

# F# Training

## *Types*

**2025 April**



# Table of contents

---

- Overview
- Tuples
- Records
- Unions
- Enums
- Anonymous records
- Value types

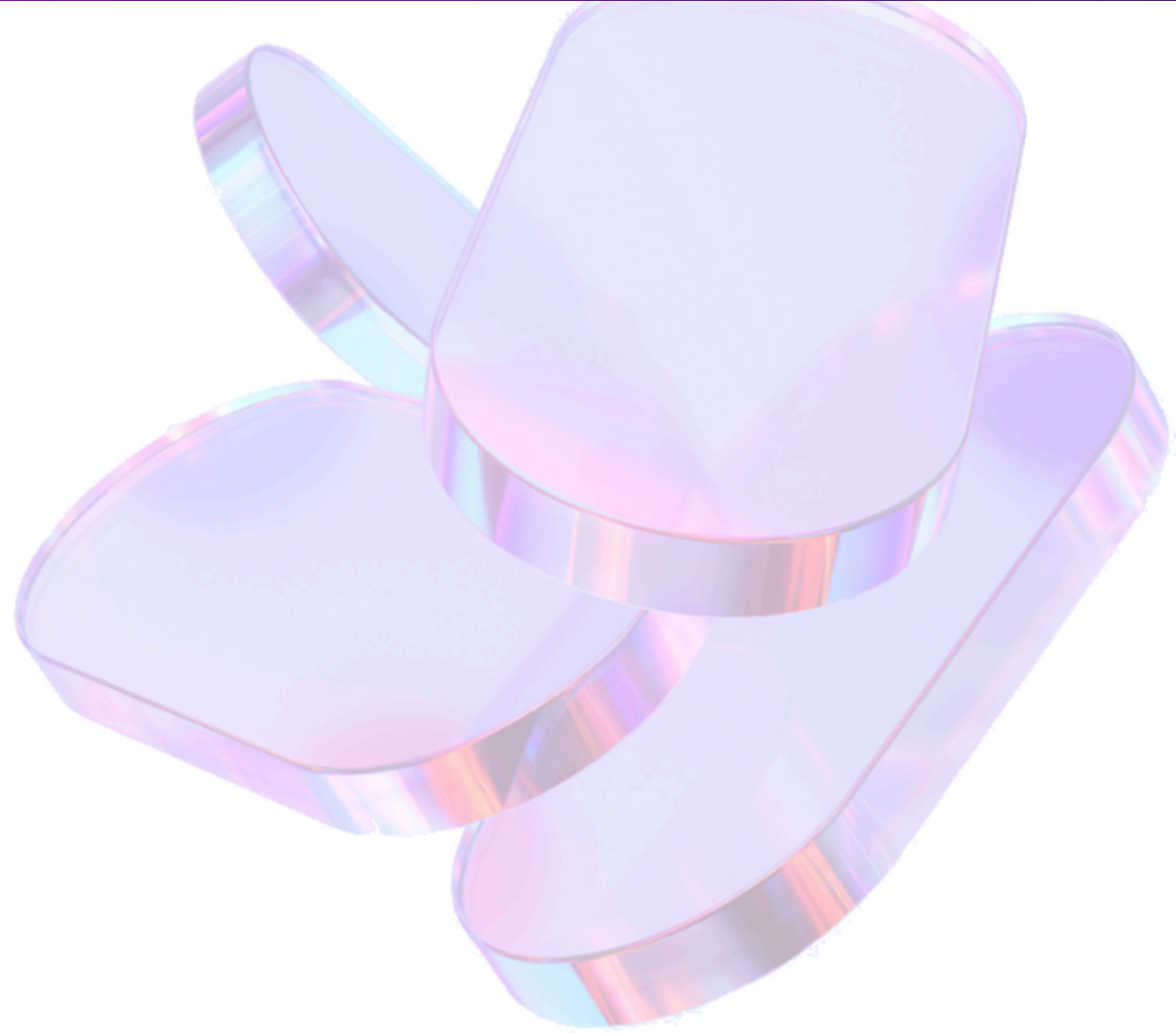


# 1. Types Overview



# .NET type classifications

1. Value types vs reference types
2. Primitive types vs composite types
3. Generic types
4. Types created from literal values
5. Algebraic types: sum vs product



# Composite types

Created by combining other types

👉 F# type features stable and mature

Types	Version	Name	Ref. type	Value type
Types .NET		<code>class</code>	✓	✗
		<code>struct</code> , <code>enum</code>	✗	✓
Specific to C#	C# 3.0	Anonymous type	✓	✗
	C# 7.0	<i>Value tuple</i>	✗	✓
	C# 9.0	<code>record (class)</code>	✓	✗
	C# 10.0	<code>record struct</code>	✗	✓
Specific to F#		<i>Tuple, Record, Union</i>	✓ (default)	✓ (opt-in)
	F# 4.6	<i>Anonymous Record</i>	✓ (default)	✓ (opt-in)

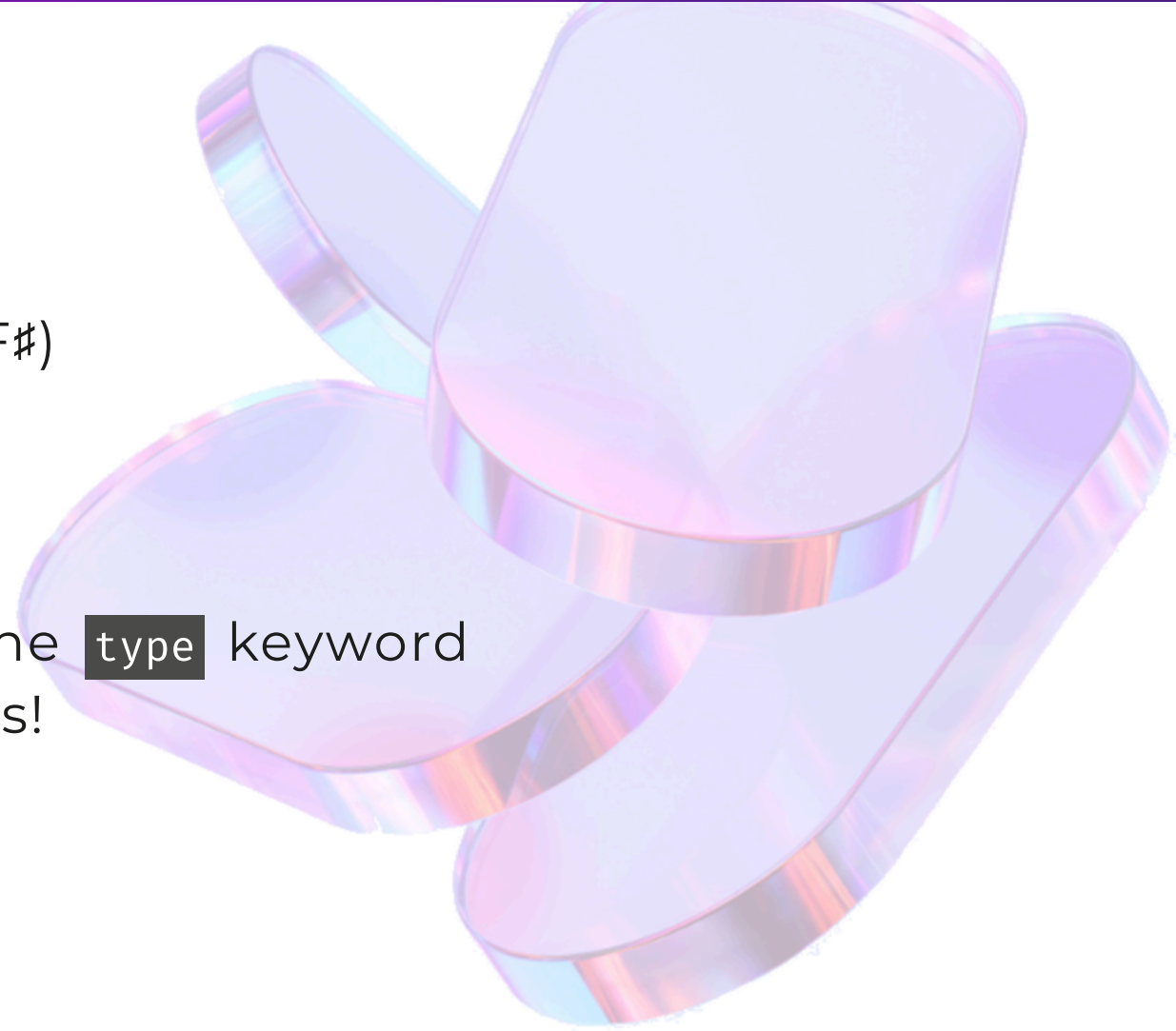
# Composite types (2)

Can be generic (except `enum`)

Location:


- *Top-level* : `namespace`, top-level `module` (F#)
- *Nested* : `class` (C#), `module` (F#)
- Not definable in `let` bindings, `member`

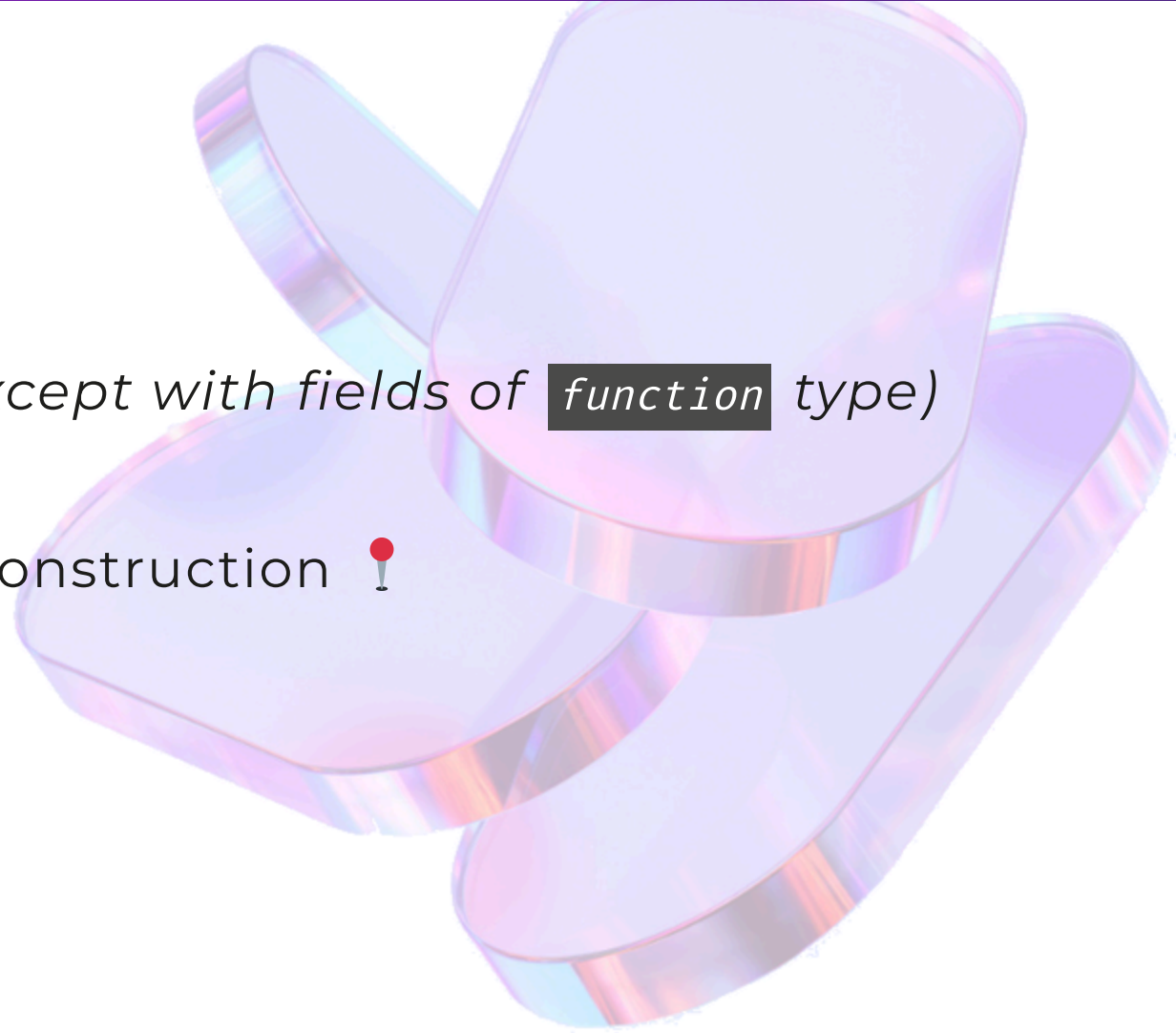
In F#, all type definitions are made with the `type` keyword  
→ including classes, enums and interfaces!  
→ but tuples don't need a type definition



