



→ Digitalize society



# Formation F# 5.0

## *Compléments sur les types*



**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



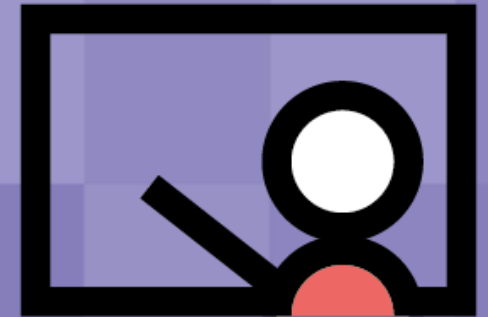
DeneauRomain



rdeneau

# Sommaire

1. Type `unit`
2. Génériques
3. Contraintes sur paramètres de type
4. Type flexible
5. Unités de mesure
6. *Casting* et conversion



# 1. Type unit



# Type `unit` : pourquoi ?

“ **Contrainte** : l'évaluation d'une expression doit produire une valeur. ”

Quid des expressions ne produisant pas de valeur significative ?

→ `void` ? Non, `void` en C#, Java n'est pas une valeur !

→ `null` ? Non, `null` n'est pas un type en .NET ! (*≠ TypeScript*)

Il faut donc un type spécifique, avec une seule valeur signifiant par convention :  
« Valeur non significative, à ignorer. »

→ Ce type s'appelle `unit`.

→ Sa valeur est notée `()`.

# Type `unit` et fonctions

Fonction `unit → 'T` ne prend pas de paramètre.

→ Ex : `System.DateTime.Now` (*fonction cachée derrière une propriété*)

Fonction `'T → unit` ne renvoie pas de valeur.

→ Ex : `printf`

👉 Fonctions impliquant un **effet de bord** !

# Type `unit` : ignorer une valeur

F# n'est pas un langage fonctionnel pur, sans effet de bord.  
Mais il encourage l'écriture de programme fonctionnel pur.

- 👉 **Règle** : Toute expression produisant une valeur doit être utilisée.
  - Sinon, le compilateur émet un warning (*sauf en console FSI*).
  - Exception: `()` est la seule valeur que le compilateur autorise à ignorer.

👉 **Avertissement** : ignorer une valeur est généralement un *code smell* en FP.

- 👉 Une expression avec effet de bord doit le signaler avec type de retour `unit`

# Type `unit` : fonction `ignore`

“ ? Comment (*malgré tout*) ignorer la valeur produit par une expression ? ”

Avec la fonction `ignore` :

→ Prend un paramètre d'entrée ignoré, "avalé"

→ Renvoie `unit`

```
let inline ignore _ = ()  
// Signature: 'T → unit
```

F#

Usage : `expression ▷ ignore`



# 2. ■ Les génériques



# Génériques

Fonctions et types peuvent être génériques, avec + de flexibilité qu'en C#.

Par défaut, généricité **implicite**

→ Inférée

→ Voire généralisée, grâce à « généralisation automatique »

Sinon, généricité peut être explicite ou résolue statiquement.

⚠ Notations différentes :

→ `'T` : paramètre de type générique

→ `^T` : paramètre de type résolu statiquement (*S RTP*)

# Généricité implicite

```
module ListHelper =  
    let singleton x = [x]  
    // val singleton : x:'a → 'a list  
  
    let couple x y = [x; y]  
    // val couple : x:'a → y:'a → 'a list
```

F#

## 👉 Explications :

- `singleton` : son argument `x` est quelconque → type générique `'a`  
→ Généralisation automatique
- `couple` : ses 2 arguments `x` et `y` doivent être du même type pour pouvoir être dans une liste → Inférence

# Généricité explicite

```
let print2 x y = printfn "%A, %A" x y  
// val print2 : x:'a → y:'b → unit
```

F#

→ Inférence de la généricité de `x` et `y` 👍

**? Comment indiquer que `x` et `y` doivent avoir le même type ?**

→ Besoin de l'indiquer explicitement :

```
let print2<'T> (x: 'T) (y: 'T) = printfn "%A, %A" x y  
// val print2 : x:'T → y:'T → unit
```

