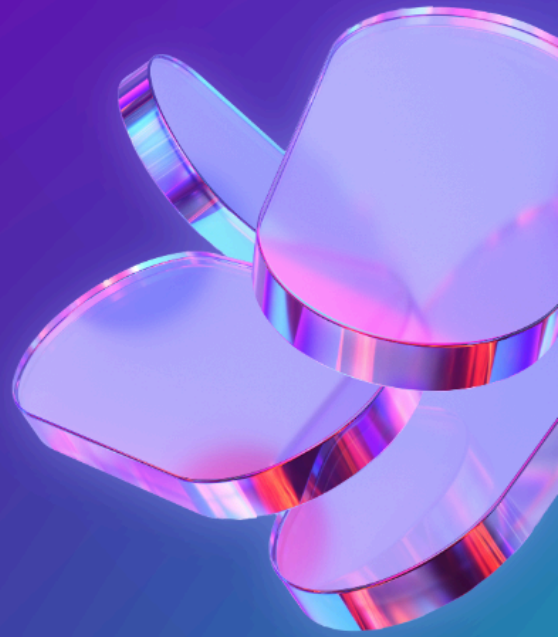
F♯ Training

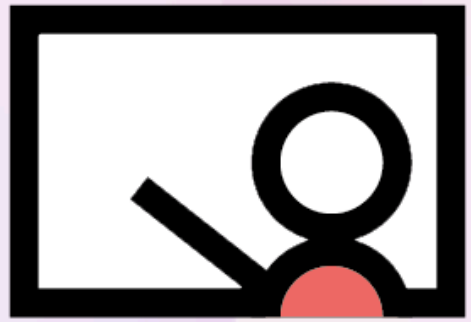
`Option` and `Result` Types

2025 April

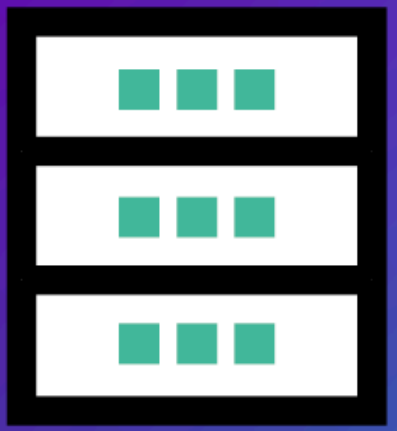


d-edge

Table of contents



1. Type `Option`

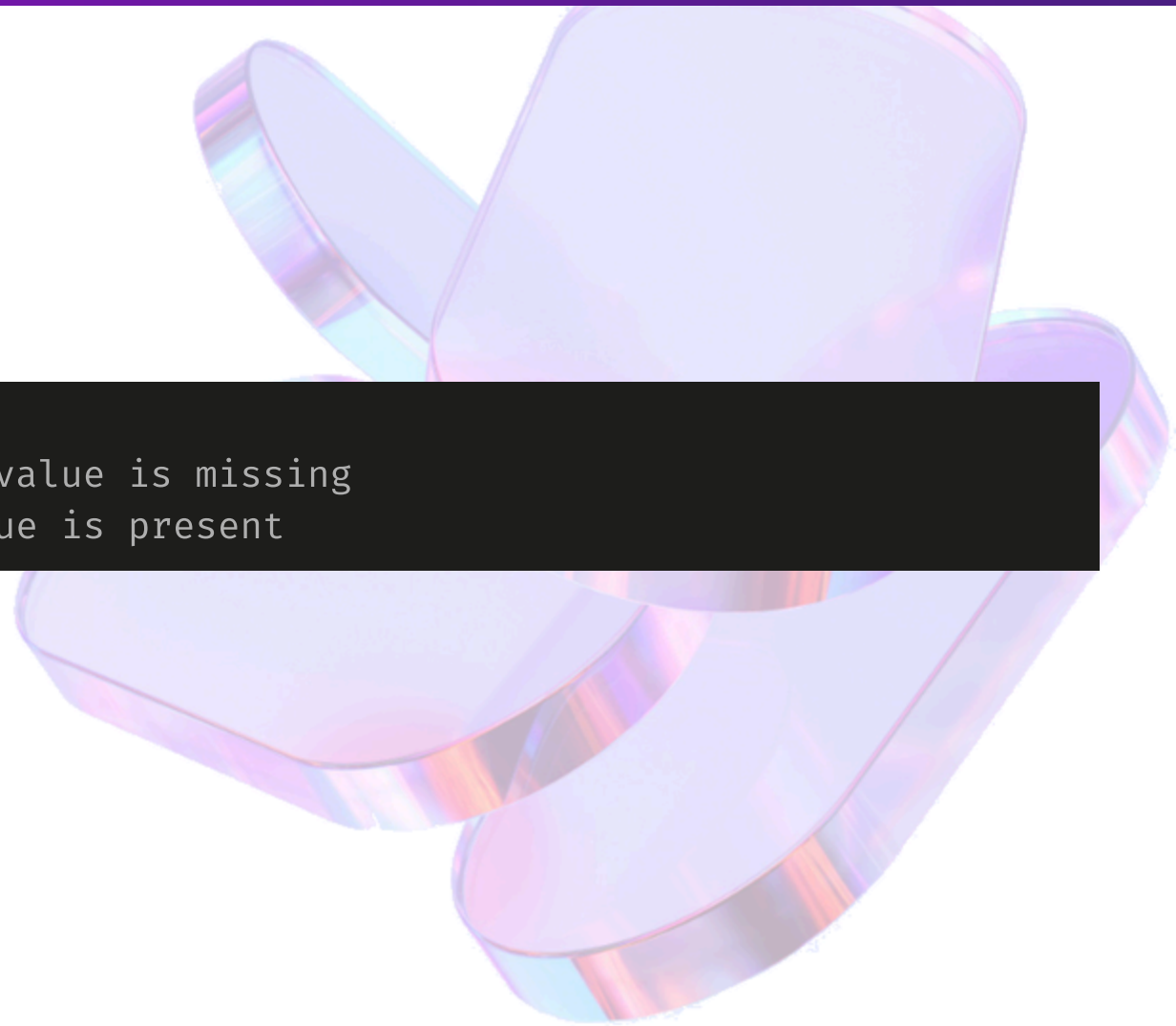


Type `Option`

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