# Particularity of F# types / .NET types

*Tuple, Record, Union* are:

- Immutable by default
- Non-nullable by default
- Equality and structural comparison (*except with fields of `function` type*)
- `sealed`: cannot be inherited
- Deconstruction, with same syntax as construction 





# Types with literal values

Literal values = instances whose type is inferred

- Primitive types: `true` (`bool`) - `"abc"` (`string`) - `1.0m` (`decimal`)
- Tuples C# / F# : `(1, true)`
- Anonymous types C# : `new { Name = "Joe", Age = 18 }`
- Records F# : `{ Name = "Joe"; Age = 18 }`

## 👉 Note :

- Types must be defined beforehand !
- Exception: tuples and C# anonymous types: implicit definition



# Algebraic data types (ADT)

“ Composite types, combining other types by *product* or *sum*. ”

Let's take the types **A** and **B**, then we can create:

- The product type **A × B**:
  - Contains 1 component of type **A** AND 1 of type **B**.
  - Anonymous or named components
- Sum type **A + B**:
  - Contains 1 component of type **A** OR 1 of type **B**.

By extension, same for the product/sum types of N types.

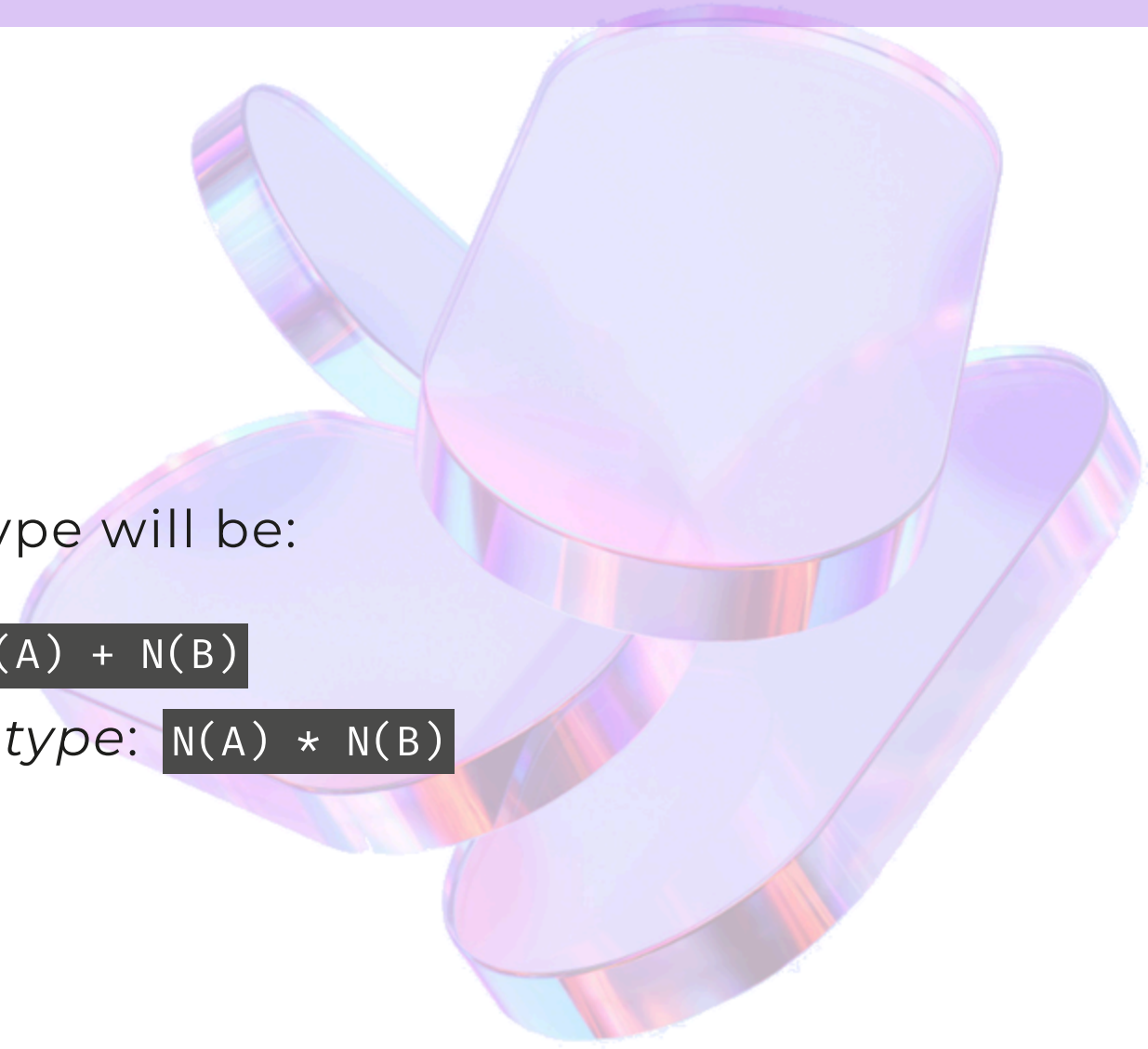
# Why *Sum* and *Product* terms?

It's related to the *number of values*:

- `bool` → 2 values: `true` and `false`
- `unit` → 1 value `()`
- `int` → infinite number of values

The number of values in the composed type will be:

- The sum of numbers for a *sum type*:  $N(A) + N(B)$
- The product of numbers for a *product type*:  $N(A) * N(B)$



# Algebraic types vs Composite types

Type	Sum	Product
<code>enum</code>	✓	✗
<i>Union</i> F#	✓	✗
<code>class</code> (1), <code>interface</code> , <code>struct</code>	✗	✓
<i>Record</i> F#	✗	✓
<i>Tuple</i> F#	✗	✓

(1) C# classes in the broadest sense:

→ including modern variations like *anonymous type*, *Value tuple* and *Record*

👉 In C#, only 1 sum type: `enum`, very limited / union type ⓘ



# Type abbreviation

**Alias** of another type: `type [name] = [existingType]`

Different use-cases:

```
// 1. Document code to avoid repetition
type ComplexNumber = float * float
type Addition<'num> = 'num → 'num → 'num // ➡ Also works with generics

// 2. Decouple (partially) usage / implementation
//    → Easier to change the implementation (for a stronger type)
type ProductCode = string
type CustomerId = int
```

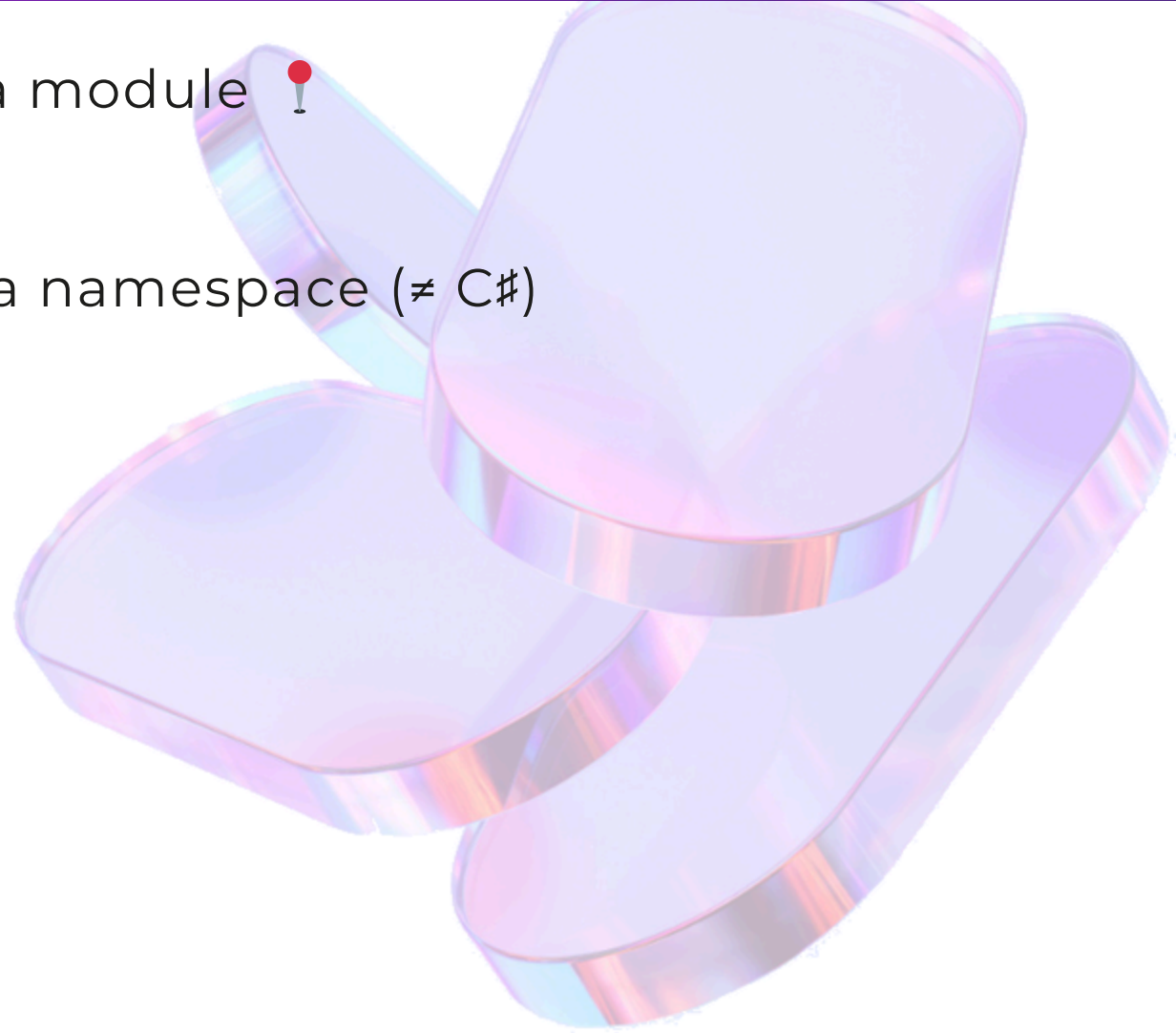
⚠ Deleted at compile time → ~~type safety~~  
→ Compiler allows `int` to be passed instead of `CustomerId`!

