SOAT

→ Digitalize society

Formation F♯ 5.0

Module & namespace

Décembre 2021

SOAT.FR

About me

Romain DENEAU

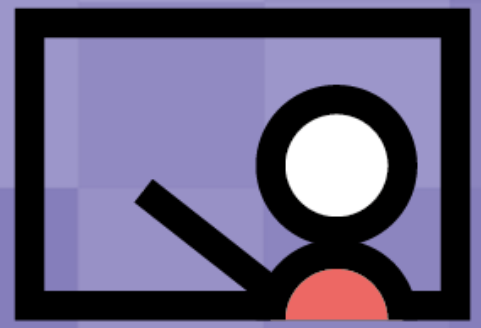
- ➔ SOAT depuis 2009
- ➔ Senior Developer C♯ F♯ TypeScript
- ➔ Passionné de Craft
- ➔ Auteur sur le blog de SOAT

DeneauRomain

rdeneau

Sommaire

- Vue d'ensemble
- Namespace
- Module



1. Vue d'ensemble

Similarités

Modules et namespaces permettent de :

- ➔ Organiser le code en zones de fonctionnalités connexes
- ➔ Éviter collision de noms

Différences

Propriété	Namespace	Module
Compilation .NET	`namespace`	`static class`
Type	*Top-level*	*Top-level* ou local
Contient	Modules, Types	Idem + Valeurs, Fonctions
Annotable	❌ Non	✅ Oui

Portée : Namespaces > Fichiers > Modules

Importer un module ou un namespace

💡 Comme en C♯ :

1. Soit qualifier les éléments individuellement à importer
2. Soit tout importer avec `open` *(placé en haut ou juste avant)*
   - ➔ En C♯ ≡ `using` pour un namespace
   - ➔ En C♯ ≡ `using static` pour un module *(classe statique .NET)*

```fsharp
// Option 1. Qualifier les usages
let result1 = Arithmetic.add 5 9

// Option 2. Importer tout le module
open Arithmetic
let result2 = add 5 9
```

Import : *shadowing*

L'import se fait sans conflit de nom mais en mode le dernier importé gagne
i.e. masque un précédent élément importé de même nom
→ ⚠️ Peut créer des problèmes difficiles à comprendre ❗

Exemple : erreur car fonction `add` appelée est celle du module `FloatHelper` !

```F#
module IntHelper =
    let add x y = x + y

module FloatHelper =
    let add x y : float = x + y

open IntHelper
open FloatHelper

let result = add 1 2 // 💥 Error FS0001: Le type 'float' ne correspond pas au type 'int'
```

2. Les Namespaces

Namespace : syntaxe

Syntaxe : `namespace [rec] [parent.]identifier`

→ `rec` pour récursif → *slide suivante*

→ `parent` permet de regrouper des namespaces

→ Tout ce qui suit appartiendra à ce namespace

Namespace : contenu

Un `namespace` F♯ ne peut contenir que des types et des modules locaux
→ Ne peut contenir ni valeurs ni fonctions

Par équivalence avec la compilation .NET
→ idem `namespace` C♯ qui ne peut contenir que des classes / enums

Quid des namespaces imbriqués ?
→ Se passe uniquement de manière déclarative `namespace [parent.]identifier`
→ 2 namespaces déclarés à la suite = ~~pas imbriqués~~ mais indépendants

Namespace : portée

- ➔ Plusieurs fichiers peuvent partager le même namespace
- ➔ Dans un fichier, on peut déclarer plusieurs namespaces
  - ➔ Ils ne seront pas imbriqués
  - ➔ Peut être source de confusion

☝ **Recommandation**

→ **Un seul** namespace par fichier, déclaré tout en haut

Namespace récursif

Permet d'étendre la visibilité par défaut unidirectionnelle, de bas en haut, pour que des éléments les uns au-dessous des autres se voient mutuellement

```fsharp
namespace rec Fruit

type Banana = { Peeled: bool }
    member this.Peel() =
        BananaHelper.peel  // `peel` non visible ici sans le `rec`

module BananaHelper =
    let peel banana = { banana with Peeled = true }
```

