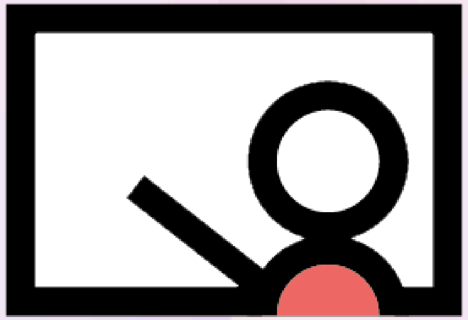
F♯ Training

Types : Addendum

2025 April

d-edge

Table of contents



1. Type `unit`

Type `unit` : why?

> **Constraint:** the evaluation of an expression must produce a value.

What about expressions that don't produce a meaningful value?
→ `void` ? No, `void` in C♯, Java is not a value!
→ `null` ? No, `null` is not a type in .NET! *(≠ TypeScript)*

So you need a specific type, with a single value meaning by convention:
"Insignificant value, to ignore."

→ This type is called `unit`.
→ Its value is noted `()`.

Type `unit` and functions

Function `unit → 'T` takes no parameters.
→ Ex : `System.DateTime.Now` *(function hidden behind a property)*

Function `'T → unit` does not return a value.
→ Ex : `printf`

👉 Functions involving a **side-effect** !

Type `unit` : ignore a value

F♯ is not a pure functional language, with no side effects.
But it does encourage the writing of pure functional programs.

👉 **Rule:** Any expression producing a value must be used.
→ `()` is the only value the compiler allows to be ignored.
→ For any other value not ignored, the compiler issues a warning.

☝️ **Warning:** ignoring a value is generally a *code smell* in FP.

👉 An expression with side-effect must signal it with the return type `unit`.

Type `unit` : function `ignore`

> ❓ How can you *(despite everything)* ignore the value produced by an expression?

With the `ignore` function:
→ Takes an ignored, "swallowed" input parameter.
→ Returns `unit`

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

Usage : `expression |> ignore`

2. Generics in F#

Generics

Functions and types can be generic, with more flexibility than in C♯.

By default, genericity **implicit**
→ Inferred
→ Even generalized, thanks to "automatic generalization"

Otherwise, genericity can be explicit or statically resolved.

⚠️ Different notations:
→ `'T`: parameter of generic type
→ `^T`: statically resolved type parameter *(SRTP)*

Implicit genericity

```
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
```

👉 **Explanations:**

➔

`singleton` : its argument `x` is any → generic type `'a`
→ Automatic generalization

➔

`couple`: its 2 arguments `x` and `y` must be of the same type to be in a list → Inference

Explicit genericity

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

→ Inference of the genericity of `x` and `y` 👍

❓ **How to indicate that `x` and `y` must have the same type?**

→ Need to indicate it explicitly :

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

Explicit genericity - Inline form

💡 **Hint:** the `'x` convention to indicate generic type parameter makes it possible to be more concise: the `<'T>` is no needed.

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

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

Explicit genericity - Type

The definition of generic types is explicit:

```fsharp
type Pair = { Item1: 'T ; Item2: 'T }
// 💥                   ~~            ~~
// Error FS0039: Parameter type `'T' is not defined.

// ✅ Records and unions with 1 or 2 generic type parameters
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
```

Genericity ignored

The *wildcard* `_` is used to replace an ignored parameter type:

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

Even more useful with flexible type 📍 :

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

// Versus
let tap action (sequence: #seq<_>) = ...
```

SRTP

F♯ offers two categories of parameter types:
→ `'X`: generic parameter type *(seen so far)*
→ `^X`: statically resolved parameter type *(by the F♯ compiler)*

☝️ **SRTP:** frequent abbreviation for *Statically Resolved Type Parameter*

SRTP - Why

W/o:

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

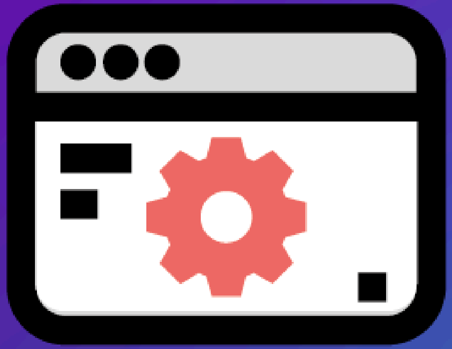
→ `int` type inference for `x` and `y`, without generalization (to `float` for example)!

With SRTP, in conjunction with `inline` function:

