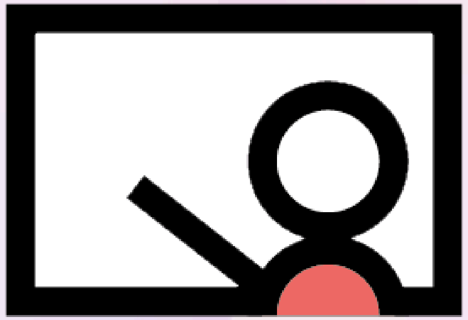
F♯ Training

Monadic Types

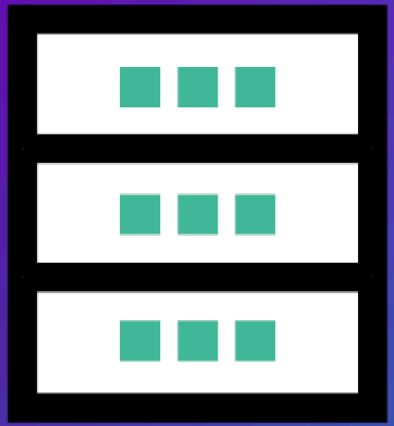
2025 April

d-edge

Table of contents



1. Type `Option`



Type `Option`

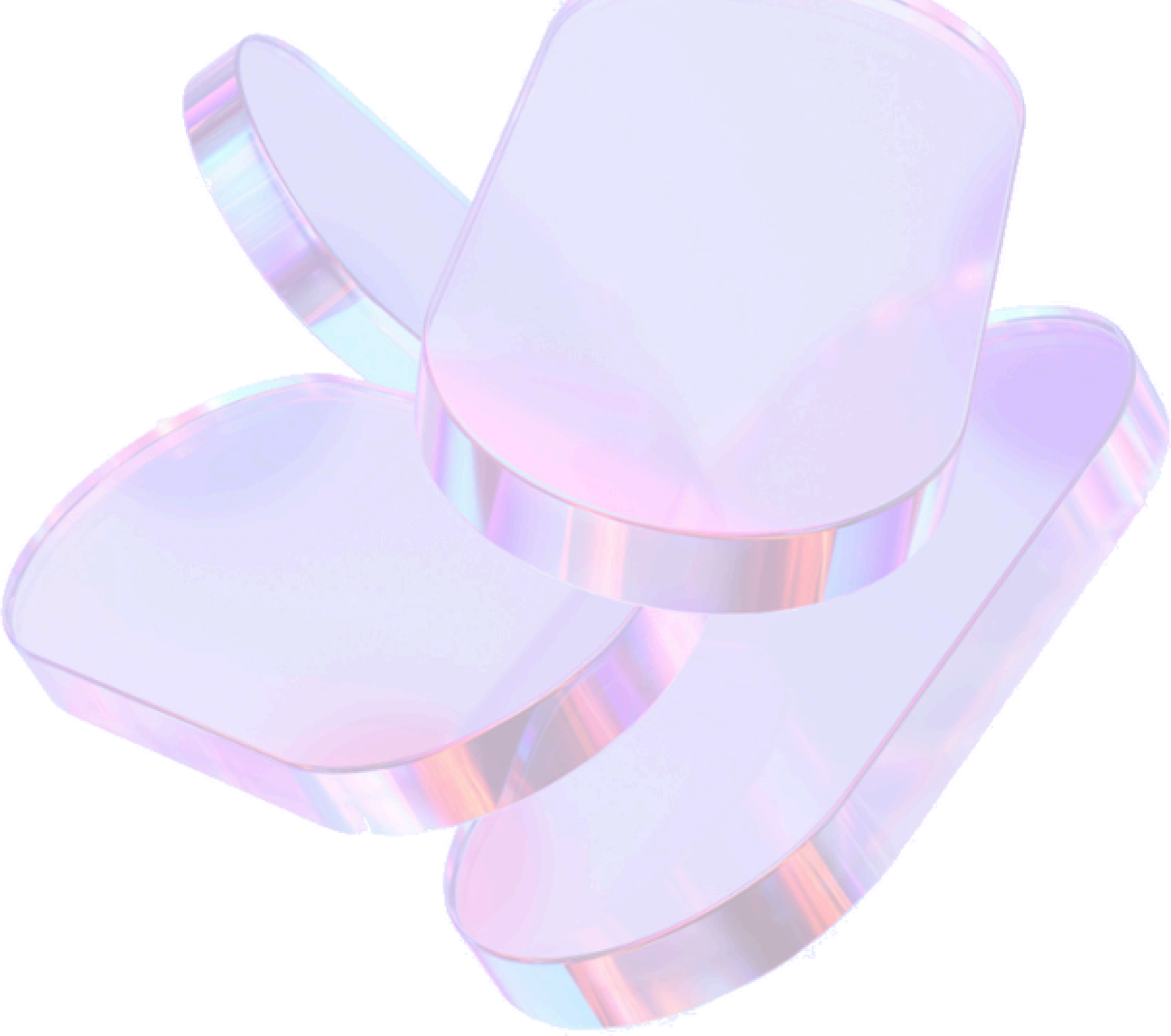
A.k.a `Maybe` *(Haskell)*, `Optional` *(Java 8)*

Models the absence of value
→ Defined as a union with 2 *cases*

```fsharp
type Option<'Value> =
    | None              // Case without data → when value is missing
    | Some of 'Value    // Case with data → when value is present
```

`Option` » Use cases

1. Modeling an optional field
2. Partial operation



Case 1: Modeling an optional field

```fsharp
type Civility = Mr | Mrs

type User = { Name: string; Civility: Civility option } with
    static member Create(name, ?civility) = { Name = name; Civility = civility }

let joey  = User.Create("Joey", Mr)
let guest = User.Create("Guest")
```

→ Make it explicit that `Name` is mandatory and `Civility` optional

☝️ **Warning:** this design does not prevent `Name = null` here *(BCL limit)*

Case 2. Partial operation

Operation where no output value is possible for certain inputs.

