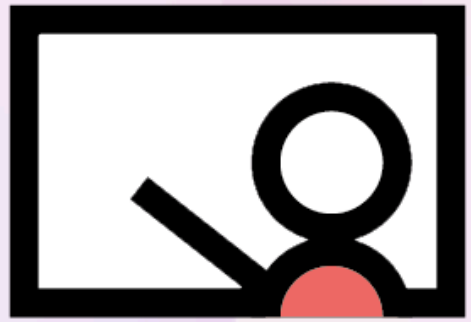
F# Training

Module & namespace

2025 April

d-edge

Table of contents

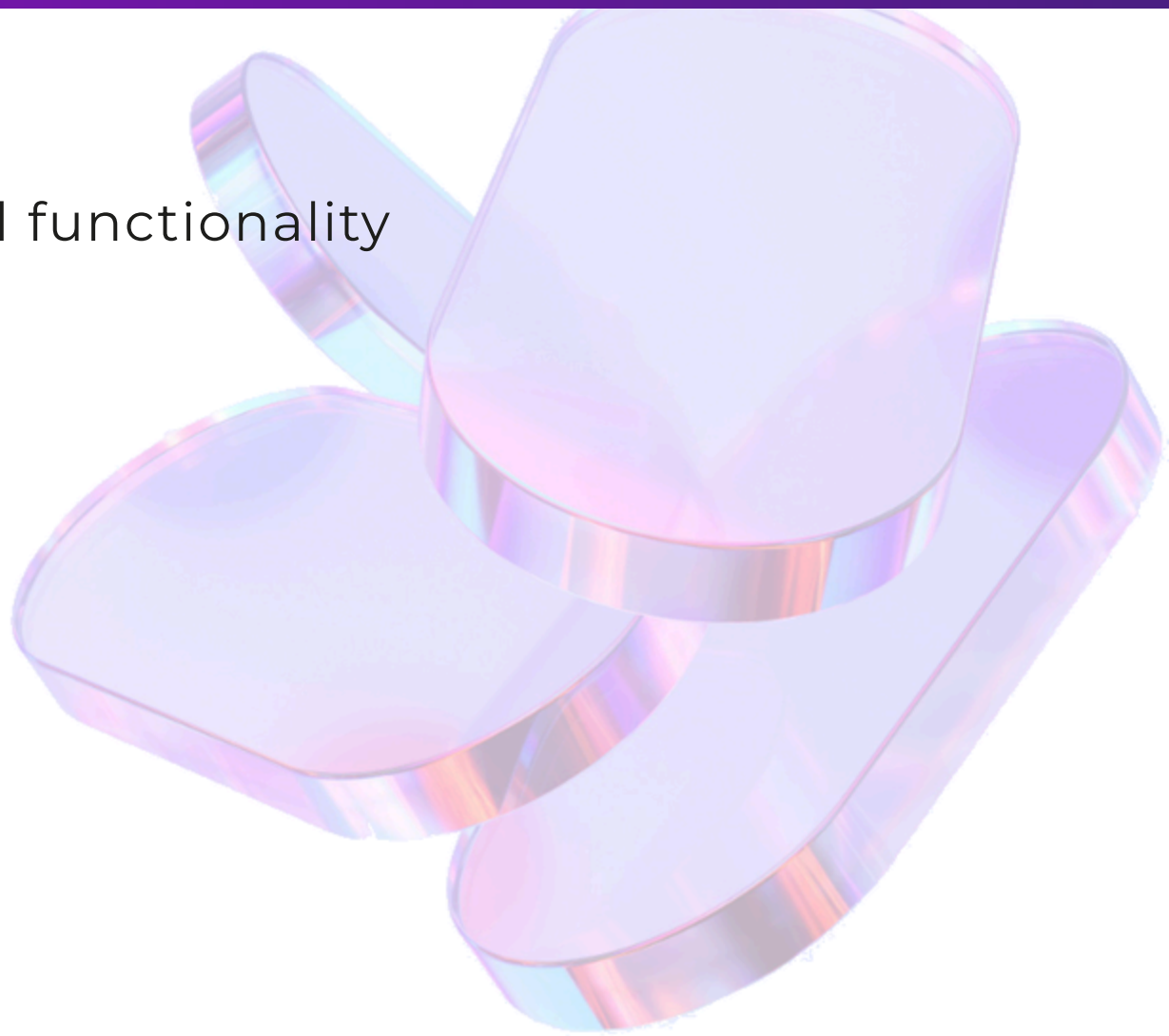


1. Overview

Similarities

Modules and namespaces allow you to:

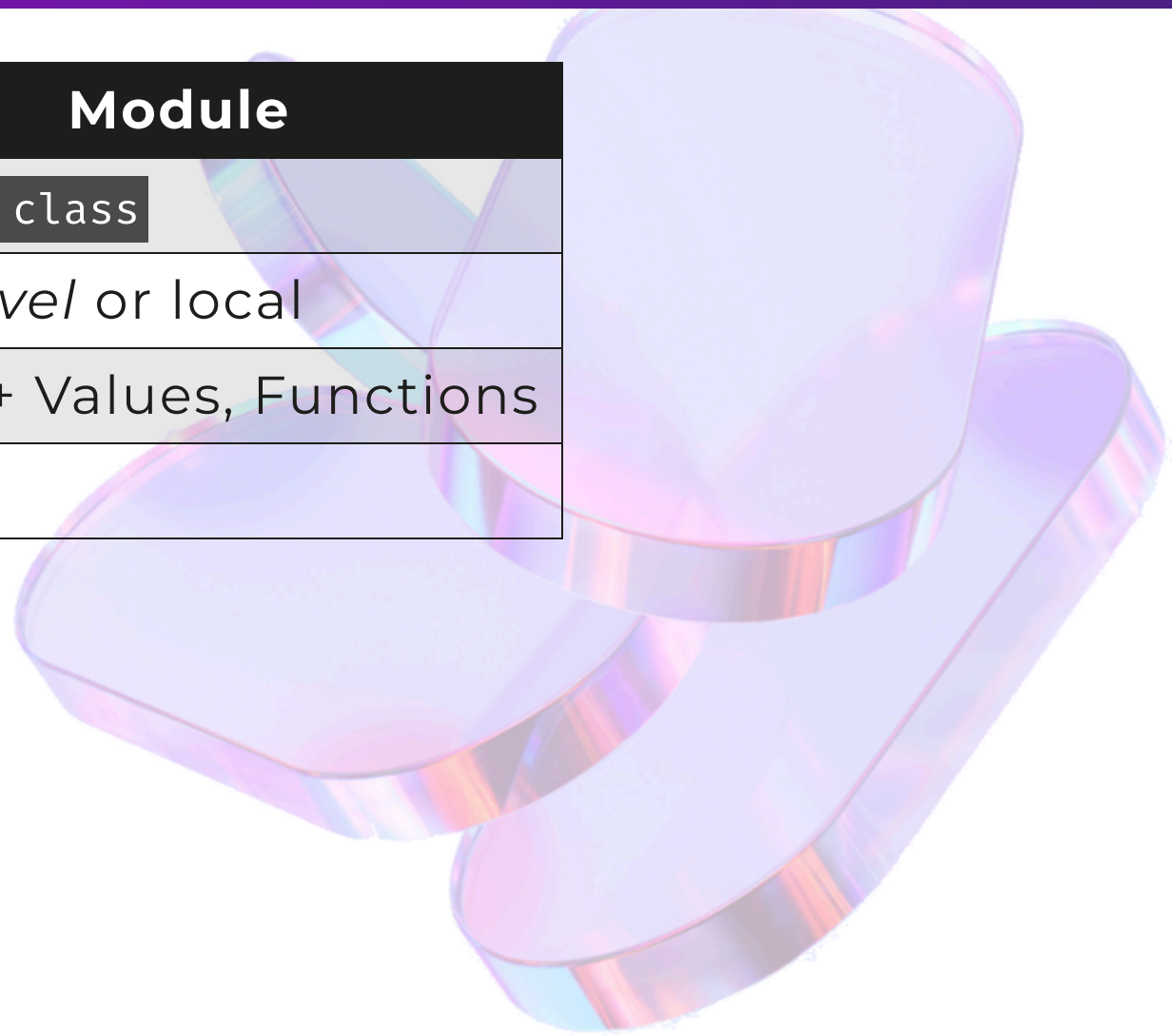
- ➔ **Organize code** into zones of related functionality
- ➔ Avoid name collisions



Differences

Property	Namespace	Module
.NET equivalent	`namespace`	`static class`
Type	*Top-level*	*Top-level* or local
Contains	Modules, Types	Idem + Values, Functions
Annotable	❌ No	✅ Yes

Scope: Namespaces > Files > Modules



Use a module or a namespace

1. Either qualify the elements individually to be imported
2. Or import everything with `open`
   - ➔ placed anywhere before the usage, at the top recommended
   - ➔ `open Name.Space` ≡ C♯ `using Name.Space`
   - ➔ `open My.Module` ≡ C♯ `using static My.Module`

```
// Option 1: Qualify usages
let result1 = Arithmetic.add 5 9

// Option 2: Import the entire module
open Arithmetic
let result2 = add 5 9
```

