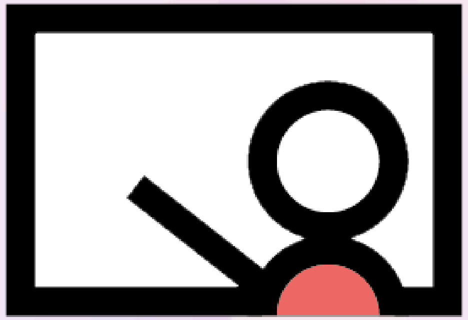
F♯ Training

F♯ collections

2025 April

Table of contents



1. *Collections* Overview

🔍 Common F♯ collections

Module	Type	-	BCL Equivalent	Immutable	Structural comparison
`Array`	`'T array`	≡	`Array<T>`	❌	✅
`List`	`'T list`	≃	`ImmutableList<T>`	✅	✅
`Seq`	`seq<'T>`	≡	`IEnumerable<T>`	✅	✅
`Set`	`Set<'T>`	≃	`ImmutableHashSet<T>`	✅	✅
`Map`	`Map<'K, 'V>`	≃	`ImmutableDictionary<K,V>`	✅	✅
❌	`dict`	≡	`IDictionary<K,V>`	☑️ ❗	❌
❌	`readOnlyDict`	≡	`IReadOnlyDictionary<K,V>`	☑️	❌
❌	`ResizeArray`	≡	`List<T>`	❌	❌

👍 Functions consistency

Common to all 5 modules:

→ `empty`/`isEmpty`, `exists`/`forall`
→ `find`/`tryFind`, `pick`/`tryPick`, `contains` (`containsKey` for `Map`)
→ `map`/`iter`, `filter`, `fold`

Common to `Array`, `List`, `Seq`:

→ `append`/`concat`, `choose`, `collect`
→ `item`, `head`, `last`
→ `take`, `skip`
→ … *a hundred functions altogether!*

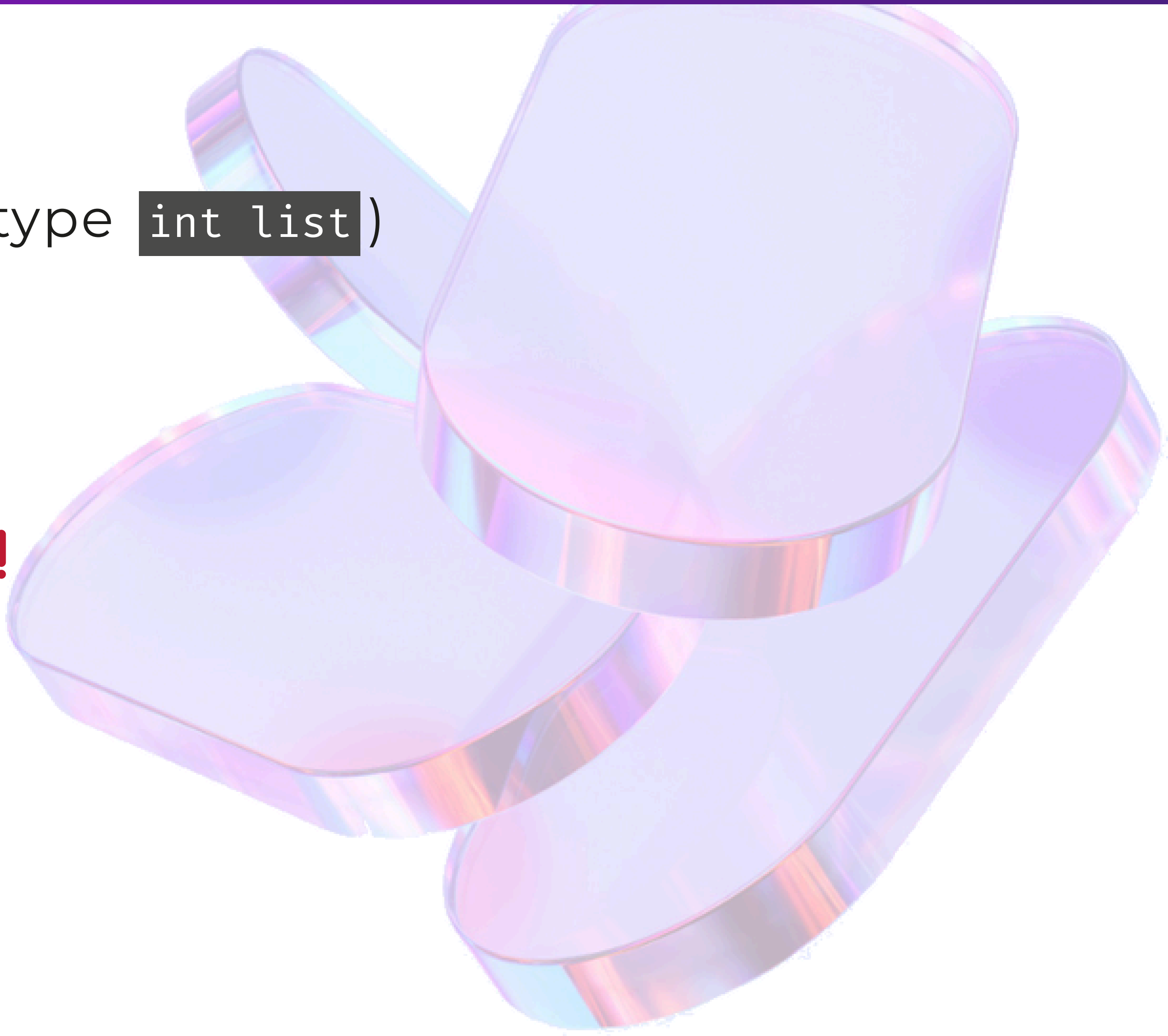
👍 Syntax consistency

Type	Construction	Range	Comprehension
`Array`	`[| 1; 2 |]`	`[| 1..5 |]`	✅
`List`	`[ 1; 2 ]`	`[ 1..5 ]`	✅
`Seq`	`seq { 1; 2 }`	`seq { 1..5 }`	✅
`Set`	`set [ 1; 2 ]`	`set [ 1..5 ]`	✅

