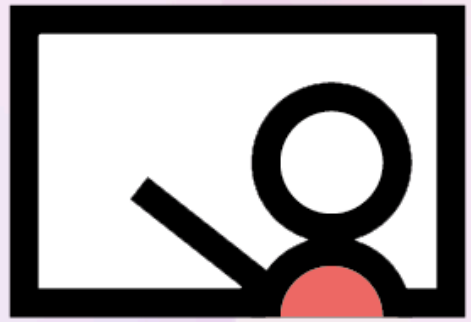
F# Training

Bases

2025 April

d-edge

Table of contents



☝ Preliminary notes

1. Symbol 📍 : indicates a concept we will see later.
2. Code is displayed using the **Fira Code** font
   🔗 github.com/tonsky/FiraCode:

```
'→' = '-' + '>'     '⩽' = '<' + '='     '⩵' = '=' + '='
'⇒' = '=' + '>'     '⩾' = '>' + '='     '≠' = '!' + '='
'◇' = '<' + '>'     '▷' = '|' + '>'     '◁' = '<' + '|'
'[|' = '[' + '|'    '|]' = '|' + ']'
```

💡 Setting in VsCode to enable ligatures: `"editor.fontLigatures": true`

💡 In Rider, same with **JetBrains Mono** font
🔗 jetbrains.com/lp/mono/

1. What is F♯?

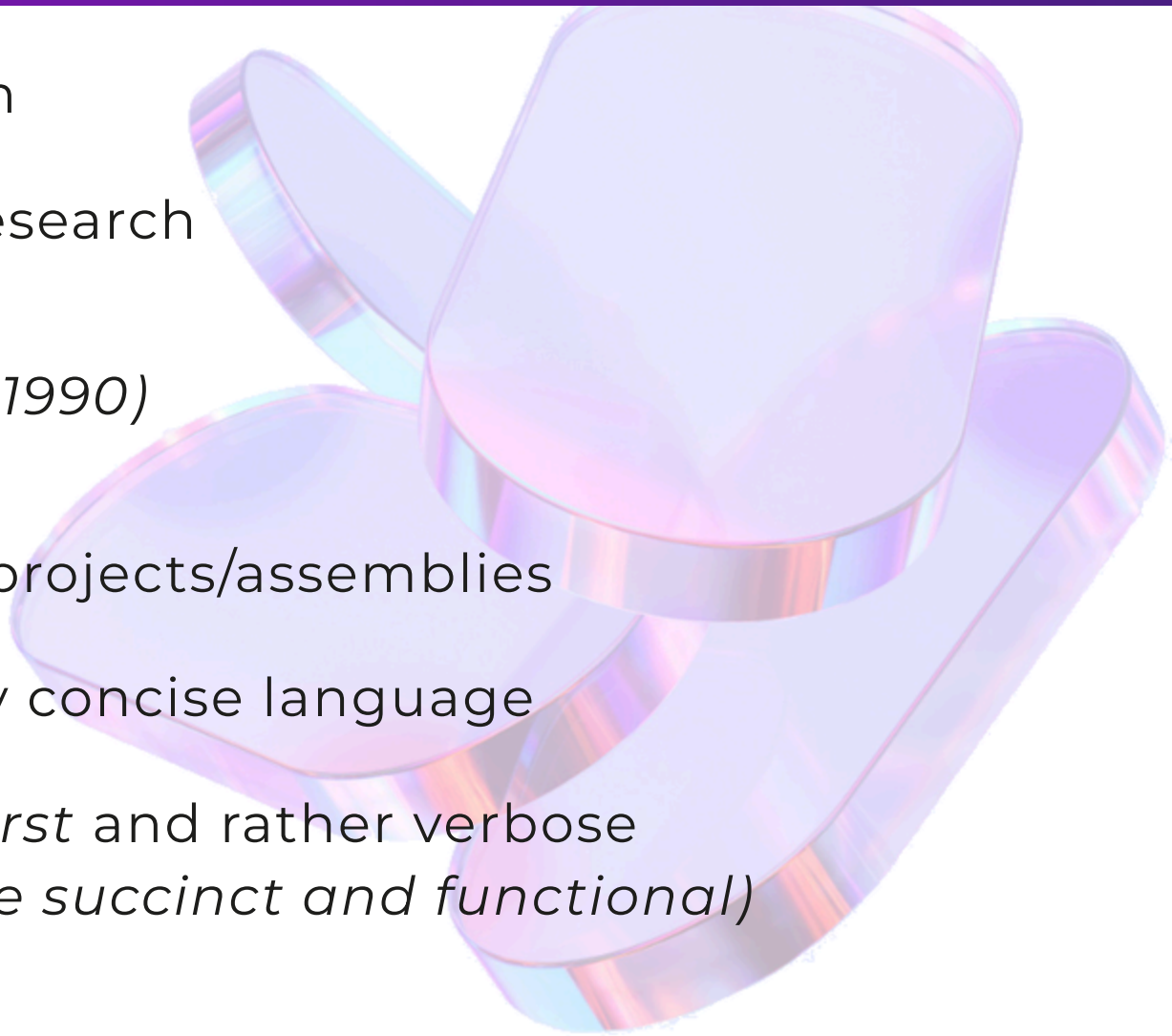
Key points

Microsoft language family - **.NET** platform

- ➔ Designer: Don Syme @ Microsoft Research
- ➔ ≃ OCaml implementation for .NET
- ➔ ≃ Inspired by Haskell *(Version 1.0 in 1990)*
- ➔ dotnet new -lang F#`
- ➔ Interoperability between C♯ and F♯ projects/assemblies

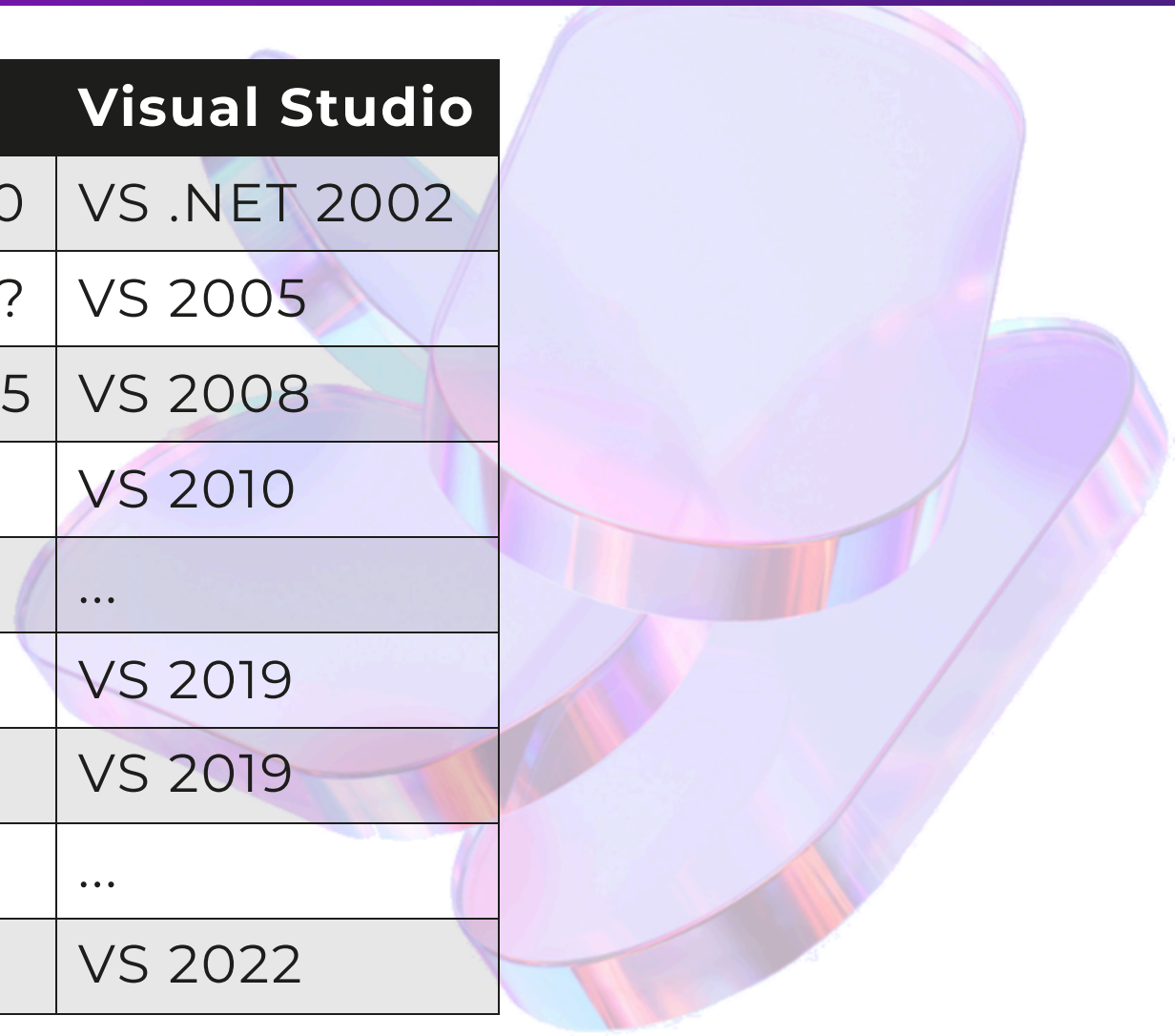
Multi-paradigm ***Functional-first*** and very concise language

Where C♯ is *imperative/object-oriented-first* and rather verbose
(even if it's inspired by F♯ to become more succinct and functional)



History

Date	C♯	F♯	.NET	Visual Studio
2002	C♯ 1.0		.NET Framework 1.0	VS .NET 2002
2005	C♯ 2.0	**F♯ 1.x**	.NET Framework ?.?	VS 2005
2007	**C♯ 3.0**		.NET Framework 3.5	VS 2008
2010	C♯ 4.0	**F♯ 2.0**	.NET Framework 4	VS 2010
…	…	…	…	…
2019	C♯ 8.0	F♯ 4.7	.NET Core 3.x	VS 2019
2020	C♯ 9.0	**F♯ 5.0**	**.NET 5.0**	VS 2019
…	…	…	…	…
2024	C♯ 13.0	F♯ 9.0	.NET 9.0	VS 2022

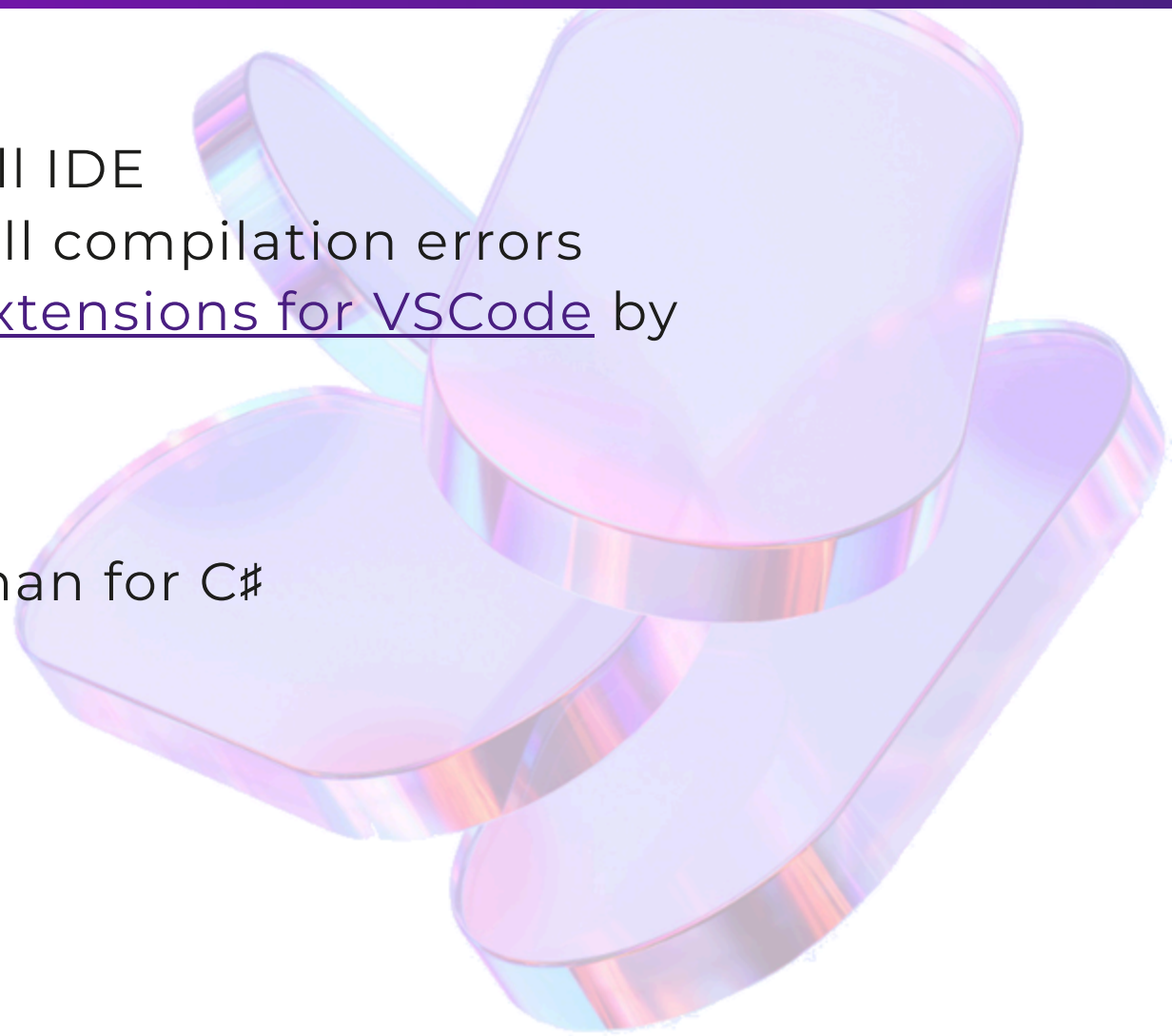


Editors / IDE

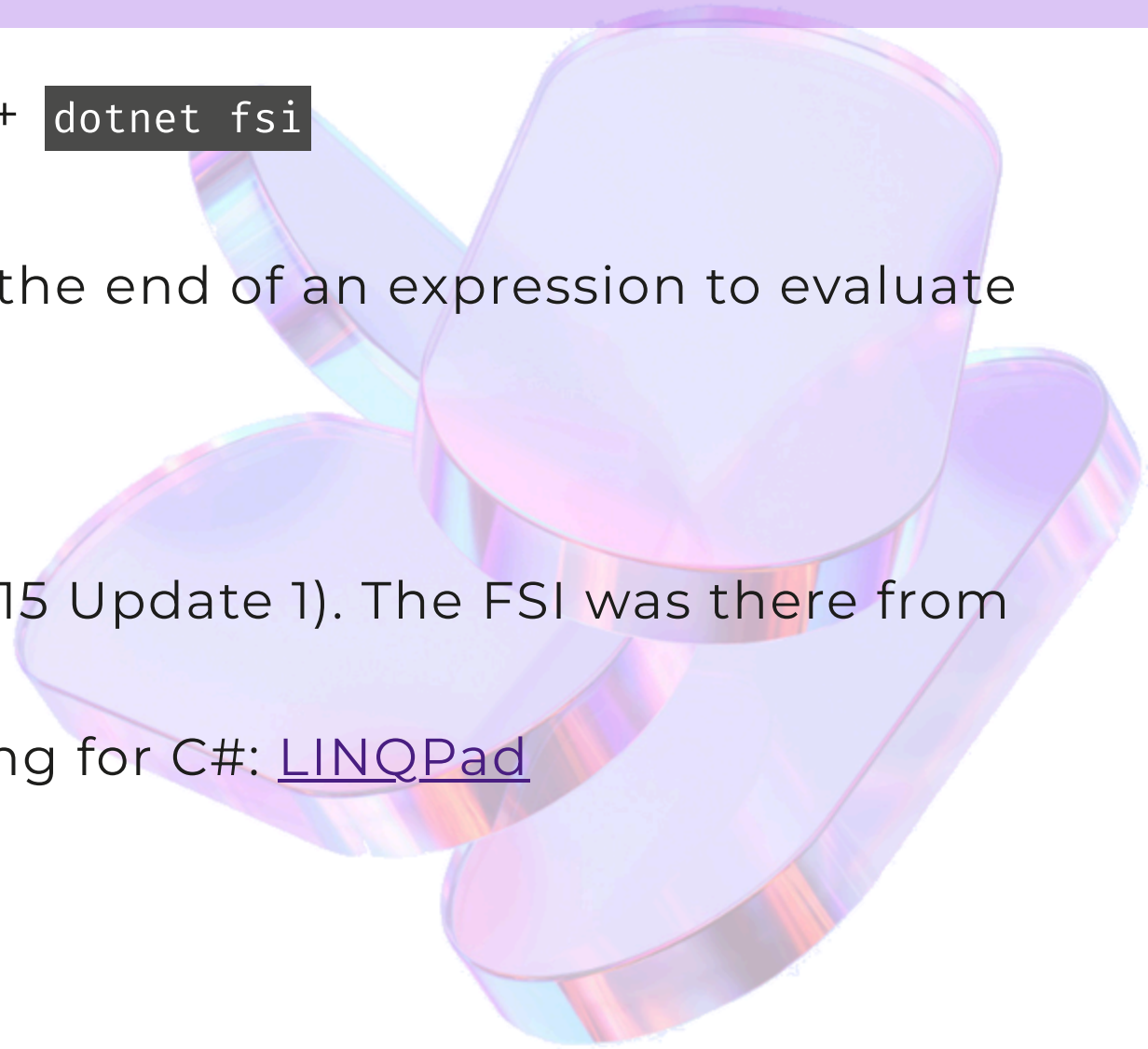
VsCode + Ionide
→ ☝ More a boosted text editor than a full IDE
→ ☝ Permissive: does not always report all compilation errors
→ 🔗 Fantastic F# and Azure Developer Extensions for VSCode by Compositional IT

Visual Studio / Rider
→ ☝ Less refactoring capabilities for F♯ than for C♯

Try F#: https://try.fsharp.org/
→ Online REPL with some examples



F♯ interactive *(FSI)*



- ➔ REPL available in VS, Rider, vscode + `dotnet fsi`
- ➔ Usage : instantly test a snippet
  - ➔ 💡 In the FSI console, enter `;;` at the end of an expression to evaluate it

☝️ **Notes:**

- ➔ *C♯ interactive* is more recent (VS 2015 Update 1). The FSI was there from the get go.
- ➔ Alternative worth trying, also working for C#: LINQPad