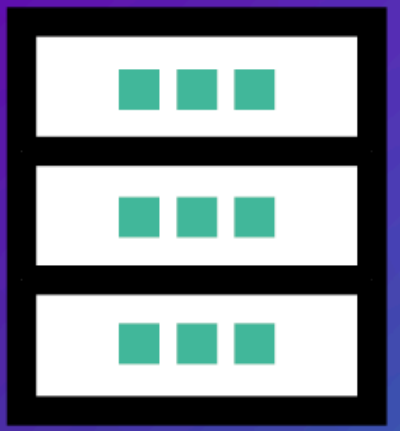
Import : *shadowing*

Imports are done without name conflicts but need **disambiguation:**

- Modules and static classes are merged ✅
- Types and functions are shadowed ❗
  → Last-imported-wins mode: see example below
  → Import order matters

```fsharp
module IntHelper =
    let add x y = x + y

module FloatHelper =
    let add x y : float = x + y

open IntHelper
open FloatHelper

// Error because function `add` called is that of module `FloatHelper`!
let result = add 1 2 // 💥 Error FS0001: The type 'float' does not match the type 'int'
```

2. *Namespaces*

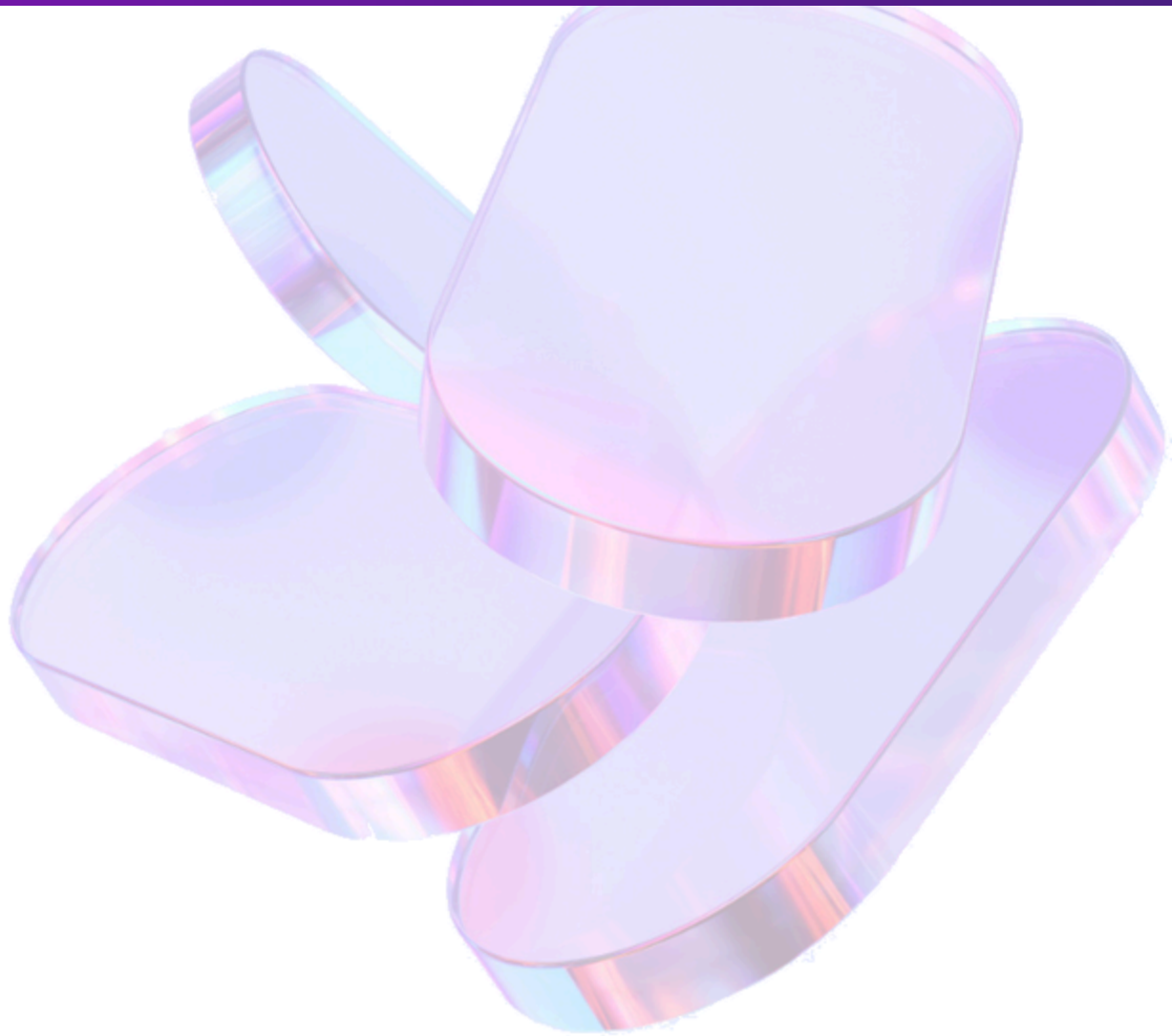


Namespace: syntax

`namespace [rec] [parent.]identifier`

→ `rec` for recursive → see *slide next*

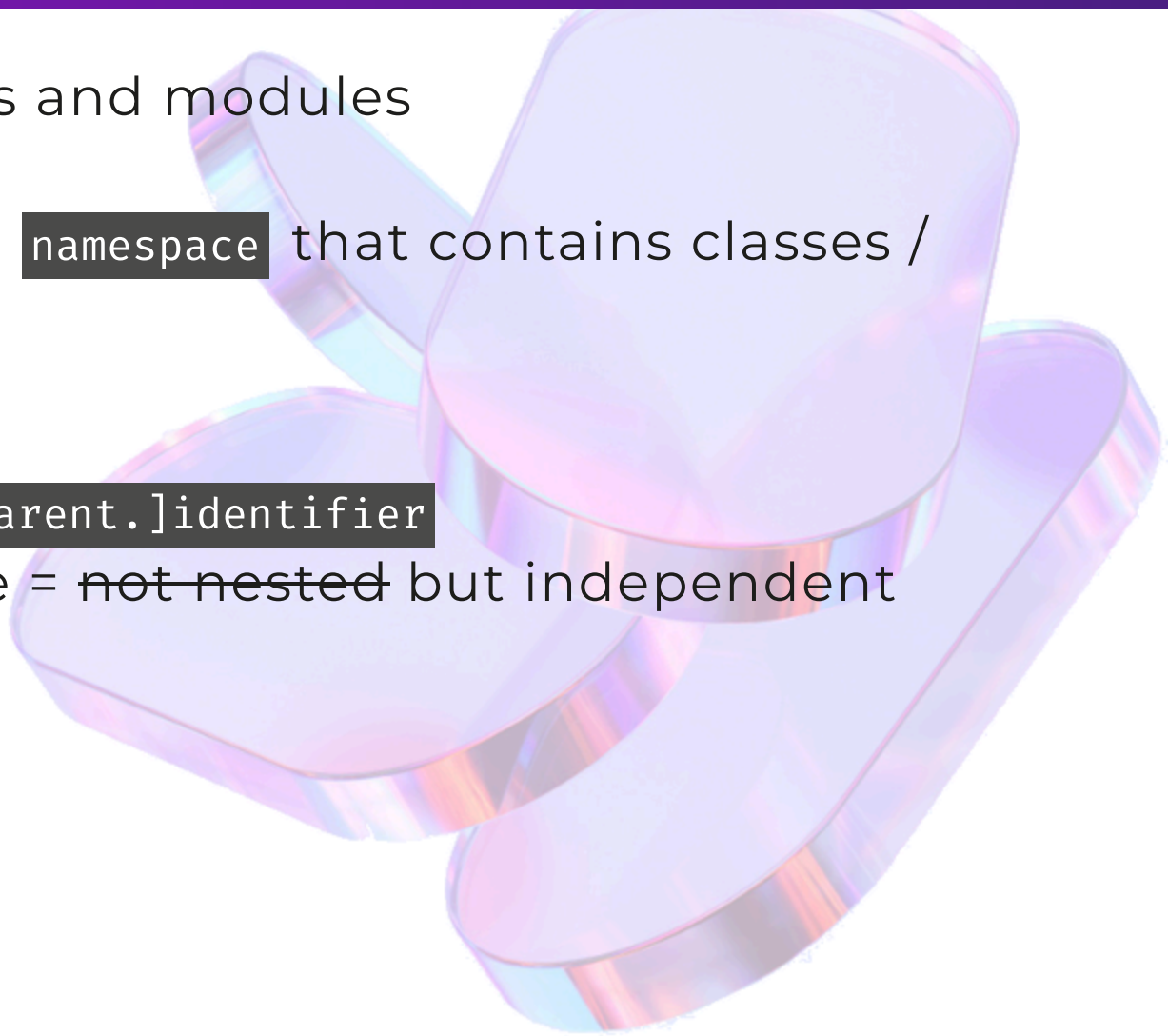
→ `parent` for grouping namespaces



Namespace: content

A `namespace` F♯ can only contain local types and modules
→ Cannot contain values or functions ❗
→ By comparison, it's the same in C♯ with `namespace` that contains classes / enums only

What about nested namespaces?
→ Only happens declaratively `namespace [parent.]identifier`
→ 2 namespaces declared in the same file = ~~not nested~~ but independent



Namespace: scope

- ➔ Several files can share the same namespace
- ➔ Several namespaces can be declared in a single file
  - ➔ They will not be ~~nested~~
  - ➔ May cause confusion ❗

☝ **Recommendation**
→ **Only** one namespace per file, declared at the top

Namespace nesting is declarative
→ `A.B.C` ⊂ `A.B` ⊂ `A`
→ Through 1 to 3 namespace declarations

Namespace: recursive

Extends the default unidirectional visibility: from bottom to top
→ each element can see all the elements in a recursive namespace

```
namespace rec Fruit

type Banana = { Peeled: bool }
    member this.Peel() =
        BananaHelper.peel  // `peel` not visible here without the `rec`

module BananaHelper =
    let peel banana = { banana with Peeled = true }
```

