F♯ Training 📎

Object-oriented

2025 April

d-edge

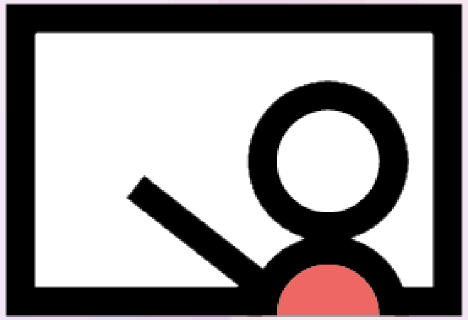
Introduction

In F♯, object-oriented sometimes + practical than functional style.

Object-oriented bricks in F♯:

1. Members
   - ➔ Methods, properties, operators
   - ➔ Attach functionalities directly to the type
   - ➔ Encapsulate the object's state (particularly if mutable)
   - ➔ Used with object dotting `my-object.my-member`
2. Interfaces and classes
   - ➔ Support abstraction through inheritance

Table of contents



Polymorphism

4th pillar of object-oriented programming

In fact, there are several polymorphisms. The main ones are:

1. By sub-typing: the one classically evoked with object-orientation
   → Basic type defining abstract or virtual members
   → Subtypes inheriting and implementing these members
2. Ad hoc/overloading → overloading of members with the same name
3. Parametric → generic in C♯, Java, TypeScript
4. Structural/duck-typing → SRTP in F♯, structural typing in TypeScript
5. Higher-kinded → type classes in Haskell

1. *Members*

Members

Additional elements in type definition *(class, record, union)*

- ➔ (Event)
- ➔ Method
- ➔ Property
- ➔ Indexed property
- ➔ Operator overload

Static and instance members

Static member: `static member member-name ...`.

Instance member:
→ Concrete member: `member self-identifier.member-name ...`
→ Abstract member: `abstract member member-name: type-signature`
→ Virtual member = requires 2 declarations
   1. Abstract member
   2. Default implementation: `default self-identifier.member-name ...`
→ Override virtual member: `override self-identifier.member-name ...`

☝️ `member-name` in PascalCase *(.NET convention)*
☝️ No `protected` member !

Self-identifier

In C♯, Java, TypeScript : `this`
In VB : `Me`
In F♯ : we can choose → `this`, `self`, `me`, any valid *identifier*...

Declaration:

1. For the primary constructor ❗: with `as` → `type MyClass() as self = ...`
   ➔ ⚠️ Can be costly
2. For a member: `member me.Introduce() = printfn $"Hi, I'm {me.Name}"`
3. For a member not using it: with `_` → `member _.Hi() = printfn "Hi!"`

Call a member

Calling a static member
→ Prefix with the type name: `type-name.static-member-name`

Calling an instance member inside the type
→ Prefix with *self-identifier*: `self-identifier.instance-member-name`

Call an instance member from outside the type
→ Prefix with instance-name: `instance-name.instance-member-name`

Method

≃ Function attached directly to a type

2 forms of parameter declaration:

1. Curried parameters = FP style
2. Parameters in tuple = OOP style
    - ➔ Better interop with C♯
    - ➔ Only mode allowed for constructors
    - ➔ Support named, optional, arrayed parameters
    - ➔ Support overloads

Method (2)

```fsharp
// (1) Tuple form (the most classic)
type Product = { SKU: string; Price: float } with
    member this.TupleTotal(qty, discount) =
        (this.Price * float qty) - discount  // (A)

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

☝️ `with` required in ① but not in ② because of indentation
→ `end` can end the block started with `with` *(not recommended)*

☝️ `this.Price` Ⓐ and `me.Price` Ⓑ
→ Access to instance via *self-identifier* defined by member

Named arguments

Calls a tuplified method by specifying parameter names:

```
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
```

Useful for :
→ Clarify a usage for the reader or compiler (in case of overloads)
→ Choose the order of arguments
→ specify only certain arguments, the others being optional

☝️ Arguments *after a named argument* are necessarily named too.

Optional parameters

Allows you to call a tuplified method without specifying all the parameters.

Optional parameter:

- Declared with `?` in front of its name → `?arg1: int`
- In the body of the method, wrapped in an `Option` → `arg1: int option`
  → You can use `defaultArg` to specify the **default value**
  → But the default value does not appear in the signature!

When the method is called, the argument can be specified either:

- Directly in its type → `Method(arg1 = 1)`
- Wrapped in an `Option` if named with prefix `?` → `Method(?arg1 = Some 1)`

☝️ Other syntax for interop .NET: `[<Optional; DefaultParameterValue( ... )>] arg`

Optional parameters: Examples

```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)
```

☝️ Notice the *shadowing* of parameters by variables of the same name

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

Parameter array

Allows you to specify a variable number of parameters of the same type
→ Via `System.ParamArray` attribute on **last** method argument

```
open System

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

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

💡 Equivalent of C♯ `public static T Max<T>(params T[] items)`

Call C♯ method *TryXxx()*

❓ How to call in F♯ a C♯ method `bool TryXxx(args, out T outputArg)`?
(Example: `int.TryParse`, `IDictionnary::TryGetValue`)

- → 👎 Use F♯ equivalent of `out outputArg` but use mutation 😵
- → ✅ Do not specify `outputArg` argument
  - → Change return type to tuple `bool * T`
  - → `outputArg` becomes the 2nd element of this tuple

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

Call method *Xxx(tuple)*

❓ How do you call a method whose 1st parameter is itself a tuple?!