F#

# Généricité explicite - Forme inline

💡 **Astuce** : la convention en `'x` permet ici d'être + concis :

```
// AVANT
let print2<'T> (x: 'T) (y: 'T) = printfn "%A, %A" x y

// APRES
let print2 (x: 'T) (y: 'T) = printfn "%A, %A" x y
```

F#

# Généricité explicite - Type

La définition des types génériques est explicite :

```
type Pair = { Item1: 'T ; Item2: 'T }  
// ✨ ~ ~  
// Error FS0039: Le paramètre de type `T` n'est pas défini.  
  
// ✅ Records et unions avec 1 ou 2 paramètres de type  
type Pair<'T> = { Item1: 'T; Item2: 'T }  
  
type Tuple<'T, 'U> = { Item1: 'T; Item2: 'U }  
  
type Option<'T> = None | Some of 'T  
  
type Result<'TOk, 'TErr> =  
| Ok of 'TOk  
| Error of 'TErr
```

F#

# Généricité ignorée

Le *wildcard* `_` permet de remplacer un paramètre de type ignoré :

```
let printSequence (sequence: seq<'T>) = sequence ▷ Seq.iteri (printfn "%i: %A")  
// Versus  
let printSequence (sequence: seq<_>) = ...
```

F#

Encore + utile avec type flexible 📌 :

```
let tap action (sequence: 'TSeq when 'TSeq :> seq<_>) =  
    sequence ▷ Seq.iteri action  
    sequence  
// action:(int → 'a → unit) → sequence:'TSeq → 'TSeq when 'TSeq :> seq<'a>  
  
// Versus  
let tap action (sequence: #seq<_>) = ...
```

F#

# SRTP

F# propose deux catégories de types de paramètre :

- `'X` : type de paramètre générique (*vus jusqu'ici*)
- `^X` : type de paramètre résolu statiquement (*par le compilateur F#*)

👉 **SRTP** : abréviation fréquente de *Statically Resolved Type Parameter*



# SRTP - Le pourquoi

Sans SRTP :

```
let add x y = x + y
// val add : x:int → y:int → int
```

F#

→ Inférence du type `int` pour `x` et `y`, sans généralisation (aux `float` par ex.) !

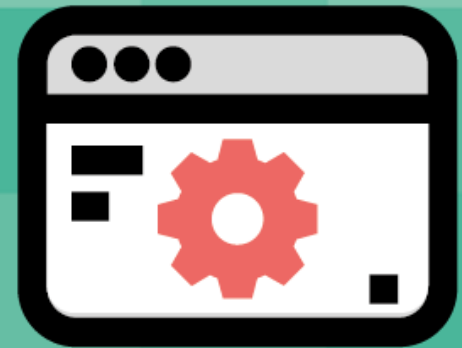
Avec SRTP, de pair avec fonction `inline` :

```
let inline add x y = x + y
// val inline add : x: ^a → y: ^b → ^c
//    when ( ^a or ^b ) : (static member (+) : ^a * ^b → ^c)
//    🙌 Contrainte de membre 📌
```

F#

```
let x = add 1 2 // ✅ val x: int = 3
let y = add 1.0 2.0 // ✅ val y: float = 3.0
```

# 3 ■ Contraintes sur paramètres de type



# Contraintes

Même principe qu'en C# avec quelques différences :

Contrainte	Syntaxe F#	Syntaxe C#
Mots clés	<code>when xxx and yyy</code>	<code>where xxx, yyy</code>
Emplacement	Juste après type :	Fin de ligne :
	<code>fn (arg: 'T when 'T ... )</code>	<code>Method&lt;T&gt;(arg: T) where T ...</code>
	Dans chevrons :	
	<code>fn&lt;'T when 'T ... &gt; (arg: 'T)</code>	

# Contraintes : vue d'ensemble

Contrainte	Syntaxe F#	Syntaxe C#
Type de base	<code>'T :&gt; my-base</code>	<code>T : my-base</code>
Type valeur	<code>'T : struct</code>	<code>T : struct</code>
Type référence	<code>'T : not struct</code>	<code>T : class</code>
Type référence nullable	<code>'T : null</code>	<code>T : class?</code>
Constructeur sans param	<code>'T : (new: unit → 'T)</code>	<code>T : new()</code>
Énumération	<code>'T : enum&lt;my-enum&gt;</code>	<code>T : System.Enum</code>
Comparaison	<code>'T : comparison</code>	<code>≈ T : System.IComparable</code>
Égalité	<code>'T : equality</code>	<i>(pas nécessaire)</i>
Membre explicite	<code>^T : member-signature</code>	<i>(pas d'équivalent)</i>