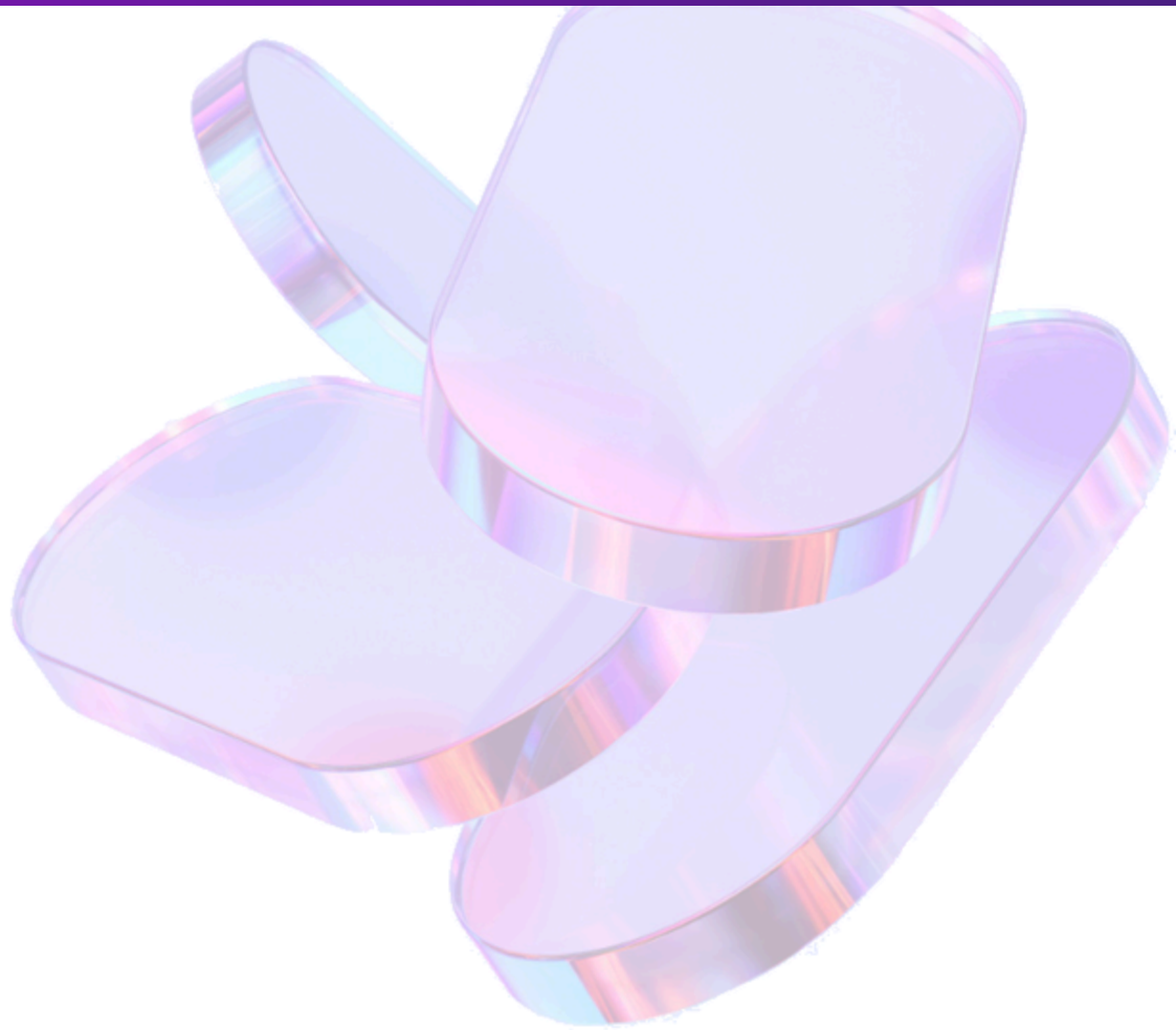
Models the absence of a value
→ Defined as a union with 2 *cases*

```
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 regarding 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:

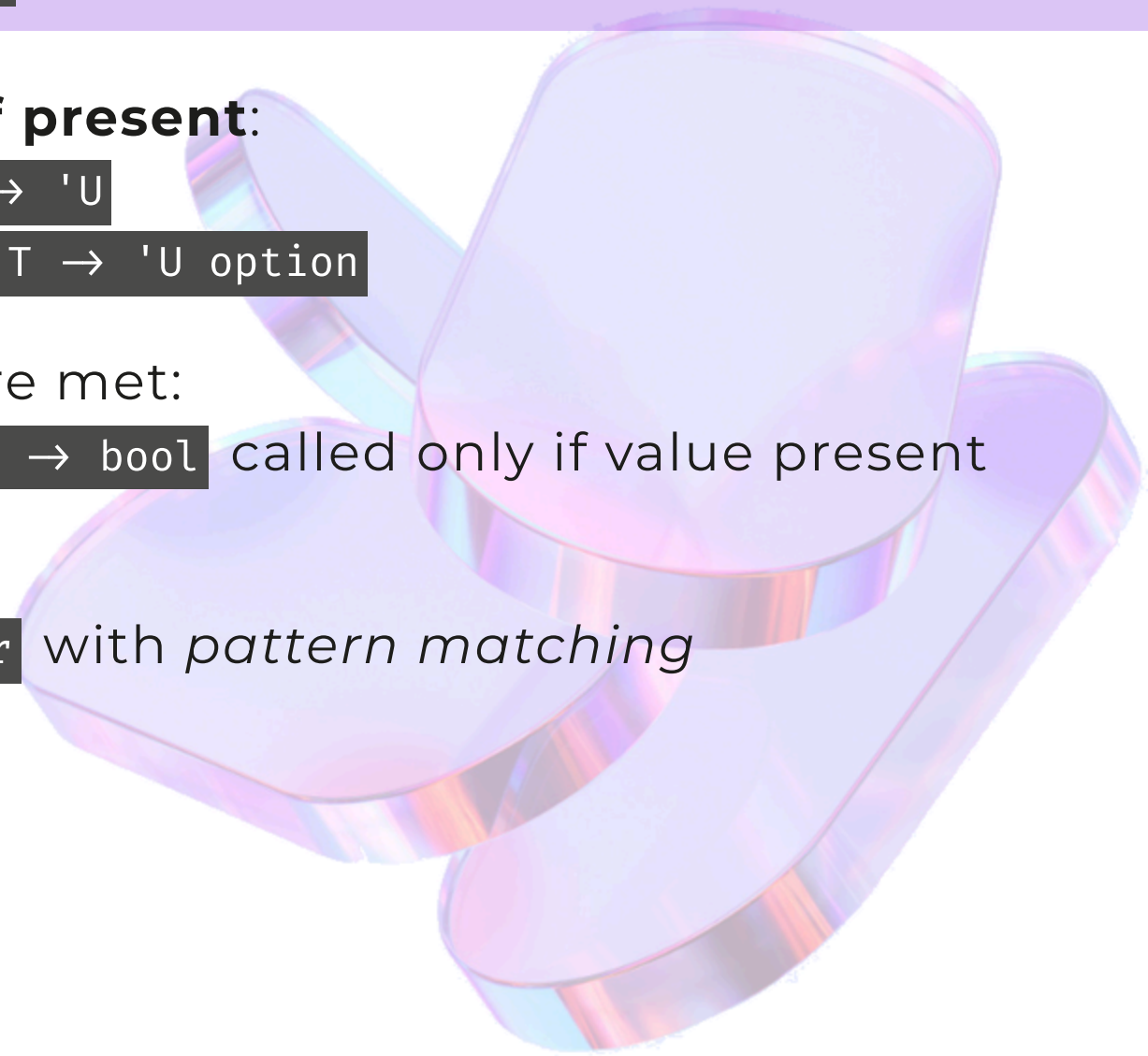
```
let print option =
    match option with
    | Some x → printfn "%A" x
    | None   → printfn "None"