Example 1: inverse of a number

```fsharp
let inverse n = 1.0 / n

let tryInverse n =
    match n with
    | 0.0 -> None
    | n   -> Some (1.0 / n)
```

Function	Operation	Signature	`n = 0.5`	`n = 0.0`
`inverse`	Partial	`float -> float`	`2.0`	`infinity` ❓
`tryInverse`	Total	`float -> float option`	`Some 2.0`	`None` 👌

Case 2. Partial operation (2)

Example 2: find an element in a collection

- Partial operation: `find predicate` → 💥 when item not found
- Total operation: `tryFind predicate` → `None` or `Some item`

Benefits 👍

- Explicit, honest / partial operation
  - No special value: `null`, `infinity`
  - No exception
- Forces calling code to handle all cases:
  - `Some value` → output value given
  - `None .....` → output value missing

`Option` » Control flow

To test for the presence of the value *(of type `'T`)* in the option

- ❌ Do not use `IsSome`, `IsNone` and `Value` (🤞💥)
  - ~~if option.IsSome then option.Value...~~
- 👌 By hand with *pattern matching*.
- ✅ `Option.xxx` functions 📍

Manual control flow with *pattern matching*

Example:

```fsharp
let print option =
    match option with
    | Some x -> printfn "%A" x
    | None   -> printfn "None"

print (Some 1.0)  // 1.0
print None        // None
```

Control flow with `Option.xxx` helpers

Mapping of the inner value (of type `'T`) **if present**:
→ `map f option` with `f` total operation `'T → 'U`
→ `bind f option` with `f` partial operation `'T → 'U option`

Keep value **if present** and if conditions are met:
→ `filter predicate option` with `predicate: 'T → bool` called only if value present

🧑‍🏫 **Demo**
→ Implementation of `map`, `bind` and `filter` with *pattern matching*

🧑‍🏫 Demo » Solution

```fsharp
let map f option =                // (f: 'T → 'U) → 'T option → 'U option
    match option with
    | Some x → Some (f x)
    | None   → None               // 🎁 1. Why can't we write `None → option`?

let bind f option =               // (f: 'T → 'U option) → 'T option → 'U option
    match option with
    | Some x → f x
    | None   → None

let filter predicate option =     // (predicate: 'T → bool) → 'T option → 'T option
    match option with
    | Some x when predicate x → option
    | _ → None                    // 🎁 2. Implement `filter` with `bind`?
```

🎁 Bonus questions » Answers

```fsharp
// 🎁 1. Why can't we write `None → option`?
let map (f: 'T → 'U) (option: 'T option) : 'U option =
    match option with
    | Some x → Some (f x)
    | None   → (*None*) option  // 💥 Type error: `'U option` given ≠ `'T option` expected
```

```fsharp
// 🎁 2. Implement `filter` with `bind`?
let filter predicate option =  // (predicate: 'T → bool) → 'T option → 'T option
    option ▷ bind (fun x → if predicate x then option else None)
```

Integrated control flow » Example

```
// Question/answer console application
type Answer = A | B | C | D

let tryParseAnswer =
    function
    | "A" -> Some A
    | "B" -> Some B
    | "C" -> Some C
    | "D" -> Some D
    | _   -> None

/// Called when the user types the answer on the keyboard
let checkAnswer (expectedAnswer: Answer) (givenAnswer: string) =
    tryParseAnswer givenAnswer
    |> Option.filter ((=) expectedAnswer)
    |> Option.map (fun _ -> "✅")
    |> Option.defaultValue "❌"

["X"; "A"; "B"] |> List.map (checkAnswer B)  // ["❌"; "❌"; "✅"]
```

Integrated control flow » Advantages

Makes business logic more readable

- No `if hasValue then / else`
- Highlight the *happy path*
- Handle corner cases at the end

💡 The *computation expressions* 📍 provide an alternative syntax + lightweight

`Option`: comparison with other types

1. `Option` *vs* `List`
2. `Option` *vs* `Nullable`
3. `Option` *vs* `null`

`Option` *vs* `List`

Conceptually closed
→ Option ≃ List of 0 or 1 items
→ See `Option.toList` function: `'t option -> 't list` (`None -> []`, `Some x -> [x]`)

💡 `Option` & `List` modules: many functions with the same name
→ `contains`, `count`, `exist`, `filter`, `fold`, `forall`, `map`

☝️ A `List` can have more than 1 element
→ Type `Option` models absence of value better than type `List`

`Option` *vs* `Nullable`

`System.Nullable<'T>` ≃ `Option<'T>` but more limited

- ❗ Does not work for reference types
- ❗ Lacks monadic behavior i.e. `map` and `bind` functions
- ❗ Lacks built-in pattern matching `Some x | None`
- ❗ In F♯, no magic as in C♯ / keyword `null`

👉 C♯ uses nullable types whereas F♯ uses only `Option`

`Option` *vs* `null`

Due to the interop with the BCL, F♯ has to deal with `null` in some cases.

👉 **Good practice**: isolate these cases and wrap them in an `Option` type.

```
let readLine (reader: System.IO.TextReader) =
    reader.ReadLine() // Can return `null`
    |> Option.ofObj   // `null` becomes None

    // Same than:
    match reader.ReadLine() with
    | null -> None
    | line -> Some line
```

2. Type `Result`

Type `Result`

A.k.a `Either` *(Haskell)*

Models a *double-track* Success/Failure

```
type Result<'Success, 'Error> = // 2 generic parameters
    | Ok of 'Success   // Success Track
    | Error of 'Error // Failure Track
```

Functional way of dealing with business errors *(expected errors)*
→ Allows exceptions to be used only for exceptional errors
→ As soon as an operation fails, the remaining operations are not launched

🔗 *Railway-oriented programming (ROP)*
https://fsharpforfunandprofit.com/rop/

Module `Result`

Contains less functions than `Option` ⁉️

`map f result` : to map the success

- `('T → 'U) → Result<'T, 'Error> → Result<'U, 'Error>`

`mapError f result` : to map the error

- `('Err1 → 'Err2) → Result<'T, 'Err1> → Result<'T, 'Err2>`

`bind f result` : same as `map` with `f` returning a `Result`

- `('T → Result<'U, 'Error>) → Result<'T, 'Error> → Result<'U, 'Error>`
- 💡 The result is flattened, like the `flatMap` function on JS arrays
- ⚠️ Same type of `'Error` for `f` and the input `result`.