```
let inline add x y = x + y
// val inline add : x: ^a → y: ^b → ^c
//    when ( ^a or ^b ) : (static member (+) : ^a * ^b → ^c)
//     ☝️ Member constraint 📍

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

3. Constraints on type parameters



Constraints

Same principle as in C♯ with a few differences:

Constraint	Syntax in F♯	Syntax in C♯
Keywords	`when xxx and yyy`	`where xxx, yyy`
Place	Just after the type:	At the end of the line:
	`fn (arg: 'T when 'T ... )`	`Method<T>(arg: T) where T ...`
	Inside chevrons:	
	`fn<'T when 'T ... > (arg: 'T)`	

Constraints: an overview

Constraint	Syntax in F♯	Syntax in C♯
Base type	`'T :> my-base`	`T : my-base`
Value type	`'T : struct`	`T : struct`
Reference type	`'T : not struct`	`T : class`
Nullable ref. type	`'T : null`	`T : class?`
Constructor w/o param	`'T : (new: unit → 'T)`	`T : new()`
Enum	`'T : enum<my-enum>`	`T : System.Enum`
Comparison	`'T : comparison`	≃ `T : System.IComparable`
Equality	`'T : equality`	*(not necessary)*
Explicit member	`^T : member-signature`	*(no equivalent)*

Type constraints

To force basic type: parent class or interface

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

→ C# equivalent:

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

💡 Alternative syntaxe: `let check condition (error : #System.Exception)`
→ Cf. *Flexible type* 📍

Enum constraint

```fsharp
open System

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 or compiler error (1)
```

(1) The `when 'T: enum<int>` constraint allows :

- To avoid the `ArgumentException` at runtime *(Type provided must be an Enum)*
- In favor of a compile-time error *(The type 'ColorUnion' is not an enum)*

Comparison constraint

Syntax: `'T: comparison`

Indicates that the `'T` type must :
- either implement `IComparable` (1)
- be a collection of comparable elements (2)

☝️ **Notes :**

1. `'T : comparison` > `'T : IComparable` ❗
2. `'T : comparison` ≠ `'T : IComparable<'T>` ❗
3. Useful for `compare` or `sort` generic methods 💡

Comparison constraint - Example

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

// Number and string comparison
let x = compare 1.0 2.0  // -1
let y = compare "a" "A"  // +1

// Integer list comparison
let z = compare [ 1; 2; 3 ] [ 2; 3; 1 ]  // -1

// Compare lists of functions
let a = compare [ id; ignore ] [ id; ignore ]
// 💥             ~~
// Error FS0001: The type '('a → 'a)' does not support the 'comparison' constraint.
// For example, it does not support the 'System.IComparable' interface.
```

Explicit member constraint

“ **Issue:** How do you specify that an object must have a certain member? ”

- .NET usual way: nominal typing
→ Constraint specifying base type (interface or parent class)

- Alternative in F♯: structural typing *(a.k.a duck-typing of TypeScript)*
→ Explicit member constraint
→ Used with SRTP *(statically resolved type parameter)*

Explicit member constraint (2)

```fsharp
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
```

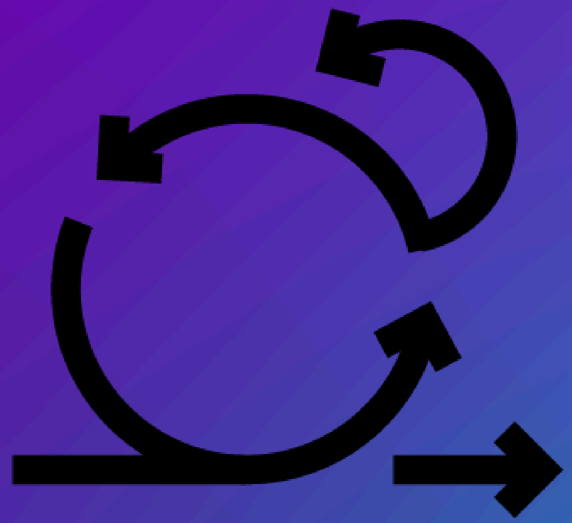
⚖️ Pros and cons :

➔ 👍 Allows same code for heterogeneous types (remove duplication).

➔ 👎 Difficult to read and maintain. Slows down compilation.

➔ 👉 To be used in a library, not to model a domain.

4. Flexible type



Flexible type - Need (1)

When creating some generic functions, it is necessary to specify that a type parameter is a subtype of a certain other type.

→ Illustration with an example:

