pas exhaustif

```
type ColorN = Red=1 | Green=2 | Blue=3 // Enum

let toHex color =
    match color with
    | ColorN.Red    → "FF0000"
    | ColorN.Green  → "00FF00"
    | ColorN.Blue   → "0000FF"
    // ⚠ Warning FS0104: Les enums peuvent accepter des valeurs en dehors des cas connus.
    // Par exemple, la valeur 'enum<ColorN> (0)' peut indiquer un cas non traité.

    // 💡 Pour enlever le warning, il faut ajouter un pattern générique
    | _ → invalidArg (nameof color) $"Color {color} not supported"
```

F#

Enum : flags

Même principe qu'en C# :

```
open System

[<FlagsAttribute>]
type PermissionFlags =
    | Read      = 1
    | Write     = 2
    | Execute   = 4

let permission = PermissionFlags.Read ||| PermissionFlags.Write

let canRead = permission.HasFlag PermissionFlags.Read
```

F#

💡 Noter l'opérateur `|||` : OU binaire (`|` en C#)

Enum vs Union

Type	Données	Qualification	Exhaustivité
Enum	Entières	Obligatoire	✗ Non
Union	Quelconques	Qu'en cas de conflit	✓ Oui

👉 Recommandation :

- Préférer une Union dans la majorité des cas
- Choisir une Enum pour :
 - Interop .NET
 - Besoin de lier données entières

6 ■ Record anonyme



Record anonyme

- Depuis F# 4.6 (*mars 2019*)
- Syntaxe : idem *Record* avec accolades "larges" `{| fields |}`
 - `{| Age: int |}` → signature
 - `{| Age = 15 |}` → instance
- Typage *inline* : pas besoin de pré-définir un `type` nommé
 - Alternative aux *Tuples*
- Autorisé en entrée/sortie de fonction
 - ≠ Type anonyme C#

Record anonyme : bénéfices

- Réduire *boilerplate*
- Améliorer interop avec systèmes externes (JavaScript, SQL...)

Exemples (*détaillés ensuite*) :

- Projection LINQ
- Personnalisation d'un record existant
- Sérialisation JSON
- Signature *inline*
- Alias par module

✓ Projection LINQ

💡 Sélectionner un sous-ensemble de propriétés

```
let names =  
    query {  
        for p in persons do  
            select { | Name = p.FirstName | }  
    }
```

F#

En C#, on utiliserait un type anonyme :

```
var names =  
    from p in persons  
    select new { Name = p.FirstName };
```

C#

<https://queil.net/2019/10/fsharp-vs-csharp-anonymous-records/>

✅ Personnalisation d'un record existant

💡 Un record anonyme peut être instancié à partir d'une instance de record

```
type Person = { Age: int; Name: string }
let william = { Age = 12; Name = "William" }

// Ajout d'un champ (Gender)
let william' = [{| william with Gender = "Male" |}]
              // [{| Age = 12; Name = "William"; Gender = "Male" |}]

// Modification de champs (Name, Age: int ⇒ float)
let jack = [{| william' with Name = "Jack"; Age = 16.5 |}]
           // [{| Age = 16.5; Name = "Jack"; Gender = "Male" |}]
```

F#

✅ S rialisation JSON

   Unions s rialis es dans un format pas pratique

```
#r "nuget: Newtonsoft.Json"
let serialize obj = Newtonsoft.Json.JsonConvert.SerializeObject obj

type CustomerId = CustomerId of int
type Customer = { Id: CustomerId; Age: int; Name: string; Title: string option }

serialize { Id = CustomerId 1; Age = 23; Name = "Abc"; Title = Some "Mr" }
```

F#

```
{
  "Id": { "Case": "CustomerId", "Fields": [ 1 ] }, //   
  "Age": 23,
  "Name": "Abc",
  "Title": { "Case": "Some", "Fields": [ "Mr" ] } //   
}
```

JSON

✅ S rialisation JSON (2)

💡 D finir un record anonyme pour s rialiser un *customer*

```
let serialisable customer =  
    let (CustomerId customerId) = customer.Id  
    [ customer with  
        Id = customerId  
        Title = customer.Title ▷ Option.toObj ]  
  
serialize (serialisable { Id = CustomerId 1; Age = 23; Name = "Abc"; Title = Some "Mr" })
```

F#

```
{  
  "Id": 1, // ✅  
  "Age": 23,  
  "Name": "Abc",  
  "Title": "Mr" // ✅  
}
```

JSON

✓ Signature *inline*

💡 Utiliser un record anonyme *inline* pour réduire charge cognitive

```
type Title = Mr | Mrs
type Customer =
{ Age : int
  Name : { First: string; Middle: string option; Last: string } // ➡
  Title: Title option }
```