# Type abbreviation (2)

💡 It's also possible to create an alias for a module 📌

```
module [name] = [existingModule]
```

⚠️ It's NOT possible to create an alias for a namespace (≠ C#)



# 2. Tuple Type



# Tuples: key points

Types constructed from **literal values**

Anonymous types

*but aliases can be defined to give them a name*

Product types by definition

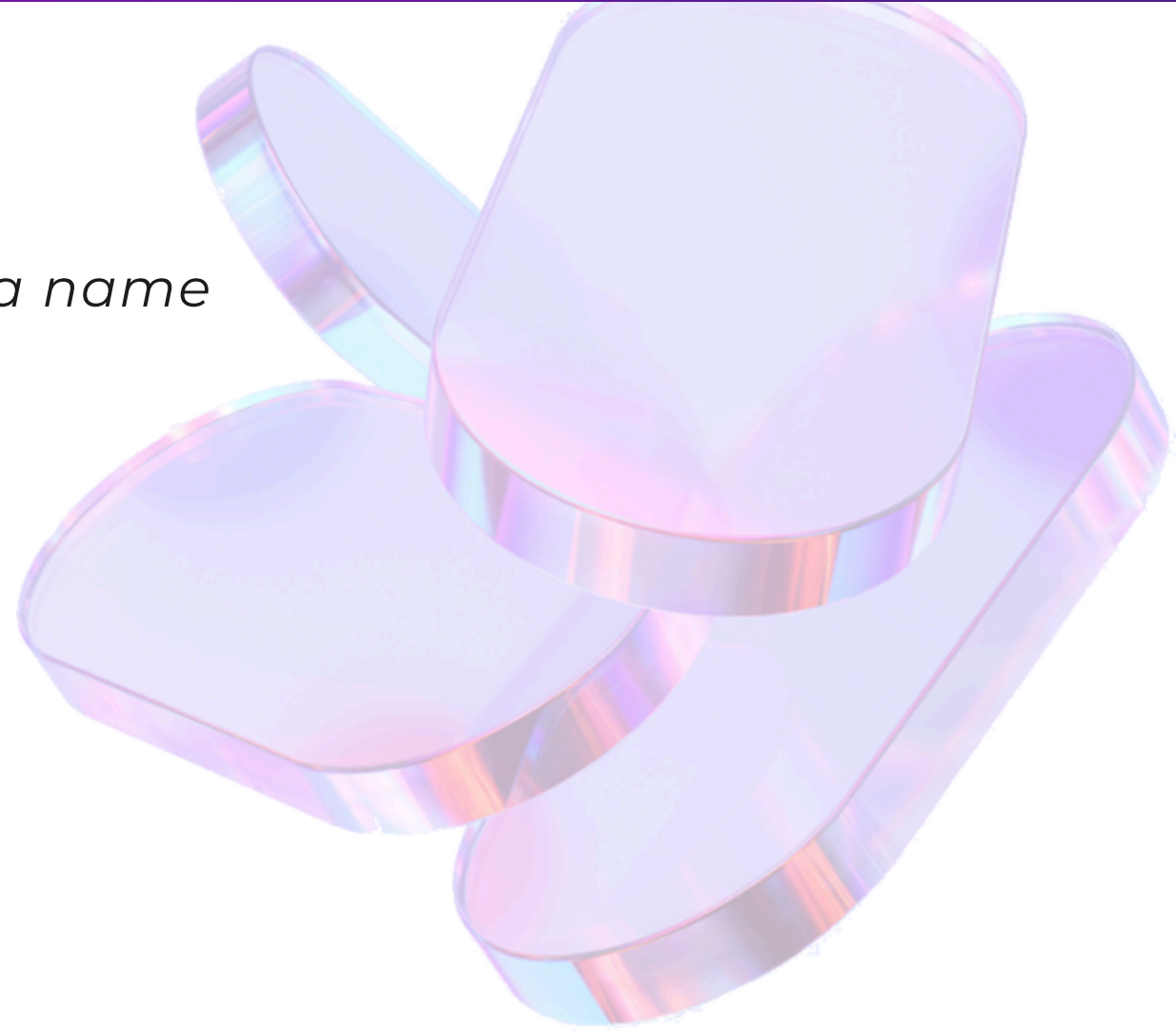
→ `*` sign in the type signature: `A * B`

Number of elements in the tuples:

- 🔥 2 or 3 (`A * B * C`)
- ⚠️ > 3 : possible but prefer *Records*

Element order matters

→ `A * B`  $\neq$  `B * A` (if `A`  $\neq$  `B`)





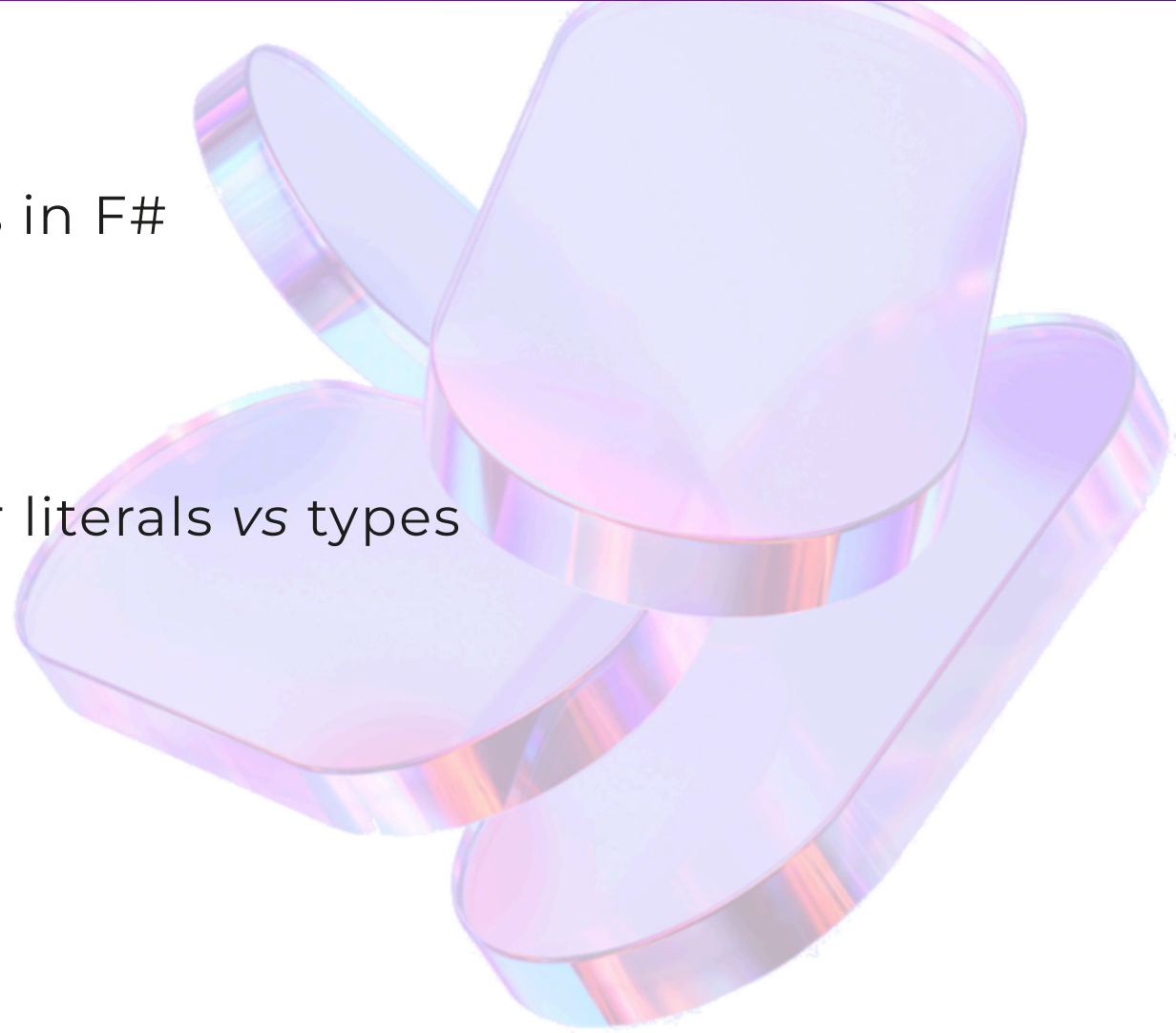
# Tuples: construction

Syntax of literals: `a,b` or `a, b` or `(a, b)`

- Comma `,`: symbol dedicated to tuples in F#
- Spaces are optional
- Parentheses `()` may be necessary

⚠ **Pitfall:** the symbol used is different for literals vs types

- `,` for literal
- `*` for signature
- E.g. `true, 1.2` → `bool * float`



# Tuples: deconstruction

- Same syntax as construction 👍
- All elements must appear in the deconstruction ⚠
  - Use `_` (*discard*) to ignore one of the elements

```
let point = 1.0, 2.5
let x, y = point

let x, y = 1, 2, 3 // ✨ Error FS0001: Type incompatibility ...
                  // ... Tuples have lengths other than 2 and 3

let result = System.Int32.TryParse("123") // (bool * int)
let _, value = result // Ignore the "bool"
```

# Tuples in practice

Use a tuple for a data structure:

- Small: 2 to 3 elements
- Light: no need for element names
- Local: small scope

Immutable tuple:

→ modifications are made by creating a new tuple

```
let addOneToTuple (x, y, z) = x + 1, y + 1, z + 1
```



# Tuples in practice (2)

**Structural equality**, but only between 2 tuples of the same signature!

```
(1,2) = (1,2)           // true
(1,2) = (0,0)           // false
(1,2) = (1,2,3)         // ✨ Error FS0001: Type incompatibility ...
                        // ... Tuples have lengths other than 2 and 3
(1,2) = (1,(2,3))        // ✨ Error FS0001: This expression was supposed to have type `int` ...
                        // ... but here it has type `a * b`
```

**Nesting** of tuples using `()`

```
let doublet = (true,1), (false, "a") // (bool * int) * (bool * string) → pair of pairs
let quadruplet = true, 1, false, "a" // bool * int * bool * string → quadruplet
doublet = quadruplet                  // ✨ Error FS0001: Type incompatibility ...
```

# Tuples: pattern matching

Patterns recognized with tuples:

```
let print move =  
    match move with  
    | 0, 0 → "No move" // Constant 0  
    | 0, y → $"Vertical {y}" // Variable y (≠ 0)  
    | x, 0 → $"Horizontal {x}"  
    | x, y when x = y → $"Diagonal {x}" // Condition x and y equal  
    // `x, x` is not a recognized pattern !  
    | x, y → $"Other ({x}, {y})"
```

## 👉 Notes:

- Patterns are ordered from specific to generic
- The last pattern `x, y` is the default one to deconstruct a tuple

# Pairs

- 2-element tuples
- So common that 2 helpers are associated with them:
  - `fst` as *first* to extract the 1st element of the pair
  - `snd` as *second* to extract the 2nd element of the pair
  - ⚠ Only works for pairs

```
let pair = 'a', "b"  
fst pair // 'a' (char)  
snd pair // "b" (string)
```

# Pair Quiz

## 1. Implement `fst` and `snd`

```
let fst ... ?  
let snd ... ?
```

## 2. What is the signature of this function?

```
let toList (x, y) = [x; y]
```





# Pair Quiz

## 1. How do you implement `fst` and `snd` yourself?

```
let inline fst (x, _) = x // Signature : 'a * 'b -> 'a  
let inline snd (_, y) = y // Signature : 'a * 'b -> 'b
```

- Tuple deconstruction: `(x, y)`
- We *discard* one element using `_` wildcard
- Functions can be `inline`



# Pair Quiz 🎲 🎲

## 2. Signature of `toList` ?

```
let inline toList (x, y) = [x; y]
```

- Returns a list with the 2 elements of the pair
- The elements are therefore of the same type
- There is no constraint on this type → generic `'a`

**Answer :** `'a * 'a → 'a list`



# 3. Record Type



# Records: key points

“ Product type with named elements called *fields*. ”

Alternative to tuples when they are imprecise, for instance `float * float`:

- Point? Coordinates? Vector?
- Real and imaginary parts of a complex number?