```
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> ❗
```

Type flexible - Need (2)

Solutions :
→ **V2 :** specify a **type constraint**
→ **V3 :** indicate a **flexible type**

```fsharp
(* 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<_>) = …
```

⚖️ **Result:**
→ **V2a**: syntax similar to C♯ → verbose and not very readable! 😖
→ **V2b**: improved version in F♯ → + readable but still a bit verbose! 😕
→ **V3**: syntax close to **V1** → concise "in the F♯ spirit"! ✅

Flexible type - Other uses (1)

Facilitate the use of the function without the need for an *upcast*.

```fsharp
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>) // 😕 Works but painful to write
```

With a flexible type:

```fsharp
let join separator (generate: unit → #seq<_>) =
    // [...]

let s1 = join ", " (fun () → [1..5]) // ✅ Works naturally
```

Flexible type - Other uses (2)

In the example below, `items` is inferred with the correct constraint:

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

💡 What about making code easier to read with a flexible type?

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

Flexible type - Other uses (3)

⚠️ Previous tip doesn't always work!

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

`x` and `y` must satisfy 2 conditions

1. `'a: comparison` ≃ types of `x` and `y` implement `IComparable`
   → `(x: #IComparable) (y: #IComparable)` ?
2. `x:'a` and `y:'a` → `x` and `y` have the same type
   → Not possible to note with flexible types! 😞

Flexible type - Summary

Flexible type
→ Used in the declaration of certain generic functions
→ Indicates that a type parameter is a subtype of a specified type
→ Syntactic sugar in `#super-type` format
→ Equivalent of `'T when`'T :> super-type`

Other uses :
→ Facilitate function usage without the need for an *upcast*.
→ Make code easier to read?

5. Units of measure

Units of measure: overview

Means of associating a **numerical type** with a unit of measure
→ Duration: `s` *aka* `second`
→ Mass: `kg`
→ Length: `m` *aka* `metre`
→ …

Units are **checked at compile time**
→ Prevents adding 🥦 to 🥕 → **code + safe**
→ Allows them to be **combined**: Speed = Distance / Duration → `m/s`

Units of measure: declaration

Syntaxe based on the `[<Measure>]` attribute

```fsharp
// 👉 New units "from scratch"
[<Measure>] type kilogram
[<Measure>] type metre
[<Measure>] type second

// 👉 Aliases of existing units
[<Measure>] type kg = kilogram
[<Measure>] type m  = metre
[<Measure>] type s  = second

// 👉 Combining existing units
[<Measure>] type Hz = / s
[<Measure>] type N = kg m / s^2
```

Units of measure: SI

International System units are predefined in the following 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

Units of measure: symbole

💡 **Tip:** use of *double 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
```

Units of measure: usage

```fsharp
// Unit defined by annotating the number
let distance = 1.0<m>               // val distance : float<m> = 1.0
let time = 2.0<s>                   // val time : float<s> = 2.0

// Combined, inferred unit
let speed = distance / time         // val speed : float<m/s> = 0.5

// Combined unit, defined by annotation
let [<Literal>] G = 9.806<m/s^2>    // val G : float<m/s ^ 2> = 9.806

// Comparison
let sameFrequency = (1<Hz> = 1</s>)   // ✅ true
let ko1 = (distance = 1.0)            // ❌ Error FS0001: Type incompatibility.
                                      // 💥 Expected 'float<m>', Given: 'float'
let ko2 = (distance = 1<m>)           // 💥 Expected 'float<m>', Given: 'int<m>'
let ko3 = (distance = time)           // 💥 Expected 'float<m>', Given: 'float<s>'
```

Units of measure: conversion

- ➔ Multiplicative factor with a `<target/source>` unit
- ➔ Conversion function using this factor

```fsharp
[<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
```

Units of measure: conversion (2)