# Contraintes de type

Pour forcer le type de base : classe mère ou interface

```
let check<'TError when 'TError :> System.Exception> condition (error: 'TError) =  
    if not condition then raise error
```

F#

→ Équivalent en C# :

```
static void check<TError>(bool condition, TError error) where TError : System.Exception  
{  
    if (!condition) throw error;  
}
```

C#

💡 Syntaxe alternative : `let check condition (error: #System.Exception)`

→ Cf. *Type flexible* 📌

# Contrainte d'enum

```
open SystemF#

let getValues<'T when 'T : enum<int>>() =
    Enum.GetValues(typeof<'T>) :> 'T array

type ColorEnum = Red = 1 | Blue = 2
type ColorUnion = Red | Blue

let x = getValues<ColorEnum>() // [] Red; Blue []
let y = getValues<ColorUnion>() // ✨ Exception ou erreur de compilation (1)
```

(1) La contrainte `when 'T : enum<int>` permet :

- D'éviter la `ArgumentException` au runtime (*Type provided must be an Enum*)
- Au profit d'une erreur dès la compilation (*The type 'ColorUnion' is not an enum*)

# Contrainte de comparaison

Syntaxe : `'T : comparison`

Indique que le type `'T` doit :

- soit implémenter `IComparable` (1)
- soit être un collection d'éléments comparables (2)

## 👉 Notes :

1. `'T : comparison` > `'T : IComparable` !
2. `'T : comparison` ≠ `'T : IComparable<'T>` !
3. Pratique pour méthodes génériques `compare` ou `sort` 💡

# Contrainte de comparaison - Exemple

```
let compare (x: 'T) (y: 'T when 'T : comparison) =  
    if x < y then -1  
    elif x > y then +1  
    else 0
```

F#

```
// Comparaison de nombres et de chaînes
```

```
let x = compare 1.0 2.0 // -1
```

```
let y = compare "a" "A" // +1
```

```
// Comparaison de listes d'entier
```

```
let z = compare [ 1; 2; 3 ] [ 2; 3; 1 ] // -1
```

```
// Comparaison de listes de fonctions
```

```
let a = compare [ id; ignore ] [ id; ignore ]
```

```
// ✨ ~
```

```
// error FS0001: Le type '('a → 'a)' ne prend pas en charge la contrainte 'comparison'.
```

```
// Par exemple, il ne prend pas en charge l'interface 'System.IComparable'
```



# Contrainte de membre explicite

“ **Pb** : Comment indiquer qu'un objet doit disposer d'un certain membre ? ”

- Manière classique en .NET : typage nominal  
→ Contrainte spécifiant type de base (interface ou classe parent)
- Alternative en F# : typage structurel (*a.k.a duck-typing du TypeScript*)  
→ Contrainte de membre explicite  
→ Utilisée avec SRTP (*statically resolved type parameter*)

# Contrainte de membre explicite (2)

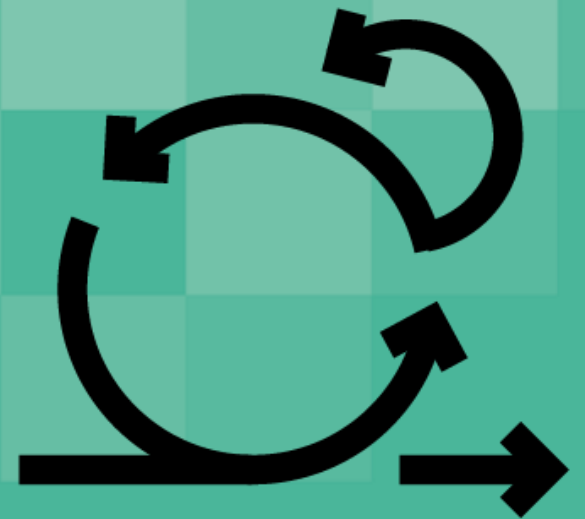
```
let inline add (value1 : ^T when ^T : (static member (+) : ^T * ^T → ^T), value2: ^T) =  
    value1 + value2  
  
let x = add (1, 2)  
// val x : int = 3  
let y = add (1.0, 2.0)  
// val y : float = 3.0
```