⚠️ **Drawbacks:** slow compilation, risk of circular reference
☝️ **Recommendation:** handy but only for very few use cases

3. Modules

Module: syntax

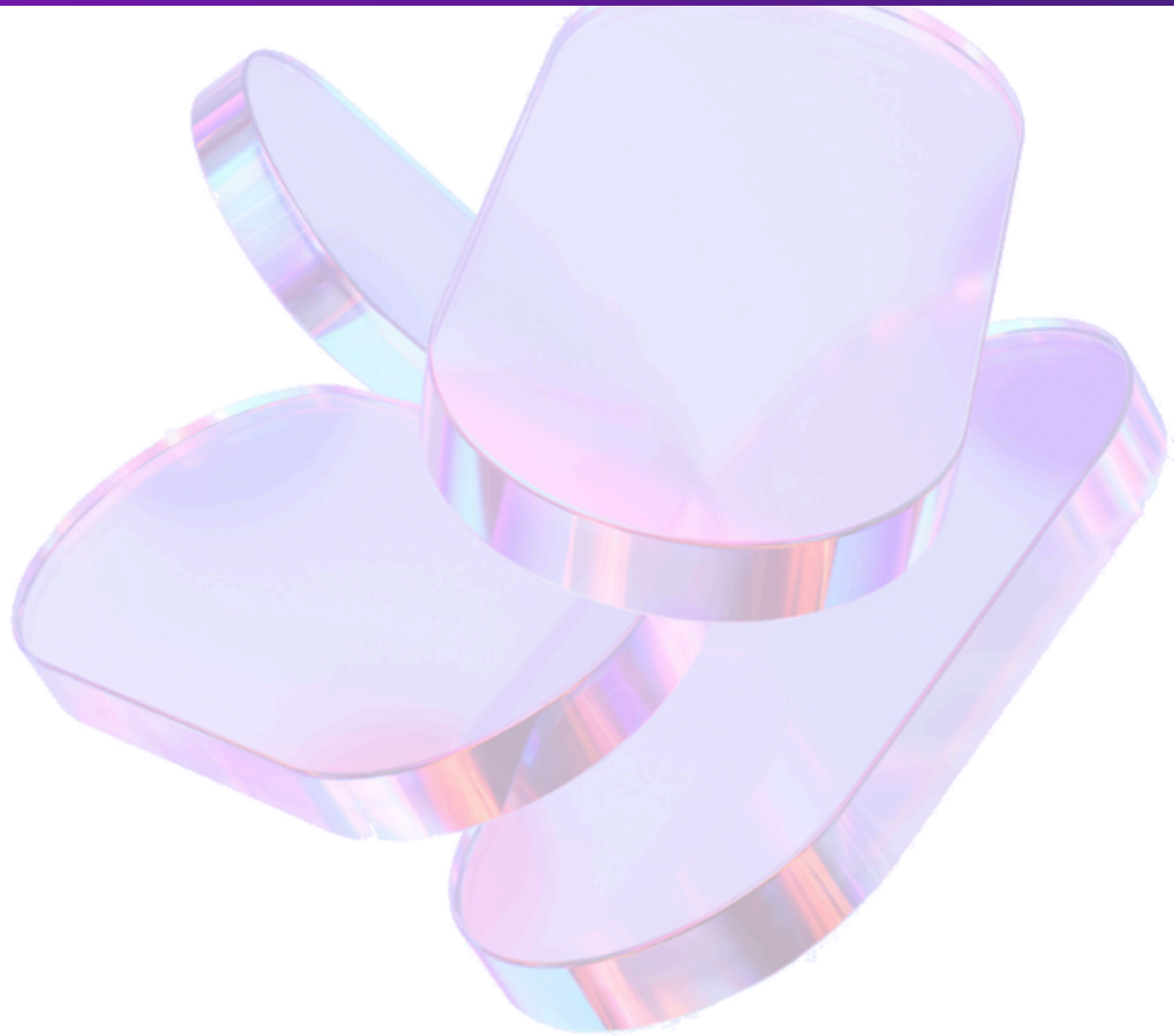
```
// Top-level module
module [accessibility-modifier] [qualified-namespace.]module-name
declarations

// Local module
module [accessibility-modifier] module-name =
    declarations
```

`accessibility-modifier`: restrict accessibility
→ `public` *(default)*, `internal` *(assembly)*, `private` *(parent)*

Full name (`[namespace.]module-name`) must be unique
→ 2 files cannot declare modules with the same name

Module kind

- Top-level module
  - Implicit top-level module
- Local module



Top-level module

- Only one top-level module per file
  → Declared on very top of the file

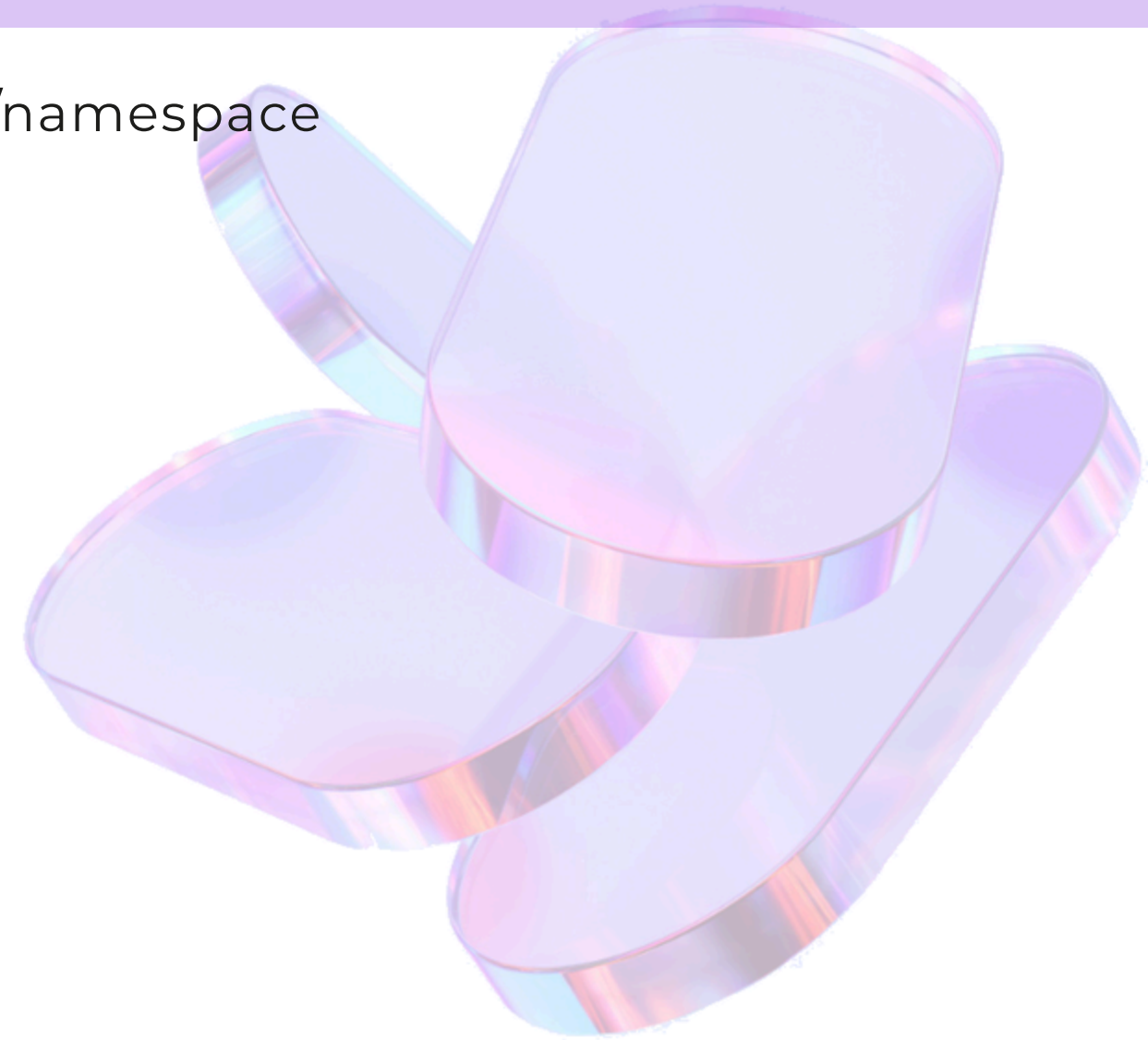
- Can (should?) be qualified
  → Attached to a parent namespace *(already declared or not)*

- Contains all the rest of the file
  → Unindented content 👍

Implicit top-level module

- ➔ For a file without top-level module/namespace
- ➔ Module name = file name
  - ➔ Without extension
  - ➔ With 1st letter in uppercase
  - ➔ E.g.: `program.fs` → `module Program`