Alleviate the doubt by naming both the type and its elements:

```
type Point = { X: float; Y: float }  
type Coordinate = { Latitude: float; Longitude: float }  
type ComplexNumber = { Real: float; Imaginary: float }
```

# Records: declaration

Base syntax:

```
type RecordName =  
    { Label1: type1  
      Label2: type2  
      ... }
```

👉 Field labels in PascalCase, not camelCase → see [MS style guide](#)

Complete syntax:

```
[ attributes ]                // [<Struct>]  
type [accessibility-modifier] RecordName = // private, internal  
    { [ mutable ] Label1: type1  
      [ mutable ] Label2: type2  
      ... }  
    [ member-list ]           // Properties, methods ...
```

# Record declaration: formatting styles

- Single-line: properties separated by `;`
- Multi-line: properties separated by line breaks
  - 3 variations: *Cramped*, *Aligned*, *Stroustrup*

```
// Single line
type PostalAddress = { Address: string; City: string; Zip: string }

// Cramped: historical
type PostalAddress =
{ Address: string
  City: string
  Zip: string }

// Aligned: C#-like
type PostalAddress =
{
    Address: string
    City: string
    Zip: string
}

// Stroustrup: C++-like
type PostalAddress = {
    Address: string
    City: string
    Zip: string
}
```

# Record: styles comparison

Criterion	Best styles 🏆
Compactness	Single-line, Cramped
Refacto Easiness <i>(re)indentation, fields (re)ordering</i>	Aligned, Stroustrup

👉 **Recommendation:** *Strive for Consistency*

- Apply consistently the same multi-line style across a repository
- In addition, use the single-line style when relevant: line with < 80 chars



# Record: styles configuration

Fantomas configuration in the `.editorconfig` file:

```
max_line_length = 180
fsharp_multiline_bracket_style = cramped | aligned | stroustrup

fsharp_record_multiline_formatter = number_of_items
fsharp_max_record_number_of_items = 3
# or
fsharp_record_multiline_formatter = character_width
fsharp_max_record_width = 120
```

[https://fsprojects.github.io/fantomas/docs/end-users/Configuration.html#fsharp\\_record\\_multiline\\_formatter](https://fsprojects.github.io/fantomas/docs/end-users/Configuration.html#fsharp_record_multiline_formatter)

# Record members declaration styles

☞ Members are declared after the fields

## Single-line

```
// `with` keyword required
type PostalAddress = { Address: string; City: string; Zip: string } with
    member x.ZipAndCity = $"{x.Zip} {x.City}"

// Or use line breaks (recommended when ≥ 2 members)
type PostalAddress =
    { Address: string; City: string; Zip: string }

    member x.ZipAndCity = $"{x.Zip} {x.City}"
    member x.CityAndZip = $"{x.City}, {x.Zip}"
```

# Record member declaration styles (2)

## Multi-line: *Cramped* and *Aligned*

👉 2 line breaks

```
type PostalAddress =  
    { Address: string  
      City: string  
      Zip: string }  
  
member x.ZipAndCity = $"{x.Zip} {x.City}"  
member x.CityAndZip = $"{x.City}, {x.Zip}"
```

# Record member declaration styles (3)

## Multi-line: *Stroustrup*

👉 **with** keyword + 1 line break + indentation

```
type PostalAddress = {  
    Address: string  
    City: string  
    Zip: string  
} with  
    member x.ZipAndCity = $"{x.Zip} {x.City}"  
    member x.CityAndZip = $"%s{x.City}, %s{x.Zip}"
```

# Record expression for instantiation

- Same syntax as an anonymous C# object without the `new` keyword
- All fields must be populated, but in any order (but can be confusing)
- Same possible styles: single/multi-lines

```
type Point = { X: float; Y: float }  
let point1 = { X = 1.0; Y = 2.0 }  
let pointKo = { Y = 2.0 }           // ✨ Error FS0764  
// ~~~~~ FS0764: No assignment given for field 'X' of type 'Point'
```

⚠ **Trap:** differences declaration / instantiation

- `:` for field type in record declaration
- `=` for field value in record expression

# Record deconstruction

- Fields are accessible by "dotting" into the object
- Alternative: deconstruction
  - Same syntax for deconstructing a *Record* as for instantiating it 👍
  - Unused fields can be ignored 💡

```
let { X = x1 } = point1  
let { X = x2; Y = y2 } = point1
```

# Record deconstruction (2)

⚠ Additional members (*properties*) cannot be deconstructed!

```
type PostalAddress =  
  {  
    Address: string  
    City: string  
    Zip: string  
  }  
  member x.CityLine = $"{x.Zip} {x.City}"  
  
let address = { Address = ""; City = "Paris"; Zip = "75001" }  
  
let { CityLine = cityLine } = address // ✨ Error FS0039  
// ~~~~~ The record label 'CityLine' is not defined  
let cityLine = address.CityLine      // 🟡 OK
```



# Record: inference

- A record type can be inferred from the fields used 👍 but not with members !
- As soon as the type is inferred, IntelliSense will work

```
type PostalAddress =  
    { Address: string  
      City: string  
      Zip: string }  
  
let department address =  
    address.Zip.Substring(0, 2) ▷ int  
    //      ^^^^ 💡 Infer that address is of type `PostalAddress`.  
  
let departmentKo zip =  
    zip.Substring(0, 2) ▷ int  
    // ~~~~~ Error FS0072: Lookup on object of indeterminate type
```

# Record: pattern matching

Let's use an example: `inhabitantOf` is a function giving the inhabitant's name (*in French*) at a given address (*in France*)

```
type Address = { Street: string; City: string; Zip: string }

let department { Zip = zip } = int zip[0..1] // Address → int

let private IleDeFrance = Set [ 75; 77; 78; 91; 92; 93; 94; 95 ]
let inIleDeFrance departmentNum = IleDeFrance.Contains(departmentNum) // int → bool

let inhabitantOf address = // Address → string
    match address with
    | { Street = "Pôle"; City = "Nord" } → "Père Noël"
    | { City = "Paris" } → "Parisien"
    | _ when department address = 78 → "Yvelinois"
    | _ when department address ▷ inIleDeFrance → "Francilien"
    | _ → "Français"
```

# Record: name conflict

In F#, typing is nominal, not structural as in TypeScript

→ Use qualification to resolve ambiguity

→ Even better: write  $\neq$  types or put them in  $\neq$  modules

```
type Person1 = { First: string; Last: string }
type Person2 = { First: string; Last: string }
let alice = { First = "Alice"; Last = "Jones" } // val alice: Person2 ... (by proximity)

// ⚠ Deconstruction
let { First = firstName } = alice // Warning FS0667 (in F# 6)
// ~~~~~ The labels of this record do not uniquely
//         determine a corresponding record type

let { Person2.Last = lastName } = alice // 🔥 OK with qualification
let { Person1.Last = lastName } = alice // ✨ Error FS0001
// ~~~~~ Type 'Person1' expected, 'Person2' given
```

# Record: modification

Record is immutable, but easy to get a modified copy

→ **copy and update** expression of a *Record*

→ use multi-line formatting for long expressions

```
// Single-line
let address2 = { address with Street = "Rue Vivienne" }

// Multi-line
let address3 =
    { address with
        City = "Lyon"
        Zip  = "69001" }
```

# Record *copy-update*: C# / F# / JS

```
// Record C# 9.0  
address with { Street = "Rue Vivienne" }
```

```
// F# copy and update  
{ address with Street = "Rue Vivienne" }
```

```
// Object destructuring with spread operator  
{ ...address, street: "Rue Vivienne" }
```

# Copy-update limits (< F# 8)

Reduced readability with several nested levels

```
type Street = { Num: string; Label: string }
type Address = { Street: Street }
type Person = { Address: Address }

let person = { Address = { Street = { Num = "15"; Label = "rue Neuf" } } }

let person' =
    { person with
        Address =
            { person.Address with
                Street =
                    { person.Address.Street with
                        Num = person.Address.Street.Num + " bis" } } }
```

# Copy-update : F# 8 improvements

```
type Street = { Num: string; Label: string }  
type Address = { Street: Street }  
type Person = { Address: Address }  
  
let person = { Address = { Street = { Num = "15"; Label = "rue Neuf" } } }  
  
let person' =  
    { person with  
        Person.Address.Street.Num = person.Address.Street.Num + " bis" }
```

👉 Usually we have to qualify the field: see `Person.`



# 4. Union Type



# Unions: key points

- Exact term: *Discriminated Union (DU)*
- Sum type: represents an **OR**, a **choice** between several Cases
  - Same principle as for an `enum`, but on steroids 🦍
- Each case must have a *Tag (a.k.a Label, Discriminator)*
- Each case **may** contain data
  - As Tuple: its elements can be named -- in camelCase 🙏

```
type Ticket =  
    | Adult           // no data → ≈ singleton stateless  
    | Senior of int   // holds an 'int' (w/o more precision)  
    | Child of age: int // holds an 'int' named 'age'  
    | Family of Ticket list // holds a list of tickets  
                                // recursive type by default (no 'rec' keyword)
```

# Unions: declaration

On several lines: 1 line / case

→ 🙌 Line indented and starting with a `|`

On a single (short) line

→ 💡 No need for the 1st `|`

```
open System

type IntOrBool =
    | Int32 of Int32           // 💡 Tag with the same name as the data
    | Boolean of Boolean

type OrderId = OrderId of int // 🙌 Single-case union
                             // 💡 Tag with the same name as the parent union

type Found<'T> = Found of 'T | NotFound // 💡 Generic union type (no auto generalization)
```

# Unions declaration (2)

Cases can be used without **qualification**: `Int32` vs `IntOrBool.Int32`

Qualification can be forced with `RequireQualifiedAccess` attribute:

- Cases using common terms (e.g. `None`) → to avoid name collision
- Cases names are designed to read better/more explicitly with qualification

Cases must be named in **PascalCase** !