F#

⚖ Pour et contre :

- 👍 Permet de rendre code générique pour types hétérogènes
- 👎 Difficile à lire, à maintenir. Ralentit la compilation
- 📌 À utiliser dans une librairie, pas pour modéliser un domaine

# 4. ■ Type flexible



# Type flexible - Besoin (1)

Lors de la création de certaines fonctions génériques, il faut spécifier qu'un paramètre de type est un sous-type d'un certain autre type.

→ Illustration grâce à un exemple :

```
open System.Collections.Generic

// V1
let add item (collection: ICollection<_>) =
    collection.Add item
    collection

let a = List([1..3]) // List<int>
let b = a ▷ add 4    // ICollection<int> ≠ List<int> !
```

F#

# Type flexible - Besoin (2)

Solutions :

- **V2** : indiquer une **contrainte de type**
- **V3** : indiquer un **type flexible**

```
(* V1 ✗ *) let add item (collection: ICollection<_>) = ...  
(* V2a 🤔 *) let add<'t, 'u when 'u :> ICollection<'t>> (item: 't) (collection: 'u) : 'u = ...  
(* V2b 😞 *) let add (item: 't) (collection: 'u when 'u :> ICollection<'t>) : 'u = ...  
(* V3 ✅ *) let add item (collection: #ICollection<_>) = ...
```

F#

## ⚖️ Bilan :

- **V2a** : syntaxe similaire au C# → verbeux et pas très lisible ! 🤔
- **V2b** : version améliorée en F# → + lisible mais encore un peu verbeux ! 😞
- **V3** : syntaxe proche de **V1** → concision « dans l'esprit F# » ✅

# Type flexible - Autres usages (1)

Faciliter l'usage de la fonction sans avoir besoin d'un *upcast*

```
let join separator (generate: unit → seq<_>) =  
    let items = System.String.Join (separator, generate() ▷ Seq.map (sprintf "%A"))  
    $"[ {items} ]"  
  
let s1 = join ", " (fun () → [1..5])           // ✨ Error FS0001  
let s2 = join ", " (fun () → [1..5] :> seq<int>) // 😞 Marche mais pénible à écrire
```

Avec un type flexible :

```
let join separator (generate: unit → #seq<_>) =  
    // [ ... ]  
  
let s1 = join ", " (fun () → [1..5])           // ✅ Marche naturellement
```

# Type flexible - Autres usages (2)

Dans l'exemple ci-dessous, `items` est inféré avec la bonne contrainte :

```
let tap f items =  
    items ▷ Seq.iter f  
    items  
// val tap : f:(('a → unit) → items:'b → 'b when 'b :> seq<'a>
```

F#

💡 Quid de faciliter la lecture du code avec un type flexible ?

```
let tap f (items: #seq<_>) =  
    // [ ... ]
```

F#

# Type flexible - Autres usages (3)

⚠ Astuce précédente ne marche pas toujours !

```
let max x y =  
    if x > y then x else y  
// val max : x:'a → y:'a → 'a when 'a : comparison
```

F#

`x` et `y` doivent satisfaire 2 conditions

1. `'a : comparison`  $\simeq$  les types de `x` et `y` implémentent `IComparable`  
→ `(x: #IComparable) (y: #IComparable) ?`
2. `x:'a` et `y:'a` → `x` et `y` ont le même type  
→ Non exprimable sous forme de type flexible ! 😞



# Type flexible - Résumé

Type flexible

- Utilisé dans la déclaration de certaine fonction générique
- Indique qu'un paramètre de type est un sous-type d'un type spécifié
- Sucre syntaxique au format `#super-type`
- Équivalent de `'T when 'T :> super-type`

Autres usages :

- Faciliter l'usage de la fonction sans avoir besoin d'un *upcast*
- Faciliter la lecture du code ?

# 5. ■ Unités de mesure



# Unités de mesure : présentation

Moyen d'associer un **type numérique** à une unité de mesure

- Durée : `s` aka `second`
- Masse : `kg`
- Longueur : `m` aka `metre`
- ...