Quiz *Result* 🕹️

Implement `Result.map` and `Result.bind`

💡 **Tips:**

- *Map* the *Success* track
- Access the *Success* value using pattern matching

Quiz *Result* 🎲

Solution: implementation of `Result.map` and `Result.bind`

```
// ('T → 'U) → Result<'T, 'Error> → Result<'U, 'Error>
let map f result =
    match result with
    | Ok x    → Ok (f x)  // ☝️ Ok → Ok
    | Error e → Error e   // ⚠️ The 2 `Error e` don't have the same type!

// ('T → Result<'U, 'Error>) → Result<'T, 'Error>
//                           → Result<'U, 'Error>
let bind f result =
    match result with
    | Ok x    → f x       // ☝️ `f x` already returns a `Result`
    | Error e → Error e
```

`Result` : Success/Failure tracks

`map` : no track change

```
Track        Input             Operation       Output
Success ─ Ok x      ───► map( x → y ) ───► Ok y
Failure ─ Error e ───► map(  ....   ) ───► Error e
```

`bind` : eventual routing to Failure track, but never vice versa

```
Track       Input                  Operation              Output
Success ─ Ok x      ─┬─► bind( x → Ok y       ) ───► Ok y
                     └─► bind( x → Error e2 ) ─┐
Failure ─ Error e ───► bind(       ....       ) ─┴─► Error ~
```

☝️ The *mapping/binding* operation is never executed in track Failure.

`Result` *vs* `Option`

`Option` can represent the result of an operation that may fail
☝️ But if it fails, the option doesn't contain the error, just `None`

`Option<'T>` ≃ `Result<'T, unit>`
→ `Some x` ≃ `Ok x`
→ `None` ≃ `Error ()`
→ See `Result.toOption` *(built-in)* and `Result.ofOption` *(below)*

```fsharp
[<RequireQualifiedAccess>]
module Result =
    let ofOption error option =
        match option with
        | Some x -> Ok x
        | None -> Error error
```

`Result` *vs* `Option` (2)

📅 **Dates:**

- The `Option` type is part of F# from the get go
- The `Result` type is more recent: introduced in F# 4.1 (2016)
  → After numerous articles on *F# for fun and profit*

📝 **Memory:**

- The `Option` type (alias: `option`) is a regular union: a reference type
- The `Result` type is a *struct* union: a value type
- The `ValueOption` type (alias: `voption`) is a *struct* union
  → `ValueNone | ValueSome of 't`

`Result` *vs* `Option` » Example

Let's change our previous `checkAnswer` to indicate the `Error`:

```fsharp
type Answer = A | B | C | D
type Error = InvalidInput of string | WrongAnswer of Answer

let tryParseAnswer =
    function
    | "A" -> Ok A
    | "B" -> Ok B
    | "C" -> Ok C
    | "D" -> Ok D
    | s   -> Error(InvalidInput s)

let checkAnswerIs expected actual =
    if actual = expected then Ok actual else Error(WrongAnswer actual)

// ...
```

`Result` vs `Option` » Example (2)

```fsharp
// ...

let printAnswerCheck (givenAnswer: string) =
    tryParseAnswer givenAnswer
    |> Result.bind (checkAnswerIs B)
    |> function
        | Ok x                   -> printfn $"%A{x}: ✅ Correct"
        | Error(WrongAnswer x)   -> printfn $"%A{x}: ❌ Wrong Answer"
        | Error(InvalidInput s) -> printfn $"%s{s}: ❌ Invalid Input"

printAnswerCheck "X";;  // X: ❌ Invalid Input
printAnswerCheck "A";;  // A: ❌ Wrong Answer
printAnswerCheck "B";;  // B: ✅ Correct
```

3. *Smart constructor*

Smart constructor: Purpose

> Making illegal states unrepresentable

🔗 https://kutt.it/MksmkG *F♯ for fun and profit, Jan 2013*

- Design to prevent invalid states
  - Encapsulate state *(all primitives)* in an object
- *Smart constructor* guarantees a valid initial state
  - Validates input data
  - If Ko, returns "nothing" (`Option`) or an error (`Result`)
  - If Ok, returns the created object wrapped in an `Option` / a `Result`

Encapsulate the state in a type

→ *Single-case (discriminated) union* 👌 : `Type X = private X of a: 'a ...`
🔗 https://kutt.it/mmMXCo *F♯ for fun and profit, Jan 2013*

→ *Record* 👍 : `Type X = private { a: 'a ... }`
🔗 https://kutt.it/cYP4gY *Paul Blasucci, Mai 2021*

☝️ `private` keyword:
→ Hide object content
→ Fields and constructor no longer visible from outside
→ Smart constructor defined in companion module or static method

Smart constructor » Example #1

Smart constructor :
→ `tryCreate` function in companion module
→ Returns an `Option`

```fsharp
type Latitude = private { Latitude: float } // 👈 A single field, named like the

[<RequireQualifiedAccess>]                   // 👈 Optional
module Latitude =
    let tryCreate (latitude: float) =
        if latitude >= -90. && latitude <= 90. then
            Some { Latitude = latitude }     // 👈 Constructor accessible here
        else
            None

let lat_ok = Latitude.tryCreate 45.  // Some { Latitude = 45.0 }
let lat_ko = Latitude.tryCreate 115. // None
```

Smart constructor » Example #2

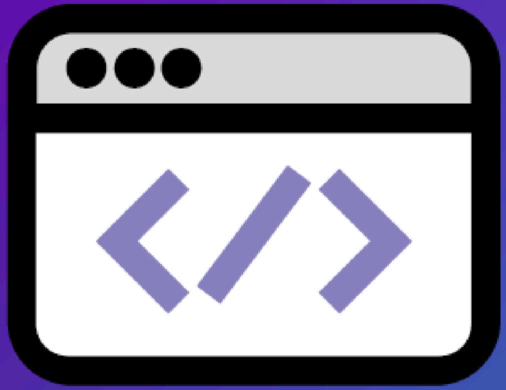
Smart constructor:
→ Static method `Of`
→ Returns `Result` with error of type `string`

```
type Tweet =
    private { Tweet: string }

    static member Of tweet =
        if System.String.IsNullOrEmpty tweet then
            Error "Tweet shouldn't be empty"
        elif tweet.Length > 280 then
            Error "Tweet shouldn't contain more than 280 characters"
        else Ok { Tweet = tweet }

let tweet1 = Tweet.Of "Hello world" // Ok { Tweet = "Hello world" }
```

4. Functional patterns 🚀



Monoid · Monad · *Functor* · *Applicative*

Languages hidden patterns

F♯ uses functional patterns under the hood

- `Option` and `Result` are monadic types
- `Async` is monadic
- Collection types `Array`, `List` and `Seq` are monadic types too!
- Computation expressions can be monadic or applicative or monoidal

C♯ uses functor and monad under the hood too, via the LINQ query syntax.