- Since F# 7.0, camelCase is allowed for `RequireQualifiedAccess` unions 💡

# Unions declaration (3)

**Field labels** are helpful for:

- Adding meaning to a primitive type:
  - See `Ticket` previous example: `Senior of int` vs `Child of age: int`
- Distinguish between two fields of the same type
  - See example below:

```
type ComplexNumber =  
    | Cartesian of Real: float * Imaginary: float  
    | Polar of Magnitude: float * Phase: float
```

# Unions: instantiation

Case  $\simeq$  **constructor**

→ Function called with any case data

```
type Shape =  
    | Circle of radius: int  
    | Rectangle of width: int * height: int  
  
let circle = Circle 12           // Type: 'Shape', Value: 'Circle 12'  
let rect = Rectangle(4, 3)       // Type: 'Shape', Value: 'Rectangle (4, 3)'  
  
let circles = [1..4] ▷ List.map Circle    // ➡ Case used as function
```

# Unions: name conflict

When 2 unions have tags with the same name  
→ Qualify the tag with the union name

```
type Shape =  
    | Circle of radius: int  
    | Rectangle of width: int * height: int  
  
type Draw = Line | Circle    // 'Circle' will conflict with the 'Shape' tag  
  
let draw = Circle            // Type='Draw' (closest type) -- ⚠ to be avoided as ambiguous  
  
// Tags qualified by their union type  
let shape = Shape.Circle 12  
let draw' = Draw.Circle
```



# Unions: get the data out

- Only via *pattern matching*.
- Matching a union type is **exhaustive**.

```
type Shape =  
    | Circle of radius: float  
    | Rectangle of width: float * height: float  
  
let area shape =  
    match shape with  
    | Circle r → Math.PI * r * r    // 💡 Same syntax as instantiation  
    | Rectangle (w, h) → w * h  
  
let isFlat = function  
    | Circle 0.  
    | Rectangle (0., _)  
    | Rectangle (_, 0.) → true  
    | Circle _  
    | Rectangle _ → false
```

# Single-case unions

Unions with a single case encapsulating a type (usually primitive)

```
type CustomerId = CustomerId of int
type OrderId = OrderId of int

let fetchOrder (OrderId orderId) = // 💡 Direct deconstruction without 'match' expression
    ...
```

- **Benefits** 👍

- Ensures *type safety* unlike simple type alias
  - Impossible to pass a `CustomerId` to a function waiting for an `OrderId`
- Prevents *Primitive Obsession* at a minimal cost

- **Trap** ⚠️

- `OrderId orderId` looks like C# parameter definition

# Enum style unions

All cases are empty = devoid of data

→ ≠ .NET `enum` based on numeric values !

Instantiation and pattern matching are done just with the `Case`.

→ The `Case` is no longer a ~~function~~ but a *singleton* value.

```
type Answer = Yes | No | Maybe
let answer = Yes

let print answer =
    match answer with
    | Yes    → printfn "Oui"
    | No     → printfn "Non"
    | Maybe  → printfn "Peut-être"
```

[!\[\]\(cbe2492b119e39e02a1dab2af4a4b296\_img.jpg\) “Enum” style unions | F# for fun and profit](#)

# Unions .Is\* properties

The compiler generates `.Is{Case}` properties for each case in a union

- Before F# 9: not accessible + we cannot add them manually 😞
- Since F# 9: accessible 👍

```
type Contact =  
    | Email of address: string  
    | Phone of countryCode: int * number: string  
  
type Person = { Name: string; Contact: Contact }  
  
let canSendEmailTo person = // Person → bool  
    person.Contact.IsEmail // `.IsEmail` is auto-generated
```

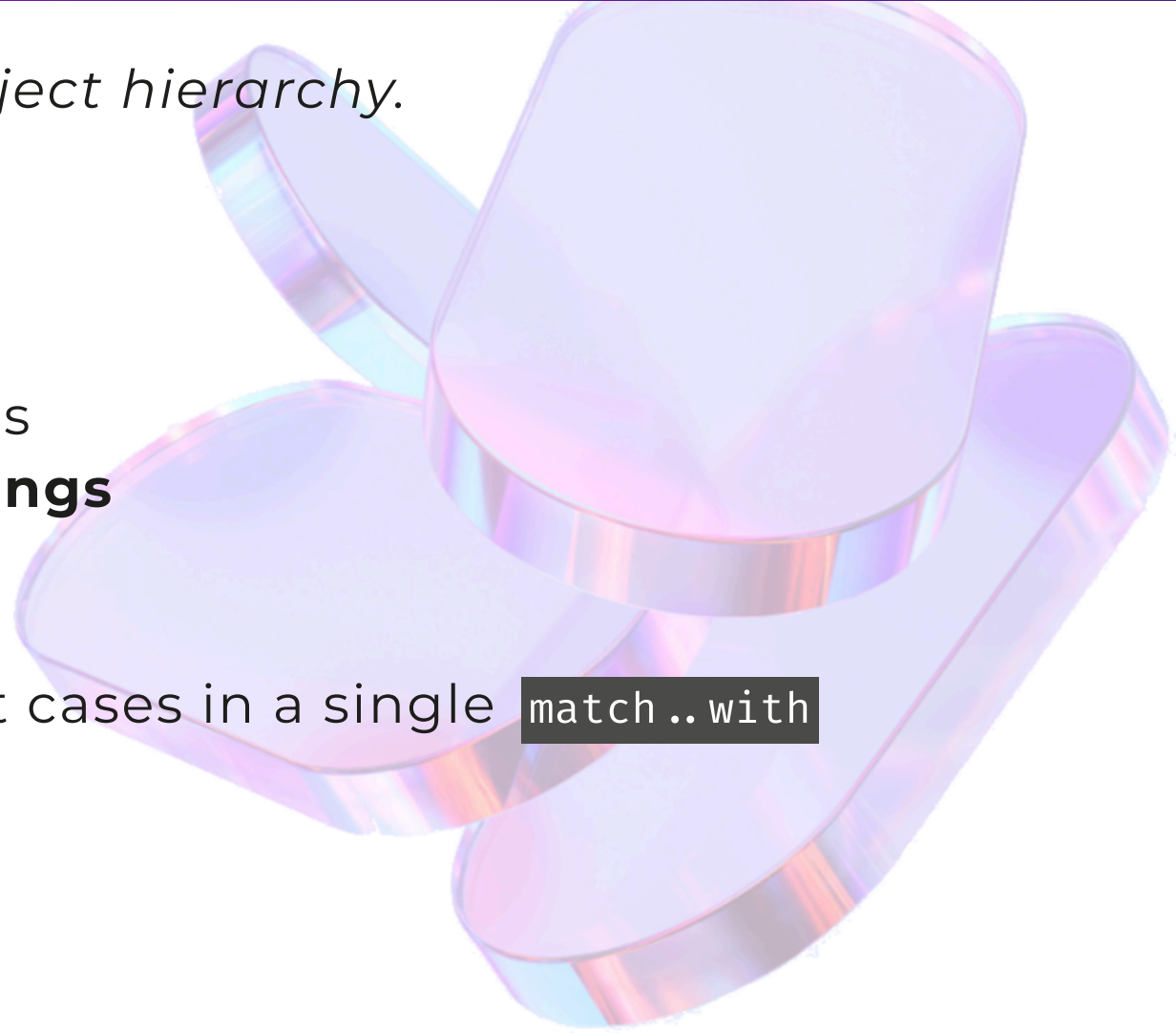
# Union (FP) vs Object Hierarchy (OOP)

👉 A union can usually replace a small *object hierarchy*.

## Explanations

Behaviors/operations implementation:

- **OO:** *virtual methods* in separated classes
- **FP:** *functions* relying on **pattern matchings**
  - exhaustivity
  - avoid duplication by grouping cases
  - improve readability by flattening split cases in a single `match .. with`



# FP vs OOP

**How we reason about the code** *(at both design and reading time)*

- **FP: by functions** → how an operation is performed for the different cases
- **OOP: by objects** → how all operations are performed for a single case

## Abstraction

- Objects are more abstract than functions
- Good abstraction is difficult to design
- The more abstract a thing is, the more stable it should be

👉 **FP is usually easier to write, to understand, to evolve**

# FP vs OOP: Open-Closed Principle

It's easier to extend what's **Open**.

**OOP:** open hierarchy, closed operations

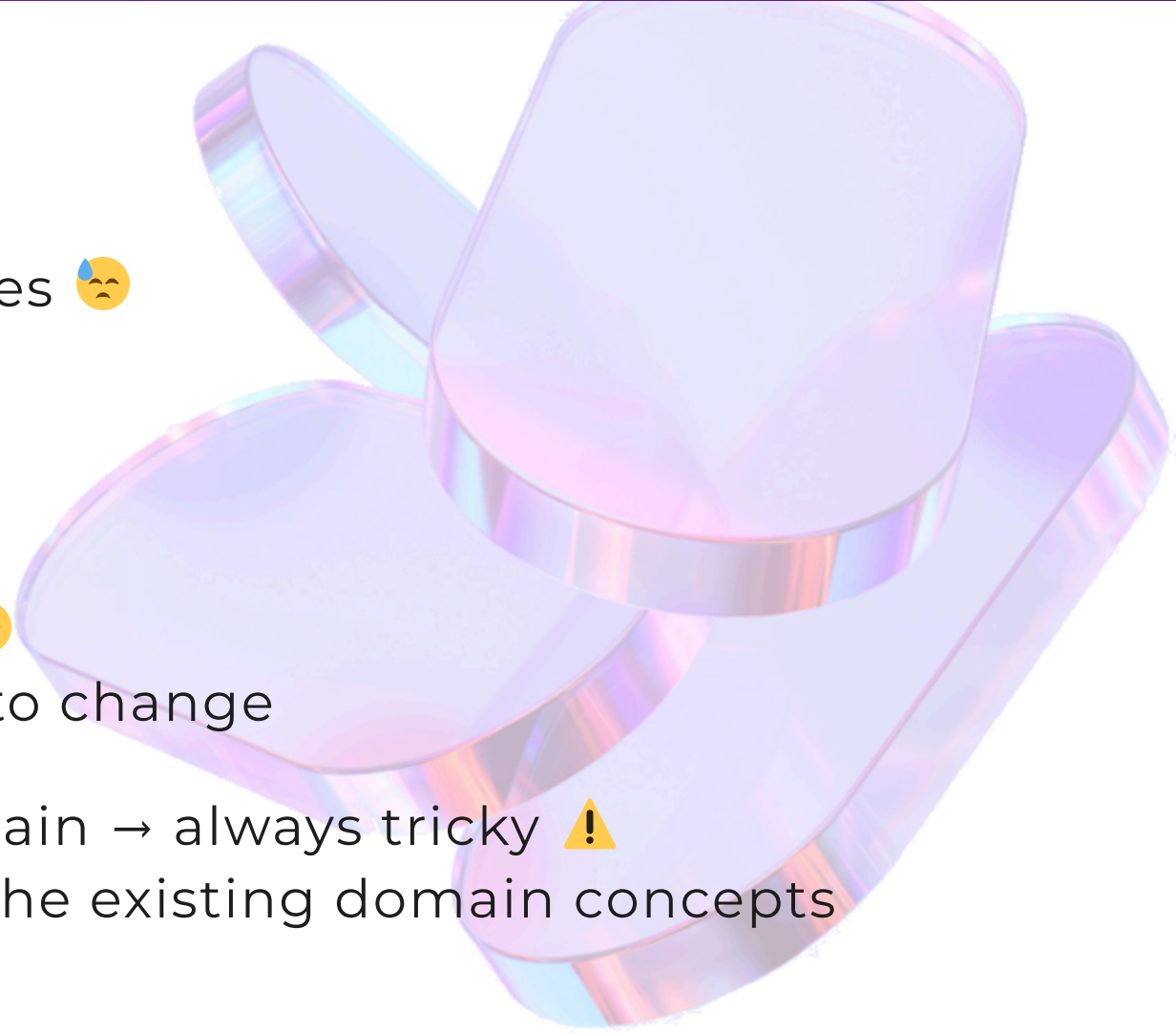
- Painful to add an operation: in all classes 😓
- Easy to add a class in the hierarchy 👍