⚠️ **Inconvénients :** compilation + lente et risque de référence circulaire

☝️ **Recommandation :** pratique mais à utiliser avec parcimonie

3. Les Modules

Module : syntaxe

```fsharp
// Top-level module
module [accessibility-modifier] [qualified-namespace.]module-name
declarations

// Local module
module [accessibility-modifier] module-name =
    declarations
```

`accessibility-modifier` : restreint l'accessibilité
→ `public` *(défaut)*, `internal` *(assembly)*, `private` *(parent)*

Le nom complet (`[namespace.]module-name`) doit être unique
→ 2 fichiers ne peuvent pas déclarer des modules de même nom

Module top-level

- ➔ Doit être déclaré en 1er dans un fichier
- ➔ Contient tout le reste du fichier
  - ➔ Contenu non indenté
  - ➔ Ne peut pas contenir de namespace
- ➔ Peut être qualifié = inclus dans un namespace parent *(existant ou non)*

Module top-level implicite

- ➔ Si fichier sans module/namespace top-level
- ➔ Nom du module = nom du fichier
  - ➔ Sans l'extension
  - ➔ Avec 1ère lettre en majuscule
  - ➔ Ex : `program.fs` → `module Program`

Module local

- ➔ Syntaxe similaire au `let` → ne pas oublier :
  - ➔ Le signe `=` après le nom du module local ❗
  - ➔ D'indenter tout le contenu du module local
    - ➔ Non indenté = ne fait pas partie du module local

Module : contenu

Un module, local comme *top-level*, peut contenir :
→ types et sous modules locaux
→ valeurs, fonctions

Différence : l'indentation du contenu
→ Module top-level : contenu non indenté
→ Module local : contenu indenté

Equivalence module / classe statique

```fsharp
module MathStuff =
    let add x y  = x + y
    let subtract x y = x - y
```

F#

Ce module F# est équivalent à la classe statique suivante :

```csharp
public static class MathStuff
{
    public static int add(int x, int y) ⇒ x + y;
    public static int subtract(int x, int y) ⇒ x - y;
}
```

C#

Cf. sharplab.io

Module imbriqué

Comme en C# et les classes, les modules F# peuvent être imbriqués

```fsharp
module Y =
    module Z =
        let z = 5

printfn "%A" Y.Z.z
```

☝️ **Notes :**

➔ Intéressant avec module imbriqué privé pour isoler/regrouper

➔ Sinon, préférer une vue aplanie

Module top-level *vs* local

Propriété	Top-level	Local
Qualifiable	✅	❌
Signe `=` + contenu indenté	❌	✅ !

Module *top-level* → 1er élément déclaré dans un fichier
Sinon *(après un module/namespace top-level)* → module local

Module récursif

Même principe que namespace récursif
→ Pratique pour qu'un type et un module associé se voient mutuellement

☝️ **Recommandation :** limiter au maximum la taille des zones récursives

Annotation d'un module

2 attributs influencent l'usage d'un module

`[<AutoOpen>]`

Import du module en même temps que ns/module parent

→ 💡 Pratique pour "monter" valeurs/fonctions au niveau d'un namespace

→ 💡 Équivalent `open type` (F♯ 5) • 🔗 + d'infos

→ ⚠️ Pollue le *scope* courant

`[<RequireQualifiedAccess>]`

Empêche l'usage non qualifié des éléments d'un module

→ 💡 Pratique pour éviter le *shadowing* pour des noms communs : `add`, `parse`...

`AutoOpen`, `RequireQualifiedAccess` ou rien ?

Soit un type `Cart` avec son module compagnon `Cart`
→ Comment appeler la fonction qui ajoute un élément au panier ?

Si `addItem item cart` : `[<RequireQualifiedAccess>]` intéressant
→ pour forcer à avoir dans le code appelant `Cart.addItem`

Si `addItemToCart item cart` : `[<AutoOpen>]` intéressant
→ car `addItemToCart` est *self-explicit*

Module et Type

> Un module sert typiquement à regrouper des fonctions agissant sur un type de donnée bien spécifique.