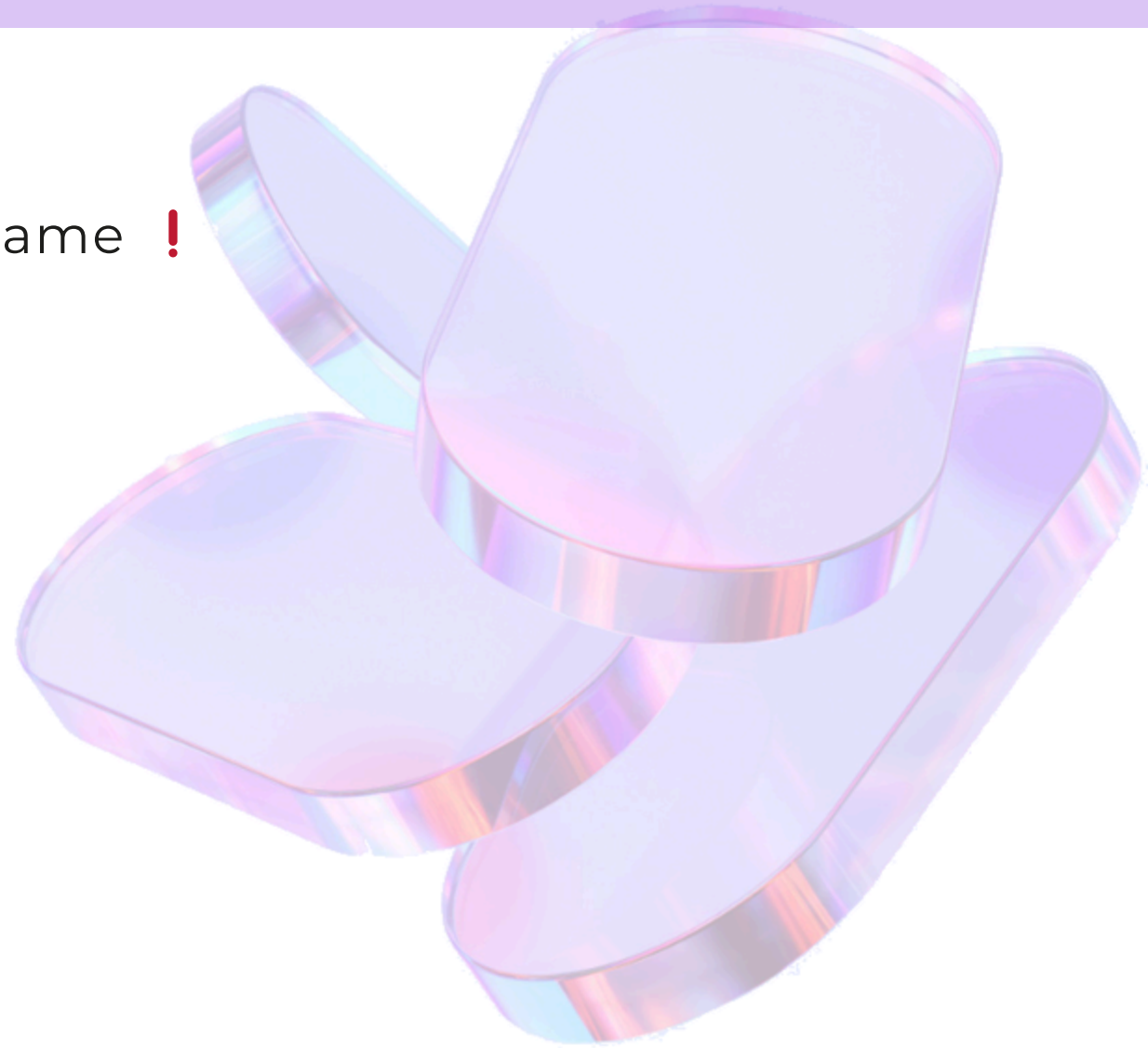
☝ Not recommended in `.fsproj`



Local module

Syntax similar to `let`

- ➔ The `=` sign after the local module name !
- ➔ Indent the entire content

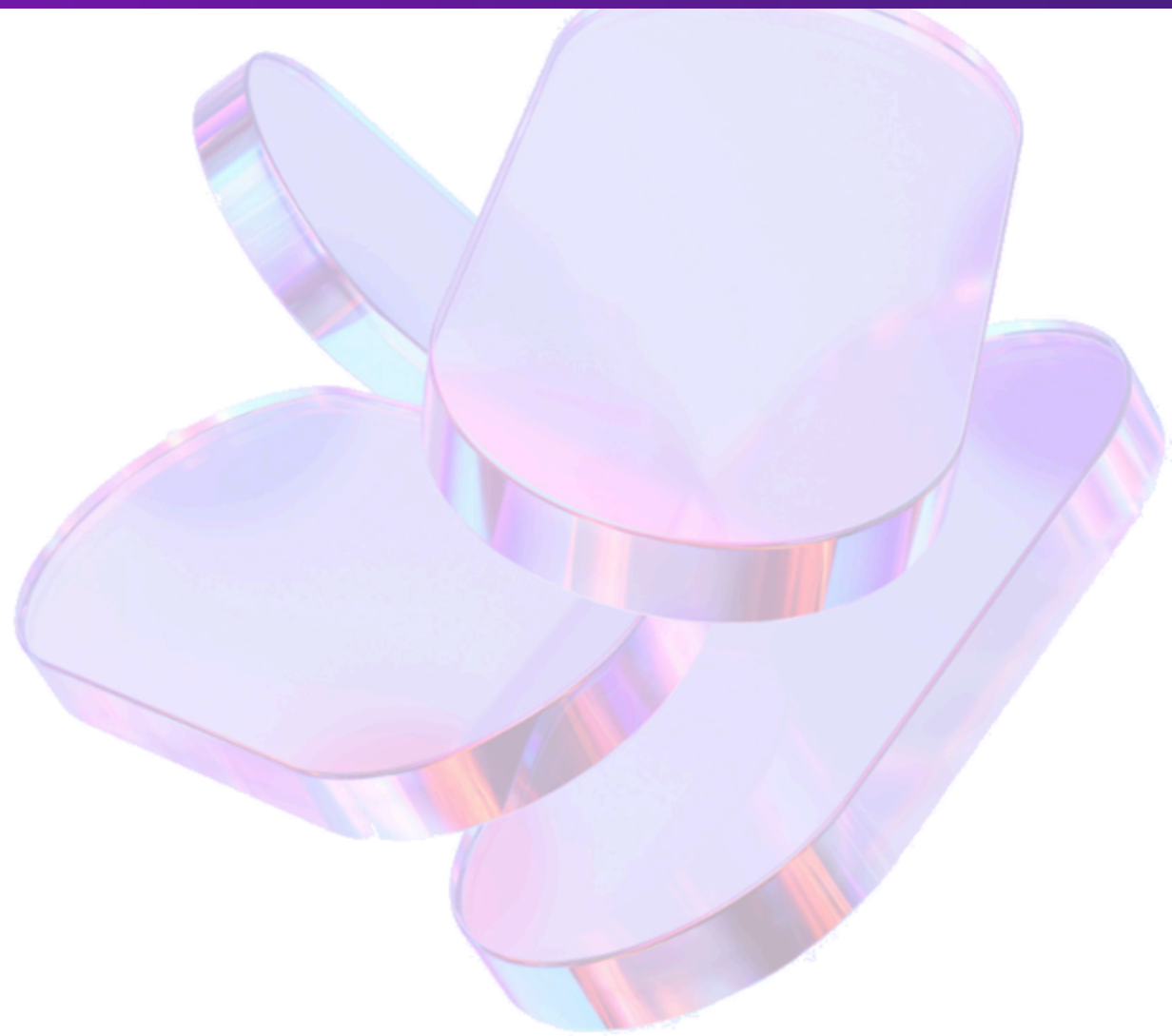


Module: content

A module, *local as top-level*, can contain:
→ local types and sub-modules
→ values, functions

Key difference:
→ content indentation

Module	Indentation
top-level	No
local	Yes



Module/static class equivalence

```
module MathStuff =
    let add x y  = x + y
    let subtract x y = x - y
```

This F# module is equivalent to the following static class in C#:

```
public static class MathStuff
{
    public static int add(int x, int y) ⇒ x + y;
    public static int subtract(int x, int y) ⇒ x - y;
}
```