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

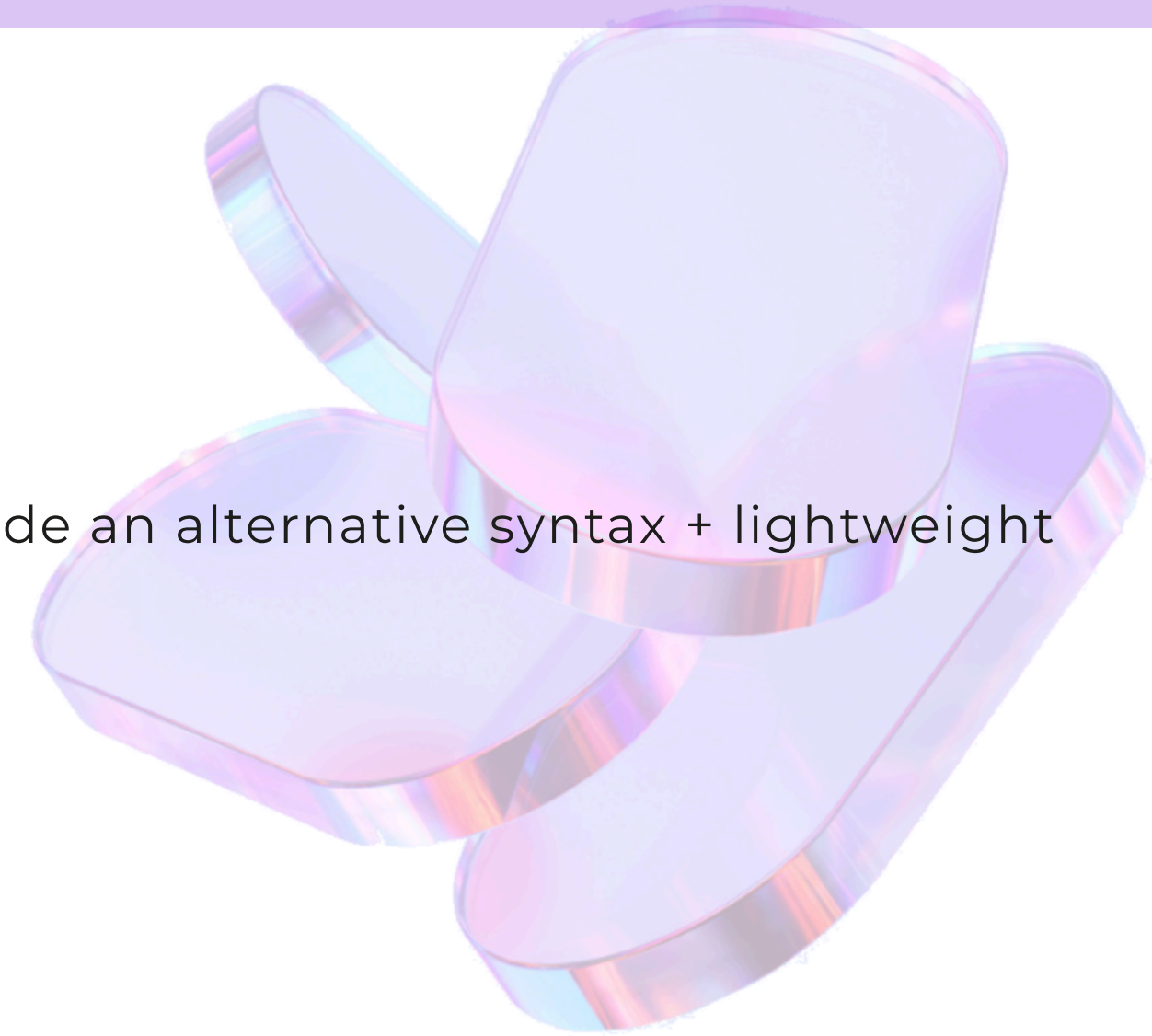
```fsharp
// Question/answer console application
type Answer = A | B | C | D

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

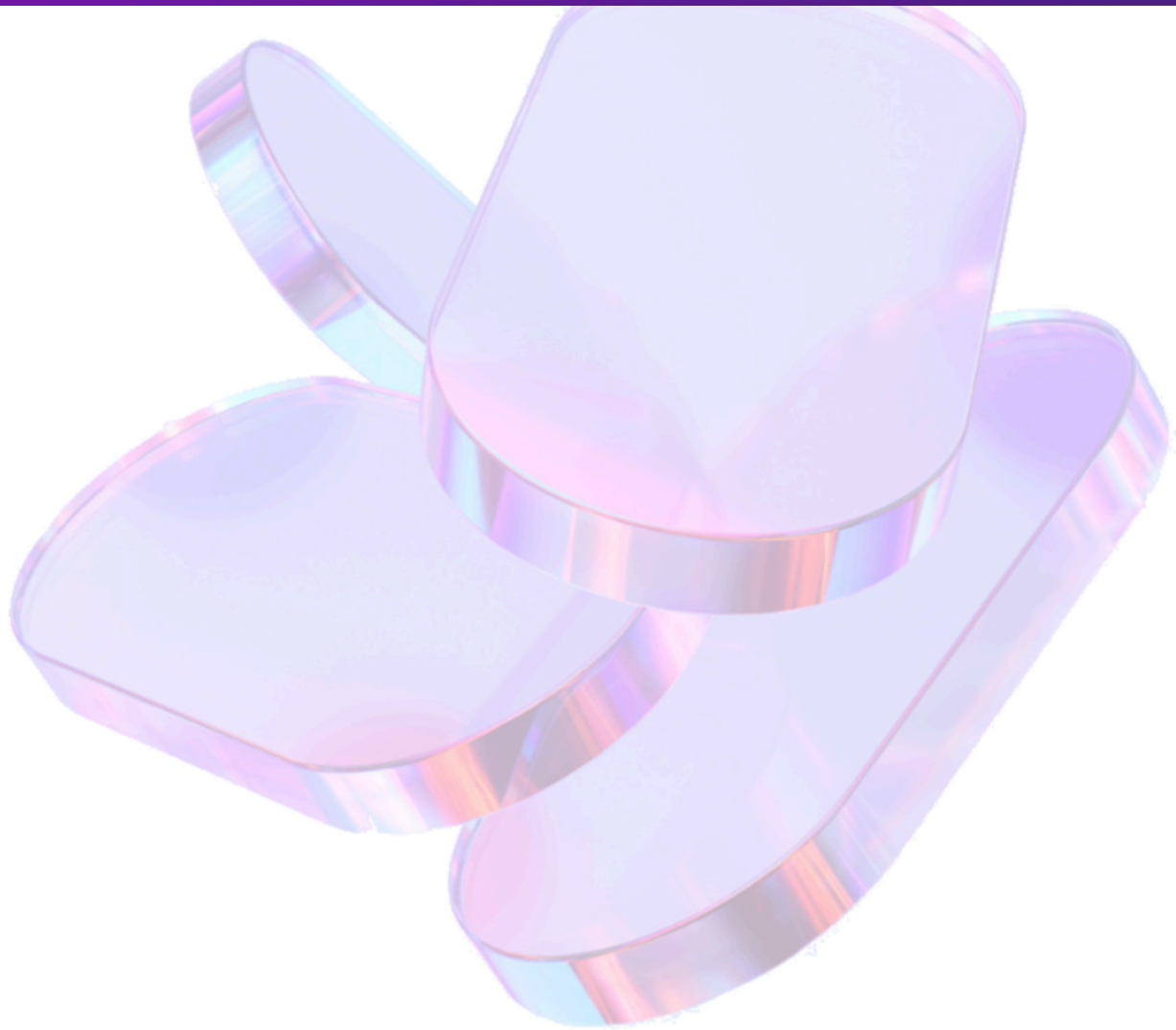
- No `if hasValue then / else`
- Highlight the *happy path*
- Handle corner cases at the end