Functional patterns overview

Studied "patterns" *(a.k.a. abstractions, concepts)*
→ *Monoid, Monad, Functor, Applicative*

- Come from the *category theory*, a branch of mathematics
- Consist of
  - A container type, mainly a *generic type*
  - 1 or 2 operations on this type
  - An eventual special element/instance/value of this type
  - Some laws

Monoid definition

Etymology (Greek): `monos` *(single, unique)* · `eidos` *(form, appearance)*

≃ Type `T` defined with:

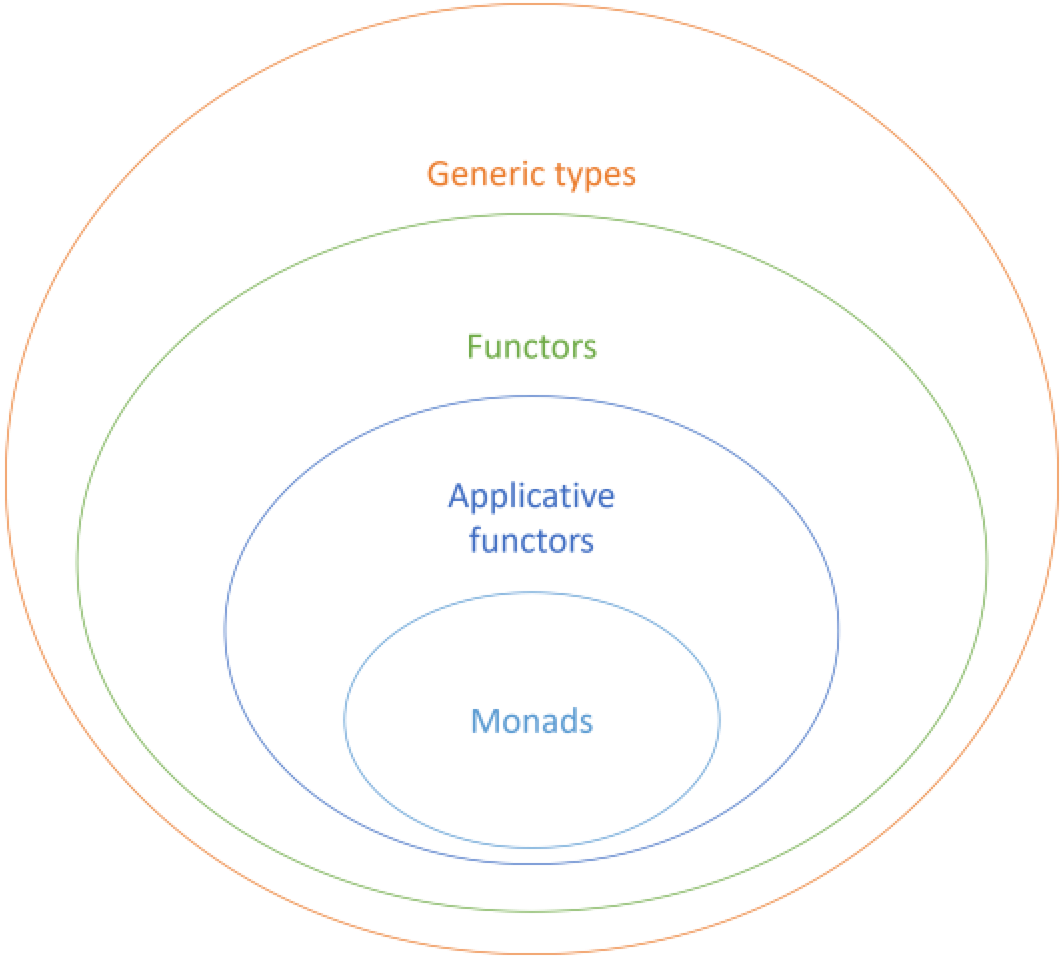
1. Binary operation `+`: `T → T → T`
   - → To *combine* 2 elements into 1
   - → **Law 1:** `+` is associative
     `a + (b + c)` ≡ `(a + b) + c`
2. *Neutral element* `e` *(a.k.a. identity)*
   - → **Law 2:** `e` is combinable with any instance `a` of `T` without effects
     `a + e` ≡ `e + a` ≡ `a`