Les unités sont **vérifiées à la compilation**

- Empêche d'ajouter des 🥦 à des 🥕 → **code + sûr**
- Permet de les **combiner** : Vitesse = Distance / Durée → `m/s`

# Unités de mesure : déclaration

Syntaxe basée sur attribut [`<Measure>`]

```
// ➡ Nouvelles unités "from scratch"
[<Measure>] type kilogram
[<Measure>] type metre
[<Measure>] type second

// ➡ Alias d'unités existantes
[<Measure>] type kg = kilogram
[<Measure>] type m = metre
[<Measure>] type s = second

// ➡ Combinaison d'unités existantes
[<Measure>] type Hz = / s
[<Measure>] type N = kg m / s^2
```

F#

# Unités de mesure : SI

Les unités du **Système International** sont prédéfinies dans les namespaces :

`FSharp.Data.UnitSystems.SI.UnitNames` :

- `ampere`, `hertz`, `joule`, `kelvin`, `kilogram`, `metre` ...
- <https://fsharp.github.io/fsharp-core-docs/reference/fsharp-data-unitsystems-si-unitnames.html>

`FSharp.Data.UnitSystems.SI.UnitSymbols`

- `A`, `Hz`, `J`, `K`, `kg`, `m` ...
- <https://fsharp.github.io/fsharp-core-docs/reference/fsharp-data-unitsystems-si-unitsymbols.html>

# Unités de mesure : symbole

💡 **Astuce** : utilisation des *doubles back ticks*

```
[<Measure>] type ``Ω``  
[<Measure>] type ``°C``  
[<Measure>] type ``°F``  
  
let waterFreezingAt = 0.0<``°C``>  
// val waterFreezingAt : float<°C> = 0.0  
  
let waterBoilingAt = 100.0<``°C``>  
// val waterBoilingAt : float<°C> = 100.0
```

F#

# Unités de mesure : usage

F#

```
// Unité définie en annotant le nombre
let distance = 1.0<m>           // val distance : float<m> = 1.0
let time = 2.0<s>               // val time : float<s> = 2.0

// Unité combinée, inférée
let speed = distance / time     // val speed : float<m/s> = 0.5

// Unité combinée, définie par annotation
let [<Literal>] G = 9.806<m/s^2> // val G : float<m/s ^ 2> = 9.806

// Comparaison
let sameFrequency = (1<Hz> = 1</s>) // ✓ true
let ko1 = (distance = 1.0)          // ✗ Error FS0001: Incompatibilité de type.
                                   // ✗ Attente de 'float<m>' mais obtention de 'float'
let ko2 = (distance = 1<m>)         // ✗ Attente de 'float<m>' mais obtention de 'int<m>'
let ko3 = (distance = time)         // ✗ Attente de 'float<m>' mais obtention de 'float<s>'
```

# Unités de mesure : conversion

- Facteur multiplicatif avec une unité `<target/source>`
- Fonction de conversion utilisant ce facteur

```
[<Measure>] type m
[<Measure>] type cm
[<Measure>] type km

module Distance =
    let toCentimeter (x: float<m>) = // (x: float<m>) → float<cm>
        x * 100.0<cm/m>

    let toKilometer (x: float<m>) = // (x: float<m>) → float<km>
        x / 1000.0<m/km>

let a = Distance.toCentimeter 1.0<m> // val a : float<cm> = 100.0
let b = Distance.toKilometer 500.0<m> // val b : float<km> = 0.5
```

F#



# Unités de mesure : conversion (2)

Exemple 2 : degré Celsius (°C) → degré Fahrenheit (°F)

```
[<Measure>] type ``°C``  
[<Measure>] type ``°F``  
  
module Temperature =  
    let toFahrenheit ( x: float<``°C``> ) = // (x: float<°C>) → float<°F>  
        9.0<``°F``> / 5.0<``°C``> * x + 32.0<``°F``>  
  
let waterFreezingAt = Temperature.toFahrenheit 0.0<``°C``>  
// val waterFreezingAt : float<°F> = 32.0  
  
let waterBoilingAt = Temperature.toFahrenheit 100.0<``°C``>  
// val waterBoilingAt : float<°F> = 212.0
```