Example 2: degree Celsius (°C) → degree Fahrenheit (°F)

```fsharp
[<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
```

Units of measure: add/delete

Add a unit to a bare number :

→ ✅ `number * 1.0<target>`

Remove a unit from a `number : float<source>` :

→ ✅ `number / 1.0<source>`

→ ✅ `float number`

Create a list of numbers with unit :

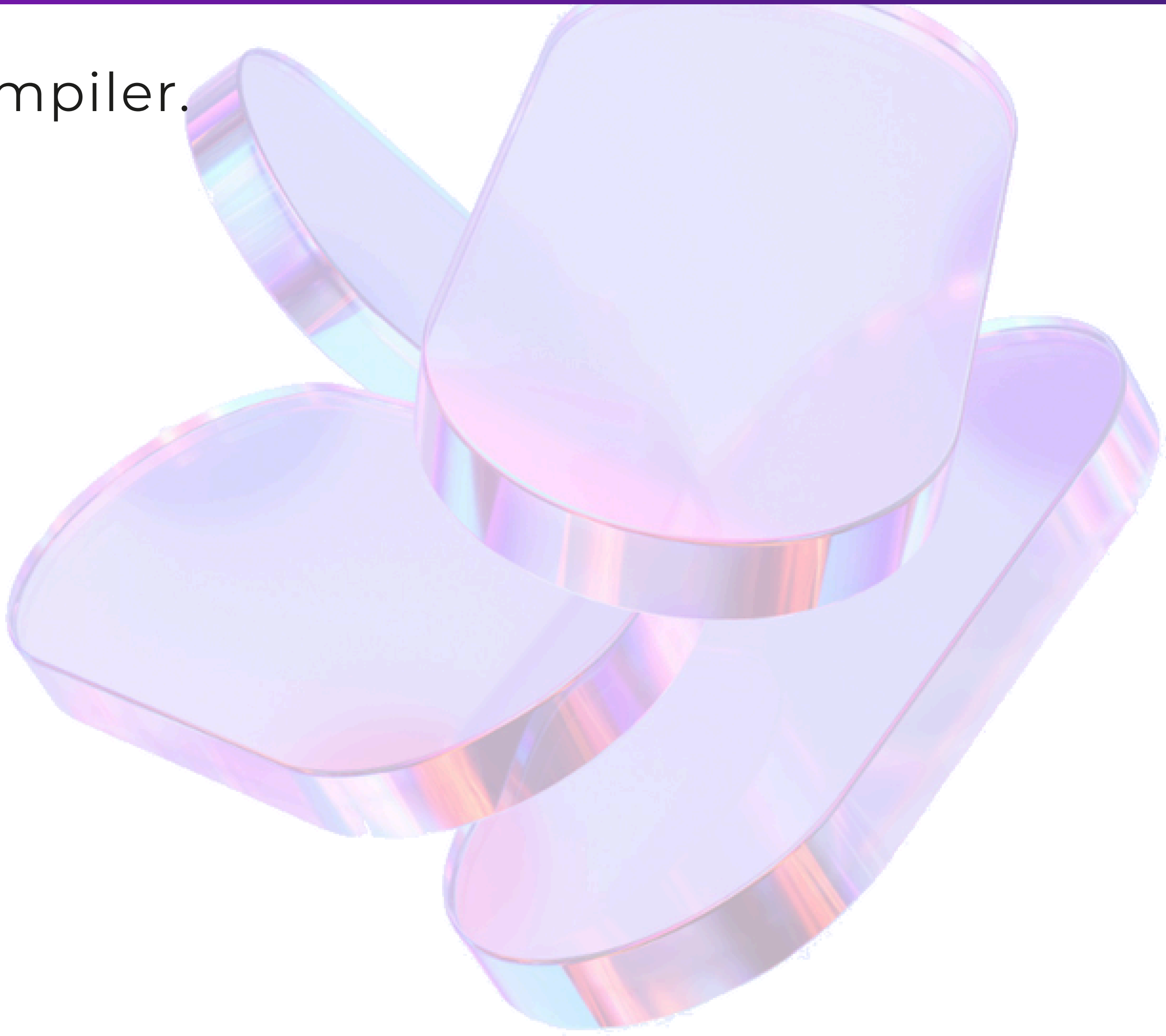
→ ✅ `[1<m>; 2<m>; 3<m>]`

→ ❌ `[1<m>..3<m>]` *(a range requires bare numbers)*

→ ✅ `[ for i in [1..3] -> i * 1<m> ]`

Units of measure: deleted at runtime ❗

Units of measure are specific to the F♯ compiler.
→ They are not compiled in .NET.