F#

✅ Alias par module

```
module Api =  
    type Customer = // 🙌 Customer est un alias  
        { Id : System.Guid  
          Name : string  
          Age : int }  
  
module Dto =  
    type Customer =  
        { Id : System.Guid  
          Name : string  
          Age : int }  
  
let (customerApi: Api.Customer) = { Id = Guid.Empty; Name = "Name"; Age = 34 }  
let (customerDto: Dto.Customer) = customerApi // 🎉 Pas besoin de convertir
```

- 💡 Instant t : même type dans 2 modules
- 💡 Plus tard : facilite personnalisation des types par module

Record anonyme : limites

```
// Inférence limitée
let nameKo x = x.Name // ✨ Error FS0072: Lookup on object of indeterminate type ...
let nameOk (x: { Name:string }) = x.Name

// Pas de déconstruction
let x = { Age = 42 }
let { Age = age } = x // ✨ Error FS0039: The record label 'Age' is not defined
let { Age = age } = x // ✨ Error FS0010: Unexpected symbol '{' in let binding

// Pas de fusion
let banana = { Fruit = "Banana" }
let yellow = { Color = "Yellow" }
let banYelKo = { banana with yellow } // ✨ Error FS0609 ...
let banYelOk = { banana with Color = "Yellow" }

// Pas d'omission
let ko = { banYelOk without Color } // ✨ Mot clé 'without' n'existe pas
```

F#

Record anonyme : limites

```
// Pas du typage structurel ⇒ tous les champs sont requis
let capitaliseFruit (x: { Fruit: string }) = x.Fruit.ToUpper()
capitaliseFruit { Fruit = "Banana" } // 🐞 "BANANA"
capitaliseFruit { Fruit = "Banana"; Origin = "Réunion" } // ✨ Too much fields... [Origin]
```

F#

7. ■ Types valeur



Type composite *struct*

Type composite : peut être déclaré en tant que type valeur

- Instances stockées dans la **pile** (*stack*) plutôt que dans le tas (*heap*)
- Permet parfois de gagner en performance
- Plutôt adapté aux types compacts : peu de champs, peu de comportements
 - Attribut `[<Struct>]`
 - Mot clé `struct`
 - Structure

Attribut [**<Struct>**]

Pour *Record* et *Union*

À placer avant ou après le mot cle `type`

```
type [<Struct>] Point = { X: float; Y: float }
```

F#

```
[<Struct>]
```

```
type SingleCase = Case of string
```

Mot clé **struct**

Pour littéral de Tuple et *Record* anonyme

```
let t = struct (1, "a")  
// struct (int * string)  
  
let a = struct { Id = 1; Value = "a" }  
// struct { Id: int; Value: string }
```

F#

Structures

Alternatives aux classes 📌 mais + limités / héritage et récursivité

👉 Cf. session sur l'orienté-objet et les classes...

8 ■ Le Récap'





Récap' Quiz

```
// Relier types et concepts
type Color1 = int * int * int
type Color2 = Red | Green | Blue
type Color3 = Red=1 | Green=2 | Blue=3
type Color4 = { Red: int; Green: int; Blue: int }
type Color5 = [| Red: int; Green: int; Blue: int |]
type Color6 = Color of Red: int * Green: int * Blue: int
type Color7 =
    | RGB of { Red: int; Green: int; Blue: int }
    | HSL of { Hue: int; Saturation: int; Lightness: int }

// A. Alias
// B. Enum
// C. Record
// D. Record anonyme
// E. Single-case union
// F. Union
// G. Union enum-like
// H. Tuple
```

F#





Récap' Quiz

Types	Concepts
<code>type Color1 = int * int * int</code>	H. Tuple + A. Alias
<code>type Color2 = Red Green Blue</code>	G. Union enum-like
<code>type Color3 = Red=1 Green=2 Blue=3</code>	B. Enum
<code>type Color4 = { Red: int; Green: int... }</code>	C. Record
<code>type Color5 = { Red: int; Green: int... }</code>	D. Record anonymé + A. Alias
<code>type Color6 = Color of Red: int * ...</code>	E. Single-case union + H. Tuple
<code>type Color7 = RGB of { Red: int... } HSL...</code>	F. Union + C. Record

Composition de types

Création de nouveaux types ?

- ❌ Les types algébriques ne supportent pas l'héritage.
- ✅ Par composition, dans *sum/product type*
- 💡 Extension d'un *Record* en un *Record* anonyme avec champs en +

Combiner 2 unions ?

- ❌ Pas "aplatissable" comme en TypeScript ①
- ✅ Nouveau type union ②

```
type Noir = Pique | Trefle
type Rouge = Coeur | Carreau
type CouleurKo = Noir | Rouge // (1) ❌ ≠ Pique | Trefle | Coeur | Carreau
type Couleur = Noir of Noir | Rouge of Rouge // (2) ✅
let c1 = Noir Pique
```

F#

Conclusion

Beaucoup de façons de modéliser !

De quoi s'adapter :

- À tous les goûts ?
- En fait surtout au domaine métier !



Merci 🙏

SOAT

→ Digitalize society



SOAT.FR