**FP:** open operations, closed cases

- Easy to add an operation 👍
- Painful to add a case: in all functions 😓
  - Still, it's usually easier in F#: only 1 file to change

Adding a class = new concept in the domain → always tricky ⚠️

Adding an operation = new behavior for the existing domain concepts





# 5. Enum Type



Real .NET `enum`

# Enum: declaration

Set of integer constants (`byte`, `int`...) or `char`

```
type ColorN =  
    | Red    = 1  
    | Green  = 2  
    | Blue   = 3
```

👉 Note the syntax difference with a enum-like union:

```
type Color = Red | Green | Blue
```

# Enum: underlying type

The underlying type is defined by means of literals defining member values:

- `1, 2, 3` → `int`
- `1uy, 2uy, 3uy` → `byte`
- Etc. - see [Literals](#)

→ Same type required for all members:

```
type ColorN =  
    | Red    = 1  
    | Green  = 2  
    | Blue   = 3uy  
// ✨ ~~~  
// This expression was expected to have type 'int' but here has type 'byte'
```

# Enum: char based

💡 Enum can be based on `char` but not on `string`

```
type AnswerChar = Yes='Y' | No='N' ✓
```

```
type AnswerStringKo = Yes="Y" | No="N" // ✨ Error FS0951  
// Literal enumerations must have type int, uint, int16, uint16, int64, uint64, byte, sbyte or char
```

# Enum members naming

💡 Enum members can be in **camelCase**

```
type File = a='a' | b='b' | c='c'
```



# Enum: usages

⚠ Unlike unions, the use of an `enum` literal is necessarily **qualified**

```
type AnswerChar = Yes='Y' | No='N'  
  
let answerKo = Yes // ✨ Error FS0039  
//           ~~~ The value or constructor 'Yes' is not defined.  
  
let answer = AnswerChar.Yes // 🟡 OK
```

💡 We can force the qualification for union types too:

```
[<RequireQualifiedAccess>] // ➡  
type Color = Red | Green | Blue
```

# Enum: matching

⚠ Unlike unions, *pattern matching* on enums is **not exhaustive**

→ See `Warning FS0104: Enums may take values outside known cases ...`

```
type ColorN =  
    | Red    = 1  
    | Green  = 2  
    | Blue   = 3  
  
let toHex color =  
    match color with  
    | ColorN.Red    → "#FF0000"  
    | ColorN.Green  → "#00FF00"  
    | ColorN.Blue   → "#0000FF"  
    | _ → invalidArg (nameof color) $"Color {color} not supported" // ➡
```

# Enum: flags

Same principle as in C#:

```
open System

[<Flags>]
type PermissionFlags =
    | Read      = 1
    | Write     = 2
    | Execute   = 4

let permission = PermissionFlags.Read ||| PermissionFlags.Write

let canRead = permission.HasFlag PermissionFlags.Read
```

💡 Note the `|||` operator called "binary OR" (same as `|` in C#)



# Enum flags: binary notation

💡 **Hint:** use binary notation for flag values:

```
[<Flags>]  
type PermissionFlags =  
    | Read      = 0b001  
    | Write     = 0b010  
    | Execute   = 0b100
```

# Enum: values

`System.Enum.GetValues()` returns the list of members of an `enum`

⚠ Weakly typed: `Array` (non-generic array)

💡 Use a helper like:

```
let enumValues<'a> () =  
    Enum.GetValues(typeof<'a>)  
    :?> ('a array)  
    ▷ Array.toList
```

```
let allPermissions = enumValues<PermissionFlags>()  
// val allPermissions: PermissionFlags list = [Read; Write; Execute]
```

# Enum: conversion

```
let redValue = int ColorN.Red           // enum → int
let redAgain = enum<ColorN> redValue    // int → enum
let red: ColorN = enum redValue         // int → enum

// ⚠ Use LanguagePrimitives for char enum
let n: AnswerChar = LanguagePrimitives.EnumOfValue 'N' // char → enum
let y = LanguagePrimitives.EnumToValue AnswerChar.Yes // enum → char
```

# Enum vs Union

Type	Data inside	Qualification	Exhaustivity
Enum	integers	Required	✗ No
Union	any	Optional	✓ Yes

## 👉 Recommendation:

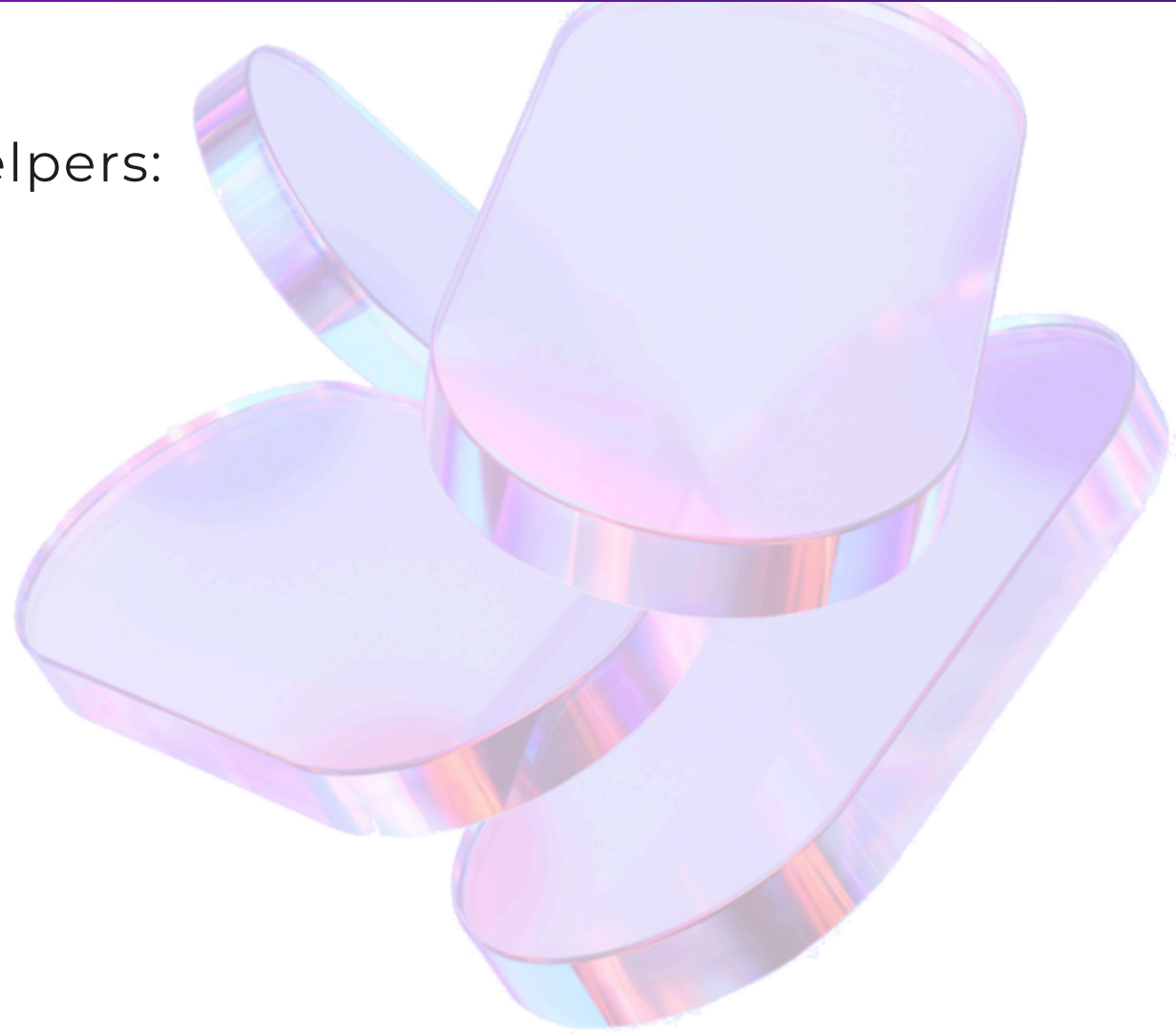
- Prefer Union over Enum in most cases
- Choose an Enum for:
  - .NET Interop
  - int data
  - Flags feature



# Enum: FSharp.Extras

💡 NuGet package [FSharp.Extras](#)  
→ Includes an `Enum` module with these helpers:

- `parse<'enum>: string → 'enum`
- `tryParse<'enum>: string → 'enum option`
- `getValues<'enum>: unit → 'enum seq`

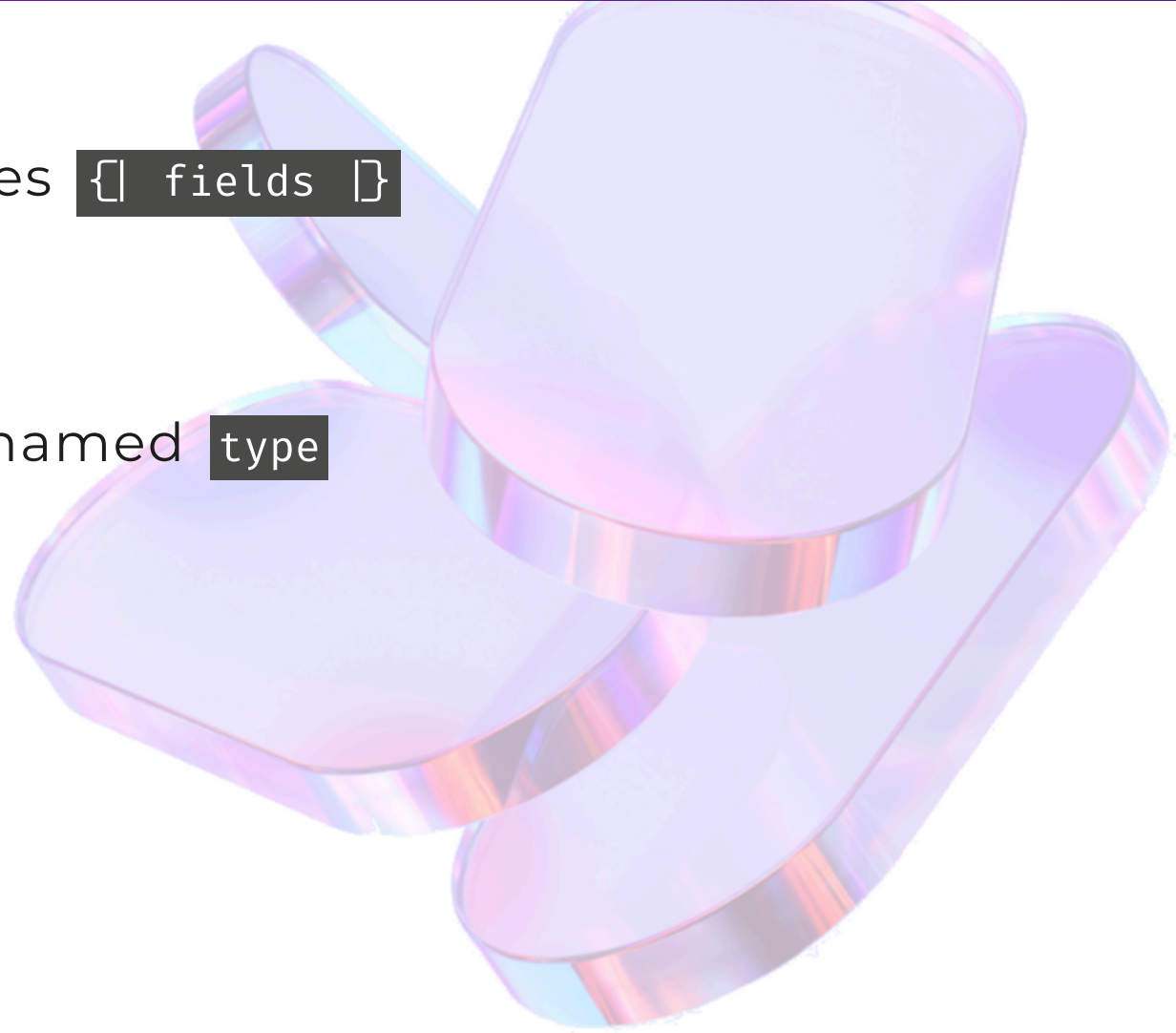


# 6. Anonymous Record



# Anonymous Record

- Since F# 4.6 (*March 2019*)
- Syntax: same as *Record* with "fat" braces `{| fields |}`
  - `{| Age: int |}` → signature
  - `{| Age = 15 |}` → instance
- Inline typing: no need to pre-define a named `type`
  - Alternative to *Tuples*
- Allowed in function input/output
  - ≠ Anonymous type C#