👩‍🏫 **Demo**

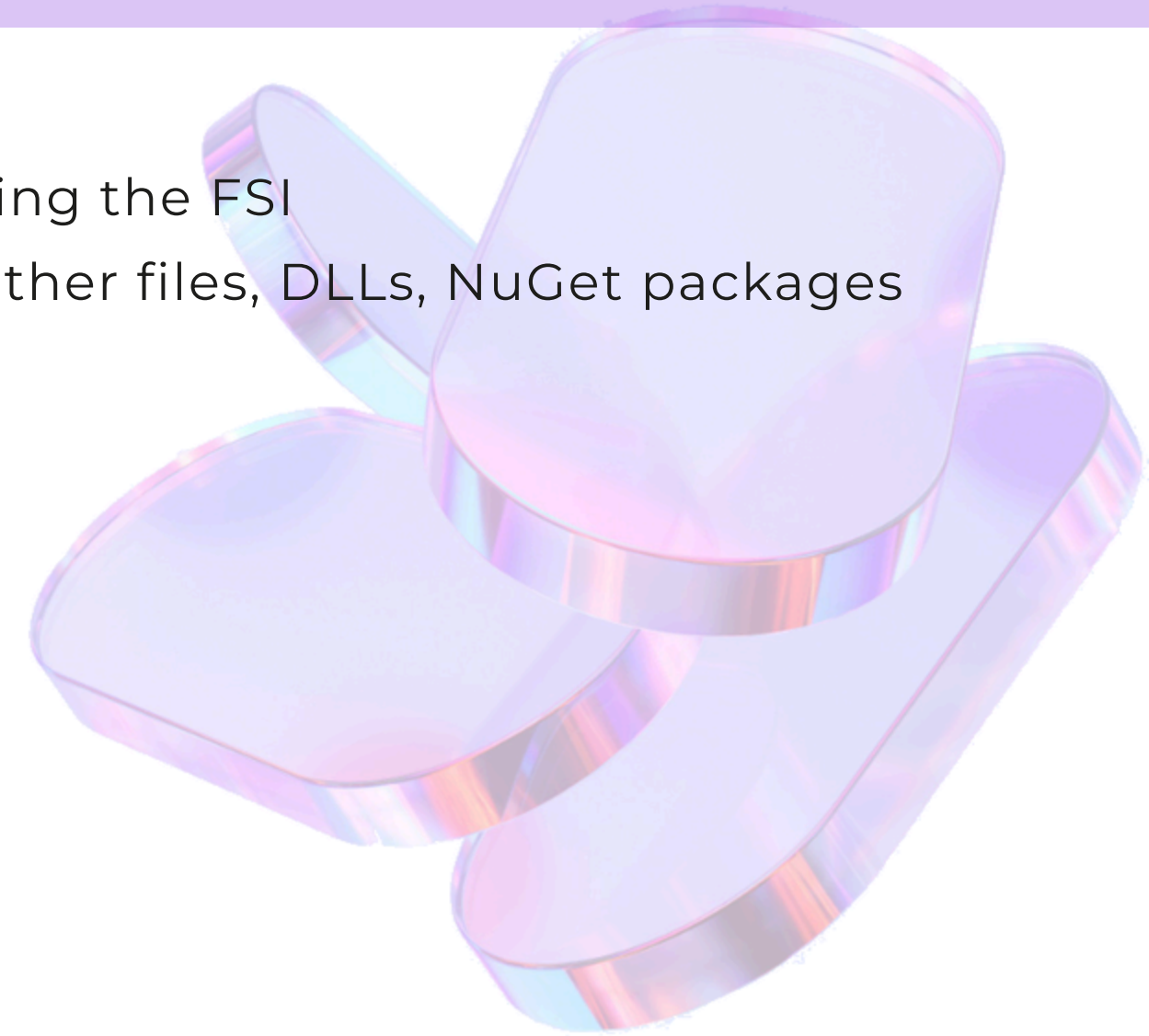
File types

4 file types: `.fs`, `.fsi`, `.fsx`, `.fsproj`

“ ⚠️ Single language : for F♯ only ”

Standalone file

- Script file `.fsx`
  - Executable *(hence the final **x**)* using the FSI
  - Independent but can reference other files, DLLs, NuGet packages



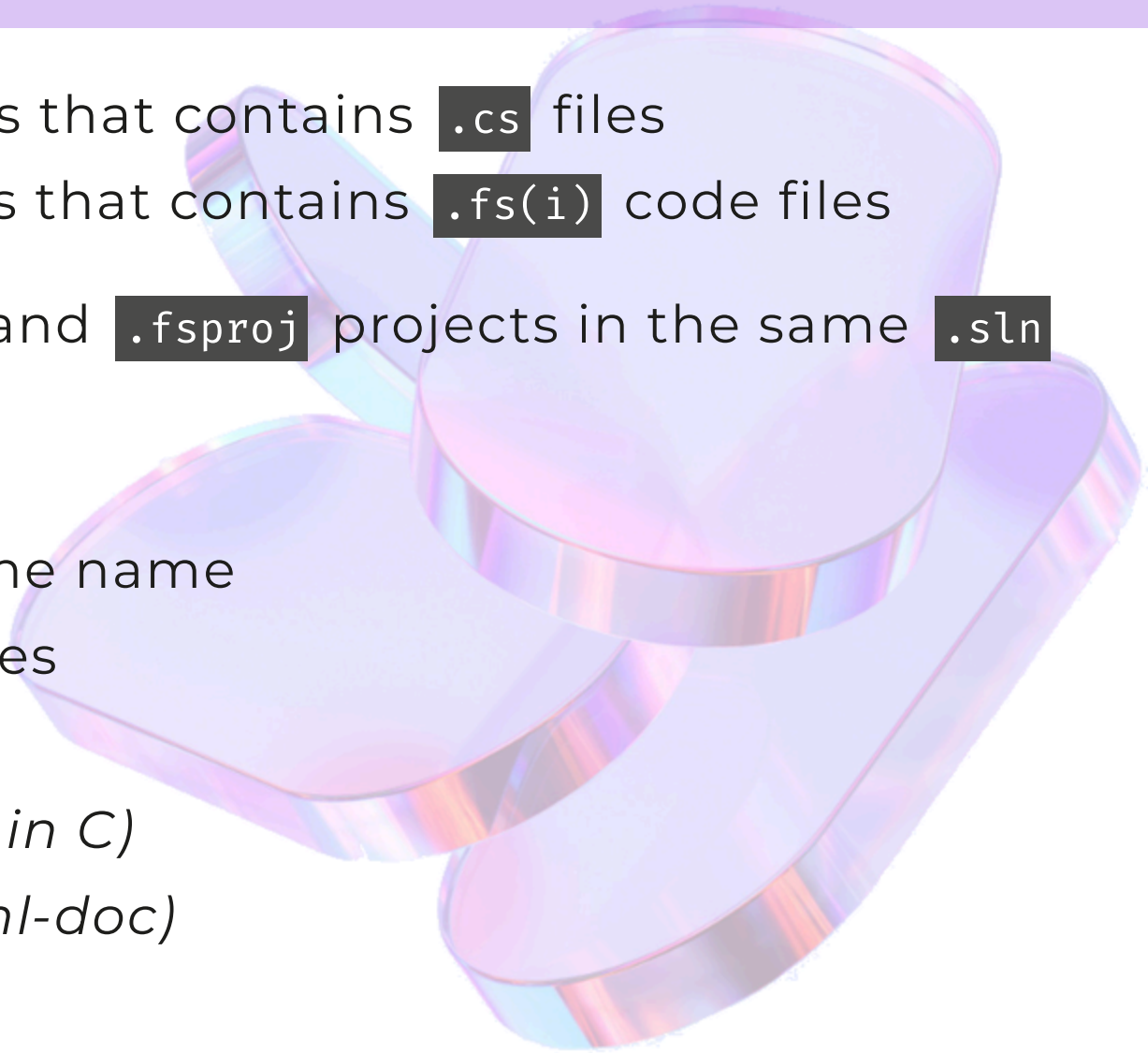
Project files

- ➔ In C♯ : `.sln` contains `.csproj` projects that contains `.cs` files
- ➔ In F♯ : `.sln` contains `.fsproj` projects that contains `.fs(i)` code files

💡 **Easy Interop** = Combine both `.csproj` and `.fsproj` projects in the same `.sln`

`.fsi` are signature files *(**i** for interface)*

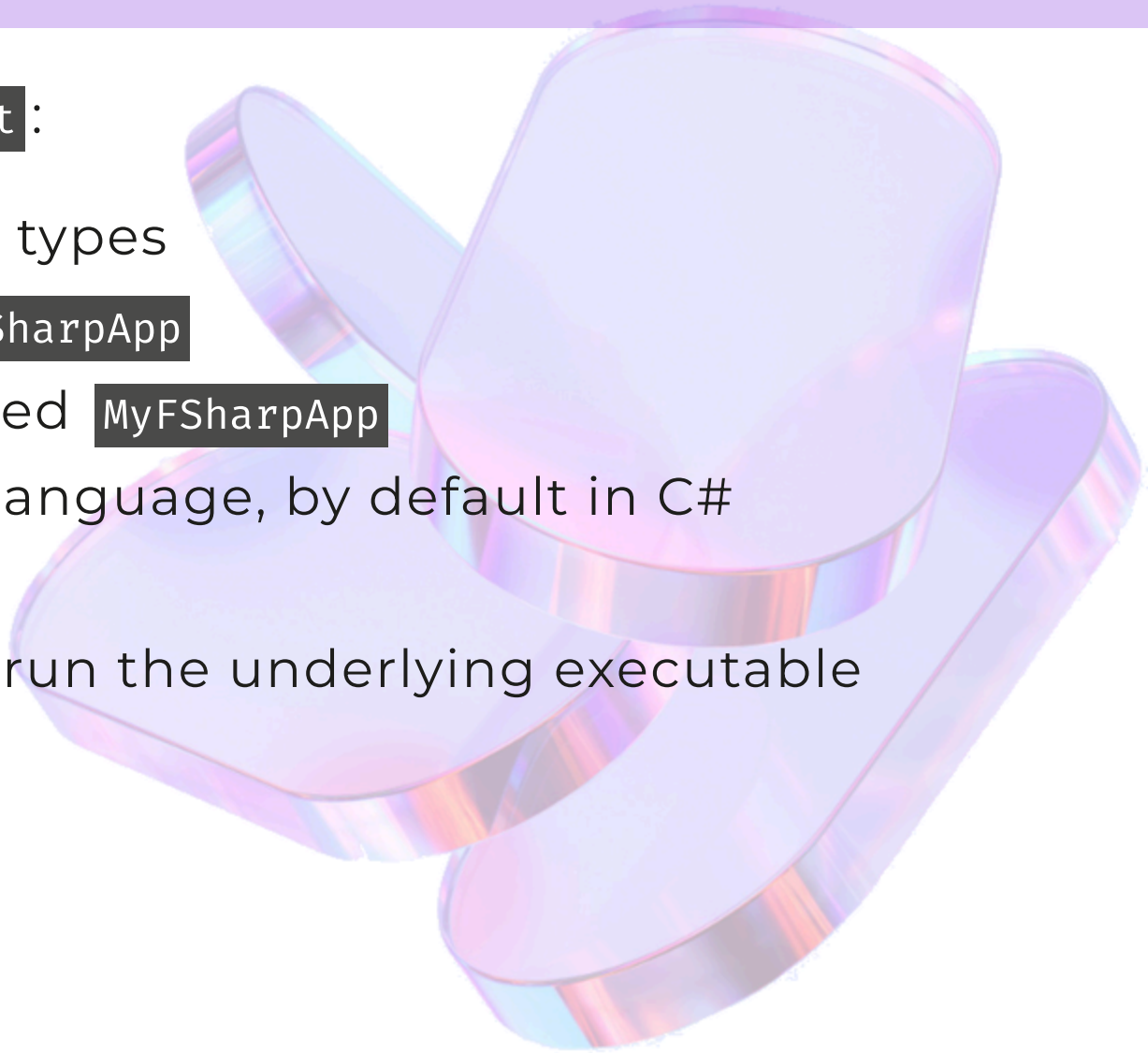
- ➔ Associated with a `.fs` file of the same name
- ➔ Optional and rather rare in codebases
- ➔ Usages
  - ➔ Reinforces encapsulation *(like `.h` in C)*
  - ➔ Separate long documentation *(xml-doc)*



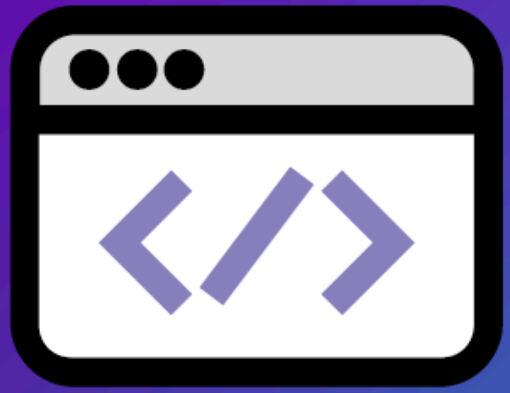
F♯ Project

Creation in the IDE or using the CLI `dotnet`:

- ➔ `dotnet new -l` : list supported project types
- ➔ `dotnet new console --language F# -o MyFSharpApp`
  - ➔ Création of a console project named `MyFSharpApp`
  - ➔ `--language F#` is key to specify the language, by default in C#
- ➔ `dotnet build` : to build the project
- ➔ `dotnet run` : to build the project and run the underlying executable



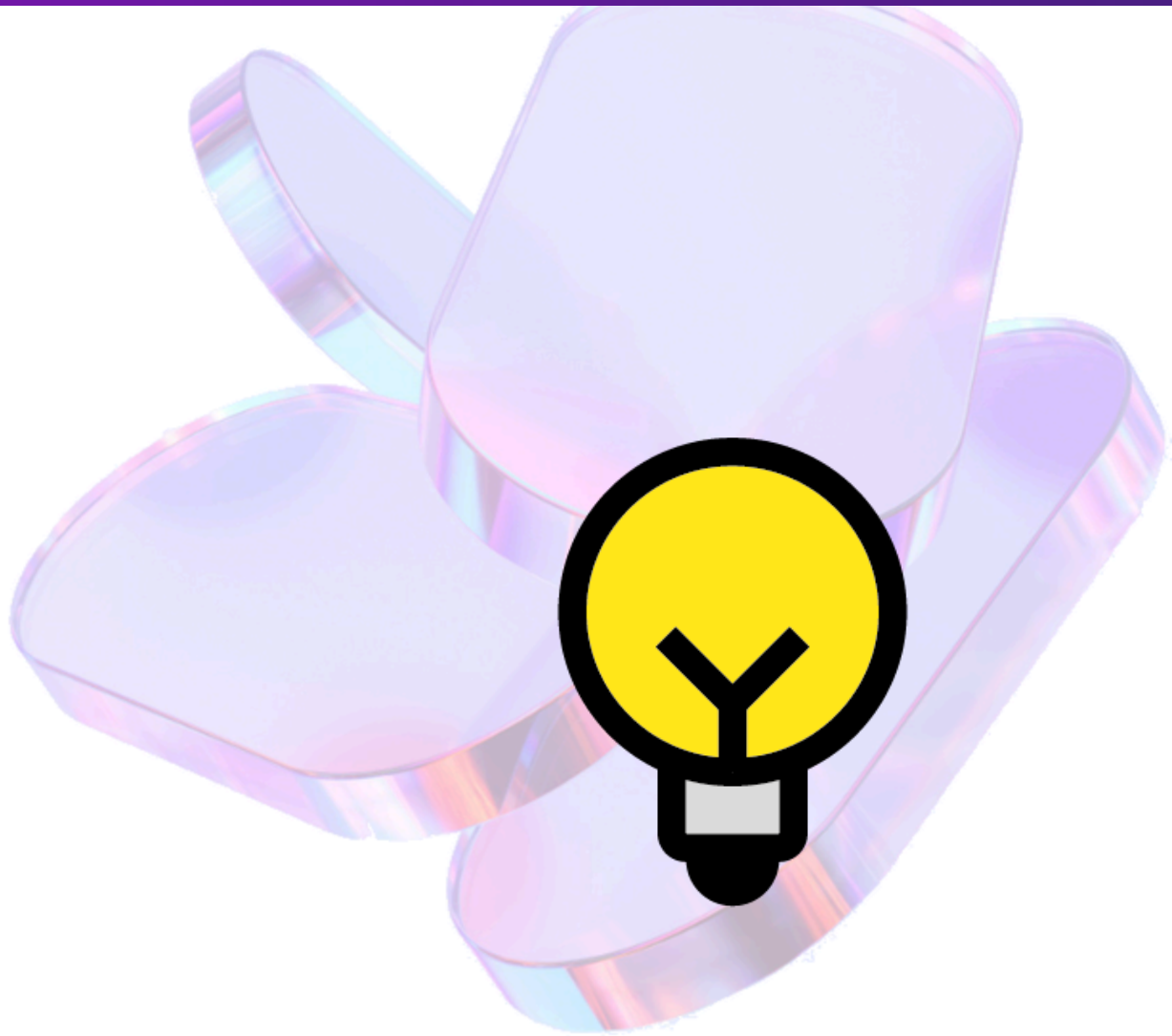
2. Syntax - Fundamentals



Syntax

F# **succinct** syntax is its first key point 💪

*(But, **explicit** is more important)*

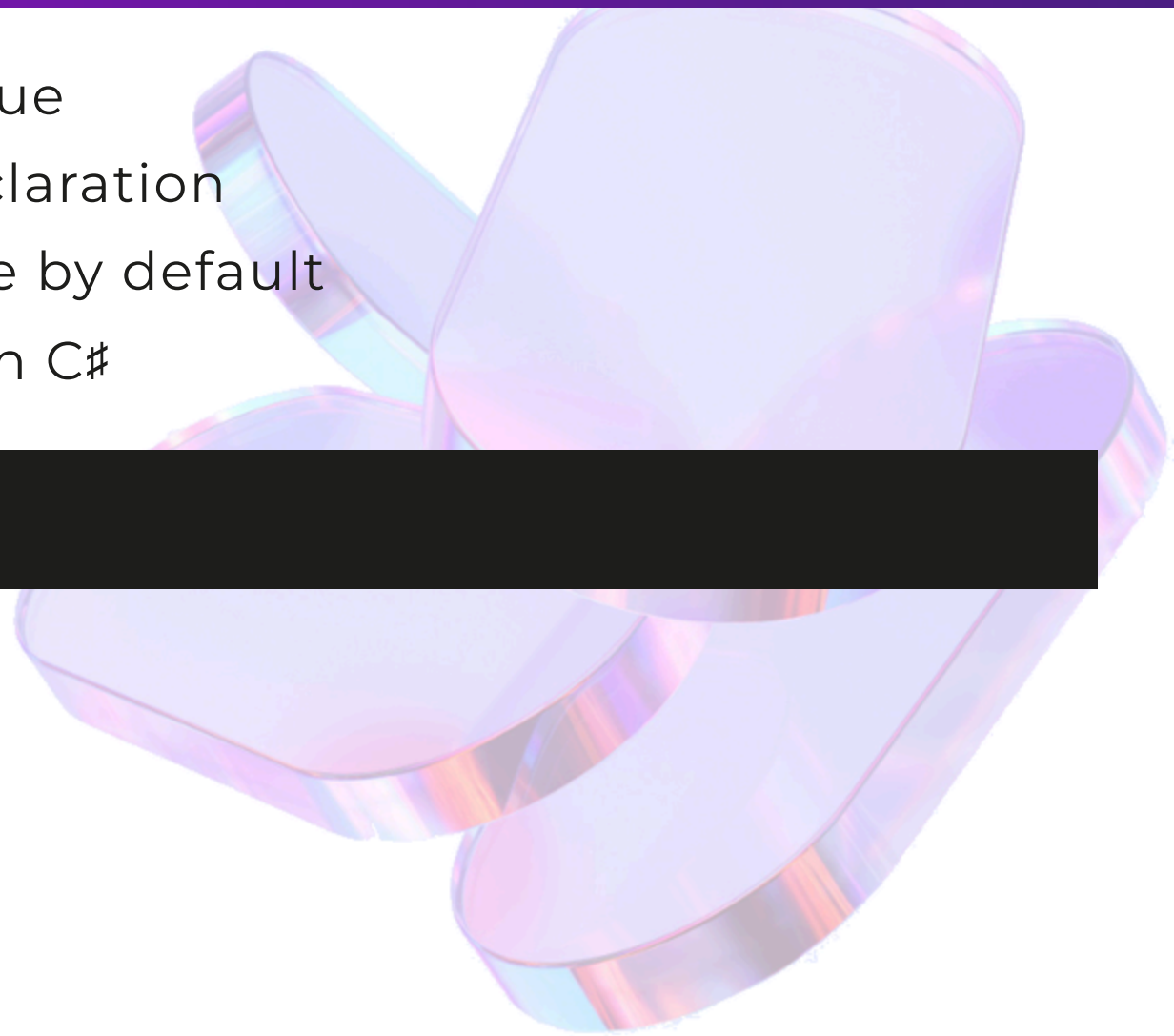


Comments

```
(* This is block
   comment *)


// And this is line comment


/// XML doc summary


/// <summary>
/// Full XML doc
/// </summary>
```