See sharplab.io

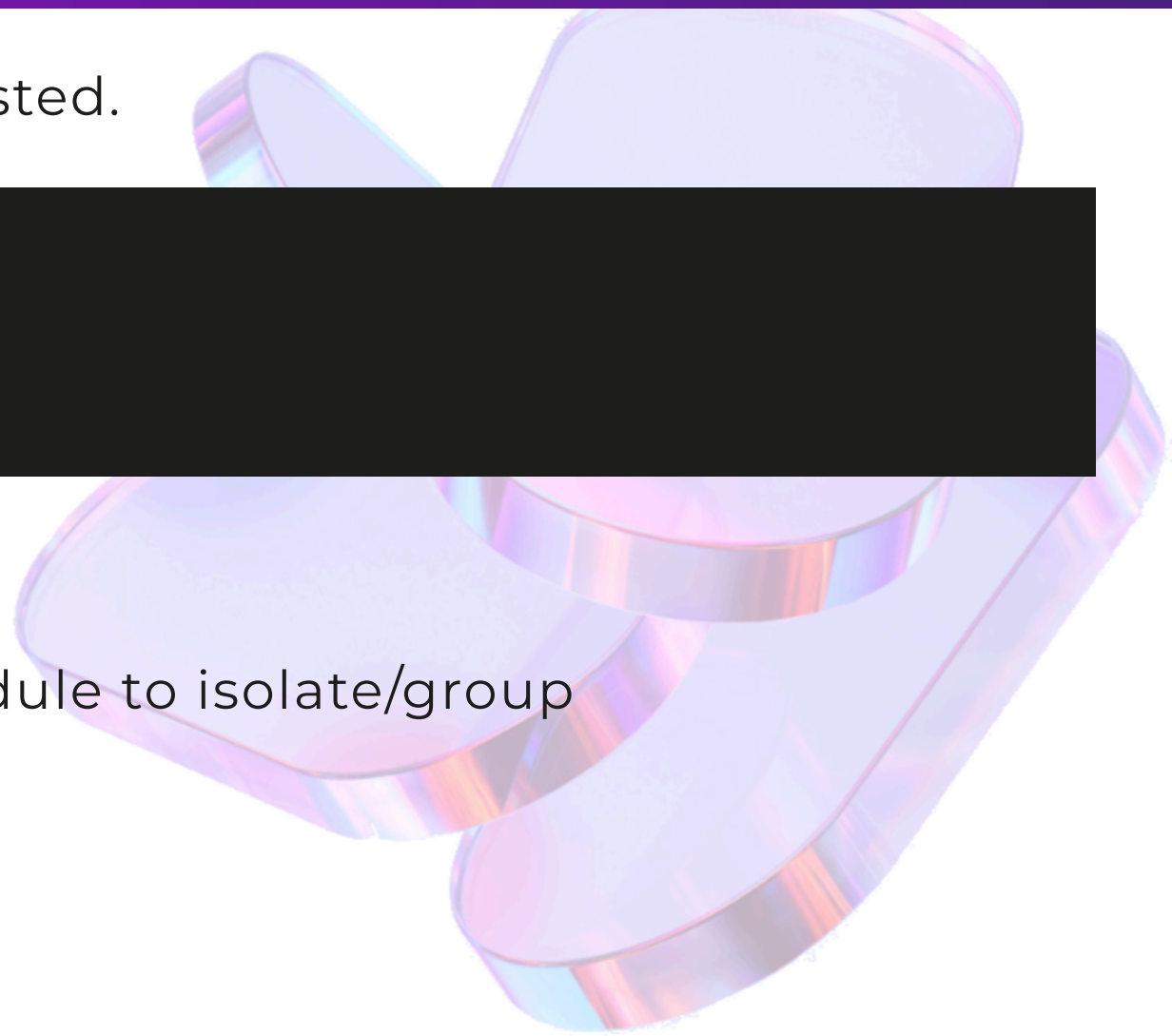
Module nesting

As with C♯ classes, F♯ modules can be nested.

```
module Y =
    module Z =
        let z = 5

printfn "%A" Y.Z.z
```

☝ **Notes :**

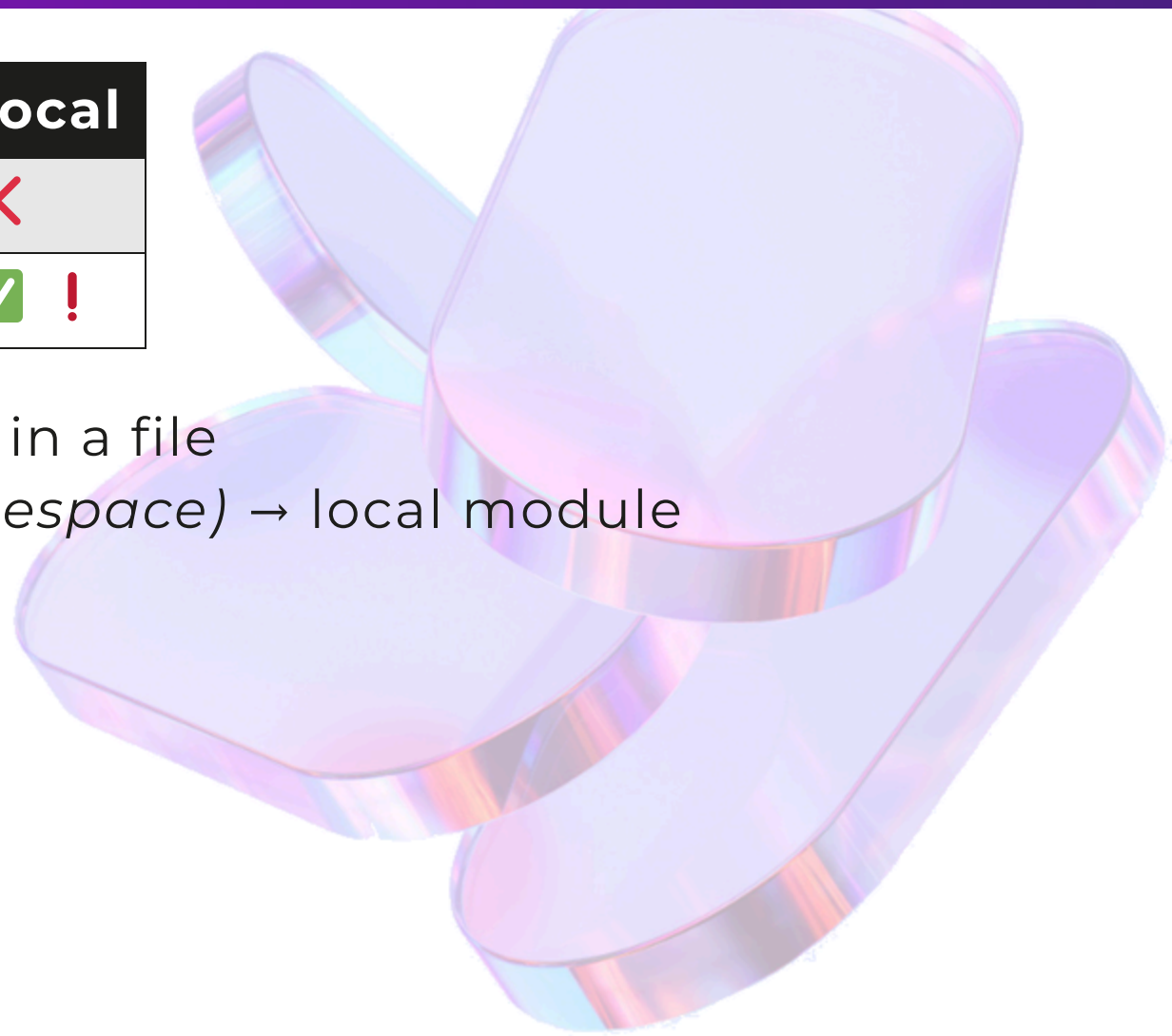
- ➔ Interesting with private nested module to isolate/group
- ➔ Otherwise, prefer a *flat view*
- ➔ F♯ classes cannot be nested



Top-level *vs* local module

Property	Top-level	Local
Qualifiable	✅	❌
`=` sign + indented content	❌	✅ !

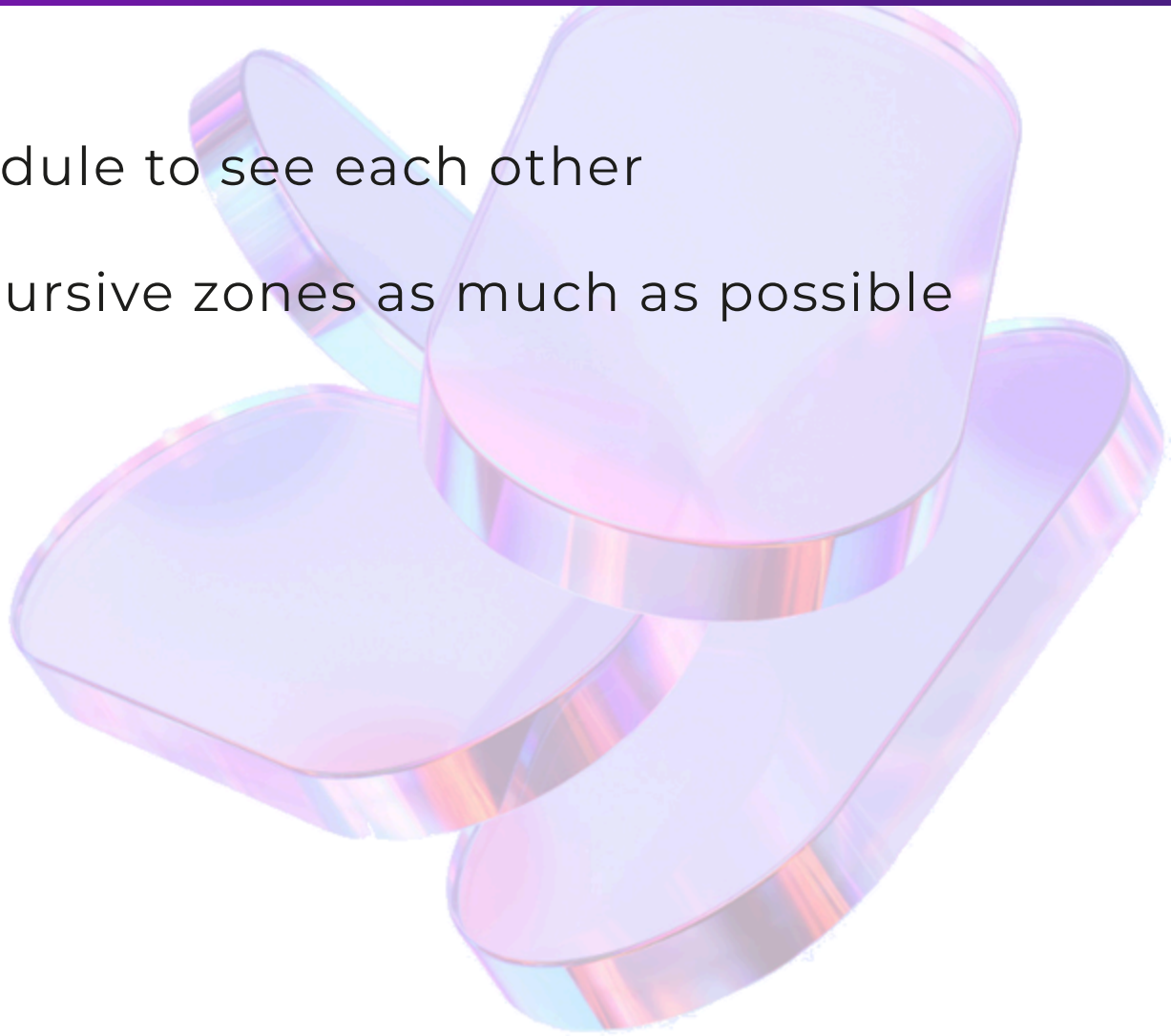
Top-level module → 1st element declared in a file
Otherwise *(after a top-level module/namespace)* → local module