Type with generic unit

Need to distinguish from a classic generic type
→ Annotate generic unit with `[<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" }
```

Unit for non-numerical primitive

💡 Nuget FSharp.UMX *(Unit of Measure Extension)*
→ For other primitives `bool`, `DateTime`, `Guid`, `string`, `TimeSpan`

```fsharp
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" }
```

6. *Casting* and conversion

Number conversion

Numeric types :

- ➔ Integer: `byte`, `int16`, `int`/`int32`, `int64`
- ➔ Floating: `float`/`double` (64b), `single` (32b), `decimal`
- ➔ Others: `char`, `enum`.

explicit conversion between these types
→ Helper with same name as the target type

```
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 = ' '
```

Number to enum conversion

Use the enum name to convert a number into an enum:

- either as a generic parameter to the `enum<my-enum>` function, ①
- Or by type annotation and the `enum` function without generic parameter. ②

The reverse operation uses the `int` function. ③

```fsharp
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
```

Object cast

→ Used for an object whose type belongs to a hierarchy

Feature	Remark	Safe	Operator	Function
Upcast	To base type	✅ Yes	`:>`	`upcast`
Downcast	To derived type	❌ No (*)	`:?>`	`downcast`
Type test	In pattern matching	✅ Yes	`:?`	

(*) The *downcast* may fail → risk of `InvalidCastException` at runtime ⚠️

Object upcast

In C♯ : *upcast* can generally be implicit

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

In F♯ : *upcast* can sometimes be implicit
but in general it must be **explicit**, with the operator `:>`

```
let o1: obj = "abc"                // Implicit 💥 Error FS0001 ...
let o2 = "abc" :> obj              // Explicit 👌

let toObject x : obj = x           // obj → obj
let o3 = "abc" ▷ toObject          // Implicit 👌

let l1: obj list = [1; 2; 3]       // Implicit 👌
let l2: int seq = [1; 2; 3]        // Implicit 💥 Error FS0001 ...
```

Object upcast (2)

Extended rules in F♯ 6

→ Example: implicit upcast from `int list` to `int seq`.

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

Object upcast - Example

```
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
```

CObject upcast - Example (2)

```fsharp
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 implicit
```

Type test

The `:?` operator performs a type test and returns a Boolean.

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

☝️ A number must be *boxed* to test its 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
```