Let's try:

```
let friendsLocation = Map.ofList [ (0,0), "Peter" ; (1,0), "Jane" ]
// Map<(int * int), string>
let peter = friendsLocation.TryGetValue (0,0)
// 💥 Error FS0001: expression supposed to have type `int * int`, not `int`.
```

💡 **Explanations:** `TryGetValue(0,0)` = method call in tuplified mode
→ Specifies 2 parameters, `0` and `0`.
→ `0` is an `int` whereas we expect an `int * int` tuple!

Call method *Xxx(tuple)* - Solutions

1. 😕 *Backward pipe*, but also confusing
   - ➔ `friendsLocation.TryGetValue ◁ (0,0)`
2. 👌 Double parentheses, but confusing syntax
   - ➔ `friendsLocation.TryGetValue((0,0))`
3. ✅ Use a function rather than a method
   - ➔ `friendsLocation ▷ Map.tryFind (0,0)`

Method *vs* Function

Feature	Function	Curried method	Tuplified method
Partial application	✅ yes	✅ yes	❌ no
Named arguments	❌ no	❌ no	✅ yes
Optional parameters	❌ no	❌ no	✅ yes
Params array	❌ no	❌ no	✅ yes
Overload	❌ no	❌ no	✅ yes ①

① If possible, prefer optional parameters

Method *vs* Function (2)

Feature	Function	Static method	Instance method
Naming	camelCase	PascalCase	PascalCase
Support of `inline`	✅ yes	✅ yes	✅ yes
Recursive	✅ if `rec`	✅ yes	✅ yes
Inference of `x` in	`f x` → ✅ yes	`K.M x` → ✅ yes	`x.M()` → ❌ no
Can be passed as argument	✅ yes : `g f`	✅ yes : `g T.M`	❌ no : `g x.M` ①

① Alternatives:
→ F♯ 8: shorthand members → `g _.M()`
→ Wrap in lambda → `g (fun x → x.M())`

Properties

≃ Syntactic sugar hiding a *getter* and/or a *setter*
→ Allows the property to be used as if it were a field

2 ways to declare a property:

- Declaration **explicit**: in relation to a *backing field*.
  - → *Getter* : `member this.Property = expression`
  - → Others: verbose *(details)* 👉 Prefer explicit methods
- Declaration **automatic** : *backing field* implicit
  - → *Read-only* : `member val Property = value`
  - → *Read/write* : `member val Property = value with get, set`

☝️ *Getter* evaluated on each call ≠ *Read-only* initialized on construction

Properties - Example

```fsharp
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"
```

Properties and pattern matching

⚠️ Properties cannot be deconstructed
→ Can only participate in pattern matching in `when` part

```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 !"
```

Indexed properties

Allows access by index, as if the class were an array: `instance[index]`
→ Interesting for an ordered collection, to hide the implementation

Set up by declaring member `Item`

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

💡 Property *read-only* (*write-only*) → declare only the *getter* (*setter*)

☝️ Notice the *setter* parameters are curried

Indexed properties: example

```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]

let digitLabel = DigitLabel()
let v1 = digitLabel[1]     // "one"
digitLabel.Lang <- Fr
let v2 = digitLabel[2]     // "deux"
```

Slice

> Same as indexed property, but with multiple indexes

Declaration: `GetSlice(?start, ?end)` method *(regular or extension)*

Usage: `..` operator

```
type Range = { Min: int; Max: int } with
    /// Defines a sub-range - newMin and newMax are optional and ignored if out-of-bounds
    member this.GetSlice(newMin, newMax) =
        { Min = max (defaultArg newMin this.Min) this.Min
        ; Max = min (defaultArg newMax 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 }
```

Operator overload

Operator overloaded possible at 2 levels:

1. In a module, as a function

```
let [inline] (operator-symbols) parameter-list = ...
```

   - ➔ 👉 See session on functions
   - ➔ ☝️ Limited: only 1 definition possible

2. In a type, as a member

```
static member (operator-symbols) (parameter-list) =
```

   - ➔ Same rules as for function form
   - ➔ 👍 Multiple overloads possible (N types × P *overloads*)