2 styles, selon localisation type / module :

- ➔ Type défini avant le module → module compagnon
- ➔ Type défini dans le module

Module compagnon d'un type

- ➔ Style par défaut - cf. `List`, `Option`, `Result`...
- ➔ Bonne interop autres langages .NET
- ➔ Module peut porter le même nom que le type

```fsharp
type Person = { FirstName: string; LastName: string }

module Person =
    let fullName person = $"{person.FirstName} {person.LastName}"

let person = { FirstName = "John"; LastName = "Doe" }   // Person
person |> Person.fullName // "John Doe"
```

Module wrappant un type

- ➔ Type défini à l'intérieur du module
- ➔ On peut nommer le type `T` ou comme le module

```fsharp
module Person =
    type I = { FirstName: string; LastName: string }

    let fullName person = $"{person.FirstName} {person.LastName}"

let person = { FirstName = "John"; LastName = "Doe" }   // Person.T !
person ▷ Person.fullName // "John Doe"
```

Module wrappant un type (2)

Recommandé pour améliorer encapsulation
→ Constructeur du type `private`
→ Module contient un *smart constructor*

F#

```
module Person =
    type I = private { FirstName: string; LastName: string }

    let create first last =
        if System.String.IsNullOrWhiteSpace first
        then Error "FirstName required"
        else Ok { FirstName = first; LastName = last }

    let fullName person =
        $"{person.FirstName} {person.LastName}".Trim()

Person.create "" "Doe"                              // Error "LastName required"
Person.create "Joe" "" ▷ Result.map Person.fullName  // Ok "Joe"
```

Module *vs* namespace

Au niveau top-level :

- ➔ Préférer un namespace à un module
- ➔ Module top-level **implicite** envisageable pour fichier `.fsx`

Cf. docs.microsoft.com/.../fsharp/style-guide/conventions#organizing-code

4. 🍔 Quiz

Q1. Valide ou non ?

```fsharp
namespace A

let a = 1
```

A. Oui

B. Non

Q1. Valide ou non ?

```fsharp
namespace A

let a = 1
```

B. Non

→ Un namespace ne peut pas contenir de valeurs !

Q2. Valide ou non ?

```fsharp
namespace A

module B

let a = 1
```

A. Oui

B. Non

Q2. Valide ou non ?

```fsharp
namespace A

module B

let a = 1
```

B. Non

→ module B est ici top-level
→ interdit après un namespace

Q2 - Code équivalent valide

Option 1 : module top-level qualifié

```fsharp
module A.B

let a = 1
```

Option 2 : namespace + module local

```fsharp
namespace A

module B =
    let a = 1
```

Q3. Nom qualifié de `add` ?

```fsharp
namespace Common.Utilities

module IntHelper =
    let add x y = x + y
```

F#

A. `add`

B. `IntHelper.add`

C. `Utilities.IntHelper.add`

D. `Common.Utilities.IntHelper.add`

Q3. Nom qualifié de `add` ?

```fsharp
namespace Common.Utilities

module IntHelper =
    let add x y = x + y
```

F#

D. **`Common.Utilities.IntHelper.add`**

→ `IntHelper` pour le module parent
→ `Common.Utilities` pour le namespace racine

5. Le récap

Modules et namespaces

- ➔ Regrouper par fonctionnalité
- ➔ Scoper : namespaces > fichiers > modules

Propriété	Namespace	Module
Compilation .NET	`namespace`	`static class`
Type	*Top-level*	Local (ou *top-level*)
Contient	Modules, Types	Valeurs, Fonctions, Type, Sous-modules
`[<RequireQualifiedAccess>]`	❌ Non	✅ Oui *(vs shadowing)*
`[<AutoOpen>]`	❌ Non	✅ Oui mais prudence ❗

Merci 🙏

SOAT

→ Digitalize society

SOAT.FR