F#

# Unités de mesure : ajouter/supprimer

Ajouter une unité à un nombre nu :

→  `number * 1.0<target>`

Supprimer l'unité d'un nombre `number : float<source>` :


→  `number / 1.0<source>`

→  `float number`

Créer une liste de nombres avec unité :

→  `[1<m>; 2<m>; 3<m>]`

→  `[1<m> .. 3<m>]` (*un range nécessite des nombres nus*)

→  `[ for i in [1..3] → i * 1<m> ]`

# Unités de mesure : effacées au runtime !

Les unités de mesure sont propres au compilateur F#.  
→ Elles ne sont pas compilées en .NET

# Type avec unité générique

Besoin de distinguer d'un type générique classique

→ Annoter l'unité générique avec `[<Measure>]`

```
type Point[<Measure>] 'u, 'data =  
    { X: float<'u>; Y: float<'u>; Data: 'data }  
  
let point = { X = 10.0<m>; Y = 2.0<m>; Data = "abc" }  
// val point : Point<m, string> = { X = 10.0; Y = 2.0; Data = "abc" }
```

F#

# Unité pour primitive non numérique

💡 Nuget [FSharp.UMX](#) (*Unit of Measure Extension*)

→ Pour autres primitives `bool`, `DateTime`, `Guid`, `string`, `TimeSpan`

```
open System

#r "nuget: FSharp.UMX"
open FSharp.UMX

[<Measure>] type ClientId
[<Measure>] type OrderId

type Order = { Id: Guid<OrderId>; ClientId: string<ClientId> }

let order = { Id = % Guid.NewGuid(); ClientId = % "RDE" }
```

F#

# 6 ■ *Casting et* conversion



# Conversion de nombre

Types numériques :

- Entier : `byte`, `int16`, `int` / `int32`, `int64`
- Flottant : `float` / `double` (64b), `single` (32b), `decimal`
- Autres : `char`, `enum`

Conversion entre eux **explicite**

→ Fonction de même nom que le type cible

```
let x = 1           // val x : int = 1
let y = float x     // val y : float = 1.0
let z = decimal 1.2 // val z : decimal = 1.2M
let s = char 160    // val s : char = ' '
```

F#

# Conversion entre nombre et enum

Il faut utiliser le nom de l'enum pour convertir un nombre en enum :

- Soit en paramètre générique de la fonction `enum<my-enum>`, ①
- Soit par annotation de type et la fonction `enum` sans paramètre générique. ②

L'opération inverse utilise la fonction `int`. ③

```
type Color =  
    | Red    = 1  
    | Green  = 2  
    | Blue   = 3  
  
let color1 = enum<Color> 1      // (1) val color1 : Color = Red  
let color2 : Color = enum 2     // (2) val color2 : Color = Green  
let value3 = int Color.Blue     // (3) val c1 : int = 3
```

F#



# Casting d'objets

→ S'utilise pour un objet dont le type appartient à une hiérarchie

Fonctionnalité	Précision	Sûr	Opérateur	Fonction
<i>Upcast</i>	Vers type de base	✓ Oui	<code>::&gt;</code>	<code>upcast</code>
<i>Downcast</i>	Vers type dérivé	✗ Non (*)	<code>::?&gt;</code>	<code>downcast</code>
Test de type	Dans pattern matching	✓ Oui	<code>::?</code>	

(\*) Le *downcast* peut échouer → risque de `InvalidCastException` au runtime ⚠

# Upcasting d'objets

En C# : *upcast* peut généralement être implicite

```
object o = "abc";
```

C#

En F# : *upcast* peut parfois être implicite  
mais en général doit être **explicite**, avec opérateur `::>`

```
let o1: obj = "abc"           // Implicite ✖ Error FS0001 ...
let o2 = "abc" ::> obj         // Explicite ✔

let toObject x : obj = x       // obj → obj
let o3 = "abc" ▷ toObject      // Implicite ✔

let l1: obj list = [1; 2; 3]   // Implicite ✔
let l2: int seq = [1; 2; 3]    // Implicite ✖ Error FS0001 ...
```

F#

# Upcasting d'objets (2)

Règles élargies/assouplies en F# 6

→ Exemple : upcast implicite de `int list` vers `int seq`

```
let l2: int seq = [1; 2; 3] // 🕒 OK en F# 6
```