Operator overload: example

```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, w: Vector) = Vector(v.X + w.X, v.Y + w.Y)
    static member (~-)(v: Vector) = -1.0 * v // 👈 Unary '-' operator

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. Type extensions

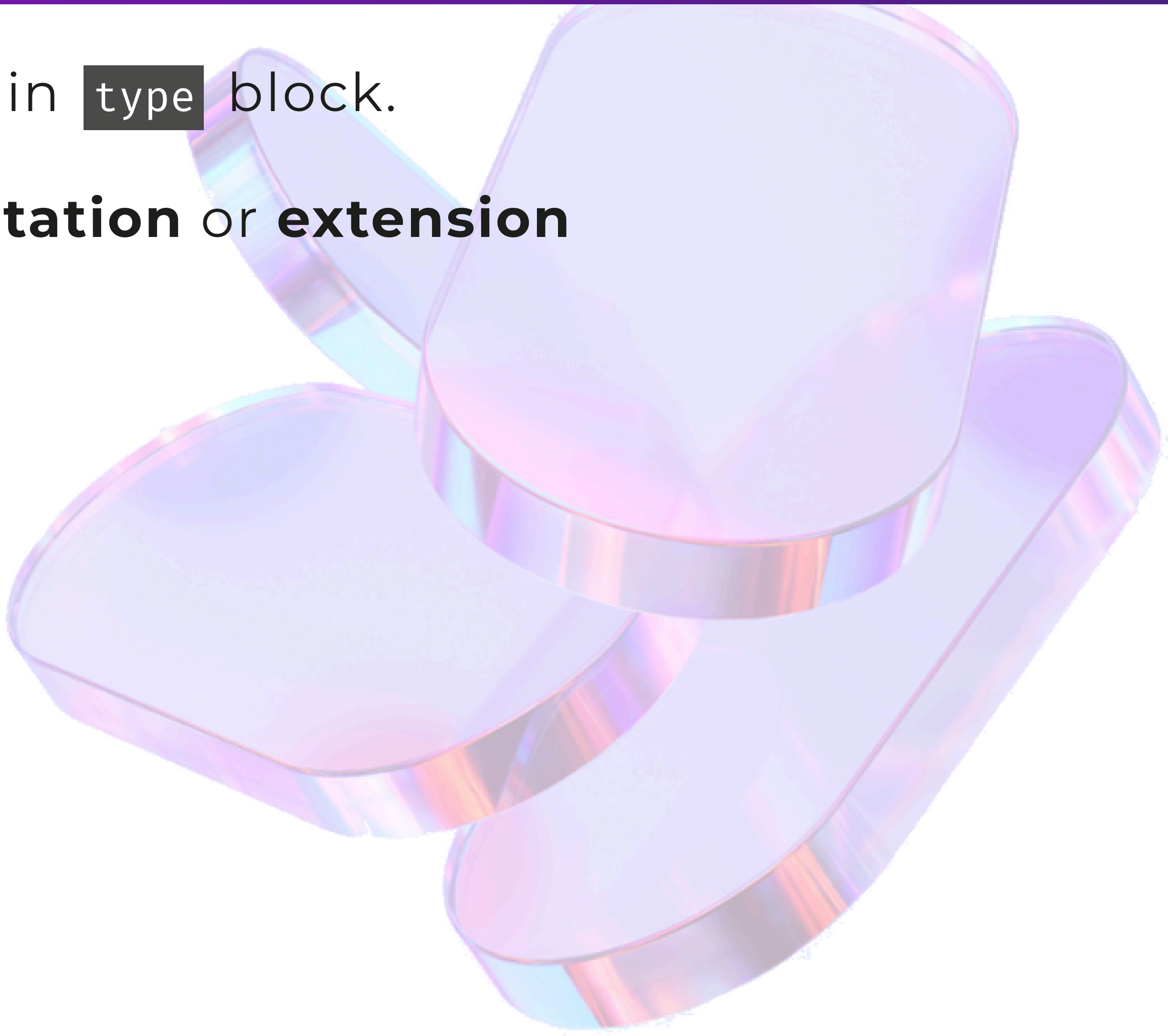
Type extension

Members of a type defined outside its main `type` block.

Each of these members is called **augmentation** or **extension**

3 categories of extension :

- ➔ Intrinsic extension
- ➔ Optional extension
- ➔ Extension methods



Intrinsic extension

> Declared in the same file and namespace as the type

Use case: Features available in both companion module and type
→ E.g. `List.length list` function and `list.Length` member

How to implement it following top-down declarations?

1. Implement in type, Redirect module functions to type members
→ More straightforward

2. Intrinsic extensions:
→ Declare type "naked", Implement in module, Augment type after
→ Favor FP style, Transparent for Interop

Intrinsic extension - Example

```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 as an extension - see `with` required keyword
type Variant with
    member x.Print() = Variant.print x
```

Optional extension

Extension defined outside the type module/namespace/assembly

Use cases

1. Types we can't modify, for instance coming from a library

2. Keep types naked - e.g. Elmish MVU pattern

```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
        }
```

Optional extension (2)

Compilation: into static methods
→ Simplified version of the previous example:

```
public static class EnumerableExtensions
{
    public static IEnumerable<T> RepeatElements<T>(IEnumerable<T> xs, int n) {…}
}
```

Usage: after the import, the member is used like a regular one