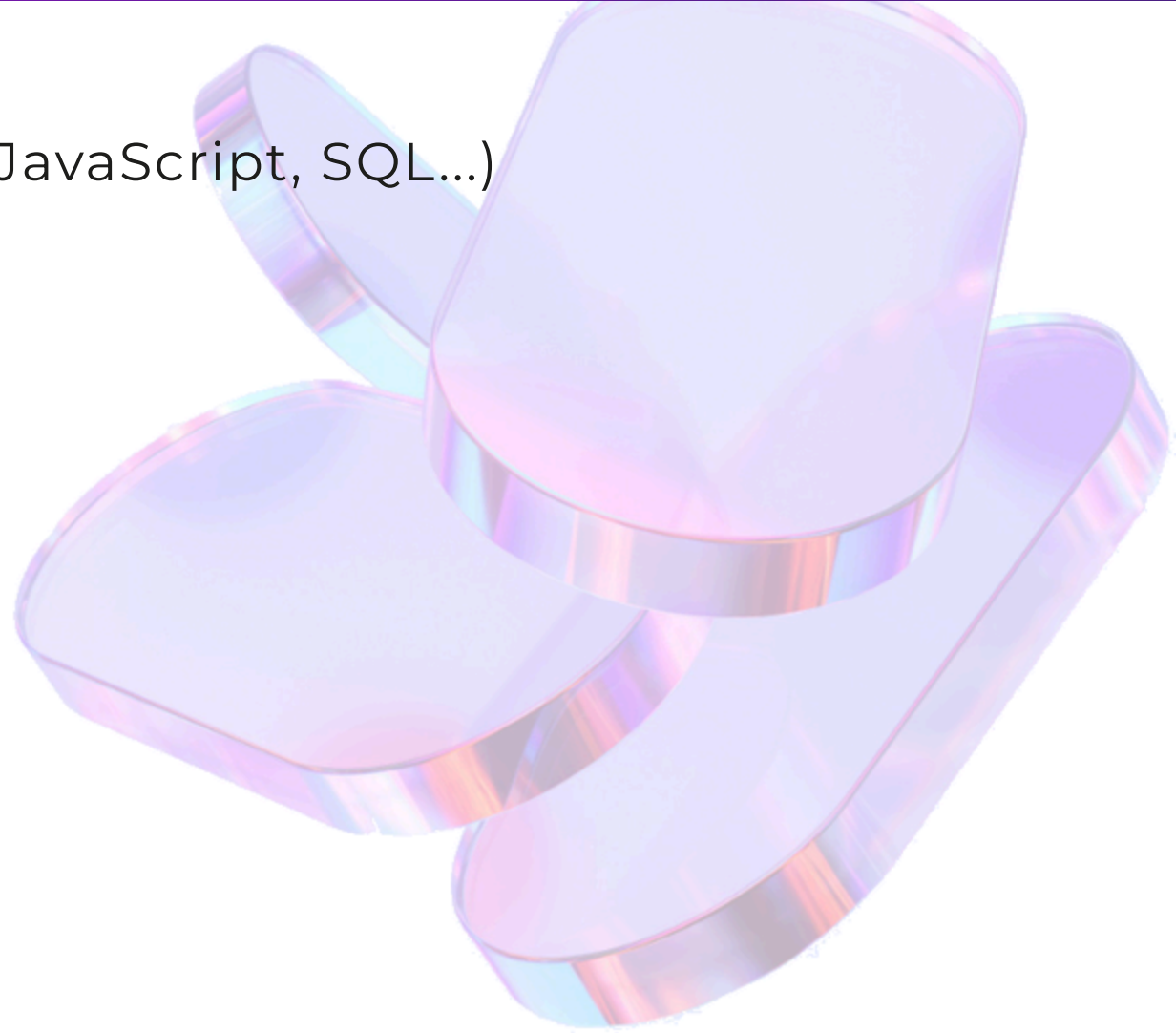


# Anonymous Record : benefits

- Reduce *boilerplate*
- Improve interop with external systems (JavaScript, SQL...)

Examples (*more on this later*) :

- LINQ projection
- Customization of an existing record
- JSON serialization
- Inline signature
- Alias by module





# ✓ LINQ Projection

💡 Select a subset of properties

```
let names =  
    query {  
        for p in persons do  
            select [| Name = p.FirstName |]  
    }
```

In C#, we would use an anonymous type:

```
var names =  
    from p in persons  
    select new { Name = p.FirstName };
```

[🔗 F# vs C#: Anonymous Records](#) by Krzysztof Kraszewski

# ✓ Customize an existing record

💡 An anonymous record can be instantiated from a record instance

```
type Person = { Age: int; Name: string }  
let william = { Age = 12; Name = "William" }  
  
// Add a field (Gender)  
let william' = [{| william with Gender = "Male" |}]  
              // [{| Age = 12; Name = "William"; Gender = "Male" |}]  
  
// Modify fields (Name, Age: int ⇒ float)  
let jack = [{| william' with Name = "Jack"; Age = 16.5 |}]  
           // [{| Age = 16.5; Name = "Jack"; Gender = "Male" |}]
```

# ✓ JSON serialization: issue

😞 Unions can be serialized in an impractical format

```
#r "nuget: Newtonsoft.Json"
let serialize obj = Newtonsoft.Json.JsonConvert.SerializeObject obj

type CustomerId = CustomerId of int
type Customer = { Id: CustomerId; Age: int; Name: string; Title: string option }

serialize { Id = CustomerId 1; Age = 23; Name = "Abc"; Title = Some "Mr" }
```

```
{
  "Id": { "Case": "CustomerId", "Fields": [ 1 ] }, // ☹️
  "Age": 23,
  "Name": "Abc",
  "Title": { "Case": "Some", "Fields": [ "Mr" ] } // ☹️
}
```

# ✅ JSON serialization: solution

💡 Define an anonymous record to serialize a *customer*

```
let serializable customer =  
    let (CustomerId customerId) = customer.Id  
    [ customer with  
        Id = customerId  
        Title = customer.Title ▷ Option.toObj ]  
  
serialize (serializable { Id = CustomerId 1; Age = 23; Name = "Abc"; Title = Some "Mr" })
```

```
{  
  "Id": 1, // ✅  
  "Age": 23,  
  "Name": "Abc",  
  "Title": "Mr" // ✅  
}
```

## ✓ Signature *inline*

💡 Use an anonymous *inline* record to reduce cognitive load

```
type Title = Mr | Mrs
type Customer =
{ Age : int
  Name : { First: string; Middle: string option; Last: string } // ➔
  Title: Title option }
```

# Anonymous Record: Limits

```
// No inference from field usage
let nameKo x = x.Name // ✨ Error FS0072: Lookup on object of indeterminate type ...
let nameOk (x: { | Name:string | }) = x.Name

// No deconstruction
let x = { | Age = 42 | }
let { | Age = age | } = x // ✨ Error FS0039: The record label 'Age' is not defined
let { | Age = age | } = x // ✨ Error FS0010: Unexpected symbol '{ | ' in let binding

// No full objects merge
let banana = { | Fruit = "Banana" | }
let yellow = { | Color = "Yellow" | }
let banYelKo = { | banana with yellow | } // ✨ Error FS0609 ...
let banYelOk = { | banana with Color = "Yellow" | }

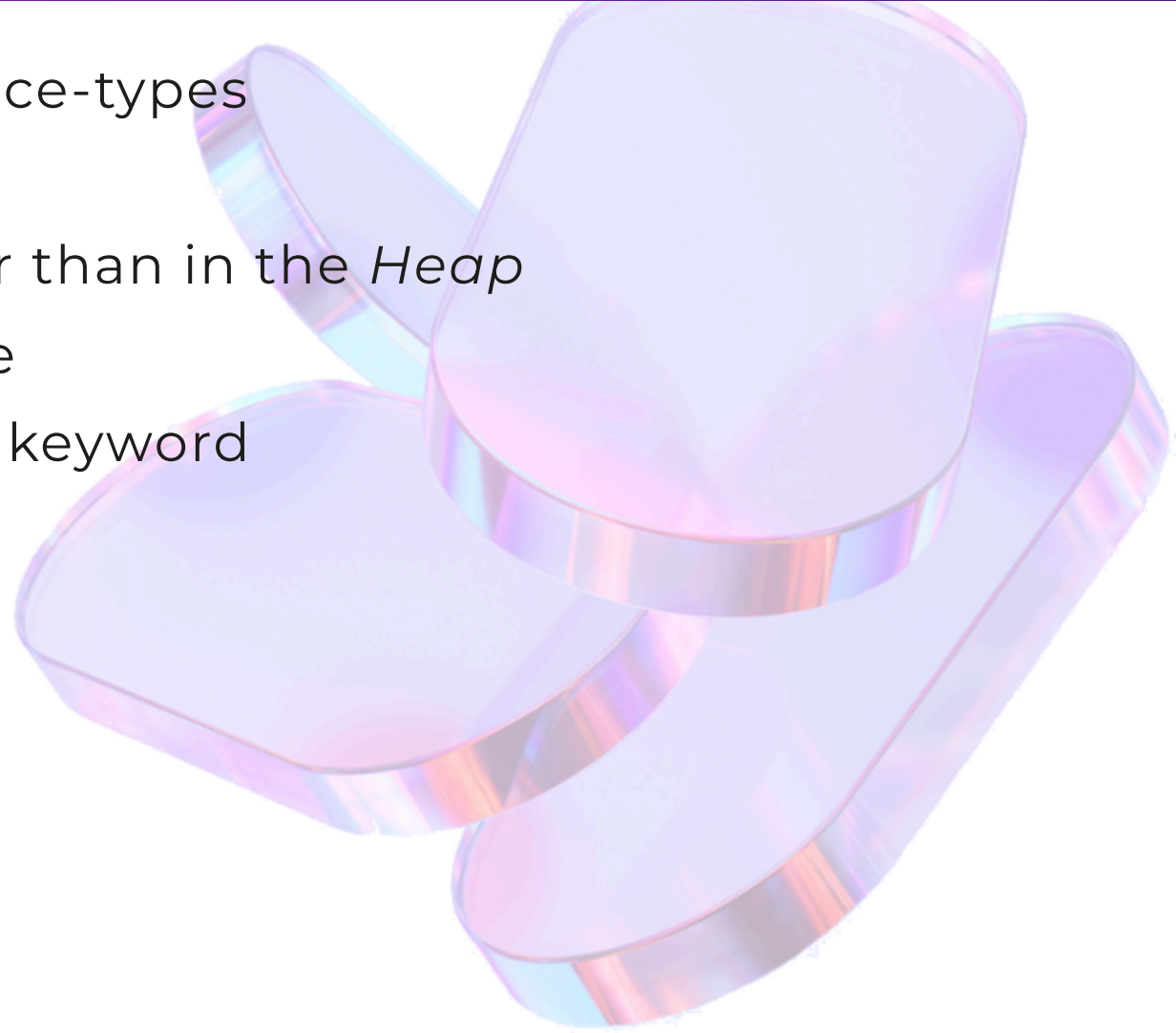
// No omissions
let ko = { | banYelOk without Color | } // ✨ No 'without' keyword
```