💡 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`

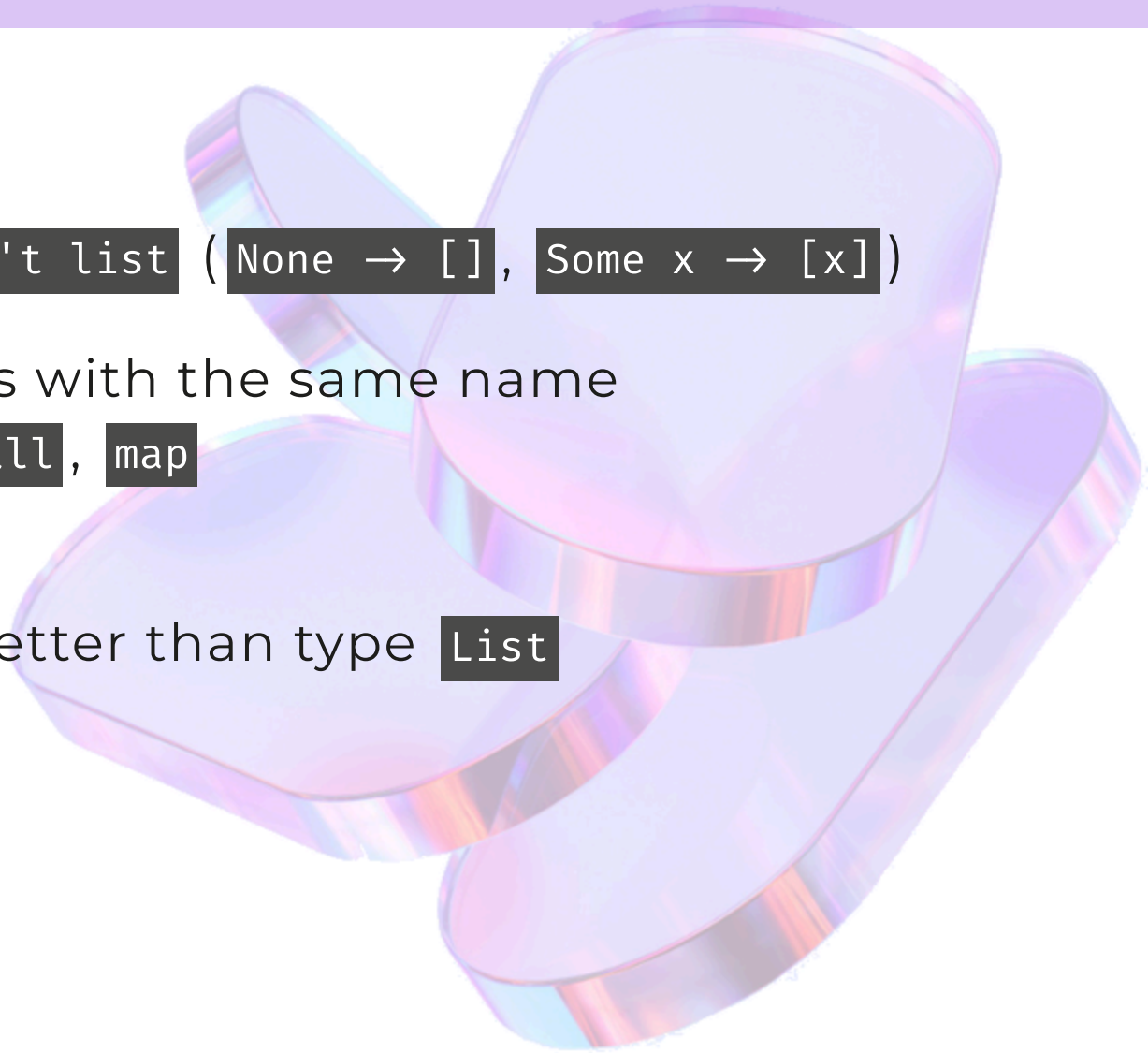


`Option` *vs* `List`

Conceptually similar
→ 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♯ with the `null` keyword

👉 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.

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

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