```
open EnumerableExtensions

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

Optional extension - Another example

```fsharp
// Person.fs ---
type Person = { First: string; Last: string }

// 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"
```

Optional extension - Limits

- ➔ Must be declared in a module
- ➔ Not compiled into the type, not visible to Reflection
- ➔ Usage as pseudo-instance members only in F♯
  - ➔ ≠ in C♯: as static methods

Type extension *vs* virtual methods

☝️ Override virtual methods:
→ in the initial type declaration ✅
→ not in a ~~type extension~~ ⛔

```
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...
```

Type extension *vs* type alias

Incompatible ❗

```fsharp
type i32 = System.Int32

type i32 with
    member this.IsEven = this % 2 = 0
// 💥 Error FS0964: Type abbreviations cannot have augmentations
```

💡 **Solution:** use the real type name

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

☝️ **Corollary:** F♯ tuples such as `int * int` cannot be augmented in this way.
→ But they can with a C♯-style extension method 📍

Type extension *vs* Generic type constraints

Extension allowed on generic type except when constraints differ:

```
open System.Collections.Generic

type IEnumerable<'T> with
//   ~~~~~~~~~~~~ 💥 Error FS0957
//   One or more of the declared type parameters for this type extension
//   have a missing or wrong type constraint not matching the original type constraints on 'IEnumerab
    member this.Sum() = Seq.sum this

// ☝️ This constraint comes from `Seq.sum`.
```

Solution: C♯-style extension method 📍

Extension method (C♯-style)

Static method:

- Decorated with `[<Extension>]`
- In F♯ < 8.0: Defined in class decorated with `[<Extension>]`
- Type of 1st argument = extended type *(`IEnumerable<'T>` below)*

Extension method - Simplified example

```fsharp
open System.Runtime.CompilerServices

[<Extension>] // 💡 Not required anymore since F♯ 8.0
type EnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: seq<_>) = Seq.sum xs

let x = [1..3].Sum()
//──────────────────────────────
// Output in FSI console (verbose syntax):
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
```

Extension method - C♯ equivalent

```csharp
using System.Collections.Generic;

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

☝ **Note:** The actual implementations of `Sum()` in LINQ are different, one per type: `int`, `float`… → *Source code*

Extension method - Tuples

An extension method can be added to any F♯ tuple:

```
open System.Runtime.CompilerServices

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

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

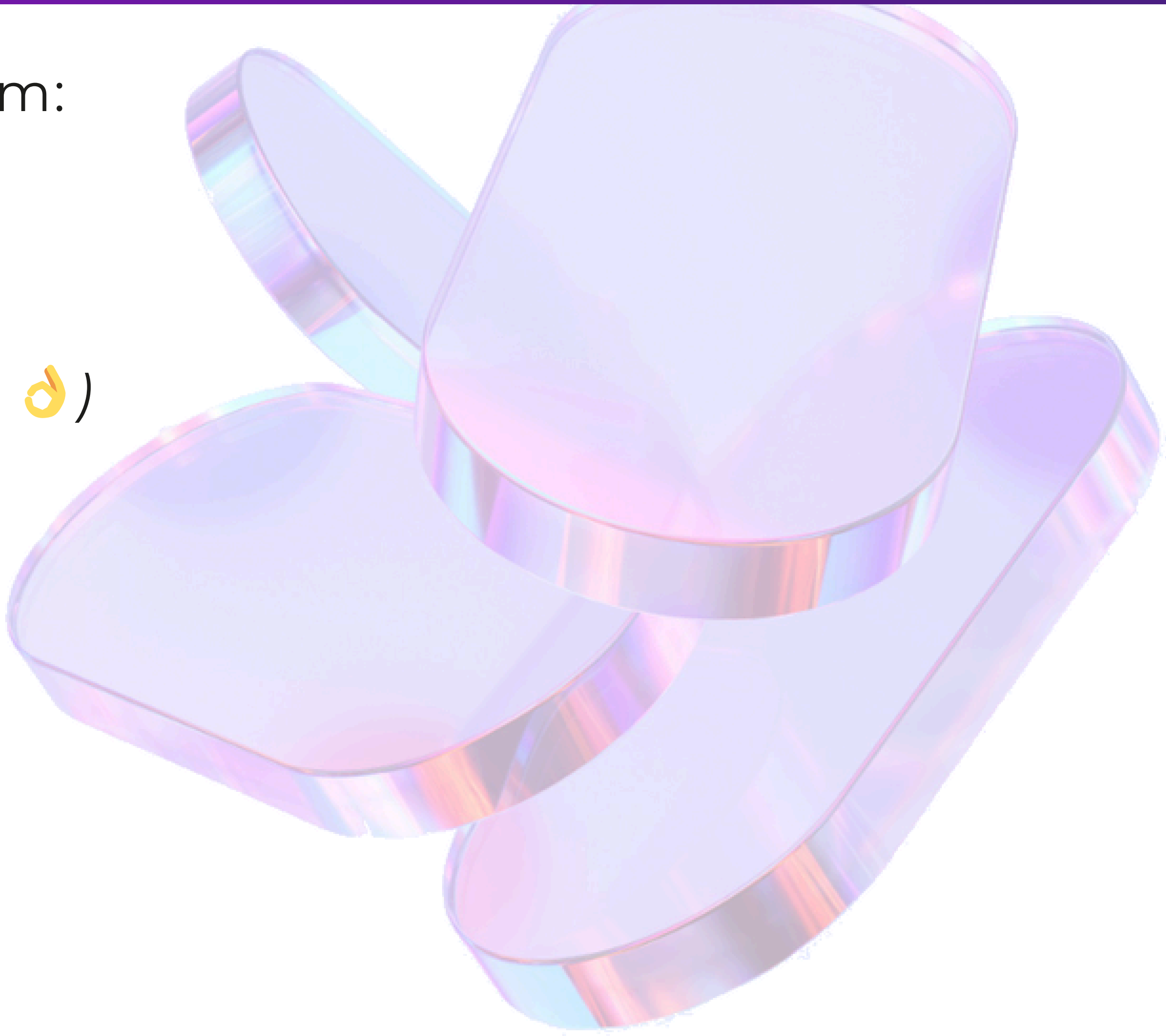
Extensions - Comparison

Feature	Type extension	Extension method
Methods	✅ instance, ✅ static	✅ instance, ❌ static
Properties	✅ instance, ✅ static	❌ *Not supported*
Constructors	✅ intrinsic, ❌ optional	❌ *Not supported*
Extend constraints	❌ *Not supported*	✅ *Support SRTP*

Extensions - Limits

Do not support (sub-typing) polymorphism:

- ➔ Not in the virtual table
- ➔ No `virtual`, `abstract` member
- ➔ No `override` member *(but overloads 👌)*



Extensions *vs* C♯ partial class

Feature	Multi-files	Compiled into type	Any type
C♯ partial class	✅ Yes	✅ Yes	Only `partial class`
Extension intrinsic	❌ No	✅ Yes	✅ Yes
Extension optional	✅ Yes	❌ No	✅ Yes

3. Class & Structure

Class

Class in F♯ ≡ class in C♯
→ Object-oriented building block
→ Constructor of objects containing data of defined type and methods

Definition of a class
→ Starts with `type` *(like any type in F♯)*
→ Class name generally followed by **primary constructor**

```
type CustomerName(firstName: string, lastName: string) =
    // Primary builder's body
    // Members ...
```

☝️ `firstName` and `lastName` parameters visible throughout class body

Generic class

No automatic generalization on type
→ Generic parameters to specify

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

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

Class: secondary constructor

Syntax for defining another constructor:

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

☝ Must call the primary constructor!

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

☝ Constructor parameters: only tuples, not curried!

Instantiation

Call one of the constructors, with tuple arguments
→ Don't forget `()` if no arguments, otherwise you get a function!

In a `let` binding: `new` optional and not recommended
→ `let v = Vector(1.0, 2.0)` 👌
→ `let v = new Vector(1.0, 2.0)` ❌

In a `use` binding: `new` mandatory
→ `use d = new Disposable()`

Property initialization

Properties can be initialized with setter at instantiation 👍
→ Specify them as **named arguments** in the call to the constructor
→ Place them after any constructor arguments

```fsharp
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")
```