Recursive module

Same principle as recursive namespace
→ Convenient for a type and a related module to see each other

☝️ **Recommendation:** limit the size of recursive zones as much as possible



Module annotation

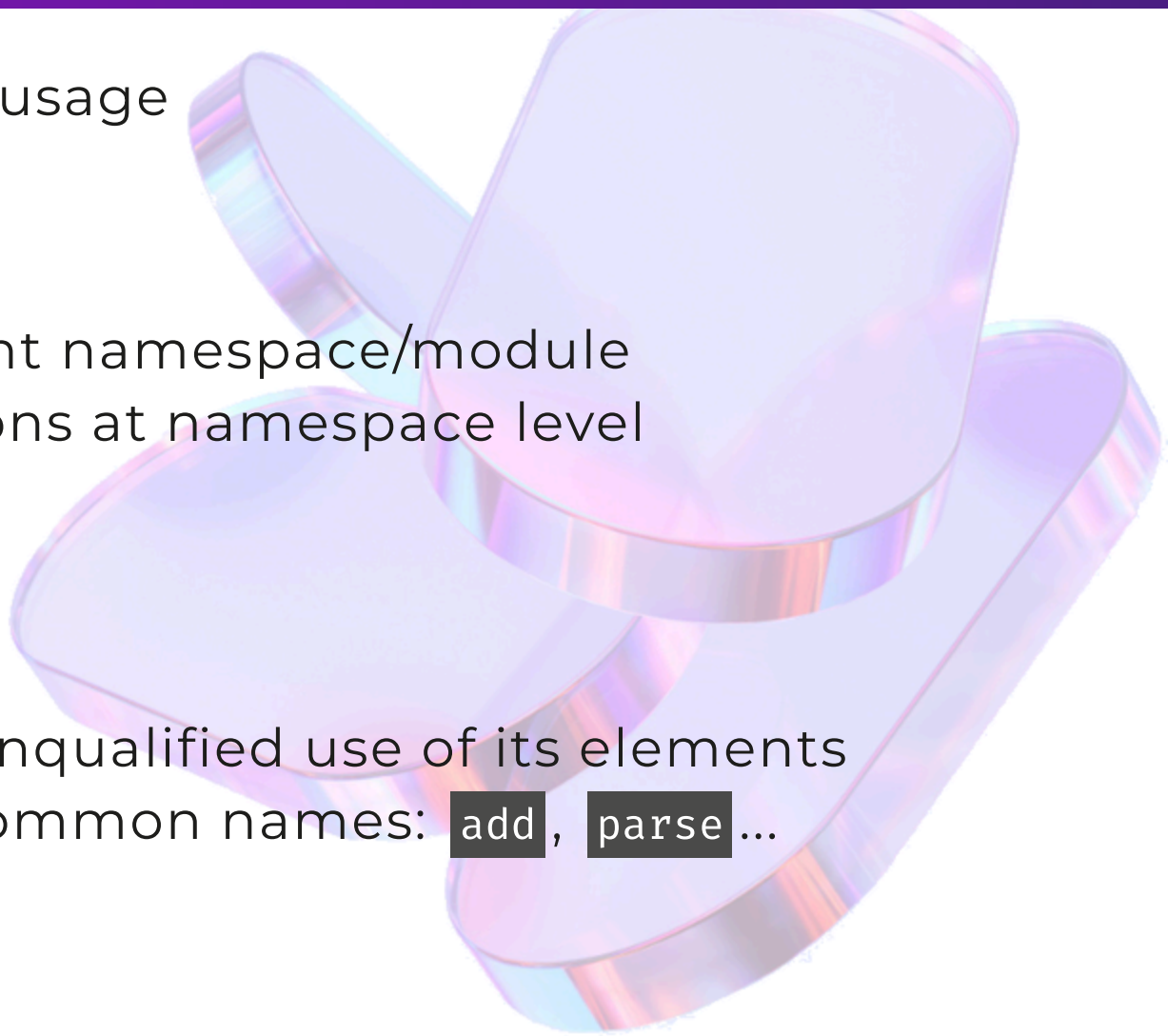
2 opposite attributes impact the module usage

`[<AutoOpen>]`

Import module at same time as the parent namespace/module
→ 💡 Handy for "mounting" values/functions at namespace level
→ ⚠️ Pollutes the current *scope*

`[<RequireQualifiedAccess>]`

Prevents the module import hence any unqualified use of its elements
→ 💡 Useful for avoiding *shadowing* for common names: `add`, `parse`...



`AutoOpen`, `RequireQualifiedAccess` or nothing?

Let's consider a `Cart` type with its `Cart` companion module.

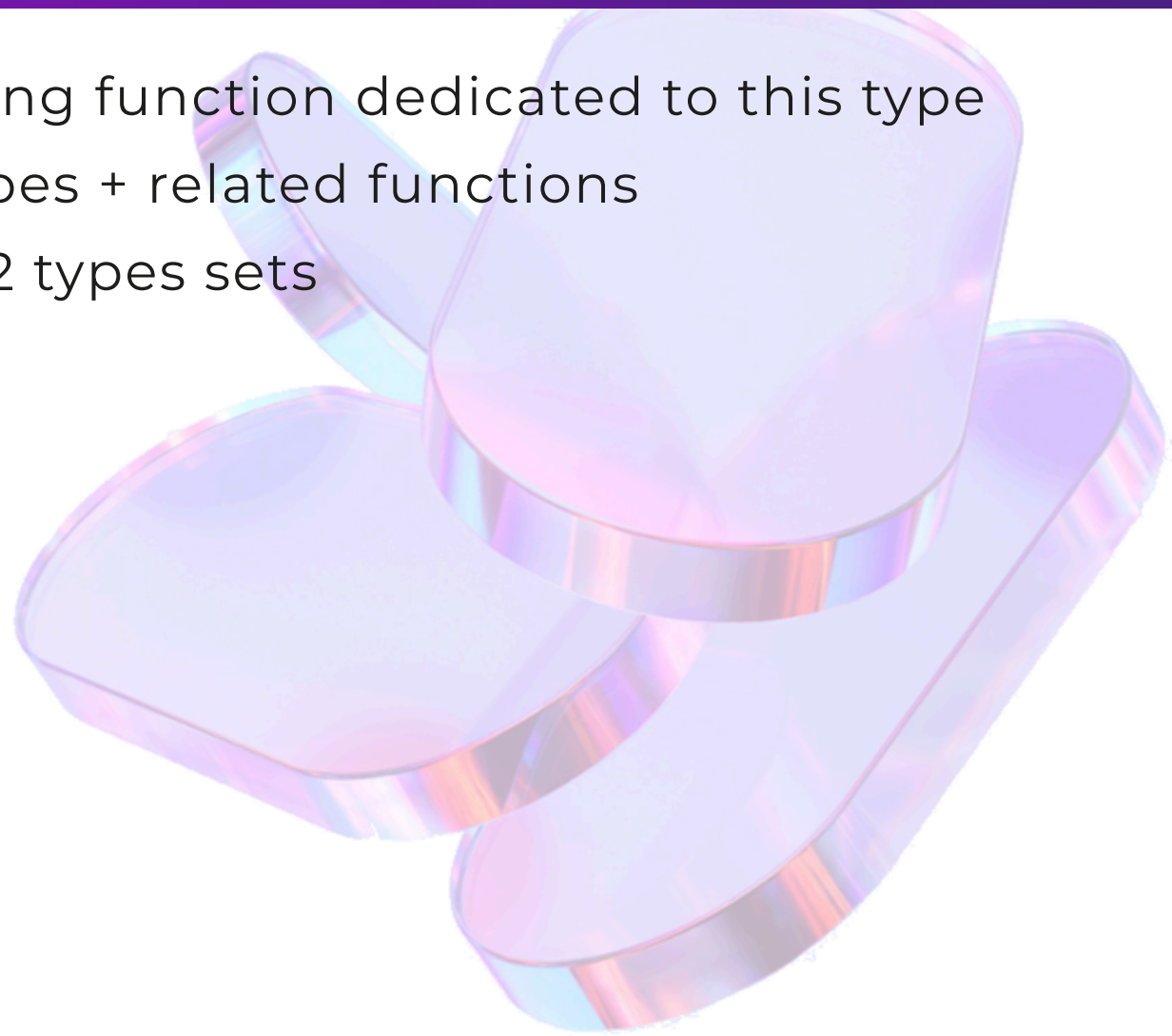
How do we call the function that adds an item to the cart?
→ It depends on the function name.

- `addItem item cart`:
  → `[<RequireQualifiedAccess>]` to consider
  → to be compelled to use `Cart.addItem`

- `addItemToCart item cart`:
  → function name is *self-explicit*
  → `[<AutoOpen>]` interesting to prevent `Cart.addItemToCart`
  → Works only if `Cart` parent *(if any)* is not `RequireQualifiedAccess` and opened

Types-Modules main typologies

- ➔ Type + Companion module containing function dedicated to this type
- ➔ Multi-type module: several small types + related functions
- ➔ Mapper modules: to map between 2 types sets



Type + Companion module

FSharp.Core style - see `List`, `Option`, `Result`...