# 7. Value Types



# Struct F# Types

- Regular tuple/record/union are reference-types
- Possible to get them as value-types
  - Instances stored on the *Stack* rather than in the *Heap*
  - Records, Unions: `[<Struct>]` attribute
  - Tuples, Anonymous Records: `struct` keyword

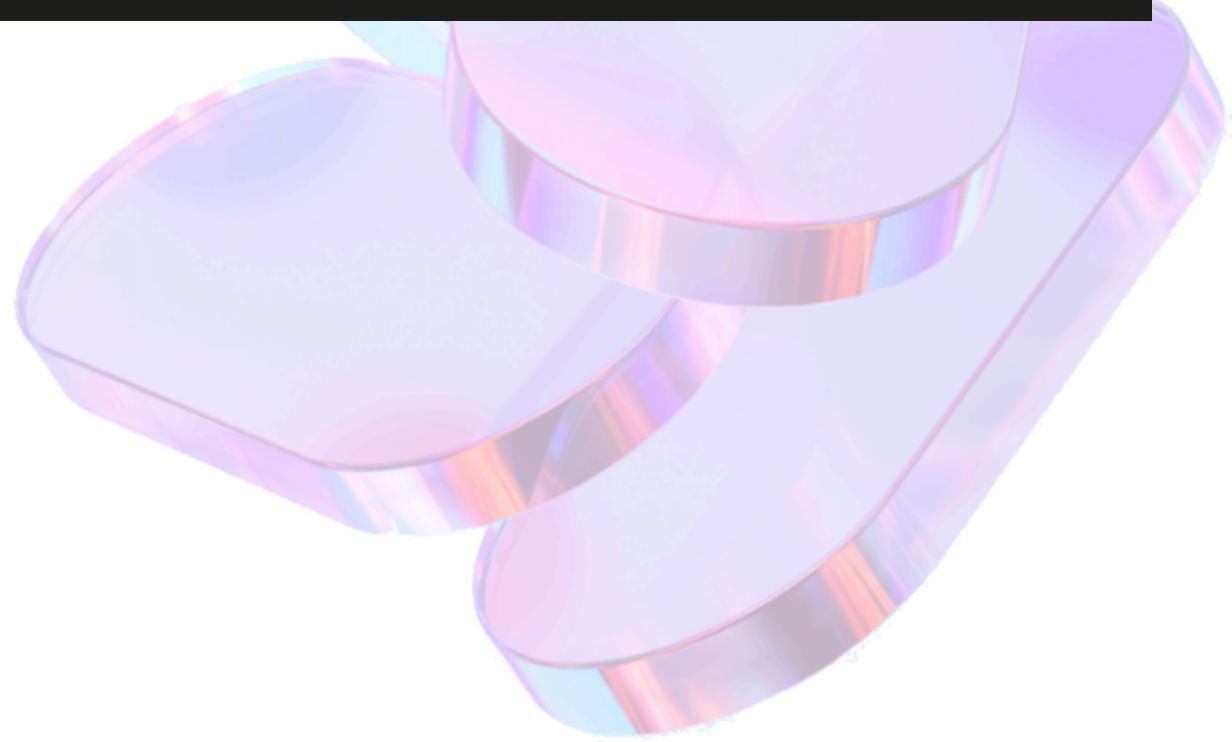




# Struct tuples & anonymous records

```
// Struct tuple
let a = struct (1, 'b', "Three") // struct (int * char * string)

// Struct anonymous record
let b = struct { Num = 1; Char = 'b'; Text = "Three" }
```



# Struct records & unions

```
// Struct record
[<Struct>]
type Point = { X: float; Y: float }
let p = { X = 1.0; Y = 2.3 } // val p: Point = { X = 1.0; Y = 2.3 }

// Struct union: unique fields labels are required !
[<Struct>]
type Multicase =
    | Int of i: int
    | Char of c: char
    | Text of s: string
let t = Int 1 // val t: Multicase = Int 1
```

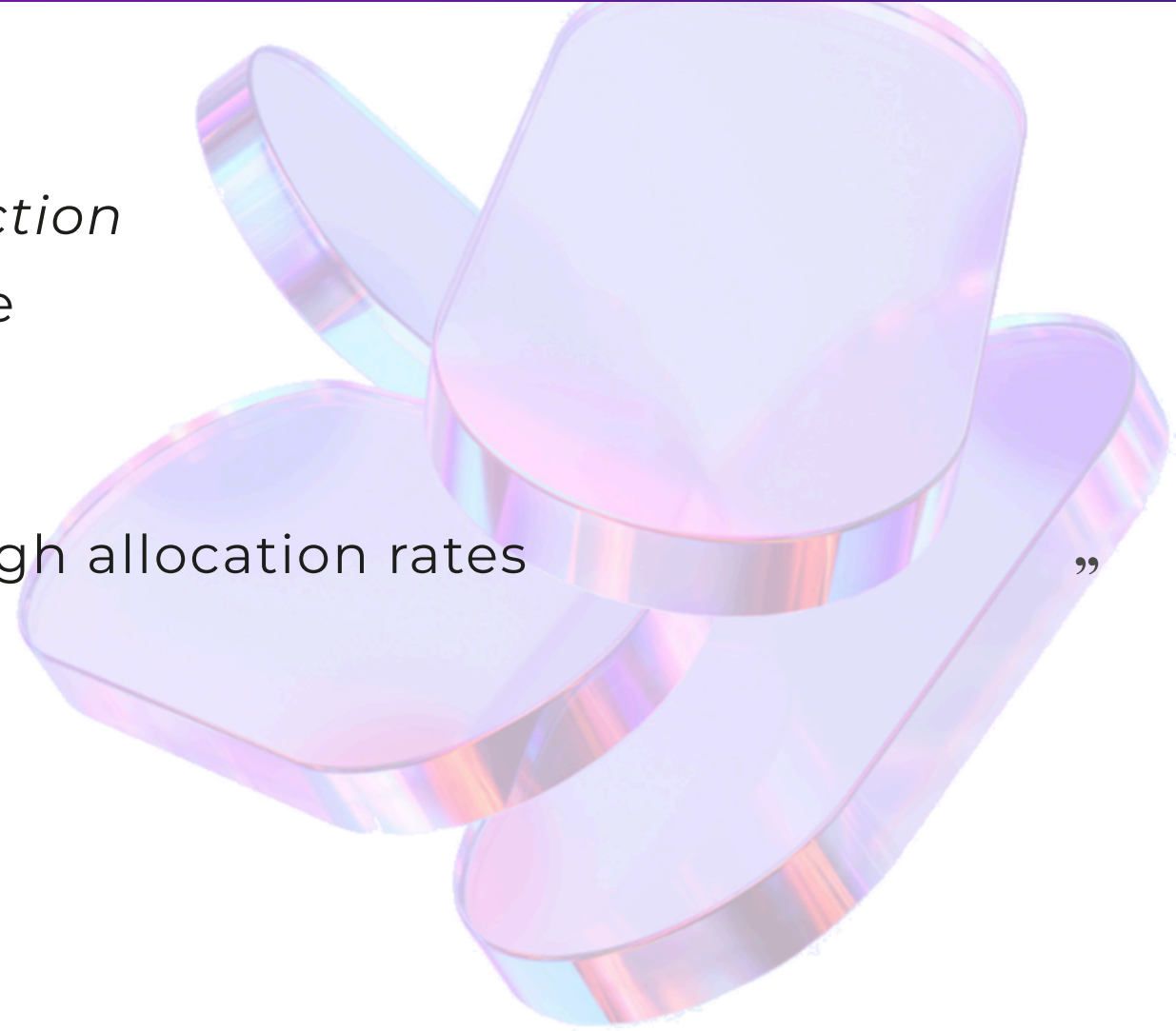
# Struct: recommendations

## ⚖️ Pros/Cons:

- ✅ Efficient because no *garbage collection*
- ⚠️ Passed by value → memory pressure

## 🔗 [F# coding conventions / Performance](#)

“ Consider structs for small types with high allocation rates ”



# 8. Wrap up 🧰





# Quiz wrap up

```
// Match types with concepts (1 to many)
type Color1 = int * int * int
type Color2 = Red | Green | Blue
type Color3 = Red=1 | Green=2 | Blue=3
type Color4 = { Red: int; Green: int; Blue: int }
type Color5 = [ | Red: int; Green: int; Blue: int | ]
type Color6 = Color of Red: int * Green: int * Blue: int
type Color7 =
  | RGB of [ | Red: int; Green: int; Blue: int | ]
  | HSL of [ | Hue: int; Saturation: int; Lightness: int | ]

// A. Alias
// B. Enum
// C. Record
// D. Record anonym
// E. Single-case union
// F. Union
// G. Union enum-like
// H. Tuple
```





# Quiz wrap up

Types	Concepts
<code>type Color1 = int * int * int</code>	<b>H.</b> Tuple + <b>A.</b> Alias
<code>type Color2 = Red   Green   Blue</code>	<b>G.</b> Union enum-like
<code>type Color3 = Red=1   Green=2   Blue=3</code>	<b>B.</b> Enum
<code>type Color4 = { Red: int; Green: int... }</code>	<b>C.</b> Record
<code>type Color5 = {   Red: int; Green: int...   }</code>	<b>D.</b> Anonymous Record + <b>A.</b> Alias
<code>type Color6 = Color of Red: int * ...</code>	<b>E.</b> Single-case union + <b>H.</b> Tuple
<code>type Color7 = RGB of {  ...  }   HSL of {  ...  }</code>	<b>F.</b> Union + <b>D.</b> Anonymous Record

# Types Composition

Creating new types?

- ❌ Algebraic data types do not support inheritance
- ✅ By composition, in *sum/product* types
- 💡 Extension of a *Record* into an anonymous *Record* with more fields

Combine 2 unions?

- ❌ Not "flattenable" as in TypeScript ①
- ✅ New parent union type ②

```
// French-suited cards
type Noir = Pique | Trefle // spades ♠ | clubs ♣
type Rouge = Coeur | Carreau // hearts ♥ | diamonds ♦
type CouleurKo = Noir | Rouge // (1) ⚠ not the expected union of Pique | Trefle | Coeur | Carreau
type Couleur = Noir of Noir | Rouge of Rouge // (2) ✅
let c1 = Noir Pique // Couleur
```

# Conclusion

Lots of ways to model!

💡 Opportunity for:

- Team discussions
- Business domain encoding in types





Thanks 🙏