💡 Equivalent in C#: `new PersonName("John") { Last = "Paul" }`

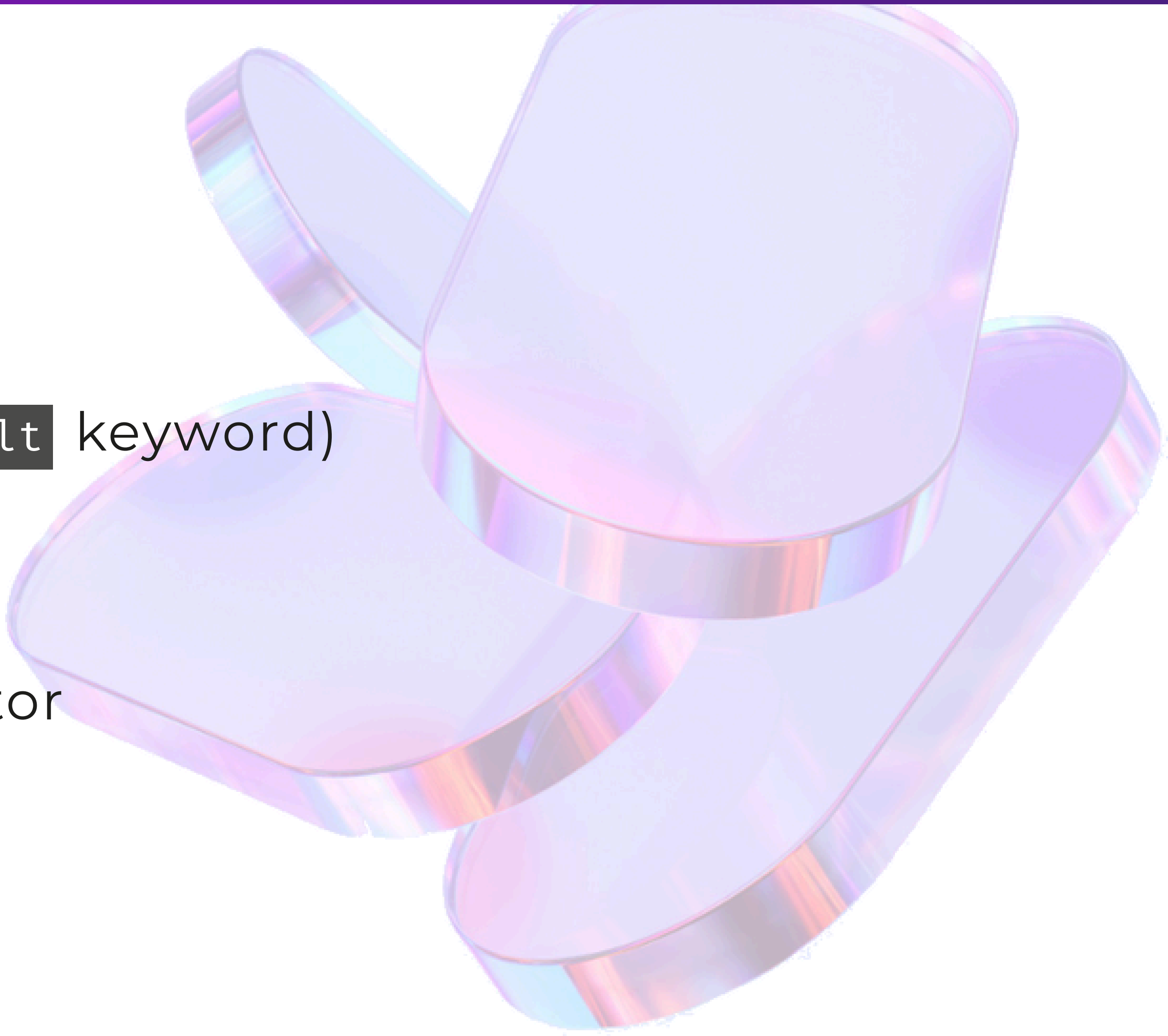
Abstract class

Annotated with `[<AbstractClass>]`

One of the members is **abstract**:

1. Declared with the `abstract` keyword
2. No default implementation (with `default` keyword)
   (Otherwise member is virtual)

Inheritance with `inherit` keyword
→ Followed by call to base class constructor



Abstract class: example

```
[<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)
```

Fields

Naming convention: camelCase

2 kind of field: implicit or explicit

- ➔ Implicit ≃ Variable inside primary constructor
- ➔ Explicit ≡ Usual class field in C♯ / Java

Implicit field

Syntax:

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

☝️ **Notes**

➔ Declared before class members

➔ Initial value mandatory

➔ Private

➔ Direct access: no need to qualify them with the `self-identifier`

Implicit instance field: example

```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!
```

Static implicit field: example

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

    // do executed for each instance at construction
    do
        count <- count + 1

    member _.CreatedCount = count

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

Explicit field

Type declaration, without initial value:

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

- ➔ `val mutable a: int` → public field
- ➔ `val a: int` → internal field `a@` + property `a => a@`

Field *vs* property

```
// Explicit fields readonly
type C1 =
    val a: int
    val b: int
    val mutable c: int
    new(a, b) = { a = a; b = b; c = 0 } // 💡 Constructor 2ndary "compact"

// VS readonly properties
type C2(a: int, b: int) =
    member _.A = a
    member _.B = b
    member _.C = 0

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

Explicit field *vs* implicit field *vs* property

Explicit field **not often used** :
→ Only for classes and structures
→ Useful with native function manipulating memory directly
   (Because fields order is preserved - see SharpLab)
→ Need a `[<ThreadStatic>]` variable
→ Interaction with F♯ class of code generated without primary constructor

Implicit field - `let` binding
→ Intermediate variable during construction

Other use cases → auto-implemented property
→ Expose a value → `member val`
→ Expose a mutable "field" → `member val ... with get, set`

Structures

Alternatives to classes, but more limited / inheritance and recursion

Same syntax as for classes, but with the addition of:

- ➔ `[<Struct>]` attribute
- ➔ Or `struct … end` block *(more frequent)*

```
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)
```

4. *Interfaces*

Interface - Syntax

Same as abstract class with:

- Only abstract members
- Without [<AbstractClass>] attribute
- With [<Interface>] attribute (optional, recommended)

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

➔ Interface name begins with I to follow .NET convention

➔ Arguments can be named *(without parentheses otherwise 💥)*

Interface implementation in a type