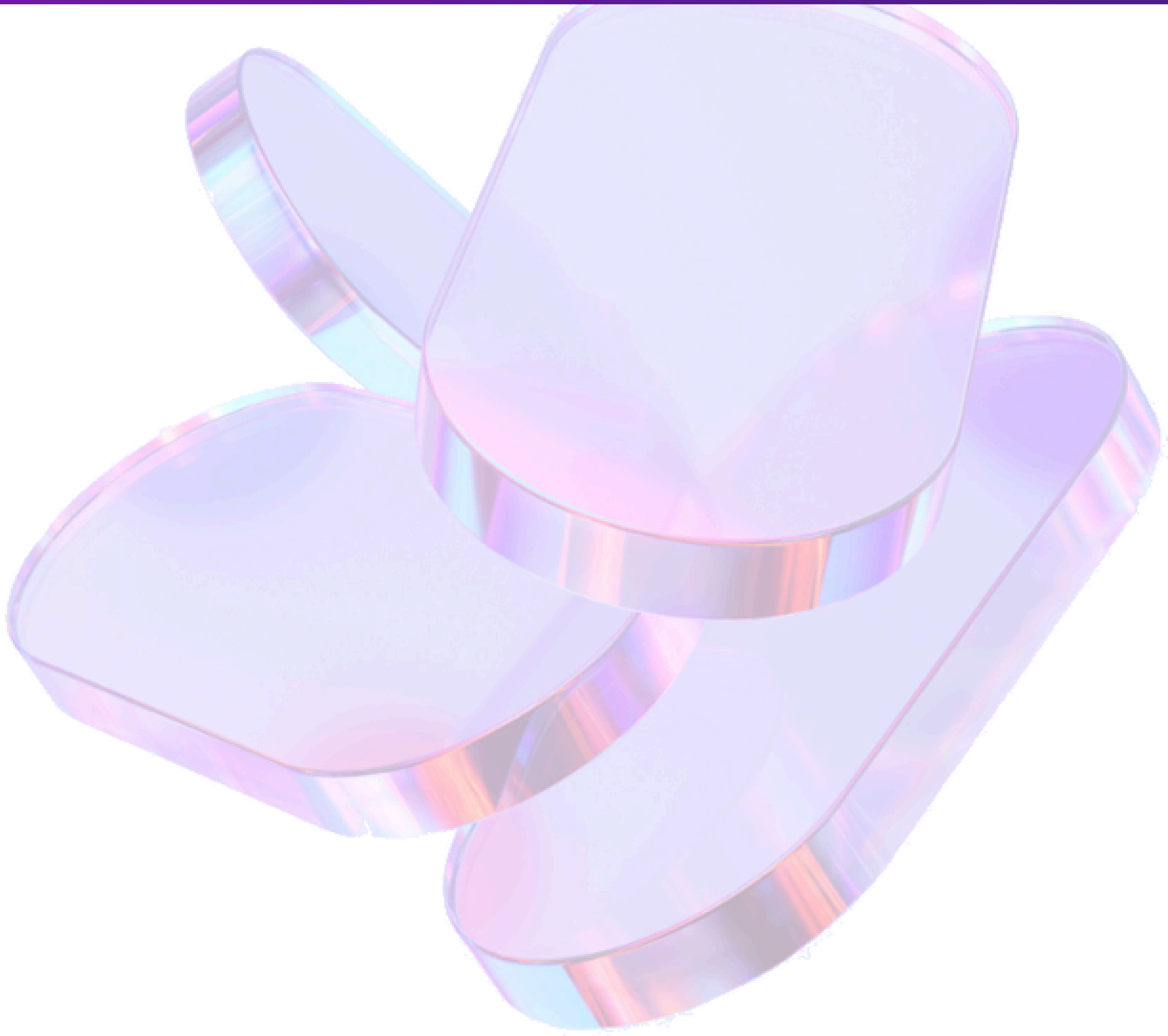
Monoid examples

Type	`+`	Identity `e`	Law 2
`int`	`+`	`0`	`i + 0 = 0 + i = i`
`int`	`*`	`1`	`i * 1 = 1 * i = i`
`string`	`+`	`""` *(empty string)*	`s + "" = "" + s = s`
`'a list`	`@` (`List.append`)	`[]` *(empty list)*	`l @ [] = [] @ l = l`
Functions	`>>` *(compose)*	`id` (`fun x -> x`)	`f >> id = id >> f = f`

💡 The monoid is a generalization of the **Composite** *OO design pattern*
🔗 Composite as a monoid *(by Mark Seemann)*

Monad big picture





🔗 [Monads Series](#) *(by Mark Seemann)*

Monad definition

≃ Any generic type, noted `M<'T>`, with:

1. Construction function `return`
   - Signature : `(value: 'T) → M<'T>`
   - ≃ Wrap a value
2. Link function `bind` *(a.k.a.* `flatMap`*)*
   - Noted `>>=` (`>` `>` `=`) as an infix operator
   - Signature : `(f: 'T → M<'U>) → M<'T> → M<'U>`
   - Take a monadic function `f`
   - Call it with the eventual wrapped value
   - Get back a new monadic value

Monad laws

`return` ≡ neutral element for `bind`

- Left: `return x |> bind f` ≡ `f x`
- Right: `m |> bind return` ≡ `m`

`bind` is associative
→ Given 2 monadic functions `f: 'a -> M<'b>` and `g: 'b -> M<'c>`

- `m >>= f >>= g` ≡ `(m >>= f) >>= g`
- `m |> bind f |> bind g` ≡ `(m |> bind f) |> bind g`

💡 `bind` allows us to chain monadic functions
☝️ Prefer using an `option` CE rather than the `>>=` bind operator

Monad *versus* Functor

A monad is also a **functor**:
→ Its `map` function can be expressed in terms of `bind` and `return`:

- Signature: `map: (f: 'T → 'U) → M<'T> → M<'U>`
- Relationship: `map f` ≡ `bind (f >> return)`

☝️ Contrary to the monad with its `return` operation, the functor is not defined with a "constructor" operation i.e. a way to put a value in the "object" of that "type". Once we have this object, the `map` preserves its structure: mapping a `List` returns another `List`.

Functor laws

Law 1 - Identity law

Mapping the `id` function over a Functor `F` should not change `F`.
→ `map id F` ≡ `F`

Law 2 - Composition law

Mapping the composition of 2 functions `f` and `g` is the same as mapping `f` and then mapping `g` over the result.
→ `map (f >> g)` ≡ `map f >> map g`

Monad alternative definition

A monad can be defined with the `flatten` operation instead of the `bind`
→ Signature: `M<M<'T>> → M<'T>`

Then, the `bind` function can be expressed in terms of `map` and `flatten`:
→ `bind` ≡ `map >> flatten`

💡 This is why `bind` is also called `flatMap`.

Monad examples

Type	Bind	Return
`Option<'T>`	`Option.bind`	`Some`
`Result<'T, _>`	`Result.bind`	`Ok`
`List<'T>`	`List.collect`	`List.singleton`

- Idem for the 2 other core collections: `Array` and `Seq`
- `Async<'T>` too but through the `async` CE 📍

Regular functions *vs* monadic functions

Pipeline

- Regular functions pipelines use the *pipe* operator `▷` (`|` `>`)
- Monadic functions pipelines use the *bind* operator `≫=` (`>` `>` `=`)

Composition

- Regular functions composition uses the compose operator `>>`
- Monadic functions composition uses the fish operator `⟹` (`>` `=` `>`)
  - Signature: `(f: 'a → M<'b>) → (g: 'b → M<'c>) → ('a → M<'c>)`
  - Definition: `let (⟹) f g = fun x → f x ▷ bind g` ≡ `f >> (bind g)`
  - A.k.a. *Kleisli composition*

Other common monads

☝️ *Rarely used in F♯, but common in Haskell*

- **Reader**: to access a read-only environment (like configuration) throughout a computation without explicitly passing it around
- **Writer**: accumulates monoidal values (like logs) alongside a computation's primary result
- **State**: manages a state that can be read and updated during a computation
- **IO**: handles side effects (disk I/O, network calls...) while preserving purity
- **Free**: to build series of instructions, separated from their execution (interpretation phase)

Applicative (Functor)

≃ Any generic type, noted `F<'T>`, with:

1. Construction function `pure` (≡ monad's `return`)
   - Signature : `(value: 'T) → F<'T>`
2. Application function `apply`
   - Noted `<*>` (same `*` than in tuple types)
   - Signature : `(f: F<'T → 'U>) → F<'T> → F<'U>`
   - Similar to functor's `map`, but whee the mapping function `'T → 'U` is wrapped in the object

Applicative laws

There are 4 laws:

- *Identity* and *Homomorphism* relatively easy to grasp
- *Interchange* and *Composition* more tricky

Law 1 - Identity

Same as the functor identity law applied to applicative:

Pattern	Equation
Functor	`map id F` ≡ `F`
Applicative	`apply (pure id) F` ≡ `F`

Applicative laws (2)

Law 2 - Homomorphism

💡 *Homomorphism* means a transformation that preserves the structure.