Variables / Values

- ➔ Keyword: `let` to declare/name a value
- ➔ No need for `;` at the end of the declaration
- ➔ Creates a *Binding* that is immutable by default
  - ➔ ≃ `const` in JS, `readonly` members in C♯



```
let x = 1
let y = "abc"
```

Variables: Mutable

Mutable binding with `let mutable`

- ➔ ≃ `let` en JS, `var` en C♯
- ➔ ⚠️ The assignment operator is `←`, not `=` used for equality
- ➔ Use it sparingly, on a limited scope

```
let x = 1
x ← 2 // 💥 error FS0027: This value is not mutable. Consider using the mutable keyword ...

let mutable y = 1
y ← 2 // ✅ OK
```

Names

- ➔ Same constraints on variable naming than in C♯
- ➔ … except the apostrophe `'` *(tick)*
  - ➔ allowed in the middle or at the end, but not at the ~~beginning~~ !
  - ➔ at the end → indicates a variant *(code convention)*
- ➔ Between double *backticks*
  - ➔ allow any character, in particular whitespaces, except line breaks

```
let x = 1
let x' = x + 1

// Works on keyword too! But avoid it because it's confusing!
let if' b t f = if b then t else f

let ``123 456`` = "123 456"
// 💡 no need to enter the ``, just the 123 to get the auto-completion
```

Shadowing

- ➔ Use to redefine a value with a name already used above
- ➔ The previous value is no longer accessible in the current scope
- ➔ Not allowed at `module` level but allowed in a sub-scope
- ➔ Convenient but can be misleading

```fsharp
let a = 2

let a = "ko"  // 💥 Error FS0037: Duplicate definition of value 'a'

let b =
    let a = "ok" // 👌 No compilation error
    // `a` is bound to the "ok" string (not the previous value: 2)
    // in all the rest of the b expression
    let a = "ko" // 👌 Consecutive shadowings are possible!
    ...
```

Type Annotation

- ➔ Optional thanks to inference
- ➔ The type is declared after the name `name: type` *(like in TypeScript)*
- ➔ The value is mandatory, even with `mutable`
  - ➔ good constraint for the code quality 👍

```
let x = 1        // Type inferred (int)
let y: int = 2   // Type explicit

let z1: int
// 💥 Error FS0010: Incomplete structured construct at or before this point
// in binding. Expected '=' or other token.

let mutable z2: int
// 💥 Same error
```

Constant

What: Variable erased during compilation, every usage is replaced by the value
≃ `const` C♯ - same idea than `const enum` in TypeScript

How: Value decorated with the `Literal` attribute

Recommended naming convention : **PascalCase**

```fsharp
[<Literal>] // Line break required before the `let`
let AgeOfMajority = 18

let [<Literal>] Pi = 3.14 // Also possible but not recommended by MS/Fantomas formatter
```

⚠️ **Attributes** are between `[< >]`
→ Frequent beginner error to use `[ ]` *(like in C♯)*

Number

```fsharp
let pi = 3.14              // val pi    : float   = 3.14  • System.Double
let age = 18               // val age   : int     = 18    • System.Int32
let price = 5.95m          // val price : decimal = 5.95M • System.Decimal
```

⚠️ No implicit conversion between number types
→ 💡 use `int`, `float`, `decimal` helper functions to do this conversion
→ ☝️ rule relaxed in some cases in F# 6

```fsharp
let i = 1
i * 1.2;;     // 💣 Error FS0001: The type 'float' does not match the type 'int'

float 3;;     // val it : float   = 3.0
decimal 3;;   // val it : decimal = 3M
int 3.6;;     // val it : int     = 3
int "2";;     // val it : int     = 2
```

String

```fsharp
let name = "Bob"                          // val name : string = "Bob"

// String formatting (available from the get go)
let name2 = sprintf "%s Marley" name      // val name2 : string = "Bob Marley"

// String interpolation (F# 5)
let name3 = $"{name} Marley"              // val name3 : string = "Bob Marley"

// Type safe string interpolation
let rank = 1
let name4 = $"%s{name} Marley, #%i{rank}" // val name4: string = "Bob Marley, #1"

// Access to a character by its index (≥ 0) (F# 6)
let initial = name2[0]                    // val initial :  char = 'B'
let initial = name2.[0]                   // Dot syntax, still supported

// String slicing (F# 6) (alternative to x.Substring(index [, length]) method)
let firstName = name2[0..2]               // val firstName : string = "Bob"
let lastName  = name2[4..]                // val lastName: string = "Marley"
```

String (2)

```fsharp
// Verbatim string: idem C#
let verbatimXml = @"<book title=""Paradise Lost"">"

// Triple-quoted string : no need to esapce the double-quotes `""`
let tripleXml = """<book title="Paradise Lost">"""

// Regular strings accept line breaks but do not trim whitespaces
let poemIndented = "
    The lesser world was daubed
    By a colorist of modest skill
    A master limned you in the finest inks
    And with a fresh-cut quill."
```

String (3)

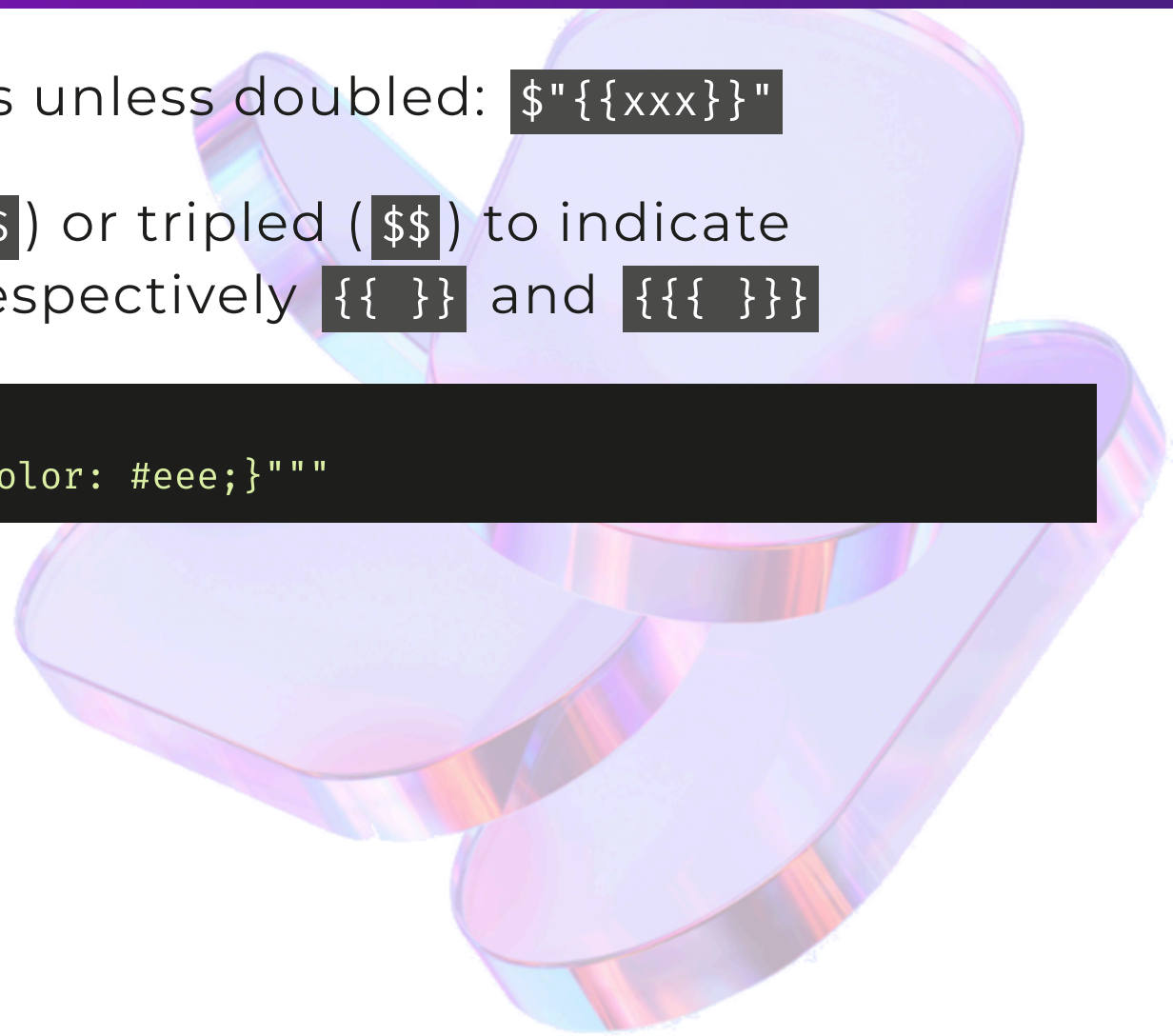
```fsharp
// Solution: backslash strings
// - Whitespaces (space and line break) are ignored between
//   the \ terminating a line and the following non-whitespace character
// - hence the \n to add line breaks
let poem = "\
    The lesser world was daubed\n\
    By a colorist of modest skill\n\
    A master limned you in the finest inks\n\
    And with a fresh-cut quill."

// We can also combine line breaks and backslash strings 🤷
let poemWithoutBackslashN = "\
    The lesser world was daubed
\
    By a colorist of modest skill
\
    A master limned you in the finest inks
\
    And with a fresh-cut quill."
```

String interpolation in F♯ 8

Interpolated string cannot contain braces unless doubled: `$"{{xxx}}"`

Since F♯ 8, the `$` character is doubled (`$$`) or tripled (`$$$`) to indicate the number of braces for interpolation, respectively `{{ }}` and `{{{ }}}`

```
let classAttr = "bold"
let cssNew = $$"""
.{{classAttr}}:hover {background-color: #eee;}
"""
```



Character encoding

String literals are encoded in **Unicode**:

```
let unicodeString1 = "abc"  // val unicodeString1: string = "abc"
let unicodeString2 = "ab✅" // val unicodeString2: string = "ab✅"
```

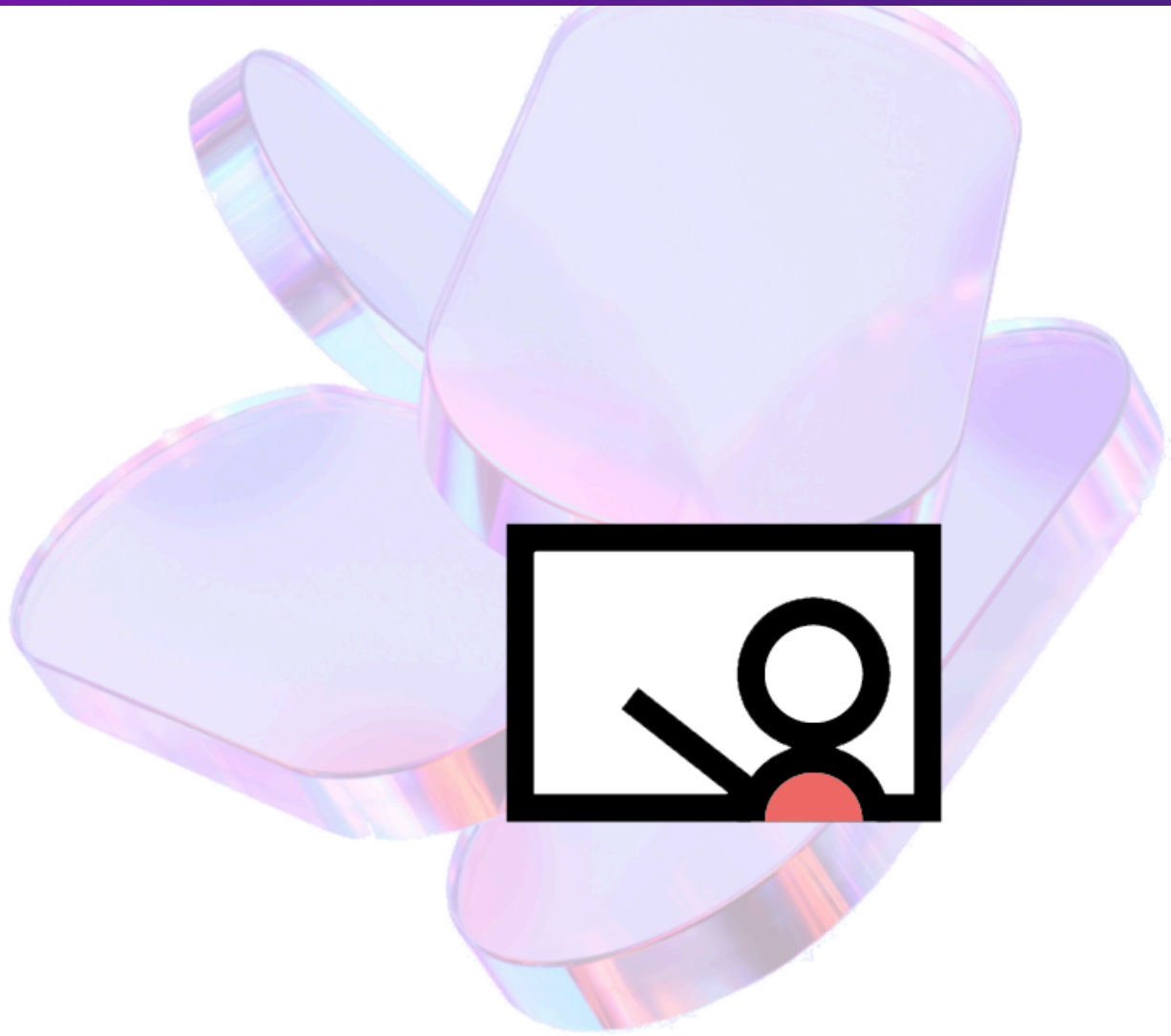
We can work in **ASCII** using the `B` suffix, but in this case we get a `byte array`:

```
let asciiBytes = "abc"B
// val asciiBytes1: byte array = [|97uy; 98uy; 99uy|]

let asciiBytesKO = "ab🚫"B
// 💥 Error FS1140: This byte array literal contains characters
//    that do not encode as a single byte
```

💡 Works also for character: `'a'B`

Collections

- Lists
- Arrays
- Sequences



Lists

A list is an immutable collection of elements of the same type.
≠ `System.Collection.Generic.List<T>` BCL type
Implemented internally as a linked list.

Creation with `[]` · Items separated by `;` or line breaks + indentation

```
let abc = [ 'a'; 'b'; 'c' ] // val abc : char list = ['a'; 'b'; 'c']
let a =
  [ 2
    3 ]  // val a : int list = [2; 3]
```

⚠️ **Trap:** using `,` to separate items = single item: a tuple 📍

☝️ **ML style type annotation:** `int list` = `List<int>`
→ Idiomatic only for some `FSharp.Core` types: `array`, `list`, `option` 📍

List operators

`::` *Cons (for "construction")*: add an item to the top of the list
`..` *Range* of numbers between `min..max` *(included)* or `min..step..max`
`@` *Append* 2 lists

```
let ints = [2..5]                  // val ints : int list = [2; 3; 4; 5]
let ints' = 1 :: ints              // val ints' : int list = [1; 2; 3; 4; 5]
let floats = [ 2. .. 5. ]          // val floats: float list = [2.0; 3.0; 4.0; 5.0]

let chars = [ 'a' .. 'd' ]         // val chars : char list = ['a'; 'b'; 'c'; 'd']
let chars' = chars @ [ 'e'; 'f' ] // val chars' : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
let e = chars'[4]                  // val e: char = 'e'
```

⚠️ **Space** required before `[]` to create a list; otherwise: access by index

Arrays

Mutable fixed-size collections of elements of the same type.
`array 't` = `'t[]`: BCL type

Creation with `[| |]` · Items separated by `;` or line breaks + indentation

```
let a1 = [| 'a'; 'b'; 'c' |]
// val a1: char array = [|'a'; 'b'; 'c'|]

let a2 =
    [| 2
       3 |]
// val a2: int array = [|2; 3|]
```

⚠️ **Trap:** `[]` used for list creation and array type!

Sequences

Series of elements of the same type

`'t seq` = alias for `System.Collections.Generic.IEnumerable<'t>` BCL type

Creation with `seq { }`

```
let seq1 = seq { 'a'; 'b'; 'c' }
// val seq1: char seq

let seq2 =
    seq {
        2
        3
    }
// val seq2: int seq
```

⚠️ **Lazy**: possible multiple enumeration • hence not evaluated in FSI console

Collections functions

Each type has its own module containing dedicated functions.