```fsharp
[<Interface>]
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}]"
```

⚠️ **Trap:** keywords are different per language

F♯ `interface`

≡ Java/TS `implements`

≠ C♯/Java/TS `interface`

Interface - Default implementation

F♯ 5.0 supports interfaces defining methods with default implementations written in C♯ 8+ but does not allow them to be defined directly in F♯.

⚠️ Don't confuse with `default` keyword: supported only in classes!

F♯ interface is explicit

F♯ interface implementation
≡ Explicit implementation of an interface in C♯

→ Interface methods are accessible only by *upcasting*:

```fsharp
[<Interface>]
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()  // upcast operator 📍
// [1..5]
```

Implementing a generic interface

```fsharp
[<Interface>]
type IValue<'T> =
    abstract member Get : unit → 'T

// Contrived example for demo purpose
type DoubleValue(i, s) =
    interface IValue<int> with
        member _.Get() = i

    interface IValue<string> with
        member _.Get() = s

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

Inheritance

Defined with `inherit` keyword

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

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

let child = B(1)
// Base: 1 2
// Child: 1
// val child: B
```

5. Object expression

Object expression

Expression used to implement an abstract type on the fly
→ Similar to an anonymous class in Java

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

☝️ The signature of `makeResource` is `string → System.IDisposable`.
💡 Upcasting not required, compared to interface implementation in a type.

Interface singleton

```fsharp
[<Interface>]
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 }
```

Implement 2 interfaces

Possible but unsafe usage → not recommended

```
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 = (o :?> System.IComparable).CompareTo("cde")  // ❗ Unsafe
// val i : int = -1
```

6. Object-oriented recommendations

No object orientation where F♯ is good

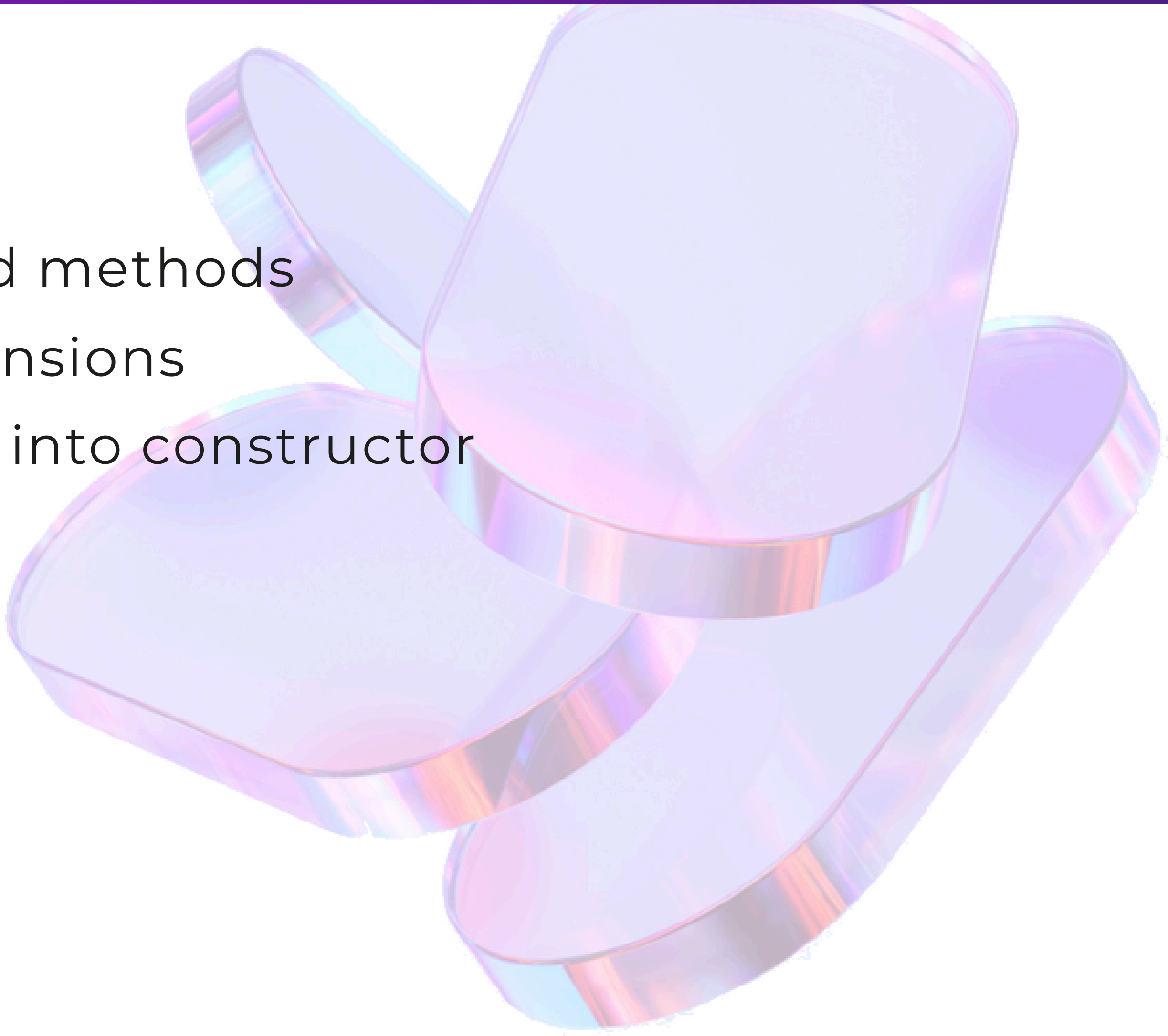
Inference works better with `function (object)` than `object.member`

Simple object hierarchy
❌ Avoid inheritance
✅ Prefer type *Union* and exhaustive *pattern matching*

Structural equality
❌ Avoid class *(equality by default reference)*
✅ Prefer a *Record* or a *Union*
❓ Consider custom structural equality for performance purposes
→ https://www.compositional-it.com/news-blog/custom-equality-and-comparison-in-f/

Object-oriented recommended use-cases

1. Encapsulate mutable state → in a class
2. Group features → in an interface
3. Expressive, user-friendly API → tuplified methods
4. API F♯ consumed in C♯ → member extensions
5. Dependency management → injection into constructor
6. Tackle higher-order functions limits



Class to encapsulate mutable state