💡 `box x` ≃ `x :> obj`

7. Wrap up

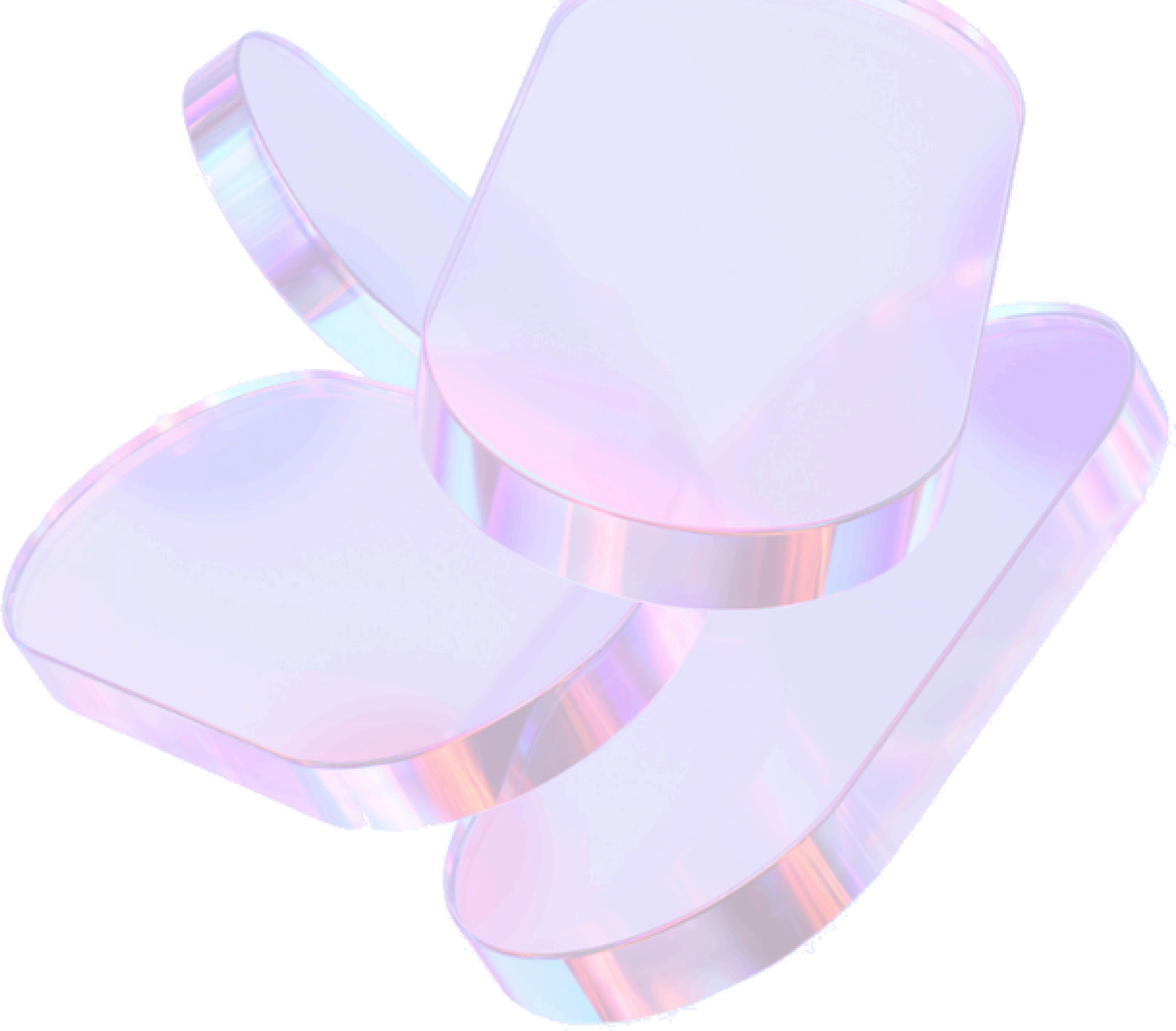
Wrap up - Type `unit`

Single instance `()`

Utility with expressions :
→ The non-significant value to return
→ Replaces `void`

In function signature :
→ Indicates side effects

Calling a function without parameters
→ Same syntax as in C♯: `my-func()`



Wrap up - Generics

Genericity of functions and types

Implicit genericity
→ Based on type inference + automatic generalization

Explicit genericity
→ `'T` annotation
→ Inline (`x: 'T`) or global (`my-func<'T> (x: 'T) = ...`, `type Abc<'T> = ...`)
→ Wilcard `_` to accept any parameter type: `seq<_>`

Static genericity
→ Annotation `^T` : statically resolved type parameter *(SRTP)*
→ Structural typing: powerful but difficult to read and slow to compile

Wrap up - Constraints

Keywords `when`, `and` ≠ `where` in C♯

Several families of constraints:

→ Type value `struct` or reference `not struct` or nullable `null`

→ Constructor `'T : (new: unit → 'T)`

→ Base type `'T :> my-base` or `#my-base` *(flexible type)*

→ Enum `'T : enum<int>`

→ Structural equality `'T: equality`

→ Structural comparison `'T: comparison`

→ Explicit member for SRTP: `^T: member-signature`

Wrap up - Units of measure

Definition `[<Measure>] type kg`

Usage `let x = 1.0<kg>`

Provides *type safety*
→ But only in F♯, deleted at compile time

Performant *(vs Single-Case Union)*

Limited to numeric types
→ Extended to other primitives with [FSharp.UMX]()

Wrap up - Conversion

- ➔ Type conversion → generally explicit
- ➔ Conversion between numeric types → like `int` helpers
- ➔ Upcast `my-object :> base-type → base-type`
- ➔ Downcast `my-object :?> derived-type → derived-type | InvalidCastException`
- ➔ Type test `my-object :? derived-type → bool`

Thanks 🙏