Common functions:

F# collections	C# LINQ (`IEnumerable<_>`)	JS `Array`
`map`, `collect`	`Select()`, `SelectMany()`	`map()`, `flatMap()`
`exists`, `forall`	`Any(predicate)`, `All()`	`some()`, `every()`
`filter`	`Where()`	`filter()`
`find`, `tryFind`	×	`find()`
`fold`, `reduce`	`Aggregate([seed]])`	`reduce()`
`average`, `sum`	`Average()`, `Sum()`	×

🔗 Full documentation on fsharp.github.io: Array · List · Seq

Named functions

- Declared in a `let` binding *(like a variable)*
- Naming convention: **camelCase**
- No `return` keyword: always returns the last expression in the body
- No `()` around all parameters, no `,` between parameters
- `()` required around parameter with type annotation (1) or deconstruction (2)

```fsharp
let square x = x * x  // Function with 1 parameter
let res = square 2    // Returns 4

// (1) Parentheses required for annotations of type
let square' (x: int) : int = x * x

// (2) Brackets required when deconstructing an object
//     (here it's a single-case discriminated union 📌)
let hotelId (HotelId value) = value
```

Functions of 2 or more parameters

Separate parameters and arguments with **spaces**:

```
// Function with 2 parameters
let add x y = x + y  // val add: x: int → y: int → int

// Call with the 2 arguments
let res = add 1 2    // val res: int = 3
```

⚠️ `,` creates another kind of functions using tuples 📌

```
let addByPair (x, y) = x + y
// val addByPair: x: int * y: int → int
```

Functions without parameter

Use `()` *(like in C#)*

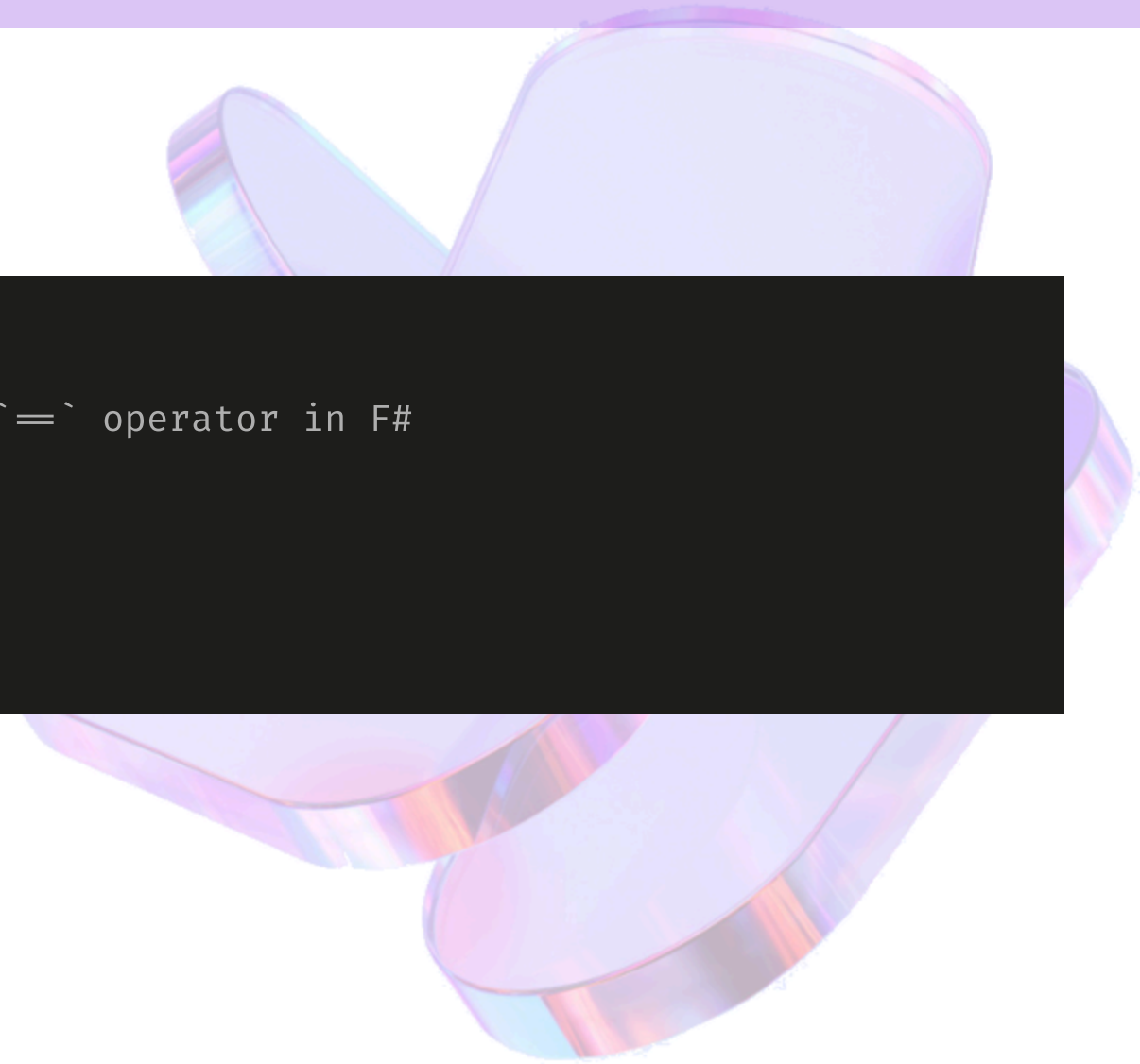
```
let printHello () = printfn "Hello"
// val printHello: unit → unit
printHello ();;
// Hello

let notAFunction = printfn "Hello"
// Hello
// val notAFunction: unit = ()
```

☝️ `unit` means "nothing" 📍

Multi-line function

Indentation required, but no need for `{}`
Can contain sub-function

```
let evens list =
    let isEven x =  // 👈 Sub-function
        x % 2 = 0   // 💡 `=` equality operator - No `==` operator in F#
    List.filter isEven list
// val evens: list: int list → int list

let res = evens [1;2;3;4;5]
// val res: int list = [2; 4]
```

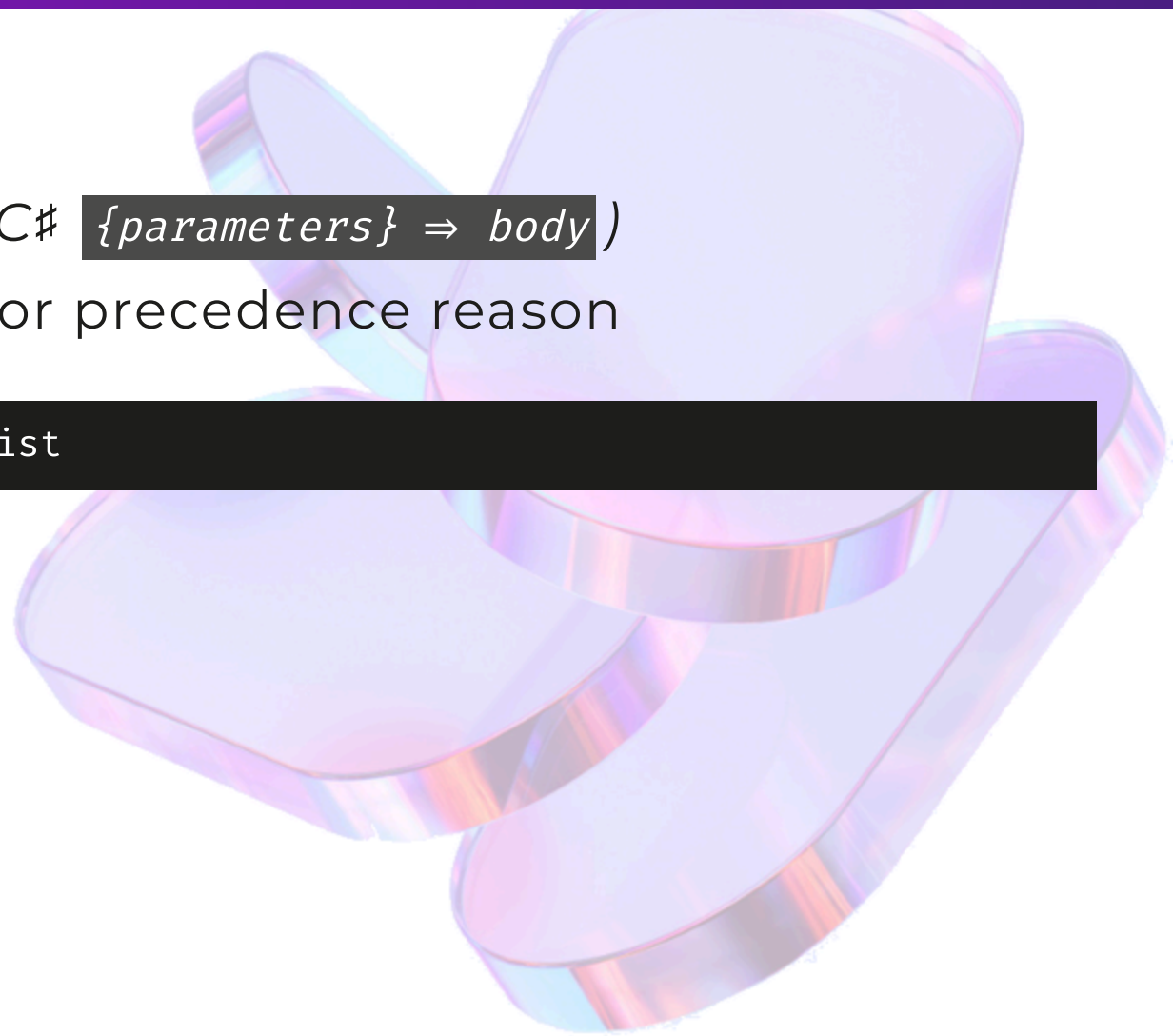


Anonymous function

A.k.a. ***Lambda****, arrow function*

- ➔ Syntax: `fun {parameters} → body` *(≠ in C♯* `{parameters} ⇒ body`*)*
- ➔ In general, `()` required all around, for precedence reason

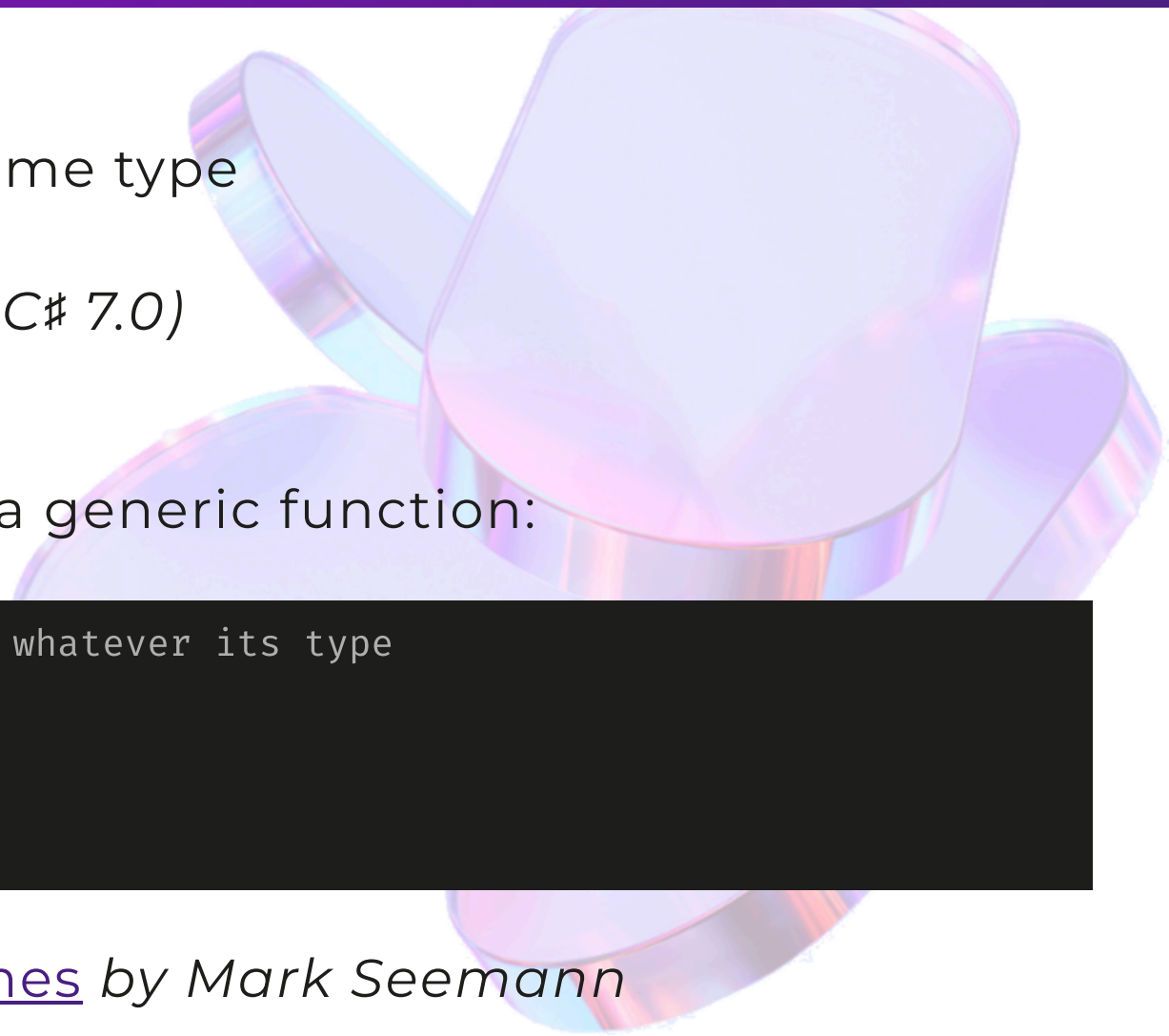
```
let evens' list = List.filter (fun x → x % 2 = 0) list
```



_.Member shorthand (F# 8)

```fsharp
type Person = { Name: string; Age: int }

let people =
    [ { Name = "Alice"; Age = 30 }
      { Name = "Billy"; Age =  5 } ]

// Regular lambda (Shorthand not possible)
let adults = people |> List.filter (fun person -> person.Age >= 18)
// val adults: Person list = [{ Name = "Alice"; Age = 30 }]

// Member chain shorthand
let uppercaseNames = people |> List.map _.Name.ToUpperInvariant() // 👉👉
// val uppercaseNames: string list = ["ALICE"; "BILLY"]
```

Naming convention related to functions



It's usual in F♯ to use short names:

- `x`, `y`, `z` : parameters for values of the same type
- `f`, `g`, `h` : parameters for input functions
- `_` : *discard* an element not used *(like in C♯ 7.0)*
- `xs` : list of `x`

☝️ Suited for a short function body or for a generic function:

```
// Function that simply returns its input parameter, whatever its type
let id x = x

// Composition of 2 functions
let compose f g = fun x → g (f x)
```

🔗 When x, y, and z are great variable names *by Mark Seemann*

Piping

Pipe operator `|>` : same idea that in UNIX with `|`

→ `value |> function` send a value to a function

→ match left-to-right reading order: "subject verb"

→ same order than when we dot an object: `object.Method`

```fsharp
let a = 2 |> add 3  // to read "2 + 3"

// We pipe a list to the "List.filter predicate" function
let evens = [1;2;3;4;5] |> List.filter (fun x -> x % 2 = 0)
```

```csharp
// ≃ C#
var a = 2.Add(3);
var nums = new[] { 1, 2, 3, 4, 5 };
var evens = nums.Where(x => x % 2 == 0);
```

Pipeline: chain of pipings

Style of coding to emphasize the data flowing from functions to functions
→ without intermediary variable 👍

Similar to a built-in *fluent API*
→ no need to return the object at the end of each method 👍

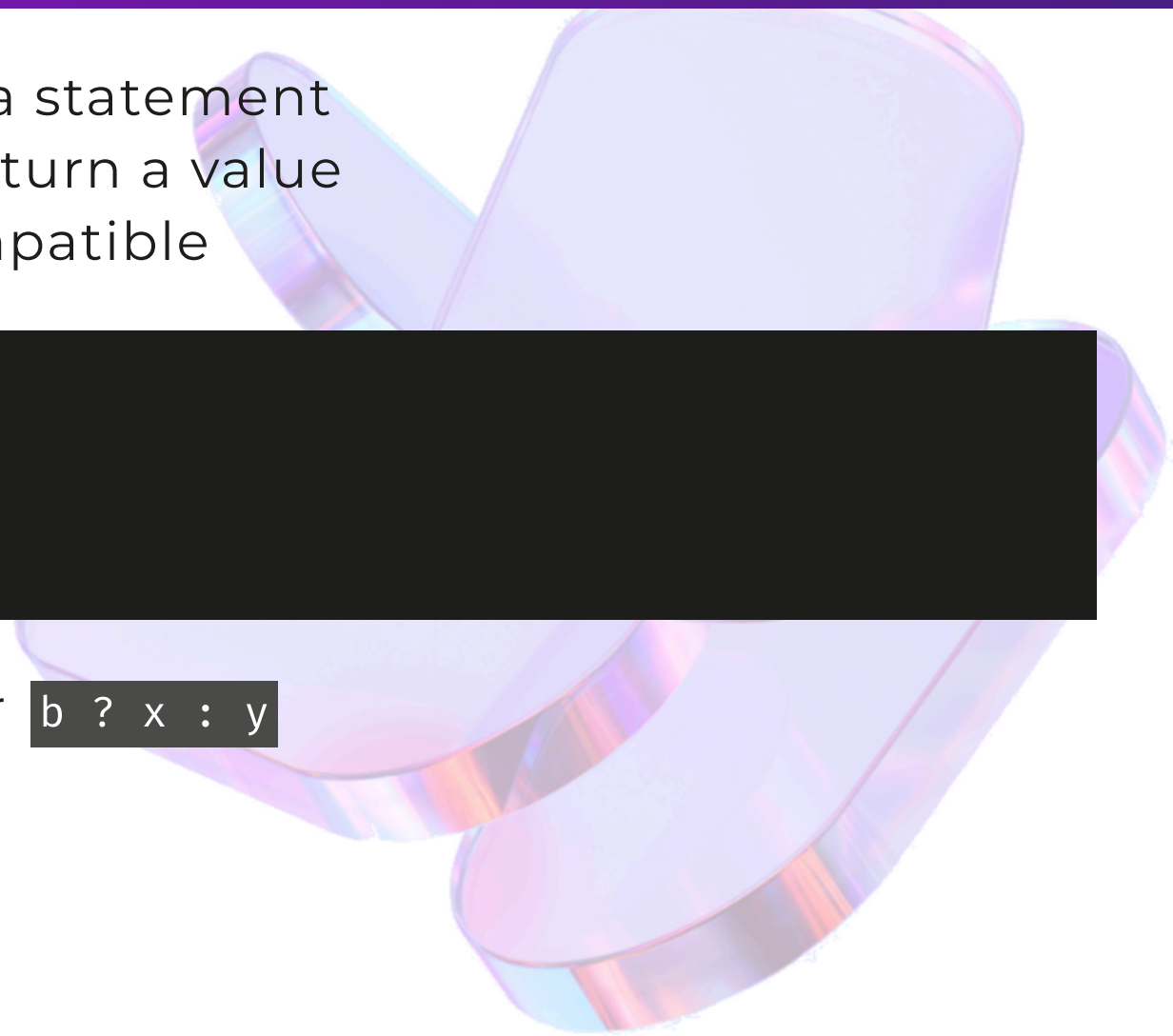
```fsharp
// Short syntax: in a single line fitting the screen width
let res = [1;2;3;4;5] ▷ List.filter (fun x → x % 2 = 0) ▷ List.sum

// More readable with line breaks
let res' =
    [1; 2; 3; 4; 5]
    ▷ List.filter isOdd  // With `let isOdd x = x % 2 <> 0`
    ▷ List.map square    //      `let square x = x * x`
    ▷ List.map addOne    //      `let addOne x = x + 1`
```

If/then/else expression

In F♯, `if/then(/else)` is an expression, not a statement
→ every branch (`then` and `else`) should return a value
→ all returned values should be type-compatible

```fsharp
let isEven n =
    if n % 2 = 0 then
        "Even"
    else
        "Odd"
```

💡 `if b then x else y` ≃ C♯ ternary operator `b ? x : y`



If/then/else expression (2)

☝️ When `then` returns "nothing" (`unit` 📍), `else` is optional:

```
let printIfEven n msg =
    if n ▷ isEven then
        printfn msg
```

💡 We can use `elif` keyword instead of `else if`.