```fsharp
// 😕 Encapsulate mutable state in a closure → impure function → counter-intuitive ⚠️
let counter =
    let mutable count = 0
    fun () ->
        count <- count + 1
        count

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

// ✅ Encapsulate mutable state in a class
type Counter() =
    let mutable count = 0 // Private field
    member _.Next() =
        count <- count + 1
        count
```

Interface grouping features

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

`serialize` and `deserialize` form a consistent group
→ Grouping them in an object makes sense

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

Interface grouping features (2)

💡 Prefer an interface to a *Record (not possible with* `Fable.Remoting`*)*

```
// ❌ Avoid: not a good use of a Record: unnamed parameters, structural comparison lost...
type Serializer<'T> = {
    Serialize: 'T → string
    Deserialize: string → 'T
}

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

→ Parameters are named in the methods
→ Object easily instantiated with an object expression

User-friendly API

```fsharp
// ❌ Avoid

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
// ✅ Favor
[<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) = ...
```

Advantages of OO implementation:

→ `Add` method overloaded *vs* `add2`, `add3` functions *(`2` and `3` = args count)*

→ Single `Log` method with `retryPolicy` optional parameter

🔗 [F♯ component design guidelines - Libraries used in C♯]()

API F♯ consumed in C♯ - Type

Do not expose this type as is:

```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)

💡 To make it easier to discover the type and use its features in C♯

- ➔ Put everything in a namespace
- ➔ Augment type with companion module functionalities

```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♯ consumed in C♯ - Type (3)

👉 The API consumed in C♯ is +/- equivalent to:

```csharp
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);
    }
}
```

Dependency management - FP based technique

Parametrization of dependencies + partial application.
→ Small-dose approach: few dependencies, few functions involved
→ Otherwise, quickly tedious to implement and to use

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

Dependency management - OO technique

Dependency injection
→ Inject dependencies into the class constructor
→ Use these dependencies in methods

👉 Offers a user-friendly API 👍

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

✅ Particularly recommended for encapsulating **side-effects** :
→ Connecting to a DB, reading settings...

⚠️ **Trap:** dependencies injected in the constructor make sense only if they are used throughout the class. A dependency used in a single method indicates a design smell.

Dependency management - Advanced FP

Dependency rejection = sandwich pattern
→ Reject dependencies in Application layer, out of Domain layer
→ Powerful and simple 👍
→ … when suitable ❗

Reader monad
→ Only if hidden inside a computation expression

Free monad + interpreter patter
→ Used in the SCM

…

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

Higher-order function limits

It's better to pass an object than a lambda
as a parameter to a higher-order function when:

1. Lambda arguments not explicit
   ❌ `let test (f: float → float → string) =...`
   ✅ Solution 1: type wrapping the 2 args `float`
   → `f: Point → string` with `type Point = { X: float; Y: float }`
   ✅ Solution 2: interface + method for named parameters
   → `type IPointFormatter = abstract Execute : x:float → y:float → string`
2. Lambda is a **command** `'T → unit`
   ✅ Prefer to trigger an side-effect via an object
   → `type ICommand = abstract Execute : 'T → unit`

Higher-order function limits (2)

3. Lambda "really" generic

```
let test42 (f: 'T → 'U) =
    f 42 = f "42"
// ❌ ^^      ~~~~
// ^^ Warning FS0064: This construct causes code to be less generic than indicated by the type annota
//                    The type variable 'T has been constrained to be type 'int'.
// ~~ Error FS0001: This expression was expected to have type 'int' but here has type 'string'
// 👉 `f: int → 'U'` expected
```

✅ Solution: wrap the function in an object

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

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

Thanks 🙏