→ `pure` does not change the nature of values and functions so that we can apply the function to the value(s) either before or after being wrapped.

`(pure f) <*> (pure x)` ≡ `pure (f x)`
`apply (pure f) (pure x)` ≡ `pure (f x)`

Law 3 - Interchange

We can provide first the wrapped function `Ff` or the value `x` wrapped directly or captured in `(▷) x` *(partial application of the `▷` operator used as function)*

`Ff <*> (pure x)` ≡ `pure ((▷) x) <*> Ff`

💡 When `Ff` = `pure f`, we can verify this law with the homomorphism law:

```
apply Ff (pure x)            | apply (pure ((▷) x)) Ff
apply (pure f) (pure x)      | apply (pure ((▷) x)) (pure f)
pure (f x)                   | pure (((▷) x) f)
                             | pure (x ▷ f)
                             | pure (f x)
```

Applicative laws (4)

Law 4 - Composition

- Cornerstone: ensures that function composition works as expected within the applicative context.
- Hardest law, involving to wrap the `<<` operator (right to left function composition)!

`Ff <*> (Fg <*> Fx)` ≡ `(pure (<<) <*> Ff <*> Fg) <*> Fx`

💡 Same verification:

```
(pure f) <*> ((pure g) <*> (pure x))     │ (pure (<<) <*> (pure f) <*> (pure g)) <*> (pure x)
(pure f) <*> (pure g x)                  │ (pure ((<<) f) <*> (pure g)) <*> (pure x)
pure (f (g x))                           │ (pure ((<<) f g)) <*> (pure x)
pure ((f << g) x)                        │ (pure (f << g)) <*> (pure x)
                                         │ pure ((f << g) x)
```

Applicative *vs* Functor

Every applicative is a functor
→ We can define `map` with `pure` and `apply`:

```
map f x ≡ apply (pure f) x
```

💡 It was implied by the 2 identity laws.

Applicative: multi-param curried function

Applicative helps to apply to a function its arguments (e.g. `f: 'x → 'y → 'res`) when they are each wrapped (e.g. in an `Option`).

Let's try by hand:

```fsharp
let call f optionalX optionalY =
    match (optionalX, optionalY) with
    | Some x, Some y → Some(f x y)
    | _ → None
```

💡 We can recognize the `Option.map2` function.

🤔 Is there a way to handle any number of parameters?

Applicative: multi-param function (2)

The solution is to use `apply` N times, for each of the N arguments, first wrapping the function using `pure`:

```fsharp
// apply and pure for the Option type
let apply optionalF optionalX =
    match (optionalF, optionalX) with
    | Some f, Some x -> Some(f x)
    | _ -> None

let pure x = Some x

// ---

let f x y z = x + y - z
let optionalX = Some 1
let optionalY = Some 2
let optionalZ = Some 3
let res = pure f |> apply optionalX |> apply optionalY |> apply optionalZ
```

Applicative: multi-param function (3)

We can "simplify" the syntax by:

- Replacing the 1st combination of `pure` and `apply` with `map`
- Using the operators for map `<!>` and apply `<*>`

```fsharp
// ...
let res = pure f |> apply optionalX |> apply optionalY |> apply optionalZ

let res' = f <!> optionalX <*> optionalY <*> optionalZ
```

Still, it's not ideal!

Applicative - 3 styles

The previous syntax is called **“Style A”** and is not recommended in modern F♯ by Don Syme - see its [Nov. 2020 design note]().

When we use the `mapN` functions, it's called **“Style B”**.

The **“Style C”** relies on `let! … and! …` in a computation expression like `option` from `FsToolkit`. It's possible since F♯ 5 and recommended over Style B when a CE is available.

```
let res'' =
    option {
        let! x = optionalX
        and! y = optionalY
        and! z = optionalZ
        return f x y z
    }
```

Applicative *vs* Monad

The `Result` type is "monadic": on the 1st error, we "unplug".

There is another type called `Validation` that is "applicative":
≃ `Result<'ok, 'error list>`
→ Allows to accumulate all errors, here in the list in the `Error` case.
→ Handy for validating user input and reporting all errors

🔗 Resources

- [FsToolkit.ErrorHandling]
- [Validation with F# 5 and FsToolkit]

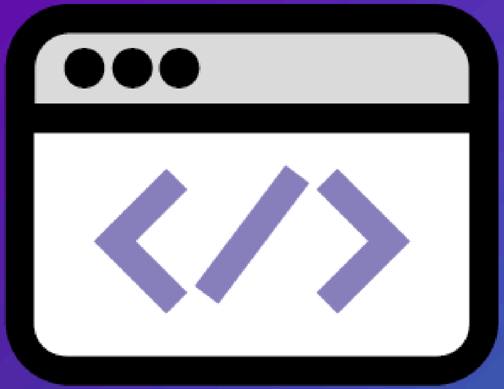
Summary

Pattern	Key points
Monoid	combinable elements: `+` operation and neutral element
Functor	mappable container
Monad	functor you can flatten
	sequential composition of effectful computations
Applicative	composition of independent effectful computations in parallel

With:

- effectful computations ≃ functions `'T → M<'T>`
- `M<'T>` the type that follows the given pattern

5. Computation expressions (CE) 🚀



CE presentation

CE = part of the F# syntax defining code blocks like `myCE { body }`

Built-in CEs: `async` and `task`, `seq`, `query`
→ Easy to use, once we know the syntax and its keywords

We can write our own CE too
→ More challenging!
→ We need to know what's behind the scene.

CE syntax

CE body looks like **imperative** F# code, with special keywords

- dedicated keywords: `yield`, `return`
- "banged" keywords: `let!`, `do!`, `yield!`, `return!`

These keywords hide a “ **machinery** ” to perform additional operations, in the background and **specific** to each CE

- Asynchronous computations like with `async` and `task`
- Effectful computations like handling a state: e.g. a sequence with `seq`
- Effectful operations like logging
- …

CE builder

A *computation expression* relies on an object called *Builder*.

⚠️ This is not exactly the *Builder* OO design pattern.