Module can have the same name as the type
→ BCL interop: module compiled name = `{Module}Module`

```fsharp
type Person = { FirstName: string; LastName: string }

module Person =
    let fullName person = $"{person.FirstName} {person.LastName}"

let person = { FirstName = "John"; LastName = "Doe" }   // Person
person |> Person.fullName // "John Doe"
```

Multi-type module

Contains several small types + related functions *(eventually)*

```
module Common.Errors

type OperationNotAllowedError = { Operation: string; Reason: string }

type Error =
    | Bug of exn
    | OperationNotAllowed of OperationNotAllowedError


let bug exn = Bug exn ▷ Error

let operationNotAllowed operation reason =
    { Operation = operation
      Reason = reason }
    ▷ Error
```

Mapper modules

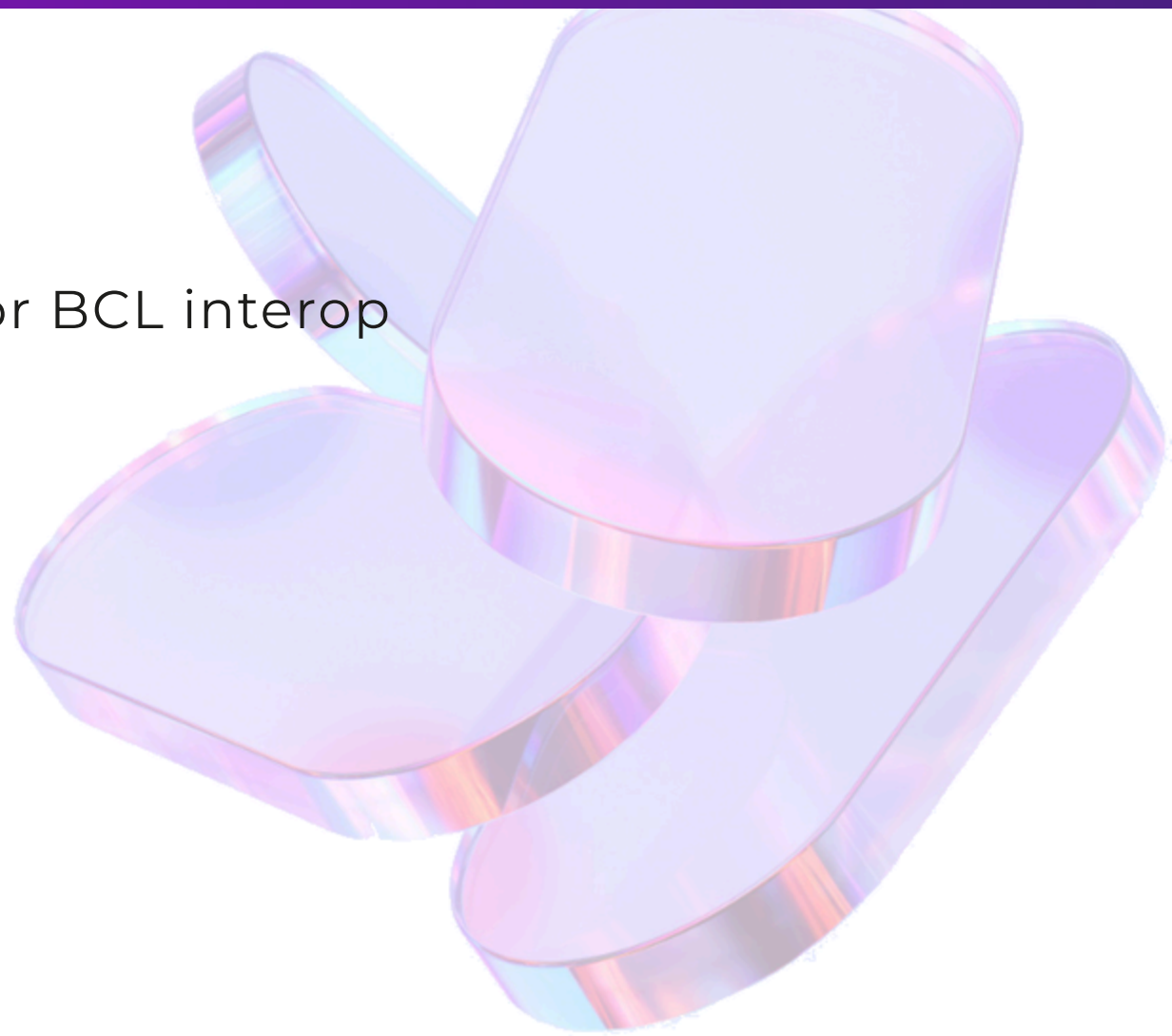
To map between 2 types sets

```
// Domain/Types/Mail.fs ---
module Domain.Types.Mail

[ types ... ]

// Data/Mail/Entities.fs ---
module Data.Mail.Entities

[ DTO types ... ]

// Data/Mail.Mappers ---
module Data.Mail.Mappers

module DomainToEntity =
    let mapXxx x : XxxDto = ...
```

Module *vs* namespace

If a file contains a single module

- ➔ Prefer top-level module in general
- ➔ Prefer namespace + local module for BCL interop



Open type *(Since F♯ 5)*

Use cases:

1. Import static classes to get direct access to methods

```
open type System.Math
let x = Max(123., 456.)
```

vs

```
open System
let x = Math.Max(123., 456.)
```

☝️ In general, use case only recommended for classes designed for this usage.

Open type - Use cases (2)

2. Cherry-pick imports

→ Import only the types needed in a module

```fsharp
// Domain/Sales.fs ---
module Domain.Sales =
    type Balance = Overdrawn of decimal | Remaining of decimal
    // Other types, functions ...

// Other/Module.fs ---
open type Sales.Balance

let myBalance = Remaining of 500. // myBalance is of type Balance.
```

4. 🍔 Quiz

Q1. Valid or not?

```
namespace A

let a = 1
```

A. Yes

B. No

Q1. Valid or not?

```
namespace A

let a = 1
```

A. ~~**Yes**~~ ❌

B. **No** ✅

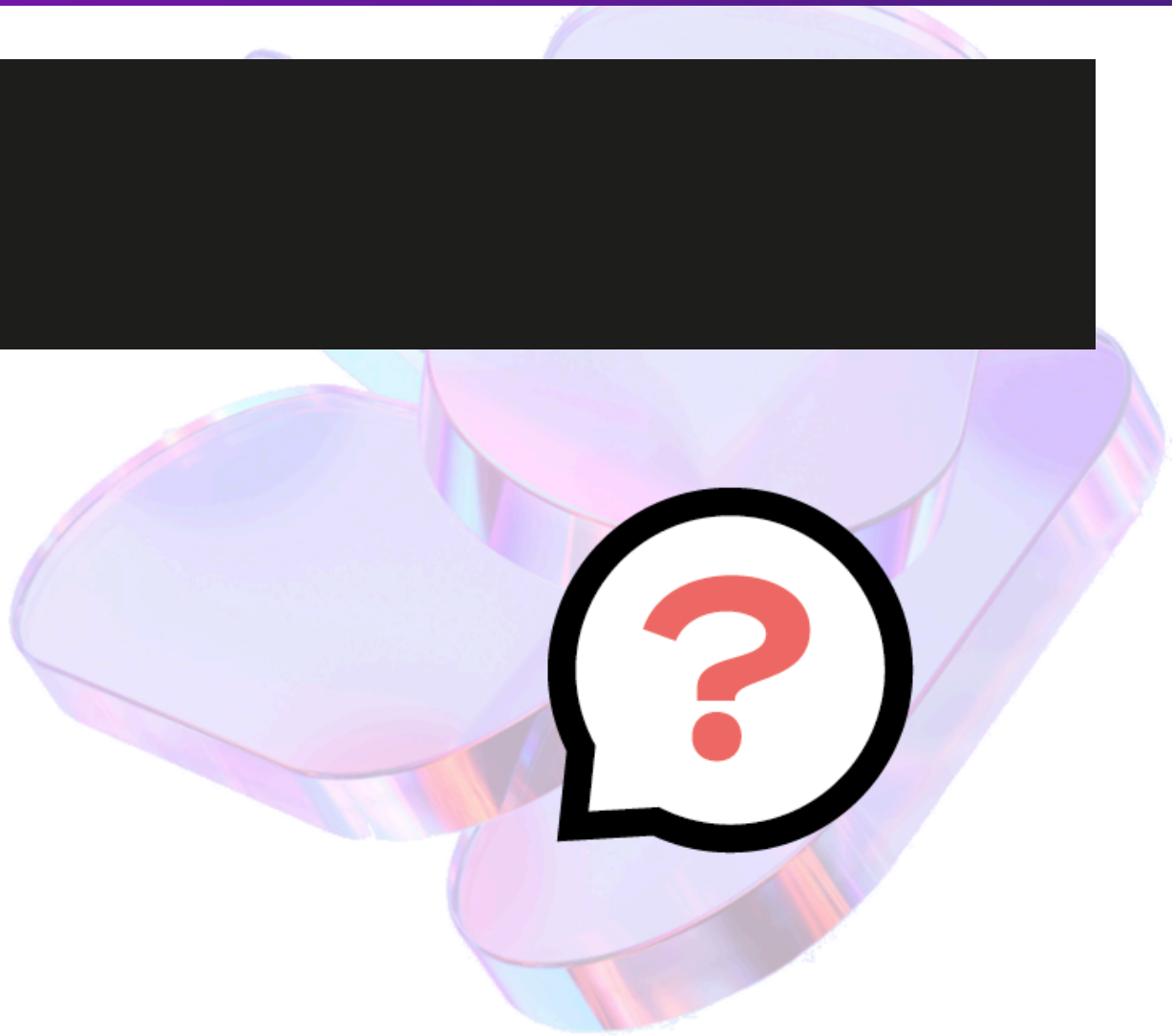
→ A namespace cannot contain values!