2. Type `Result`

Type `Result`

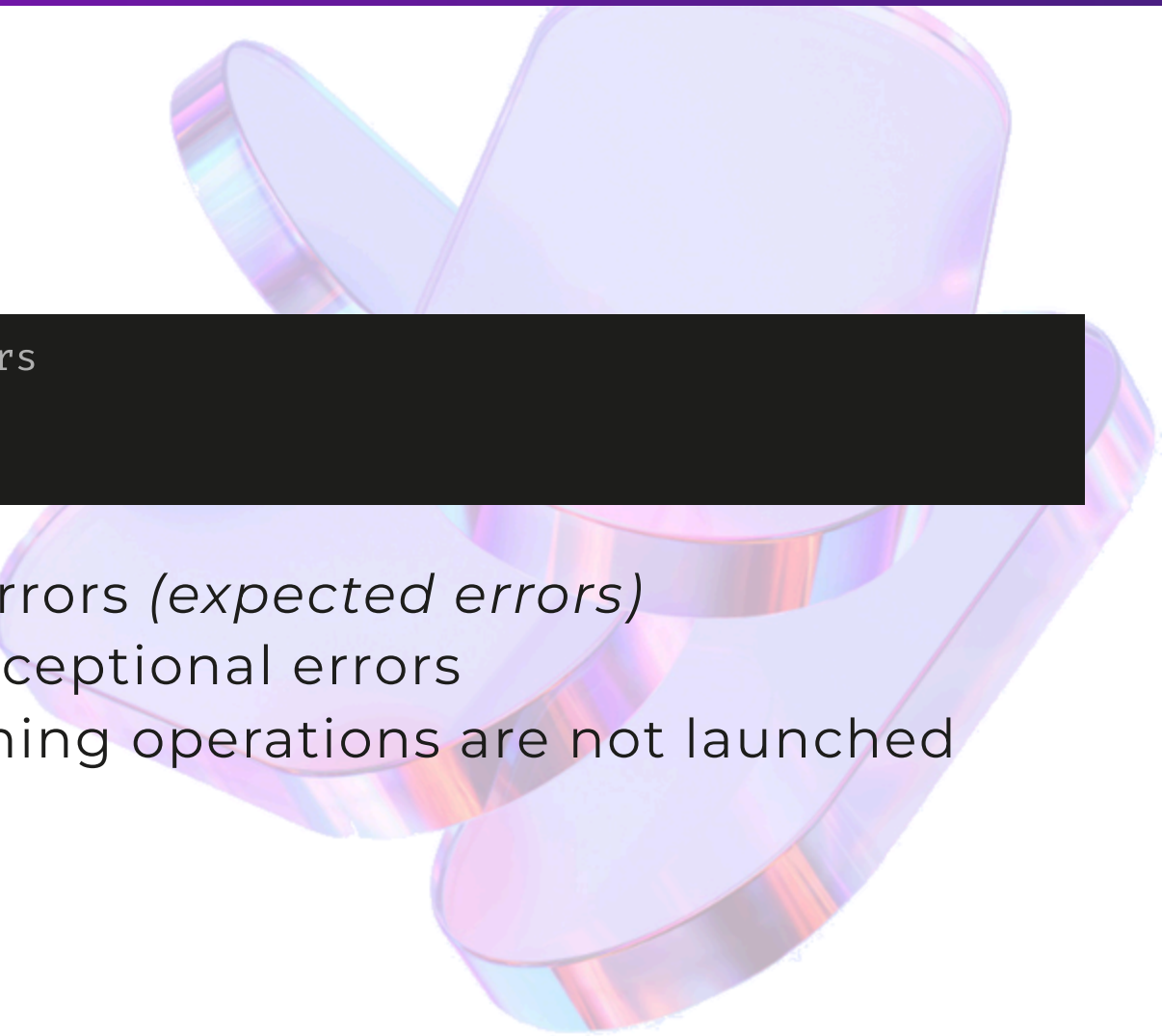
A.k.a `Either` *(Haskell)*

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 fewer functions than `Option` ⁉️

`map f result` : to map the success

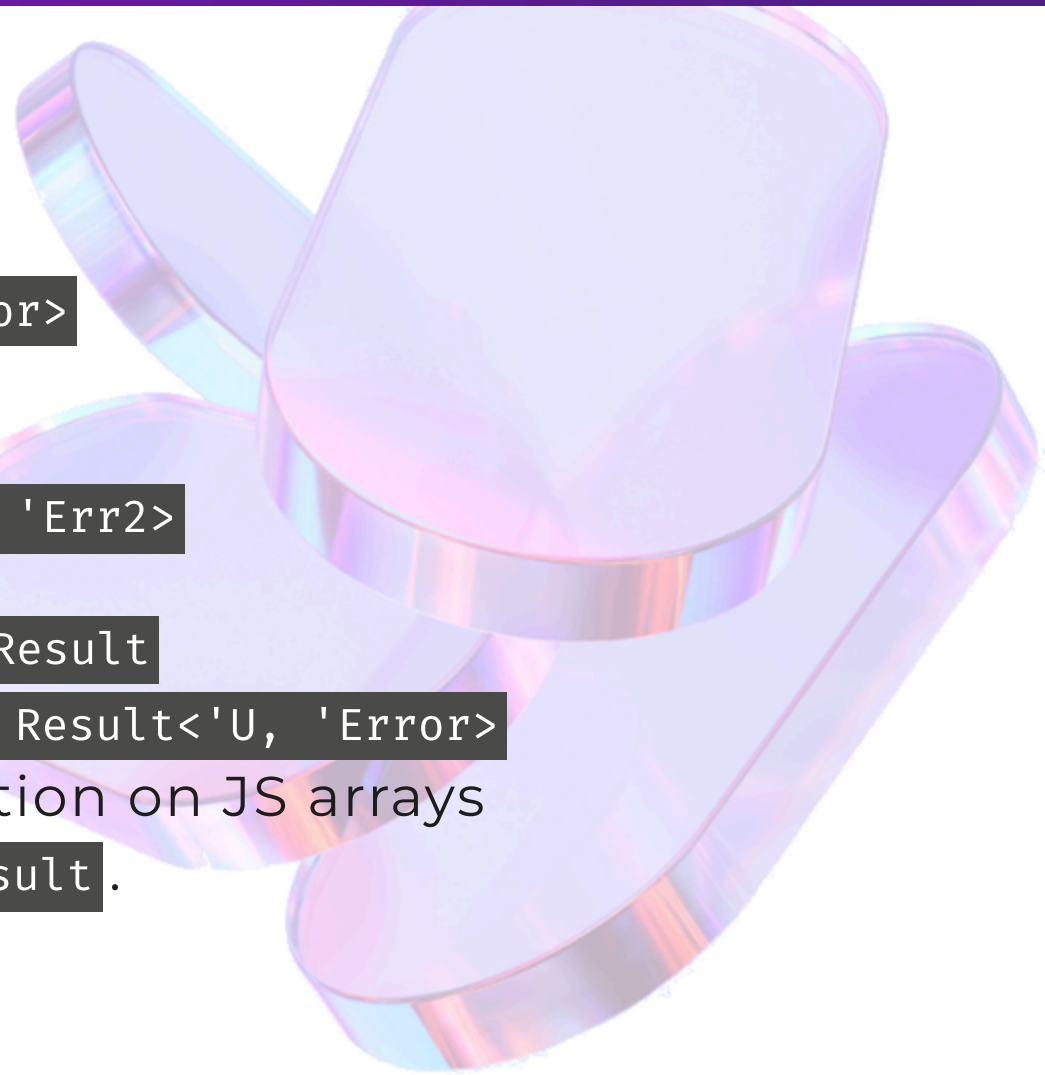
- `('T → 'U) → Result<'T, 'Error> → Result<'U, 'Error>`

`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`