For each supported **keyword** (`let!`, `return`...), the *Builder* implements one or more related **methods**.

The 2 fundamental methods to know when writing our own CE:

- `builder.Return(expr)` used to handle the `return` keyword
- `builder.Bind(expr, f)` used for `let!` keyword

💡 Looks familiar, no? Hello, monads!

CE builder (2)

The *builder* can handle a state of its own type:

- `async { return x }` returns an `Async<'X>` type
- `seq { yield x }` is a `'x seq`

The logger CE *(next slide)* has no underlying state, no wrapping type.

Builder example: `logger`

Need: log the intermediate values of a calculation

```
let log value = printfn $"{value}"

let loggedCalc =
    let x = 42
    log x   // ❶
    let y = 43
    log y   // ❶
    let z = x + y
    log z   // ❶
    z
```

Issues ⚠️

① *Verbose*: the `log x` interfere with reading
② *Error prone*: easy to forget to log a value,
or to log the wrong variable after a bad copy-paste-update...

Builder example: `logger` - Code

💡 Make logs implicit in a CE by implementing a custom `let!` / `Bind()` :

```fsharp
type LoggingBuilder() =
    let log value = printfn $"{value}"; value
    member _.Bind(x, f) = x ▷ log ▷ f
    member _.Return(x) = x

let logger = LoggingBuilder()

//---

let loggedCalc = logger {
    let! x = 42
    let! y = 43
    let! z = x + y
    return z
}
```

Builder example: `logger` - Desugaring

The 3 consecutive `let!` is translated into 3 **nested** `Bind`:

```
// let! x = 42
logger.Bind(42, (fun _arg1 →
    let x = _arg1

    // let! y = 43
    logger.Bind(43, (fun _arg2 →
        let y = _arg2

        // let! z = x + y
        logger.Bind(x + y, (fun _arg3 →
            let z = _arg3
            logger.Return(z))
        ))
    ))
)
```

CE desugaring: tips 💡

I found a simple way to desugar a computation expression:
→ Write a failing unit test and use [Unquote](#) - 🔗 [Example](#)

```fsharp
open Swensen.Unquote
open Xunit
open Xunit.Abstractions

type LoggerBuilder(logMessage: string -> unit) = ...

type LoggerTests(output: ITestOutputHelper) =
    [<Theory>]
    [<InlineData("✅", 85)>]
    [<InlineData("❌", 850)>]
    member _.``3 Binds`` (_, expected) =
        test
          <@ logger {
              let! x = 42
              let! y = 43
              let! z = x + y
              return z
          } = expected @>
```

```
LoggerTests (2 tests) Failed: One or more child tests fail
  3 Binds (2 tests) Failed: One or more child tests failed
    ✓ 3 Binds<String>(_arg1: "✅", expected: 85) Success
    ⊖ 3 Binds<String>(_arg1: "❌", expected: 850) Failed
```

```
(fun builder@ -> builder@.Bind(42, (fun _arg2 -> let x = _arg2 in builder@.Bind(43, (fun _arg3
 -> let y = _arg3 in builder@.Bind(x + y, (fun _arg4 -> let z = _arg4 in builder@.Return(z))))
 )))) _.logger = expected
(fun builder@ -> builder@.Bind(42, (fun _arg2 -> let x = _arg2 in builder@.Bind(43, (fun _arg3
 -> let y = _arg3 in builder@.Bind(x + y, (fun _arg4 -> let z = _arg4 in builder@.Return(z))))
 )))) FSharpTraining.LoggerTests.logger = 850
```

Builder example: `option`

Need: successively find in maps by identifiers

1. `roomRateId` → `policyCode`
2. `policyCode` → `policyType`
3. `policyCode` and `policyType` → `result`

```fsharp
// 1: with match expressions → nesting!
match policyCodesByRoomRate.TryFind(roomRateId) with
| None -> None
| Some policyCode ->
    match policyTypesByCode.TryFind(policyCode) with
    | None -> None
    | Some policyType -> Some(buildResult policyCode policyType)
```

Builder example: `option` (2)

```fsharp
// 2: with Option module helpers → terser but harder to read
policyCodesByRoomRate.TryFind(roomRateId)
|> Option.bind (fun policyCode -> policyCode, policyTypesByCode.TryFind(policyCode))
|> Option.map (fun (policyCode, policyType) -> buildResult policyCode policyType)
```

Builder example: `option` (3)

```fsharp
// 3: with an option CE → both terse and readable 🎉

type OptionBuilder() =
    member _.Bind(x, f) = x ▷ Option.bind f
    member _.Return(x) = Some x

let option = OptionBuilder()

// ---

option {
    let! policyCode = policyCodesByRoomRate.TryFind(roomRateId)
    let! policyType = policyTypesByCode.TryFind(policyCode)
    return buildResult policyCode policyType
}
```

CE monoidal

The builder of a monoidal CE *(such as `seq`)* has *at least* :

- `Yield` to build the collection element by element
- `Combine` ≡ `(+)` (`Seq.append`)
- `Zero` ≡ neutral element (`Seq.empty`)

Generally added (among others):

- `For` to support `for x in xs do ...`
- `YieldFrom` to support `yield!`

CE monadic

The builder of a monadic CE has `Return` and `Bind` methods.

The `Option` and `Result` types are monadic.
→ We can create their own CE :

```fsharp
type OptionBuilder() =
    member _.Bind(x, f) = x ▷ Option.bind f
    member _.Return(x) = Some x

type ResultBuilder() =
    member _.Bind(x, f) = x ▷ Result.bind f
    member _.Return(x) = Ok x
```

FSharpPlus `monad` CE

FSharpPlus provides a `monad` CE
→ Works for all monadic types: `Option`, `Result`, ... and even `Lazy` 🎉

⚠️ **Limits:**

- Several monadic types cannot be mixed!
- Based on SRTP: can be very long to compile!

👉 Not recommended.

FsToolkit specific CEs

FsToolkit.ErrorHandling library provides:

- CE `option {}` specific to type `Option<'T>` *(example below)*
- CE `result {}` specific to type `Result<'Ok, 'Err>`

☝ Recommended as it is more explicit than the `monad` CE.

```
open FsToolkit.ErrorHandling

let addOptionalInt x' y' = option {
    let! x = x'
    let! y = y'
    return x + y
}

let v1 = addOptionalInt (Some 1) (Some 2) // = Some 3
let v2 = addOptionalInt (Some 1) None     // = None
```

Applicative CE

Library [FsToolkit.ErrorHandling]() offers:

- Type `Validation<'Ok, 'Err>` ≡ `Result<'Ok, 'Err list>`
- CE `validation {}` supporting `let! ... and! ...` syntax.