Match expression

```fsharp
let translateInFrench civility =
    match civility with
    | "Mister" → "Monsieur"
    | "Madam"  → "Madame"
    | "Miss"   → "Mademoiselle"
    | _        → ""   // 👈 wilcard `_`
```

Equivalent in C♯ 8 :

```csharp
public static string TranslateInFrench(string civility) ⇒
    civility switch {
        "Mister" ⇒ "Monsieur"
        "Madam"  ⇒ "Madame"
        "Miss"   ⇒ "Mademoiselle"
        _        ⇒ ""
    }
```

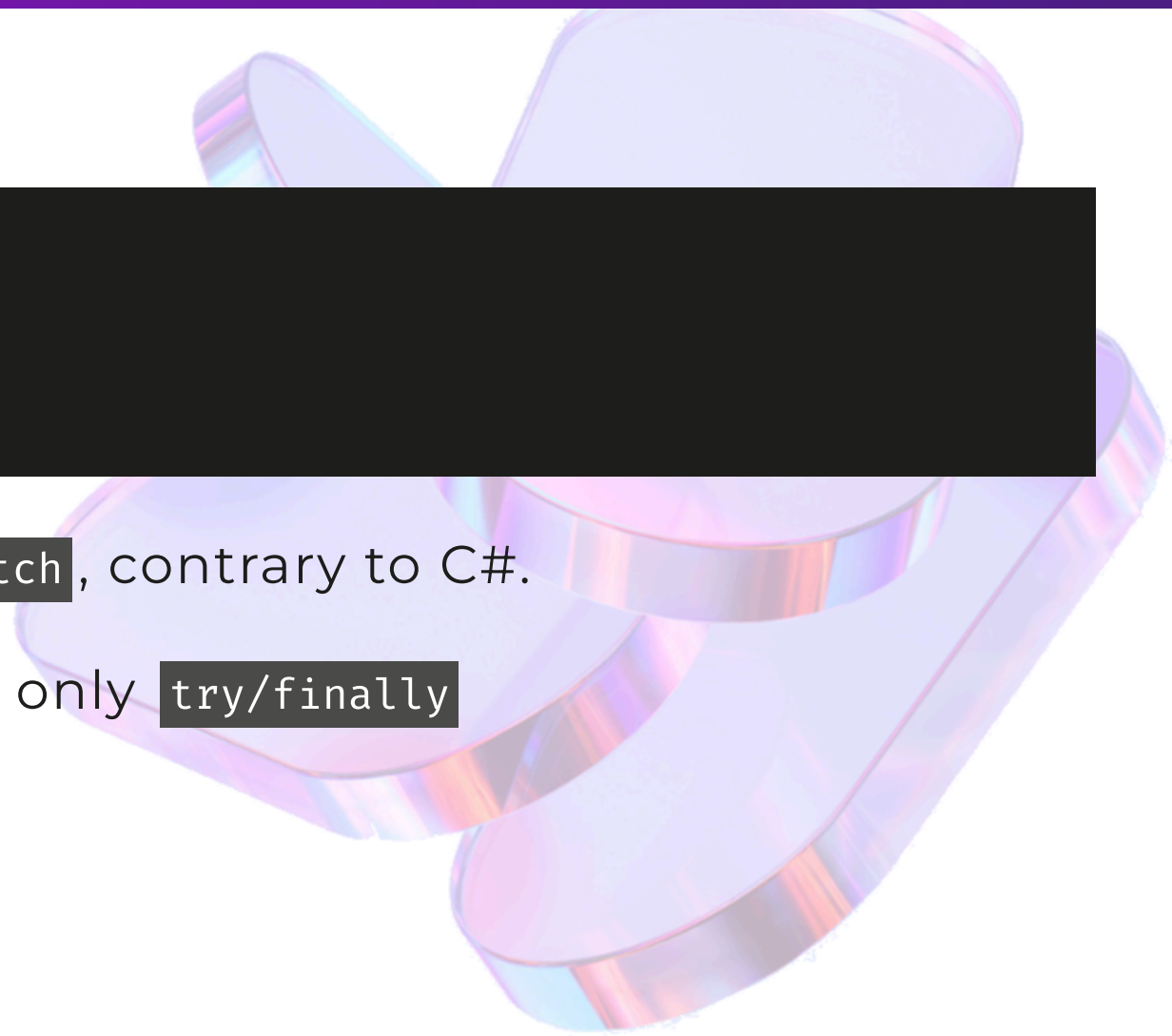
Handling Exception

→ `try/with` expression

```
let tryDivide x y =
    try
        Some (x / y)
    with :? System.DivideByZeroException →
        None
```

⚠️ **Trap**: the keyword used is `with`, not `catch`, contrary to C#.

💡 There is no `try/with/finally` expression, only `try/finally`
→ Nest a `try/finally` in a `try/with`



Throwing Exception

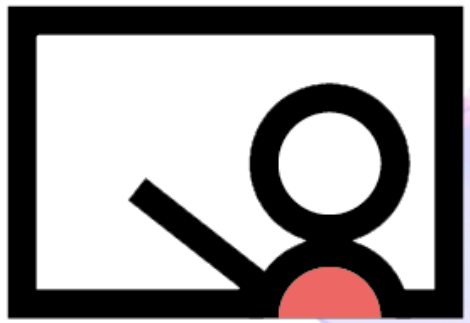
→ Helpers `failwith`, `invalidArg`, `nullArg`

```
let fn arg =
    if arg = null then nullArg (nameof arg)
    failwith "Not implemented"

let divide x y =
    if y = 0
    then invalidArg (nameof y) "Divisor cannot be zero"
    else x / y
```

🔗 Handling Errors Elegantly https://devonburriss.me/how-to-fsharp-pt-8/

Syntax rules

- ➔ Declarations order
- ➔ Indentation



Declarations order

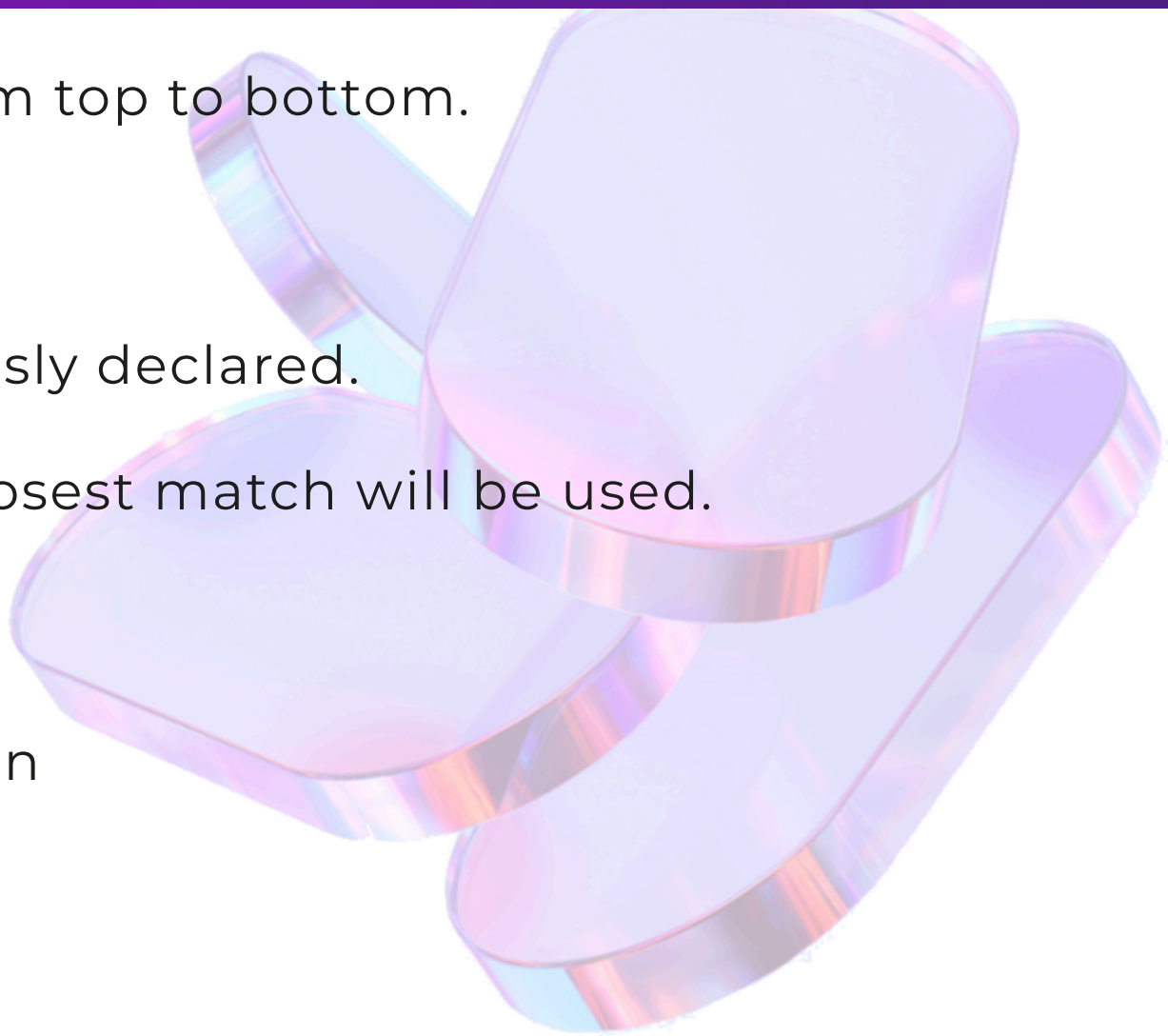
In a file, the declarations are ordered, from top to bottom.
→ Declaration comes before the usages.

In a `.fsproj`, the files are ordered too.
→ We can import only something previously declared.

Type inference works by proximity: the closest match will be used.

👍 Pros:
→ no cyclic dependencies
→ faster and more predictable compilation
→ code easier de reason about

☝️ Cons:
→ need more coding discipline



Declarations order: example

We try to use the `fn` before its declaration:

```
let result = fn 2
//           ~~ 💥 Error FS0039: The value or constructor 'fn' is not defined.

let fn i = i + 1
```

Indentation

In general, indentation is very important for code readability:
→ It creates visual structures that match the logical structure, the hierarchy.

In C♯: indentation is optional; logical blocks defined with `{ }` and `;`
→ It's the indentation that matters for readability, then `{ }` can help
→ A code not properly indented can be mis-interpreted, that can lead to bugs!

In F♯, indentation is required to define code blocks and nesting level.
→ Compiler ensures indentation is correct
→ Reader can really trust the indentation to understand the code structure

👉 **Conclusion:**
F♯ forces us to do what matters the most for the code readability 👍

Vertical line of indentation

Concept related to the way F♯ understands the indentation.

- In general, a block starts in a new line, at a greater indentation level.
- But sometimes a block can start in a middle of a line.
  → This position defines the expected vertical indentation line.

```
let f =
   let x=1
   x+1
// ^ Vertical line here

let f = let x=1
        x+1
//      ^ Vertical line here!
```

Vertical line of indentation (2)

There are some exceptions to this rule, for instance with operators.
🔗 F♯ syntax: indentation and verbosity | F♯ for fun and profit.

☝️ Indentation rules have been relaxed in F♯ 6.

Indentation Guideline

F♯ allows to have different:
→ whitespace **characters**: tabs and spaces.
→ **number** of whitespaces per indentation level.

Recommendations:
- In temporary `fsx`, writing speed > code readability
- In `fsproj`, proper and consistent formatting is important for maintainability
  → Use consistently only spaces
  → Use the same number of spaces for all indentation level
  → 4 spaces is idiomatic; exception: vertical indentation line *(next slide)*
  → Use a code formatter like **Fantomas** to ensure this consistency.

Indentation Guideline (2)

Avoid naming-sensible indentation a.k.a *Vanity Alignment*:

- ➔ They can break compilation after a renaming.
- ➔ Blocks too far at the right: less readable *(left-to-right language)*

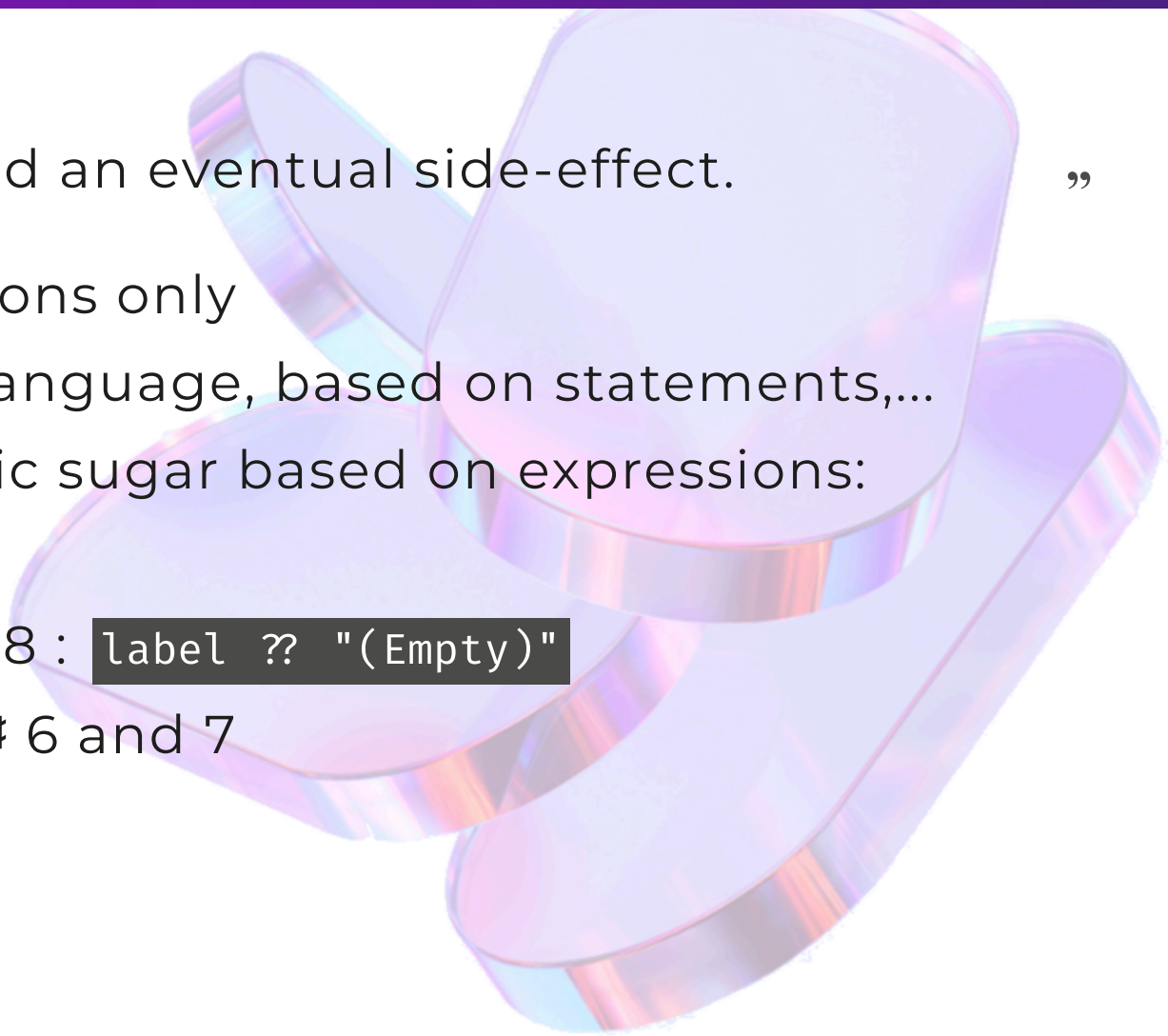
```
// 👌 OK
let myLongValueName =
    someExpression
    ▷ anotherExpression

// ⚠️ To avoid
let myLongValueName = someExpression
                      ▷ anotherExpression  // 👈 Depend on the length of `myLongValueName`
```

3. *F♯ language* design traits

Expression *vs* Statement

> A **statement** will produce a side effect.
> An **expression** will produce a value... and an eventual side-effect.