```fsharp
// ('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
```

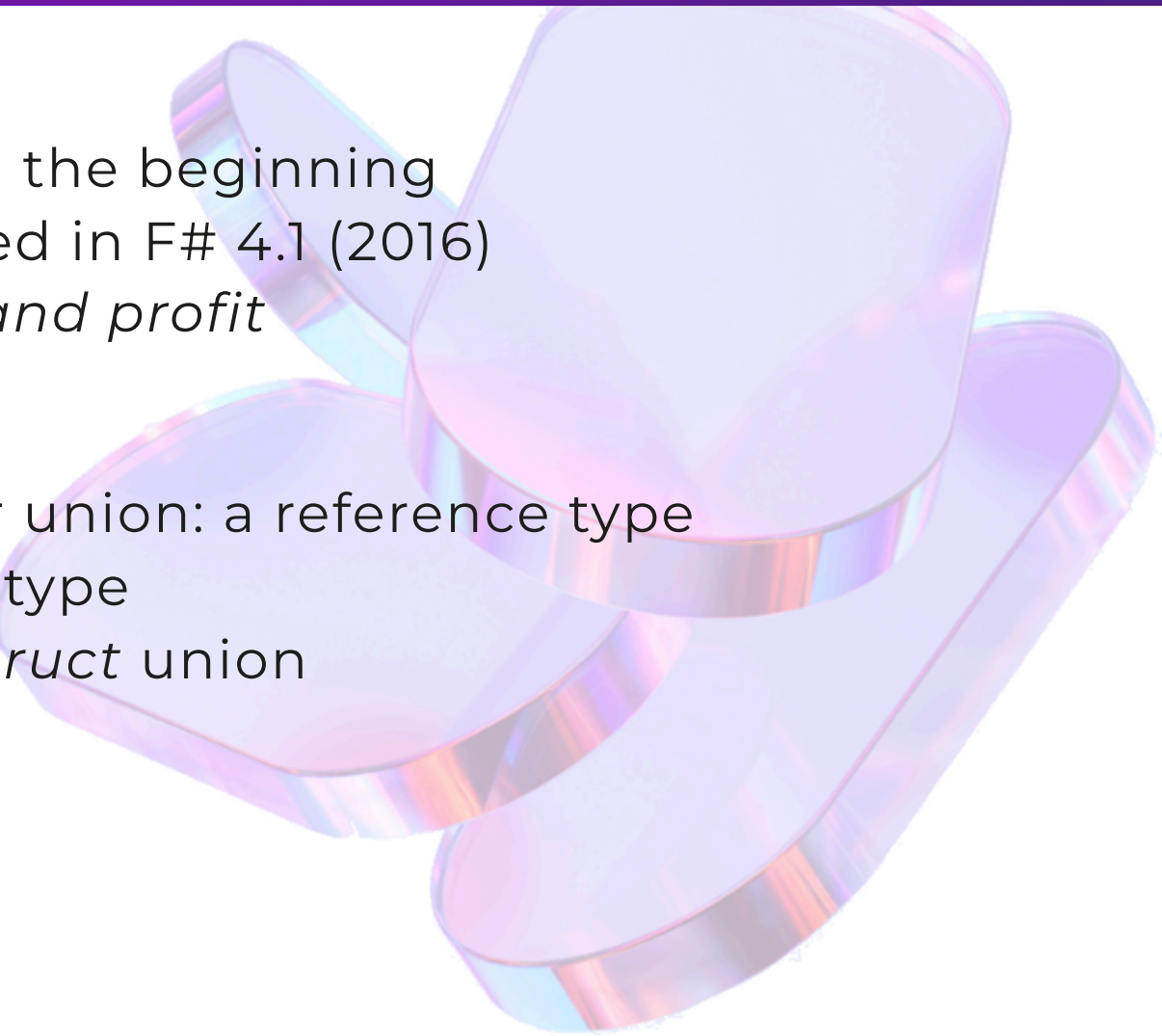
`Result` *vs* `Option` (2)

📅 Dates:

- The `Option` type has been part of F# from the beginning
- 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`:

```
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
```

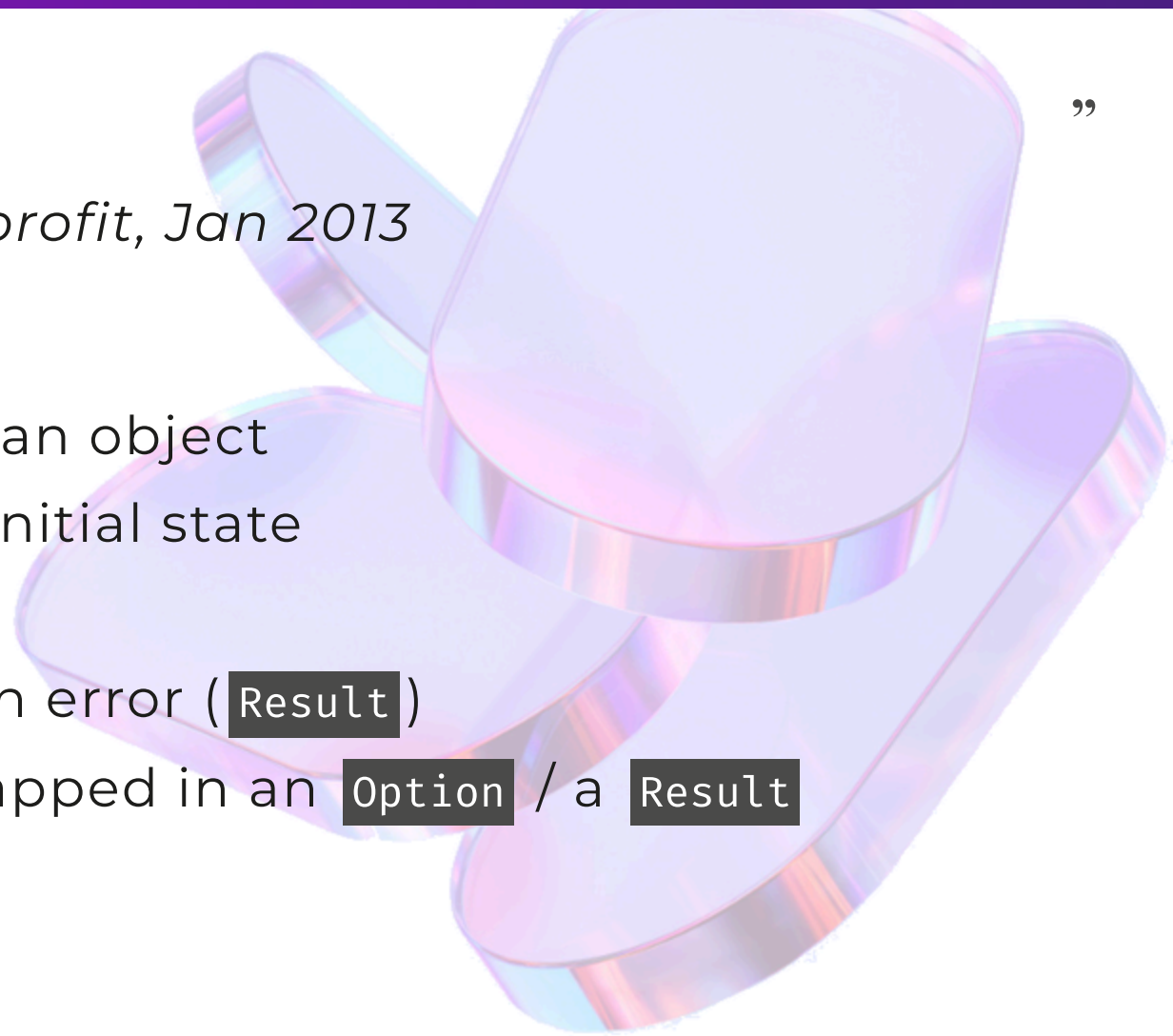
3. *Smart constructor*

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`

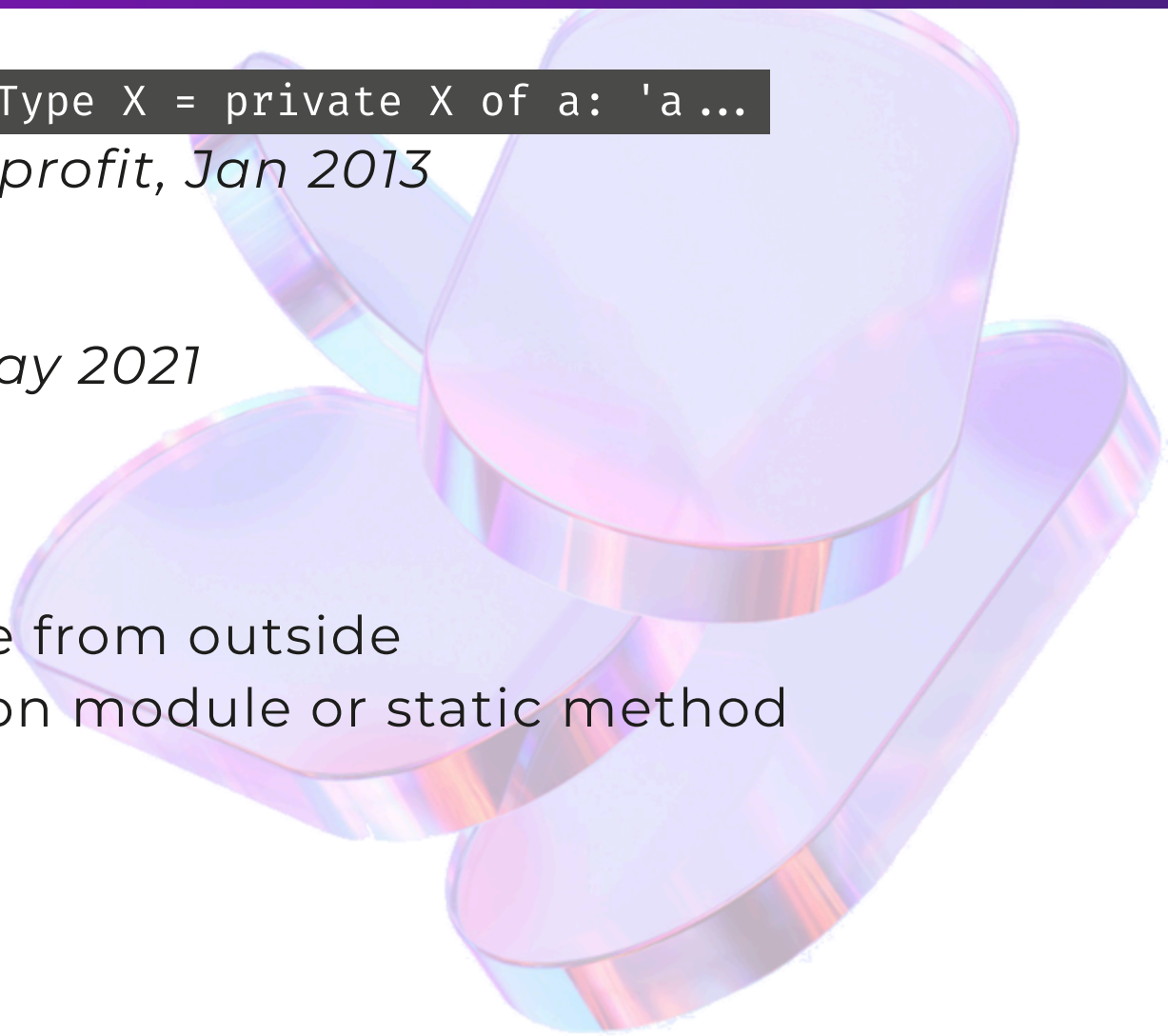


Encapsulate the state in a type

→ *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, May 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 type

[<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`

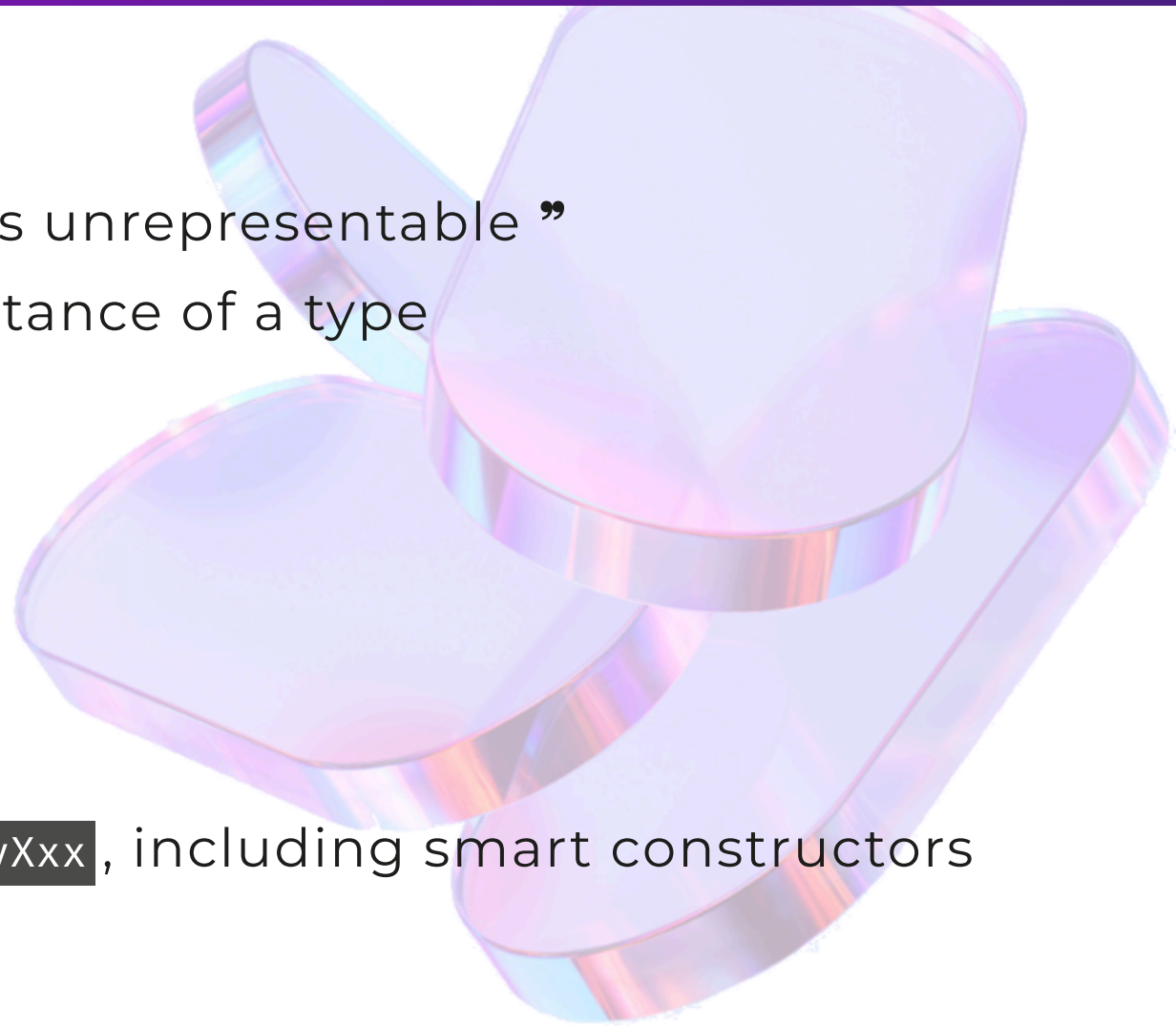
```
type Tweet =
    private { Tweet: string }