Allows errors to be accumulated in use cases like:

- Parsing external inputs
- *Smart constructor (example on the next slide…)*

Applicative CE: example

```fsharp
#r "nuget: FSToolkit.ErrorHandling"
open FsToolkit.ErrorHandling

type [<Measure>] cm
type Customer = { Name: string; Height: int<cm> }

let validateHeight height =
    if height <= 0<cm>
    then Error "Height must me positive"
    else Ok height

let validateName name =
    if System.String.IsNullOrWhiteSpace name
    then Error "Name can't be empty"
    else Ok name

module Customer =
    let tryCreate name height : Result<Customer, string list> =
        validation {
            let! validName = validateName name
            and! validHeight = validateHeight height
            return { Name = validName; Height = validHeight }
        }

let c1 = Customer.tryCreate "Bob" 180<cm>  // Ok { Name = "Bob"; Height = 180 }
let c2 = Customer.tryCreate "Bob" 0<cm> // Error ["Height must me positive"]
let c3 = Customer.tryCreate "" 0<cm>    // Error ["Name can't be empty"; "Height must me positive"]
```

Other CE

We've seen 2 libraries that extend F♯ and offer their CEs:

- FSharpPlus → `monad`
- FsToolkit.ErrorHandling → `option`, `result`, `validation`

Many libraries have their own DSL *(Domain Specific Language)*
Some are based on computation expression(s):

- Expecto: Testing library (`test "..." {...}`)
- Farmer: Infra as code for Azure (`storageAccount {...}`)
- Saturn: Web framework on top of ASP.NET Core (`application {...}`)

Writing our own CE

- Choose the main **behaviour**: monoidal? monadic? applicative?
  - Prefer a single behaviour unless it's a generic/multi-purpose CE
- Create a **builder** class
- Implement the main **methods** to get the selected behaviour
- Implement additional methods like `Delay` and `Run`
  - If needed to fine tune the behaviour or the performances,
  - or by the compiler!
- There is a flexibility in the signature of the methods
  - It can be tricky to get them right!
  - 🔗 Computation Expressions Workshop: 2 - Choice Builder | GitHub

Writing our own CE - Tips 💡

- Overload methods to support more use cases like different input types
  - `Async<Return<_,_>>` + `Async<_>` + `Result<_,_>`
  - `Option<_>` and `Nullable<_>`
- Get inspired by the existing codebases that provide CEs
  E.g. Tips found in [FsToolkit/OptionCE.fs]():
  - Undocumented `Source` methods
  - Force the method overload order with extension methods, to get a better code completion assistance.

🔗 [Computation Expressions Workshop: 6 - Extensions | GitHub]()

Writing our own CE - Custom operations 🚀

What: builder methods annotated with `[<CustomOperation("myOperation")>]`

Use cases: add new keywords, build a custom DSL
→ Example: the `query` core CE supports `where` and `select` keywords like LINQ

⚠️ **Warning:** you may need additional things that are not well documented:

- Additional properties for the `CustomOperation` attribute:
  - `AllowIntoPattern`, `MaintainsVariableSpace`
  - `IsLikeJoin`, `IsLikeGroupJoin`, `JoinConditionWord`
  - `IsLikeZip`...
- Additional attributes on the method parameters, like `[<ProjectionParameter>]`

🔗 [Computation Expressions Workshop: 7 - Query Expressions | GitHub]

CE benefits ✅

- **Increased Readability**: imperative-like code
- **Reduced Boilerplate**: hides a "machinery"
- **Extensibility**: we can write our own "builder" for specific logic

CE limits ⚠️

- **Compiler error messages** within a CE body can be cryptic
- **Nesting different CEs** can make the code more cumbersome
  - E.g. `async` + `result`
  - Alternative: custom combining CE - see `asyncResult` in FsToolkit
- Writing our own CE can be **challenging**
  - Implementing the right methods, each the right way
  - Understanding the underlying concepts

CE additional resources 🔗

- [Computation expressions series | F# for fun and profit]
- [F# computation expressions | Microsoft Learn]
- [Computation Expressions Workshop | GitHub]

6. Wrap up

Union types: `Option` and `Result`

- What they are used for:
  - Model absence of value and business errors
  - Partial operations made total `tryXxx`
    - *Smart constructor* `tryCreate`
- How to use them:
  - Chaining: `map`, `bind`, `filter` → *ROP*
  - Pattern matching
- Their benefits:
  - `null` free, `Exception` free → no guard clauses Cluttering the code
  - Makes business logic and *happy path* more readable

Functional patterns

Embedded in F♯ without necessarily realizing it:

- Monoids with `int`, `string`, `list` and functions
- Monads with `Async`, `List`, `Option`, `Result`...

Still, useful to know when dealing with computation expressions.

Key words to associate with each:

- Monoid: single-form, combine, composite pattern ++
- Functor: map, preserve structure
- Monad: functor, flatten, bind, sequential composition
- Applicative: functor, apply, multi-params function, parallel composition

Computation expression (CE)

- Syntactic sugar: inner syntax: standard or "banged" (`let!`)
  → Imperative-like · Easy to use
- CE is based on a *builder*
  - instance of a class with standard methods like `Bind` and `Return`
- *Separation of Concerns*
  - Business logic in the CE body
  - Machinery behind the scene in the CE builder
- Little issues for nesting or combining CEs
- Underlying functional patterns
  - Monoid → `seq` *(of composable elements and with a "zero ")*
  - Monad → `async`, `option`, `result`
  - Applicative → `validation`/`Result<'T, 'Err list>`
- Libraries: FSharpPlus, FsToolkit, Expecto, Farmer, Saturn

🔗 Additional resources

- Compositional IT *(Isaac Abraham)*
  - *Writing more succinct C# – in F#! (Part 2)* · 2020
- F# for Fun and Profit *(Scott Wlaschin)*
  - *The Option type* · 2012
  - *Making illegal states unrepresentable* · 2013
  - *The "Map and Bind and Apply, Oh my!" series* · 2015
  - *The "Computation Expressions" series* · 2013
- Extending F# through Computation Expressions
  - 📹 Video
  - 📜 Article
- Computation Expressions Workshop
- Applicatives IRL by Jeremie Chassaing

Thanks 🙏