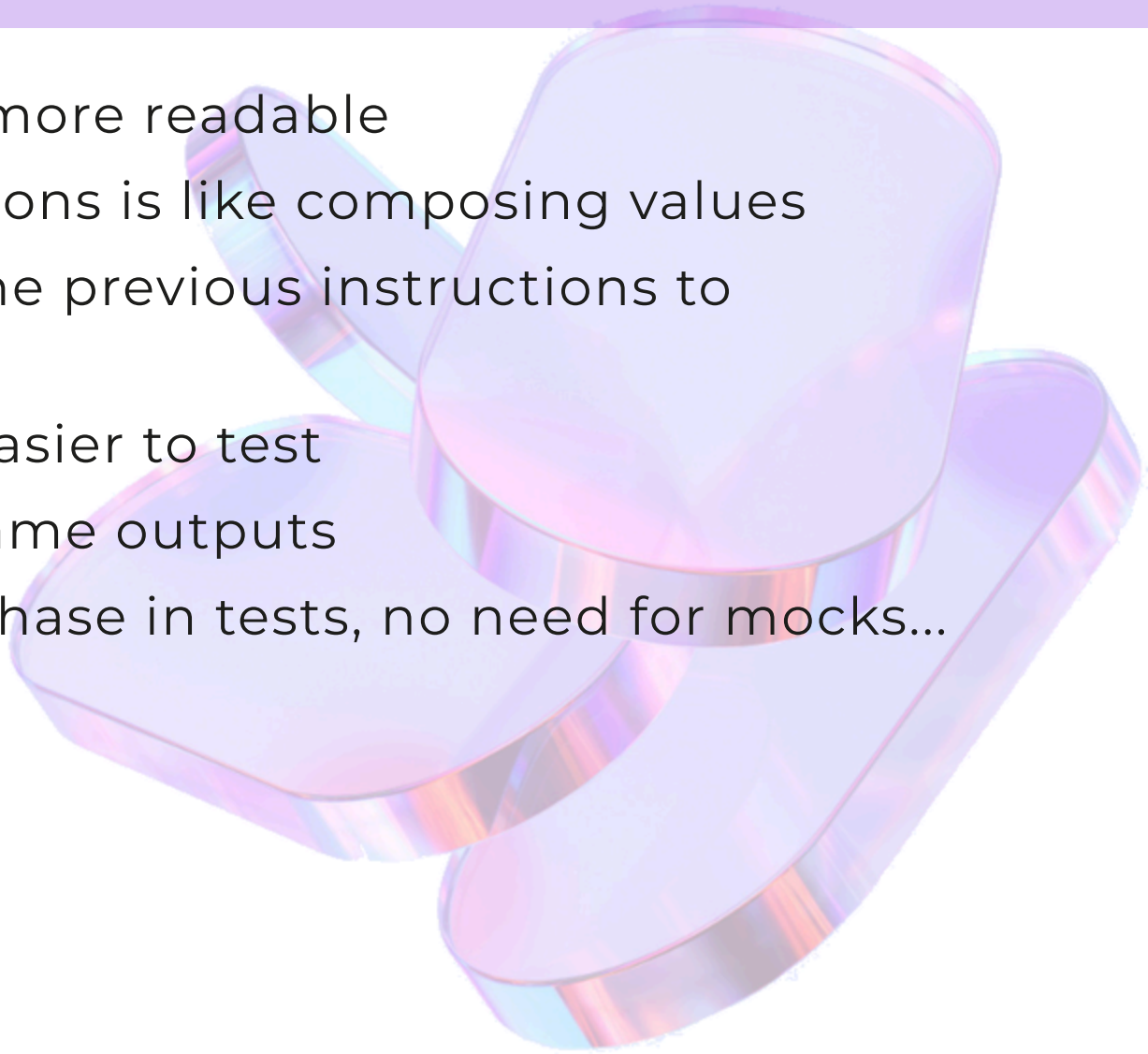
- ➔ F♯ is a functional, based on expressions only
- ➔ In comparison, C♯ is an imperative language, based on statements,...
- ➔ ... including more and more syntactic sugar based on expressions:
  - ➔ Ternary operator `b ? x : y`
  - ➔ Null-coalescing operator `??` in C♯ 8 : `label ?? "(Empty)"`
  - ➔ Expression-bodied members in C♯ 6 and 7
  - ➔ `switch` expression in C♯ 8



⚖️ Benefits of expressions over instructions

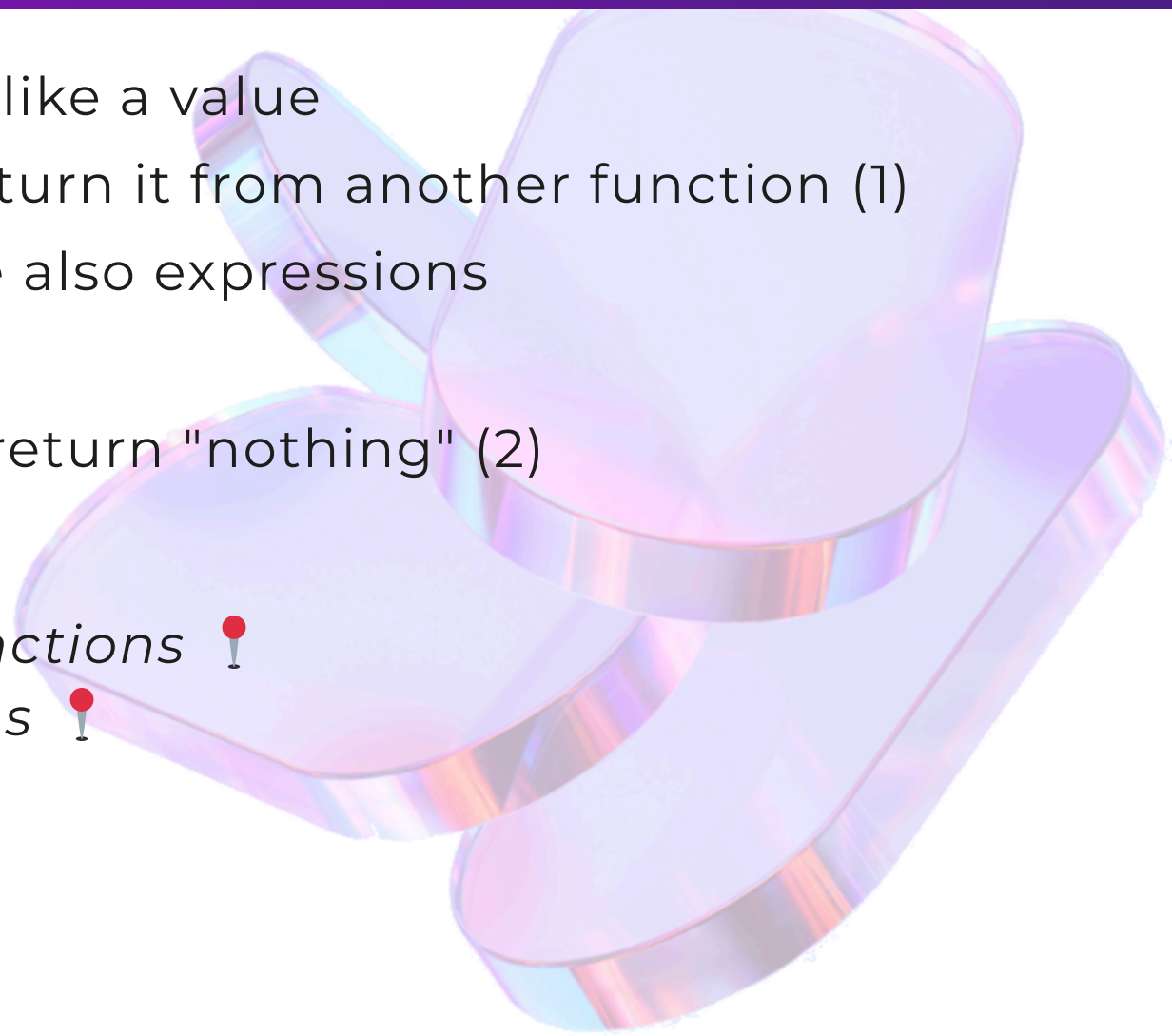
- ➔ **Conciseness**: less visual clutters → more readable
- ➔ **Composability**: composing expressions is like composing values
- ➔ **Understanding**: no need to know the previous instructions to understand the current one
- ➔ **Testability**: pure expressions (*) → easier to test
  - ➔ *Predictable*: same inputs mean same outputs
  - ➔ *Isolated*: shorter *Arrange/Setup* phase in tests, no need for mocks…

(*) **Pure:** with no side-effects



In F♯ « Everything is an expression »

- ➔ A function is declared and behaves like a value
  - ➔ We can pass it as parameter or return it from another function (1)
- ➔ The *control flow* building blocks are also expressions
  - ➔ `if … then/else` , `match … with`
  - ➔ `for … in`, `for … to`, `while … do` just return "nothing" (2)



☝ **Notes:**

- (1) See *1st-class citizens, high-order functions* 📌
- (2) Except in *collection comprehensions* 📌

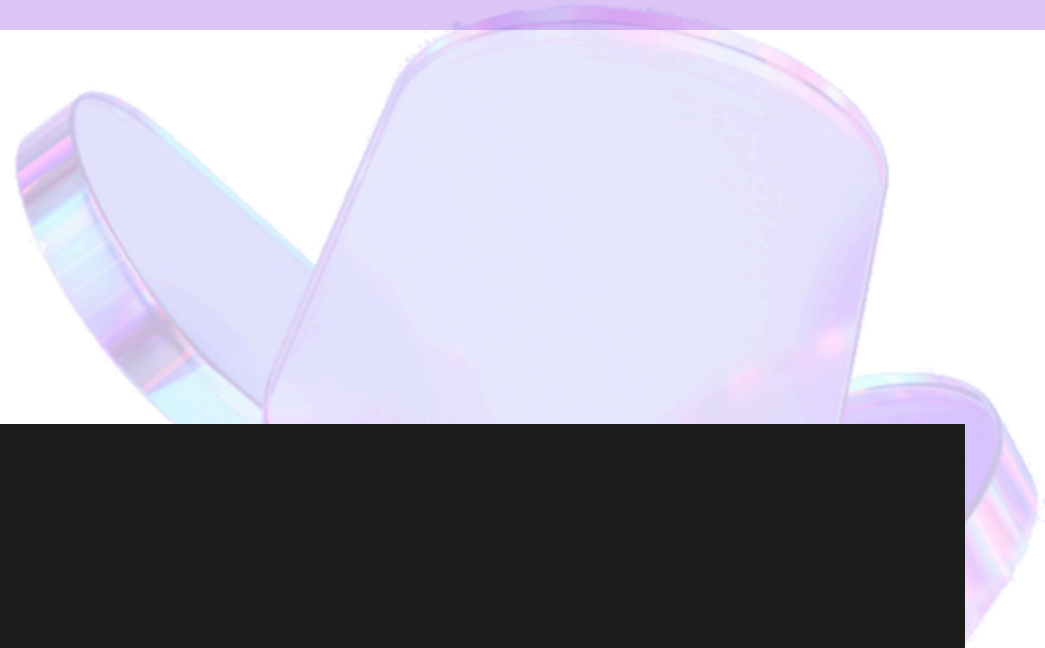
Everything is an expression » Consequences

No `void`
→ Best replaced by the `unit` type 📍

No *early exit*

- In C#, you can exit a function with `return` and exit a `for/while` loop with `break`.
- In F♯, these keywords do not exist. *(Alternatives: next slides)*

Early exit alternatives » Imperative style



💩 `BreakException`

→ *See code in StackOverflow*

😕 **Mutable variables**

```
let firstItemOr defaultValue predicate (items: 't array) =
    let mutable result = None
    let mutable i = 0
    while i < items.Length && result.IsNone do
        let item = items[i]
        if predicate item then
            result ← Some item
        i ← i + 1

    result
    ▷ Option.defaultValue defaultValue

let test1' = firstItemOr -1 (fun x → x > 5) [| 1 |]     // -1
```

Early exit alternatives » Functional style

✅ Recursive function 📍

```fsharp
[<TailCall>]
let rec firstItemOr defaultValue predicate list =
    let loop list =
        firstItemOr defaultValue predicate list

    match list with
    | [] → defaultValue            // 👈 Exit
    | x :: _ when predicate x → x  // 👈 Exit
    | _ :: rest → loop rest        // 👈 Continue recursion

// Tests
let test1 = [1]    ▷ firstItemOr -1 (fun x → x > 5)  // -1
let test2 = [1..7] ▷ firstItemOr -1 (fun x → x > 5)  // 6
```

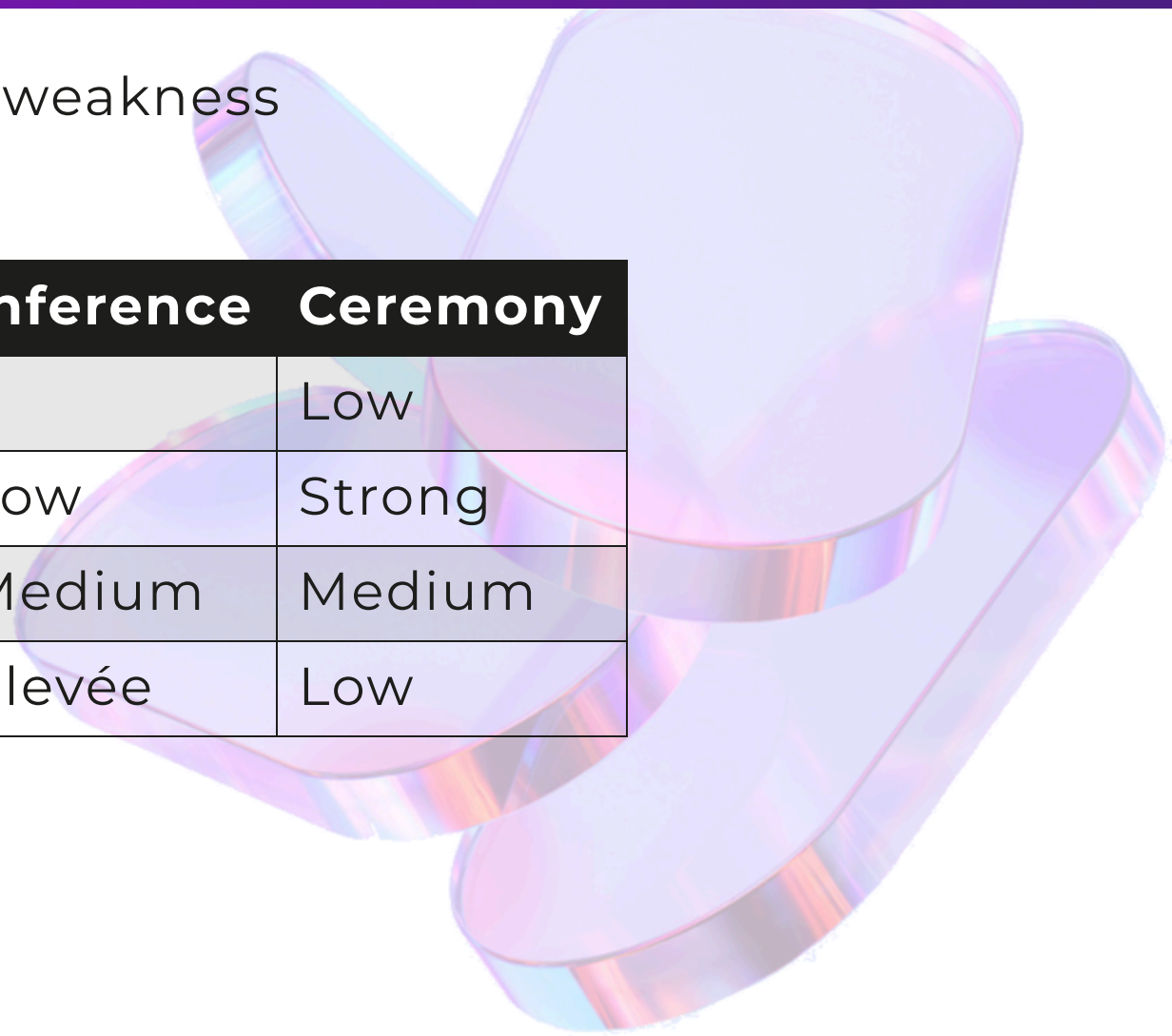
Typing, inference and ceremony

The ceremony is correlated to the typing weakness

🔗 Zone of Ceremony *by Mark Seemann*

Lang	Typing strength	Inference	Ceremony
JS	Low (dynamic)	×	Low
C♯	Medium (static nominal)	Low	Strong
TS	Strong (static structural + ADT)	Medium	Medium
F♯	Strong (static nominal + ADT)	Élevée	Low