Q2. Valid or not?

```
namespace A

module B

let a = 1
```

A. Yes

B. No



Q2. Valid or not?

```
namespace A

module B

let a = 1
```

A. ~~**Yes**~~ ❌

B. **No** ✅

→ module B is declared as top-level
→ forbidden after a namespace

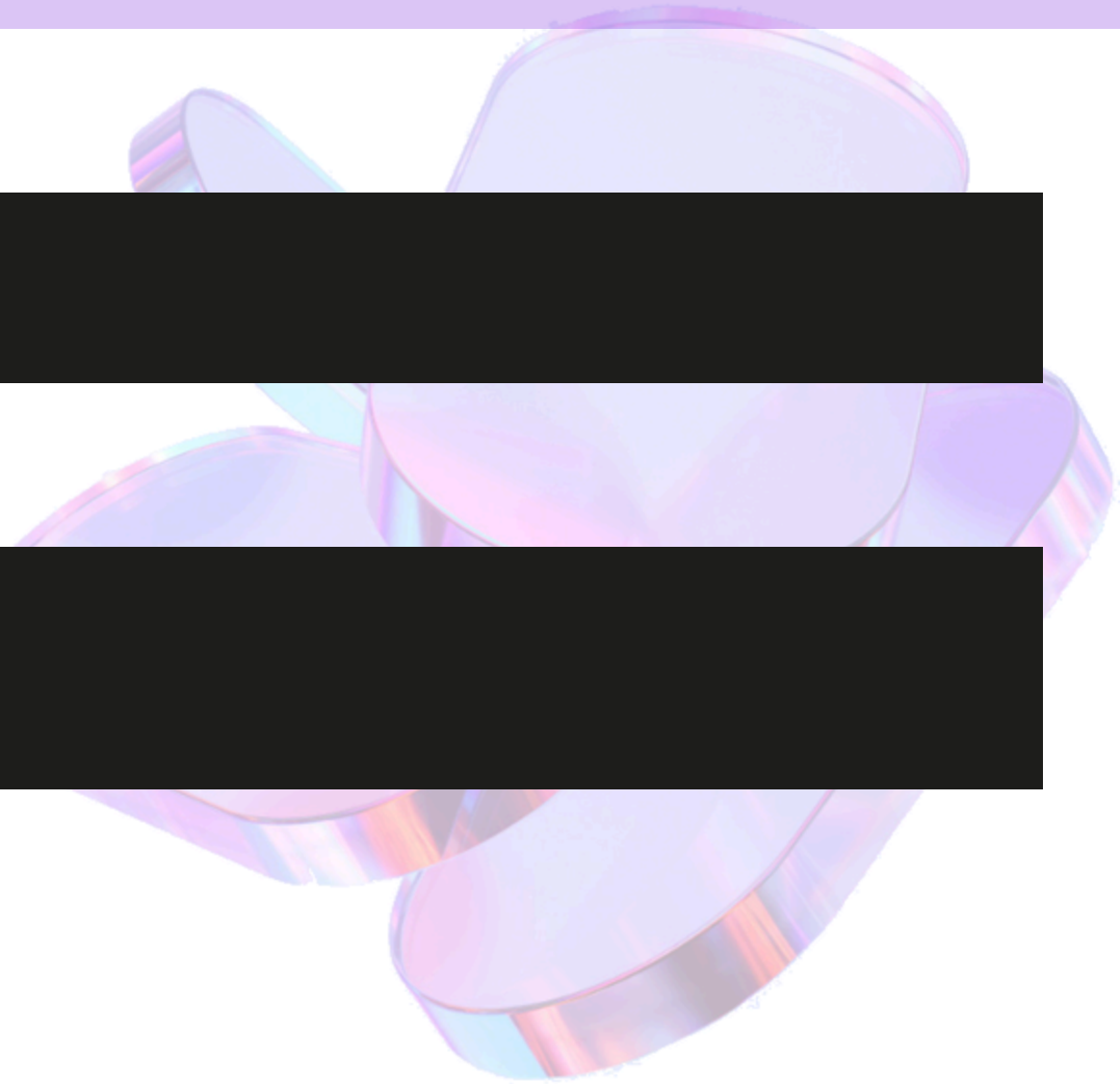
Q2 - Valid equivalent code

Option 1: top-level module

```
module A.B

let a = 1
```

Option 2: namespace + local module

```
namespace A

module B =
    let a = 1
```



Q3. Give the fully-qualified name for `add`?

```
namespace Common.Utilities

module IntHelper =
    let add x y = x + y
```

A. `add`

B. `IntHelper.add`

C. `Utilities.IntHelper.add`

D. `Common.Utilities.IntHelper.add`

Q3. Give the fully-qualified name for `add` ?

```
namespace Common.Utilities

module IntHelper =
    let add x y = x + y
```

A. **`add`** ❌

B. **`IntHelper.add`** ❌

C. **`Utilities.IntHelper.add`** ❌

D. **`Common.Utilities.IntHelper.add`** ✅

→ `IntHelper` for the parent module

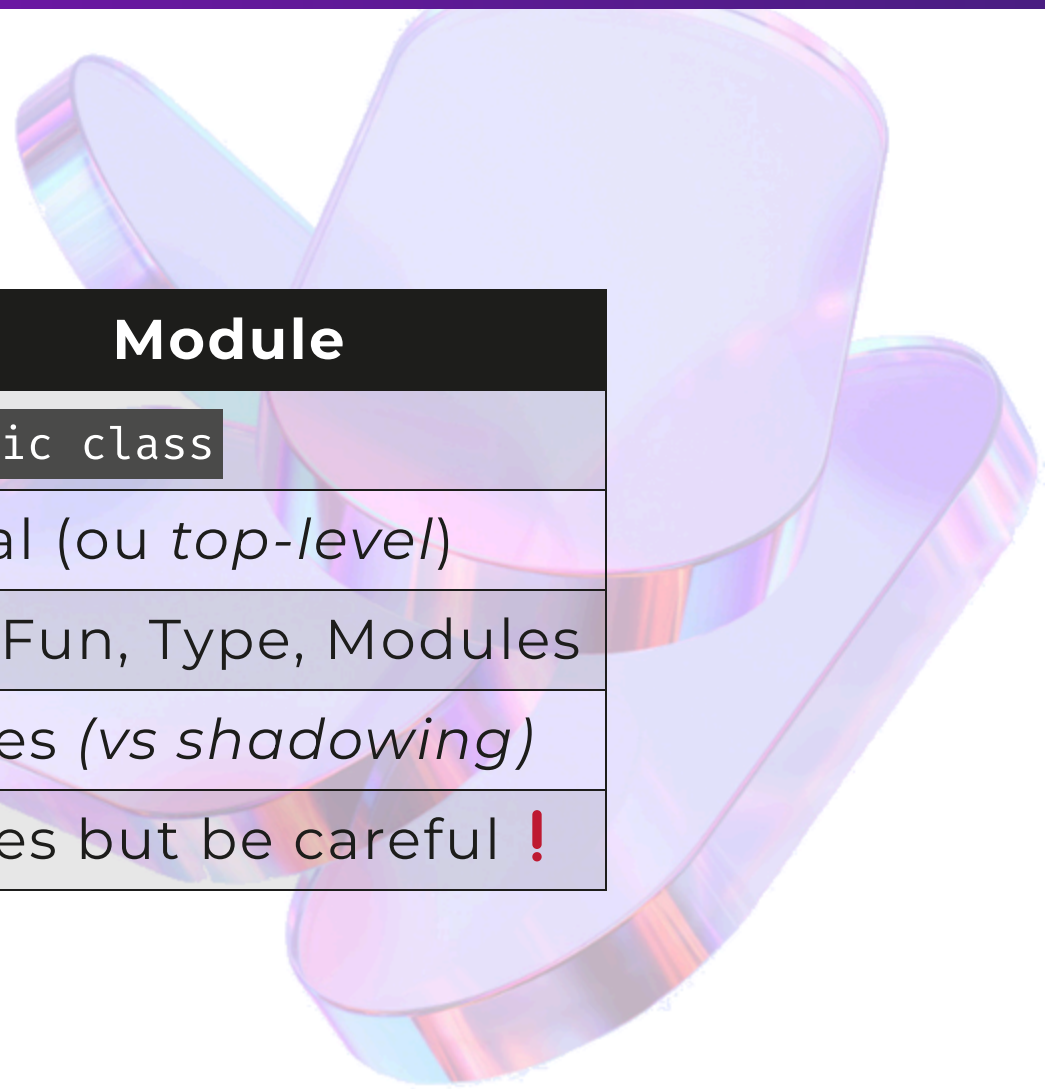
→ `Common.Utilities` for the root namespace

5. Recap

Modules and namespaces

- ➔ Group by functionality
- ➔ Scope: namespaces > files > modules

Property	Namespace	Module
.NET Compilation	`namespace`	`static class`
Type	*Top-level*	Local (ou *top-level*)
Contains	Modules, Types	Val, Fun, Type, Modules
`[<RequireQualifiedAccess>]`	❌ No	✅ Yes *(vs shadowing)*
`[<AutoOpen>]`	❌ No	✅ Yes but be careful ❗



🔗 Additional ressources

docs.microsoft.com/.../fsharp/style-guide/conventions#organizing-code

Thanks 🙏