    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. Wrap up

`Option` & `Result`

Smart constructors

- 1 of the patterns to “ Make illegal states unrepresentable ”
- Function that tries to create a valid instance of a type
- Otherwise, return `None` or `Error ...`



When to use

- `Option`: model the absence of value
- `Result`: handle business errors
- Both: partial operations made total `tryXxx`, including smart constructors

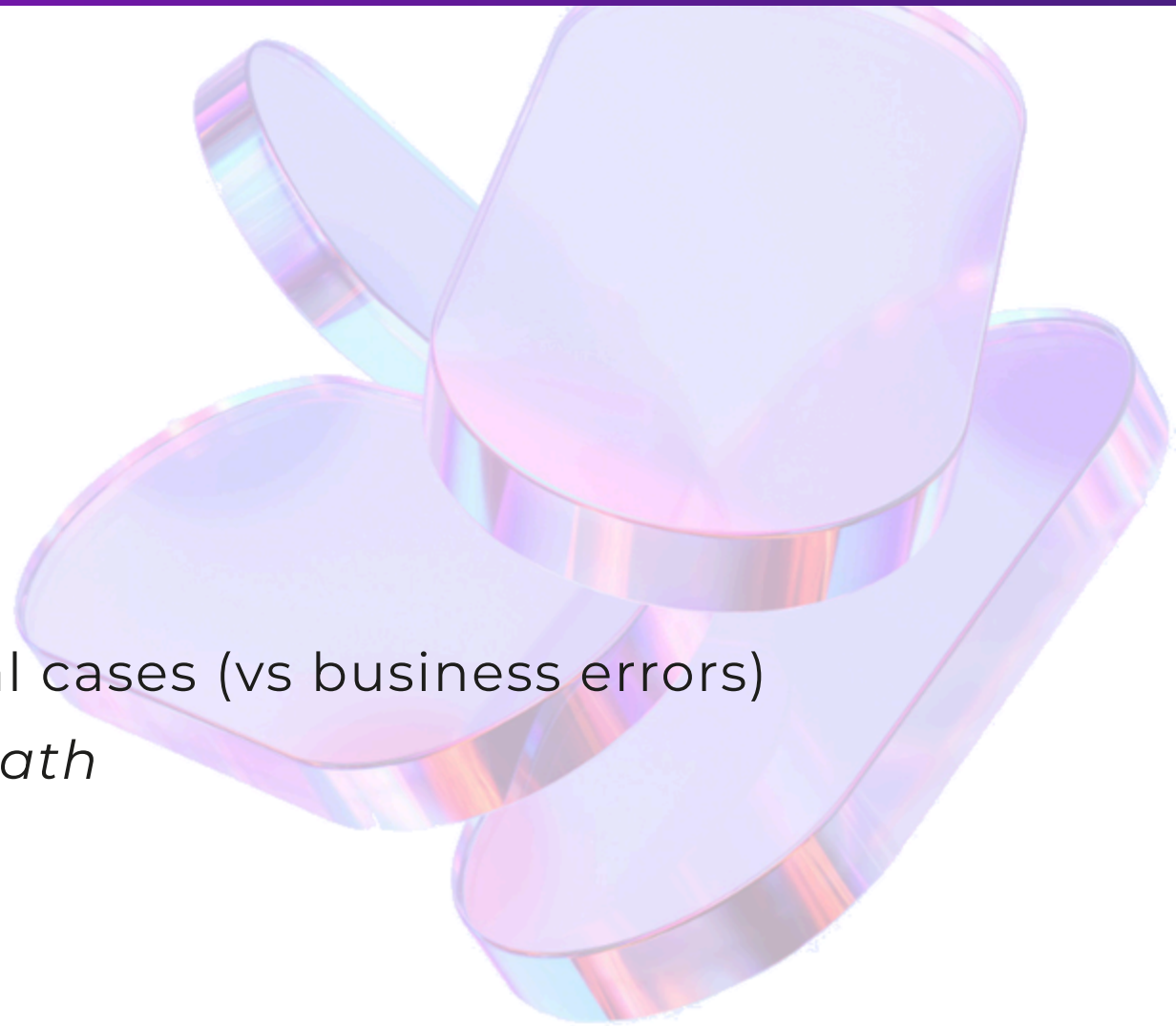
`Option` & `Result` (2)

How to use them

- Chaining helpers: `map`, `bind`, `filter`
- Pattern matching
- Computation expressions 📌

Benefits

- `null` free, `Exception` only for exceptional cases (vs business errors)
- Highlights business logic and *happy path*



🔗 Additional resources

- F# for Fun and Profit *(Scott Wlaschin)*
  - *The Option type* · 2012
  - *Making illegal states unrepresentable* · 2013
- Compositional IT *(Isaac Abraham)*
  - *Writing more succinct C# – in F#! (Part 2)* · 2020

Thanks 🙏