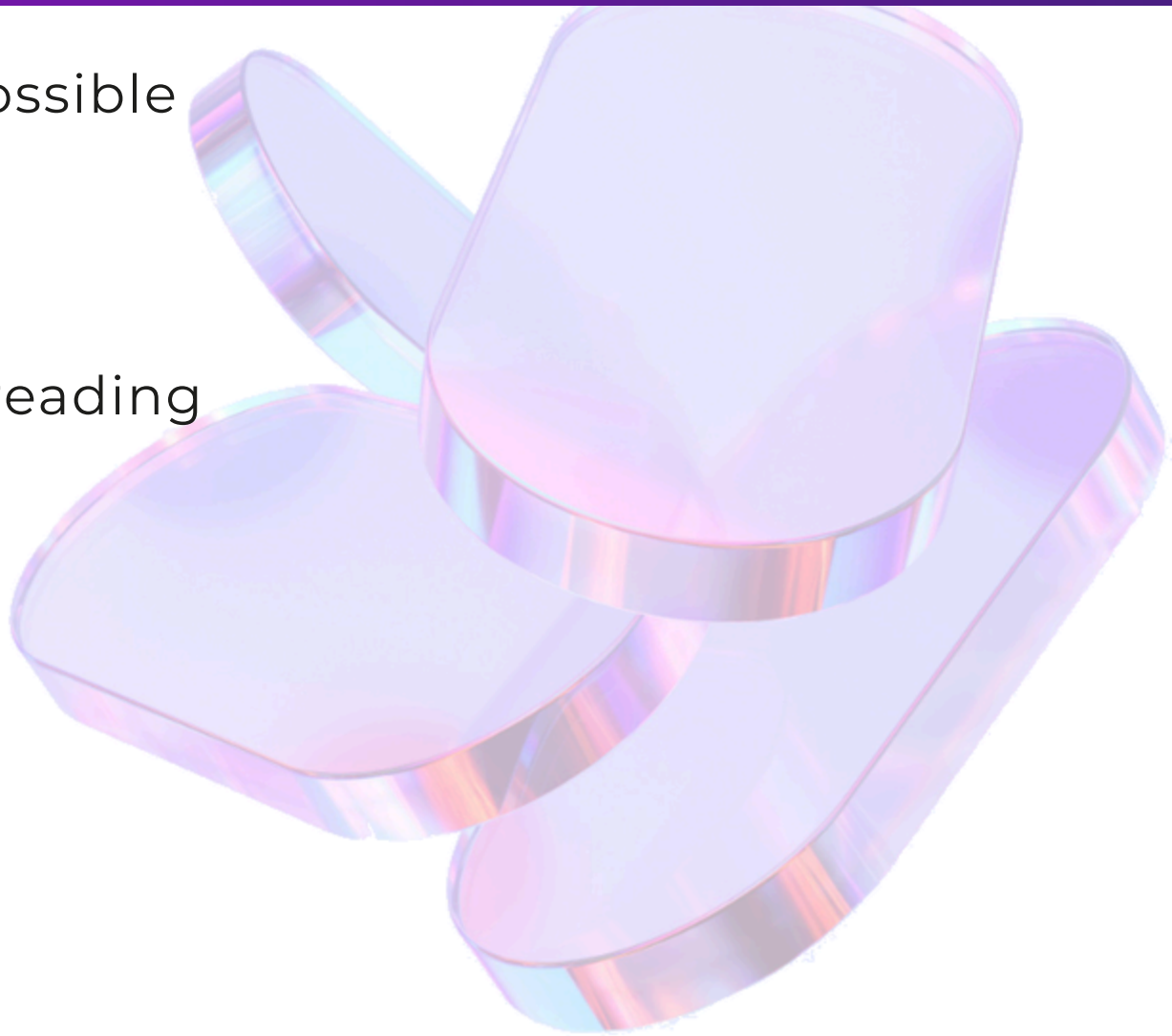
ADT = *Algebraic Data Types* 📌



Type inference

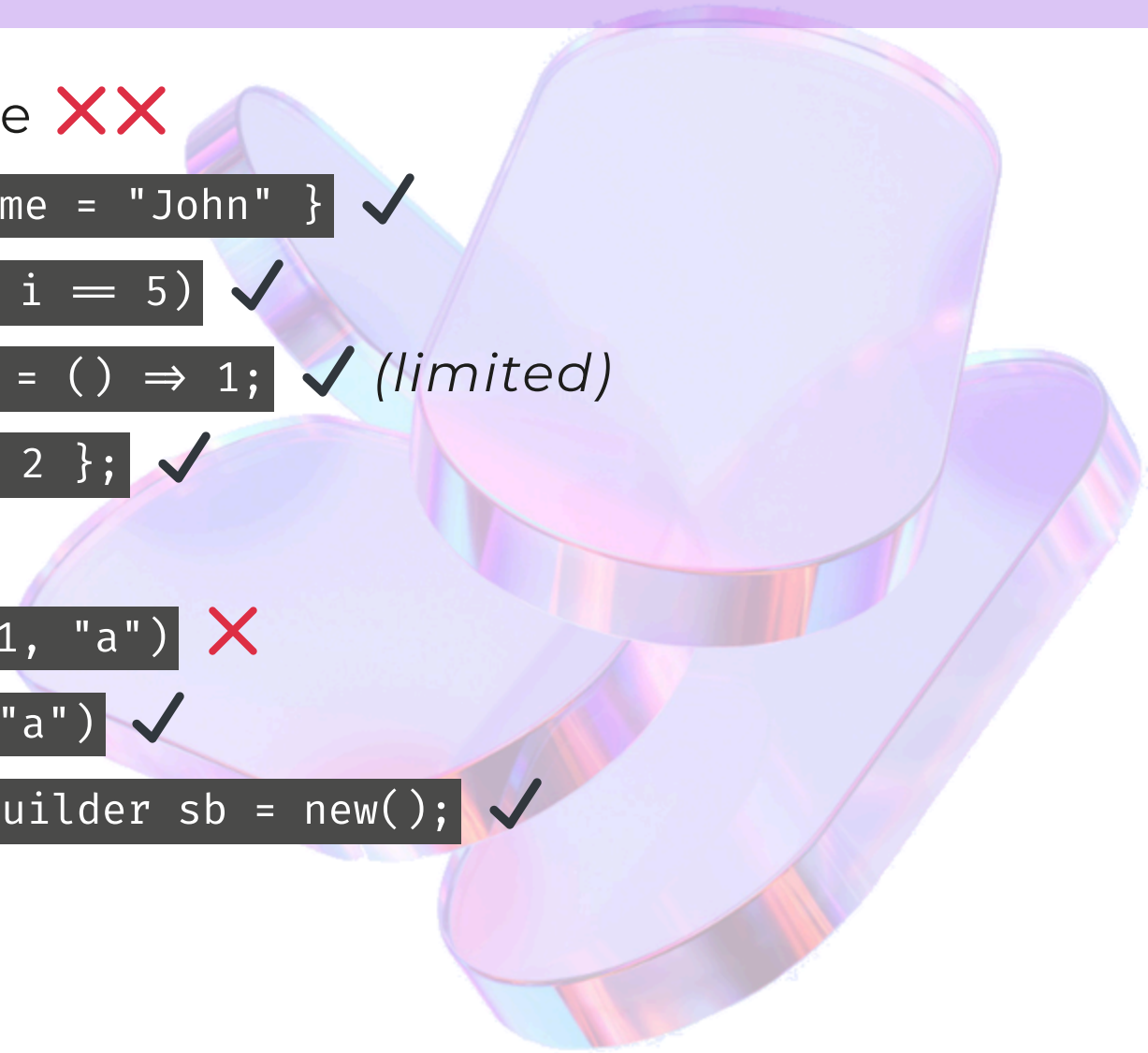
Goal: write type annotations as little as possible

- ➔ Less code to write
- ➔ Compiler ensures consistency
- ➔ IntelliSense helps with coding and reading



Type inference in C♯

- Method parameters and return value ✗✗
- Variable declaration: `var o = new { Name = "John" }` ✓
- Lambda as argument: `list.Find(i ⇒ i == 5)` ✓
- Lambda declaration in C# 10: `var f3 = () ⇒ 1;` ✓ *(limited)*
- Array initialisation: `var a = new[] { 1, 2 };` ✓
- Generic classes:
  - constructor: `new Tuple<int, string>(1, "a")` ✗
  - static helper class: `Tuple.Create(1, "a")` ✓
- C♯ 9 target-typed expression `StringBuilder sb = new();` ✓



Type inference in F♯

Hindley–Milner method

- ➔ Able to deduce the type of variables, expressions and functions
  - ➔ without any type annotation
- ➔ Based on both the implementation and the usage

Example:

```
let helper instruction source =
    if instruction = "inc" then // 1. `instruction` has the same type than `"inc"` ⇒ `string`
      source + 1                // 2. `source` has the same type than `1` ⇒ `int`
    elif instruction = "dec" then
      source - 1
    else
      source                    // 3. `return` has the same type than `source` ⇒ `int`
```

F♯ inference - Automatic generalization

If something can be inferred as generic, it will be
→ Open to more cases 🥳

```fsharp
// Generic value
let a = [] // 'a list

// Generic function with both parameters generic
let listOf2 x y = [x; y]
// val listOf2: x: 'a → y: 'a → 'a list

// Generic type constraint inference: 'a must be "comparable"
let max x y = if x > y then x else y
```

☝️ In F♯, a generic type starts with an apostrophe `'` *(a.k.a. tick)*

- Can be in camelCase (`'a`) or PascalCase (`'T`)
- C♯ `TXxx` → F♯ `'xxx` or `'Xxx`

Inference *vs* type annotation

Pros 👍

→ code terser

→ automatic generalization

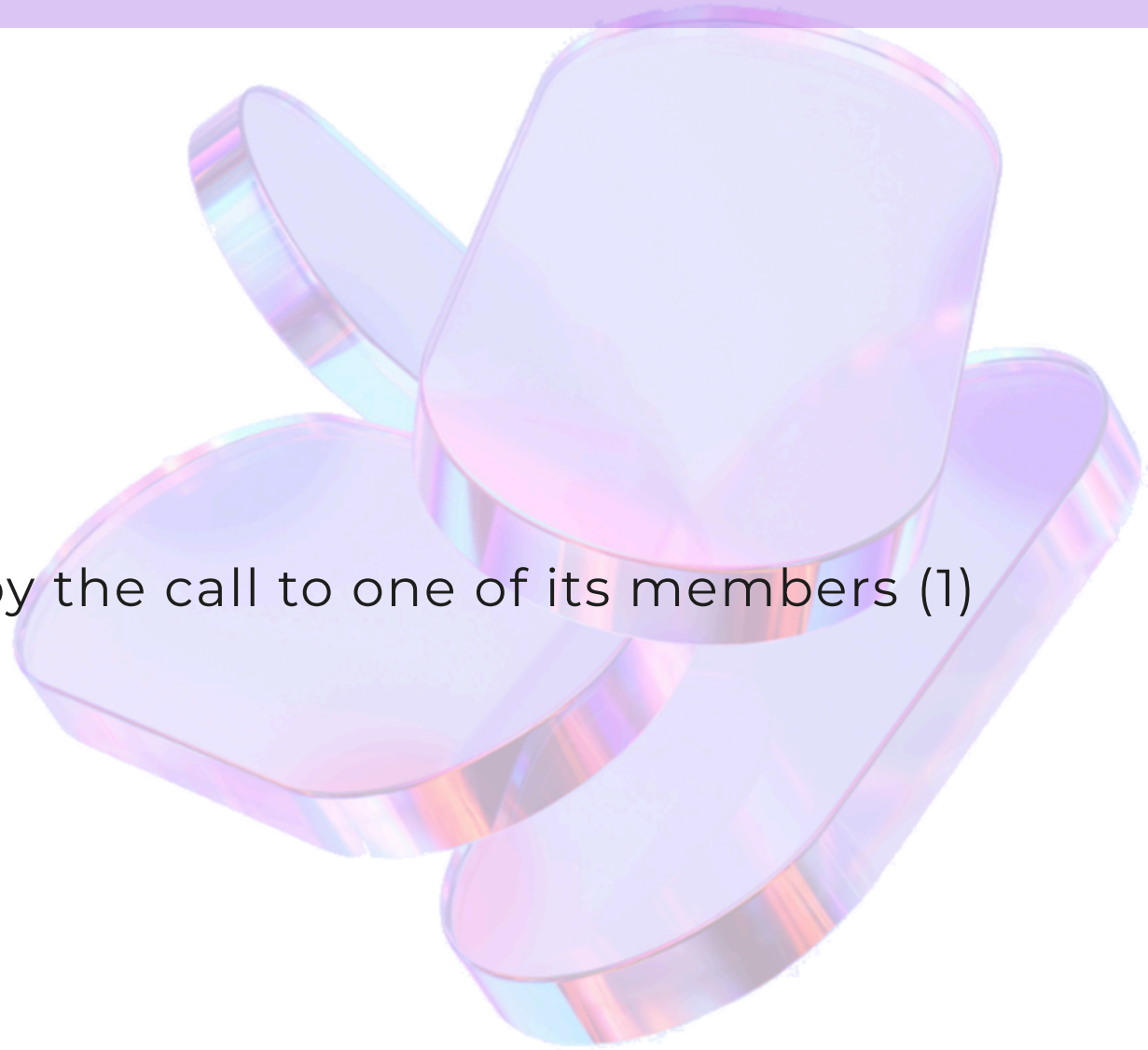
Cons ⚠️

→ we can break code in cascade

→ inference limited:

- an object type cannot be determine by the call to one of its members (1)
- sensible to the instructions order (2)

(1)(2) Example on next slides



Inference *vs* type annotation - Limit #1

⚠️ No inference from *"object dotting" (Exception: records 📍)*

```
let helperKO instruction source =
    match instruction with
    | 'U' → source.ToUpper()
    //       ~~~~~~~~~~~~~~~ 💥
    // Error FS0072: Lookup on object of indeterminate type based on information prior to this program point.
    // A type annotation may be needed prior to this program point to constrain the type of the object.
    | _   → source

let helperOk instruction (source: string) = [ ... ]
// Type annotation needed here  : ^^^^^^

// If there is a function equalivalent to the method, it will work
let info list = if list.Length = 0 then "Vide" else " ... " // 💥 Error FS0072 ...
let info list = if List.length list = 0 then "Vide" else $"{list.Length} éléments" // 👌
```

Inference *vs* type annotation - Limit #2

⚠️ Sensitivity to the instructions order

```fsharp
let listKo = List.sortBy (fun x → x.Length) ["three"; "two"; "one"]
//                               ~~~~~~~~ 💥 Error FS0072: Lookup on object of indeterminate type ...

// Solution 1: reverse the order by piping the list
let listOk = ["three"; "two"; "one"] ▷ List.sortBy (fun x → x.Length)

// Solution 2: use a named function  instead of a lambda
let listOk' = List.sortBy String.length ["three"; "two"; "one"]
```

4. 🍔 Quiz

1. Who is the father of the F♯? ⏱️ 10”

A. Anders Hejlsberg

B. Don Syme

C. Scott Wlaschin

1. Who is the father of the F♯ ?

A. Anders Hejlsberg ❌

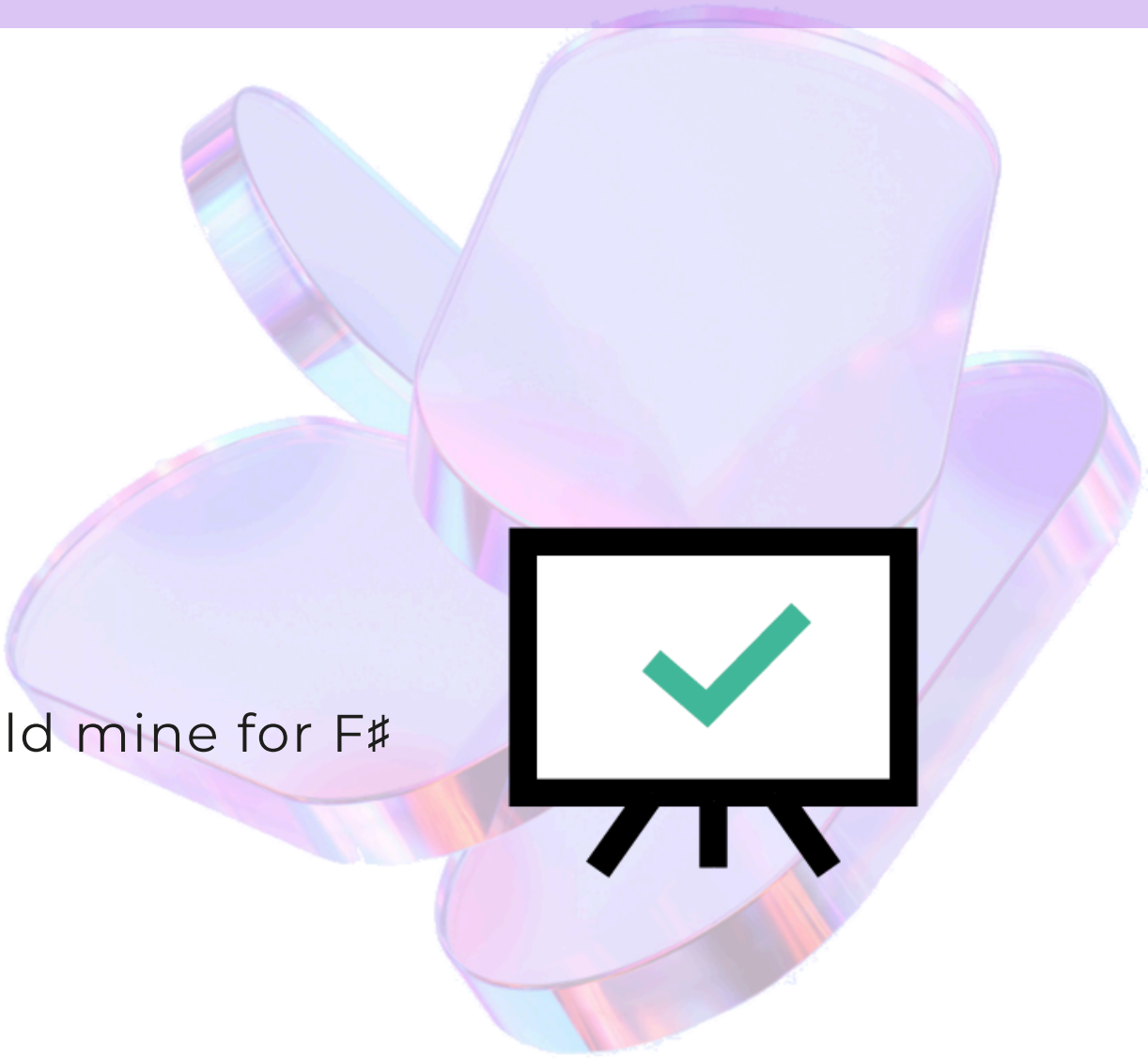
→ Father of C♯ and TypeScript!

B. Don Syme ✅

→ dsymetweets · 📽 F♯ Code I Love

C. Scott Wlaschin ❌

→ Famous blog F♯ for Fun and Profit, a gold mine for F♯



2. What is the name of the `::` operator? ⏱️ 10''

A. Append

B. Concat

C. Cons

2. What is the name of the `::` operator?

A. Append ❌

`List.append` : concatenation of 2 lists

B. Concat ❌

`List.concat` : concatenation of a set of lists

C. Cons ✅

`newItem :: list` is the fasted way to add an item at the top of

3. Find the intruder! ⏱️ 15”

A. `let a = "a"`

B. `let a () = "a"`

C. `let a = fun () → "a"`

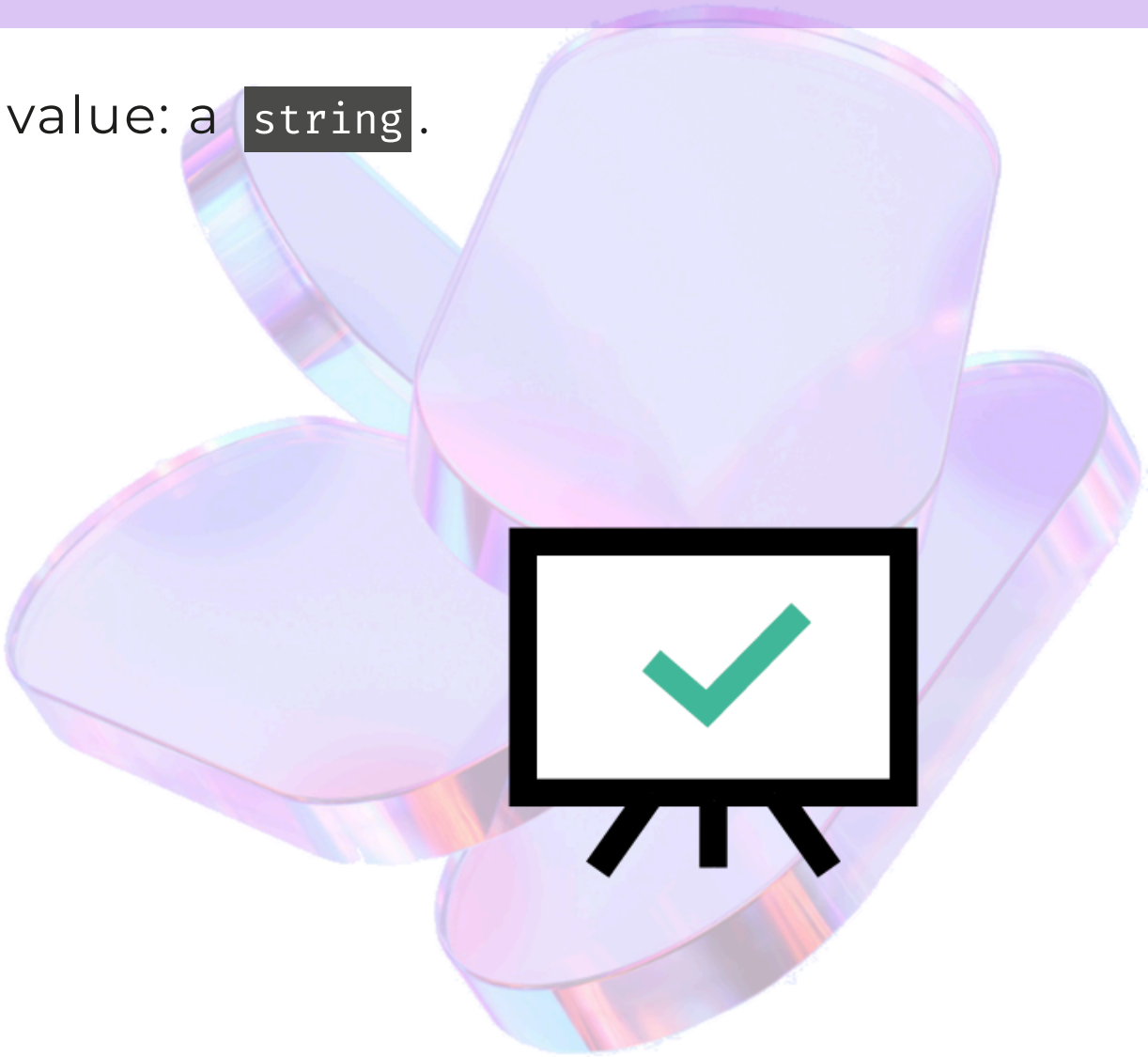
3. Find the intruder!