# Casting d'objets - Exemple

```
type Base() =  
    abstract member F : unit → string  
    default _.F() = "F Base"
```

```
type Derived1() =  
    inherit Base()  
    override _.F() = "F Derived1"
```

```
type Derived2() =  
    inherit Base()  
    override _.F() = "F Derived2"
```

```
let d1 = Derived1()  
let b1 = d1 :> Base           // val b1 : Base  
let b1' : Base = upcast d1    // val b1' : Base
```

```
let t1 = b1.GetType().Name    // val t1 : string = "Derived1"
```

```
let one = box 1 // val one : obj = 1
```

F#

# Casting d'objets - Exemple (2)

```
let d1' = b1 :?> Derived1           // val d1' : Derived1
let d2' = b1 :?> Derived2           // ✨ System.InvalidCastException

let d1'': Derived1 = downcast b1    // val d1'' : Derived1

let f (b: Base) =
    match b with
    | :? Derived1 as derived1 → derived1.F()
    | :? Derived2 as derived2 → derived2.F()
    | _ → b.F()

let x = f b1                        // val x : string = "F Derived1"
let y = b1.F()                      // val y : string = "F Derived1"
let z = f (Base())                  // val z : string = "F Base"
let a = f (Derived2())              // val a : string = "F Derived2"
// 🙌      ^^^^^^^ Upcast implicite
```

F#

# Test de type

L'opérateur `:?` réalise un test de type et renvoie un booléen.

```
let isDerived1 = b1 :? Derived1 // val isDerived1 : bool = true
let isDerived2 = b1 :? Derived2 // val isDerived2 : bool = false
```

F#

👉 Il faut *boxer* un nombre pour tester son type :

```
let isIntKo = 1 :? int // ✨ Error FS0016
let isInt32 = (box 1) :? int // val isInt32 : bool = true
let isFloat = (box 1) :? float // val isFloat : bool = false
```

F#

💡 `box x`  $\simeq$  `x :> obj`

# 7 ■ Le Récap'



# Récap' - Type `unit`

Instance unique `()`

Utilité avec expressions :

- LA valeur non significative à renvoyer
- Remplace `void`

Dans signature de fonctions :

- Indique effets de bord

Appel d'une fonction sans paramètres

- Même syntaxe qu'en C# : `my-func()`



# Récap' - Génériques

Généricité de fonctions et de types

Généricité implicite

→ Basée sur inférence de type + généralisation automatique

Généricité explicite

→ Annotation `'T`

→ Inline (`x: 'T`) ou globale (`my-func<'T> (x: 'T) = ...`, `type Abc<'T> = ...`)

→ Wilcard `_` pour accepter n'importe quel paramètre de type : `seq<_>`

Généricité statique

→ Annotation `^T` : paramètre de type résolu statiquement (*SRTP*)

→ Typage structurel : puissant mais difficile à lire et lent à la compilation

# Récap' - Contraintes

Mots-clés `when`, `and`  $\neq$  `where` en C#

Plusieurs familles de contraintes :

- Type valeur `struct` ou référence `not struct` ou nullable `null`
- Constructeur `'T : (new: unit → 'T)`
- Type de base `'T :> my-base` ou `#my-base` (*type flexible*)
- Énumération `'T : enum<int>`
- Égalité `'T : equality` et comparaison `'T : comparison` structurelles
- Membre explicite pour SRTP : `^T : member-signature`

# Récap' - Unité de mesure

Définition `[<Measure>] type kg`

Usage `let x = 1.0<kg>`

Apporte *type safety*

→ Mais qu'en F#, effacées à la compilation

Performant (*vs Single-Case Union*)

Limitée aux types numériques

→ Étendue aux autres primitives avec [FSharp.UMX](#)

# Récap' - Conversion

- Conversion de type → généralement explicite
- Conversion entre types numériques → fonctions tq `int`
- Upcast `my-object :> base-type → base-type`
- Downcast `my-object :?> derived-type → derived-type | InvalidCastException`
- Test de type `my-object :? derived-type → bool`

Merci 🙏

**SOAT**

→ Digitalize society



**SOAT.FR**