B and C are functions, while A is a simple value: a `string`.

A. `let a = "a"` ✅

B. `let a () = "a"` ❌

C. `let a = fun () → "a"` ❌



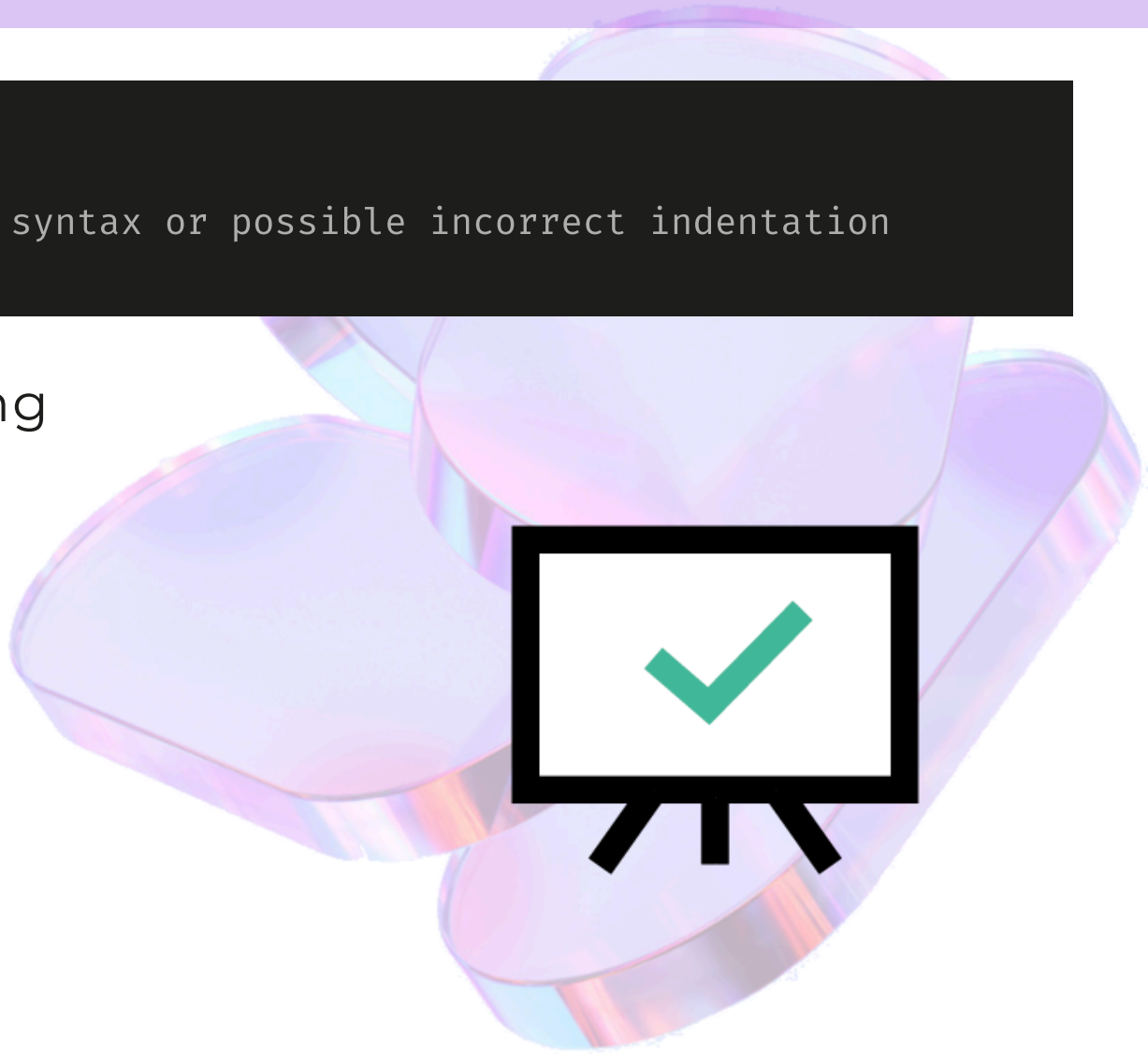
4. What line does not compile? ⏱️ 20”

```
(* 1 *) let evens list =
(* 2 *)     let isEven x =
(* 3 *)     x % 2 = 0
(* 4 *)     List.filter isEven list
```

4. What line does not compile?

```
(* 1 *) let evens list =
(* 2 *)     let isEven x =
(* 3 *)     x % 2 = 0 // 💥 Error FS0058: Unexpected syntax or possible incorrect indentation
(* 4 *)     List.filter isEven list
```

Line **3.** `x % 2 = 0` : an indentation is missing



5. What is the name of ▷ operator? ⏱️ 10”

A. Compose

B. Chain

C. Pipeline

D. Pipe

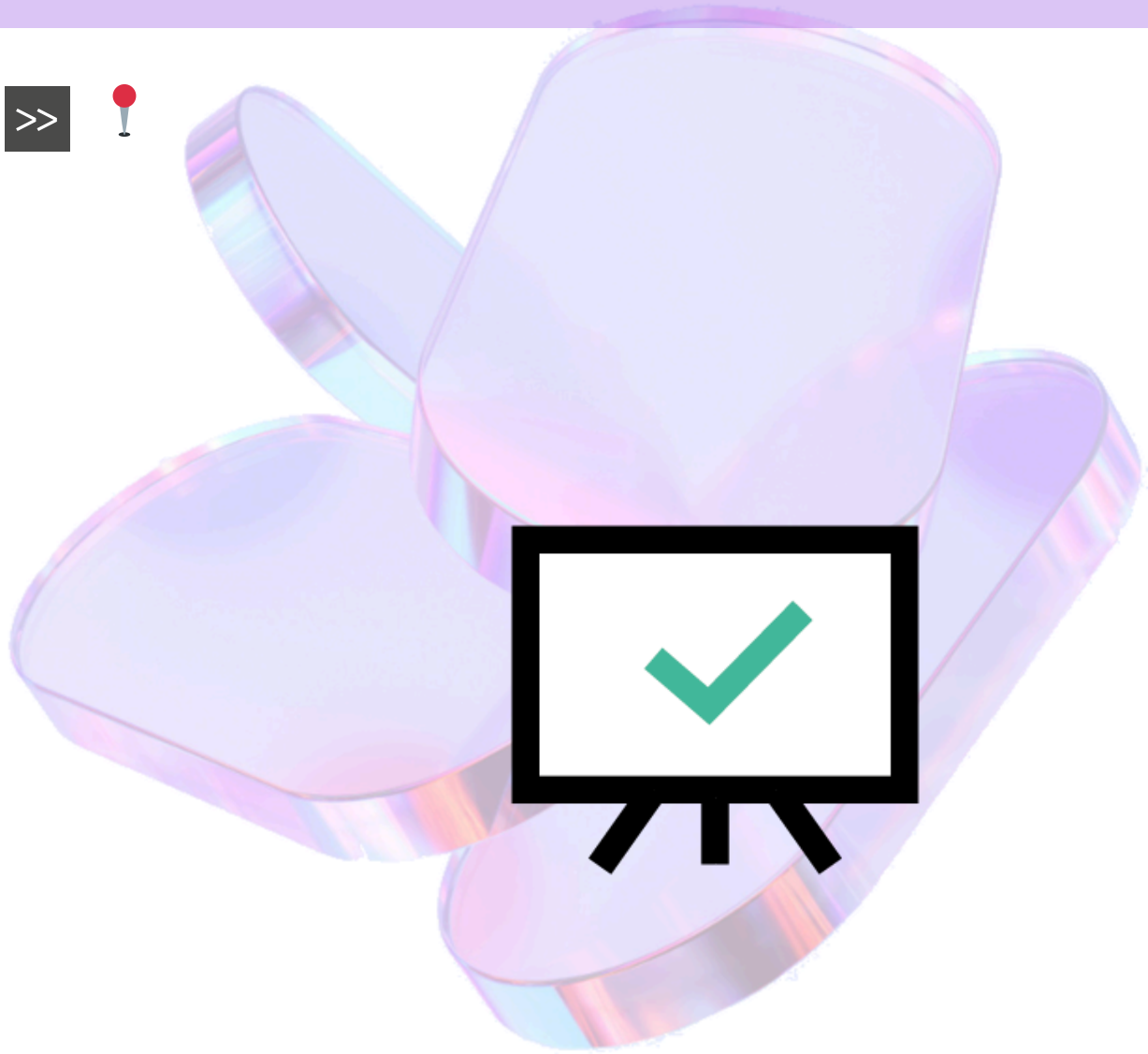
5. What is the name of `|>` operator?

A. Compose ❌ - Composition operator is `>>` 📍

B. Chain ❌

C. Pipeline ❌

D. Pipe ✅



6. Which expression compiles? ⏱️ 20”

A.
```
a == "a" && b ≠ "*"
```

B.
```
a == "a" && b <> "*"
```

C.
```
a = "a" && b <> "*"
```

D.
```
a = "a" && b ≠ "*"
```

6. Which expression compiles?

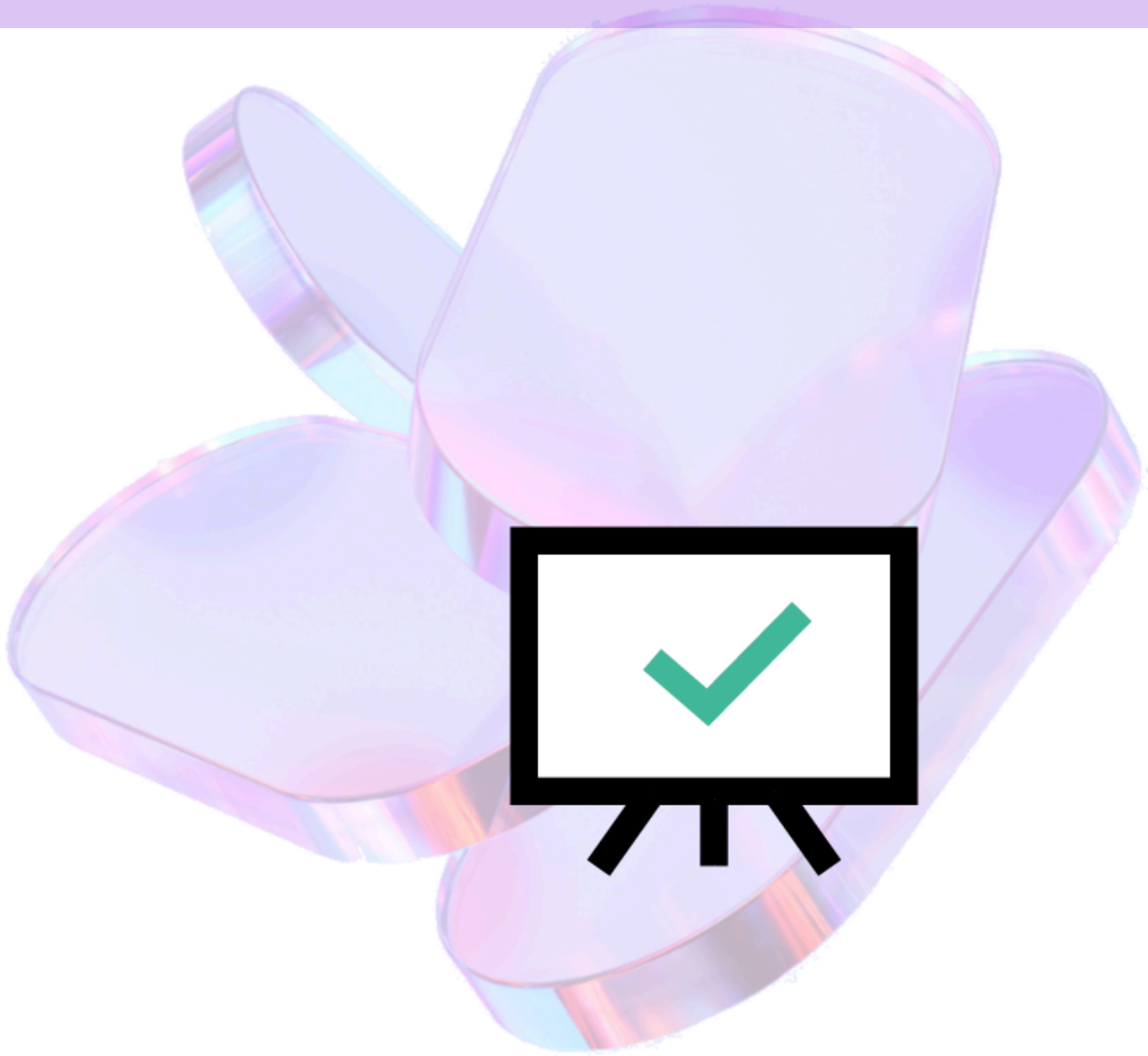
A. `a == "a" && b != ""` ❌

B. `a == "a" && b <> ""` ❌

C. `a = "a" && b <> ""` ✅

D. `a = "a" && b != ""` ❌

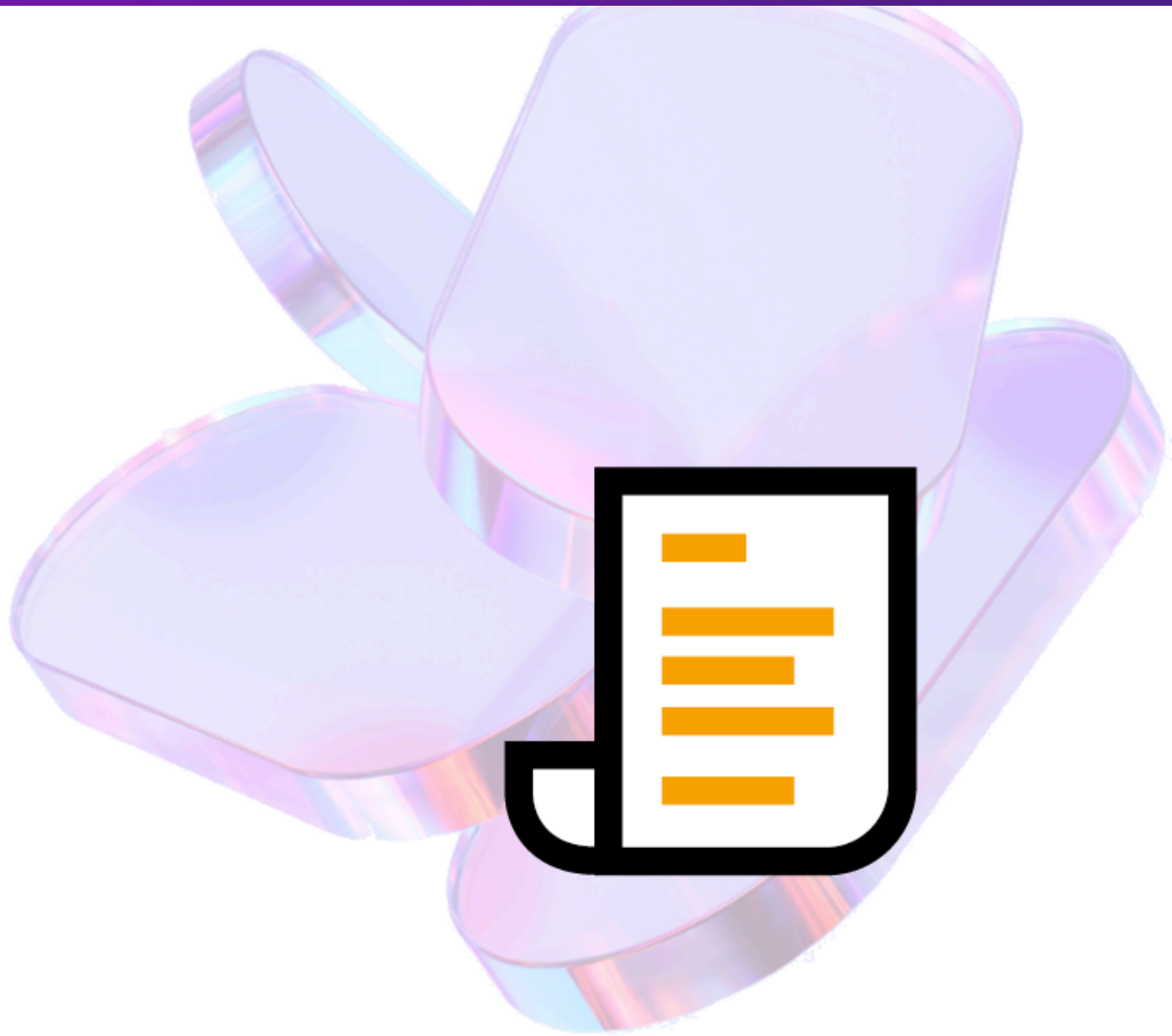
Operator	C♯	F♯
Equality	`==`	`=`
Inequality	`!=` (`!` `=`)	`<>` (`<` `>`)



5. Wrap up

Recap

- F♯ Syntax
- F♯ Language design traits
  - Everything is an expression!
  - Type inference



Addendum

- F♯ Cheatsheet
- Troubleshooting F♯ - Why won't my code compile?

Thanks 🙏