⚠️ Syntax trap

Square brackets `[]` are used for:

- ➔ *Value:* instance of a list `[ 1; 2 ]` (of type `int list`)
- ➔ *Type:* array `int []`, e.g. of `[| 1; 2 |]`



☝️ Recommendations

- ➔ Distinguish between type *vs* value ❗
- ➔ Write `int array` rather than `int[]`

Comprehension

- ➔ **Purpose:** syntactic sugar to construct collection
  - ➔ Handy, succinct, powerful
  - ➔ Syntax includes `for` loops, `if` condition
- ➔ Same principle as generators in C♯, JS
  - ➔ `yield` keyword but often **optional** (since F♯ 4.7)
  - ➔ `yield!` keyword *(pronounce "yield bang")* ≡ `yield*` in JS
  - ➔ Works for all collections 👍

Comprehension: examples

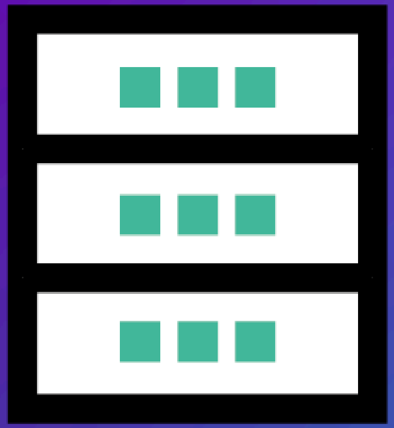
```fsharp
// Multi-line (recommended)
let squares =
    seq { for i in 1 .. 10 do
        yield i * i // 💡 'yield' can be omitted most of the time 👍
    }

// Single line
let squares = seq { for i in 1 .. 10 -> i * i }

// Can contain 'if'
let halfEvens =
    [ for i in [1..10] do
        if (i % 2) = 0 then i / 2 ]  // [1; 2; 3; 4; 5]

// Nested 'for'
let pairs =
    [ for i in [1..3] do
      for j in [1..3] do
        i, j ]              // [(1, 1); (1; 2); (1; 3); (2, 1); ... (3, 3)]
```

Comprehension: examples (2)

Flattening:

```fsharp
// Multiple items
let twoToNine =
    [ for i in [1; 4; 7] do
        if i > 1 then i
        i + 1
        i + 2 ]  // [2; 3; 4; 5; 6; 7; 8; 9]

// With 'yield! collections'
let oneToSix =
    [ for i in [1; 3; 5] do
        yield! [i; i+1] ]
```

2. The Types



Type `List`

Implemented as a **linked list:**
→ 1 list = 1 element *(Head)* + 1 sub-list *(Tail)*
→ Construction using `::` *Cons* operator

To avoid infinite recursion, we need an "exit" case:
→ Empty list named *Empty* and noted `[]`

👉 **Generic and recursive union type:**

```fsharp
type List<'T> =
  | ( [] )
  | ( :: ) of head: 'T * tail: List<'T>
```

☝️ **Note:** this syntax with cases as operator is only allowed in `FSharp.Core`.

`List` : Type alias

`List` *(big L)* : reference to the F♯ type (`List<'t>`) or its companion module.
`list` *(small l)* : alias of F♯'s `List` type, often used with OCaml notation:
→ `let l : string list = ...`

⚠️ **Warnings:** After `open System.Collections.Generic`:
→ `List` is the C♯ mutable list, hiding the F♯ type!
→ The `List` F♯ companion module remains available → confusion!

💡 **Tips:** Use the `ResizeArray` alias 📍

`List` : Immutability

A `List` is **immutable**:
→ It is not possible to modify an existing list.

Adding an element in the list:
= Cheap operation with the *Cons* operator (`::`)
→ Creates a new list with:

- *Head* = given element
- *Tail* = existing list

🏷️ **Related concepts:**

- ➔ linked list
- ➔ recursive type

`List` : Literals

#	Notation	Equivalent	Meaning (*)
0	`[]`	`[]`	Empty
1	`[1]`	`1 :: []`	Cons (1, Empty)
2	`[2; 1]`	`2 :: 1 :: []`	Cons (2, Cons (1, Empty))
3	`[3; 2; 1]`	`3 :: 2 :: 1 :: []`	Cons (3, Cons (2, Cons (1, Empty)))

(*) We can verify it with SharpLab.io :

```
// ...
v1@2 = FSharpList<int>.Cons(1, FSharpList<int>.Empty);
v2@3 = FSharpList<int>.Cons(2, FSharpList<int>.Cons(1, FSharpList<int>.Empty));
// ...
```

List : Initialisation

```fsharp
// Range: Start..End (Step=1)
let numFromOneToFive = [1..5]     // [1; 2; 3; 4; 5]

// Range: Start..Step..End
let oddFromOneToNine = [1..2..9]  // [1; 3; 5; 7; 9]

// Comprehension
let pairsWithDistinctItems =
    [ for i in [1..3] do
      for j in [1..3] do
        if i <> j then
            i, j ]
// [(1; 2); (1; 3); (2, 1); (2, 3); (3, 1); (3, 2)]
```

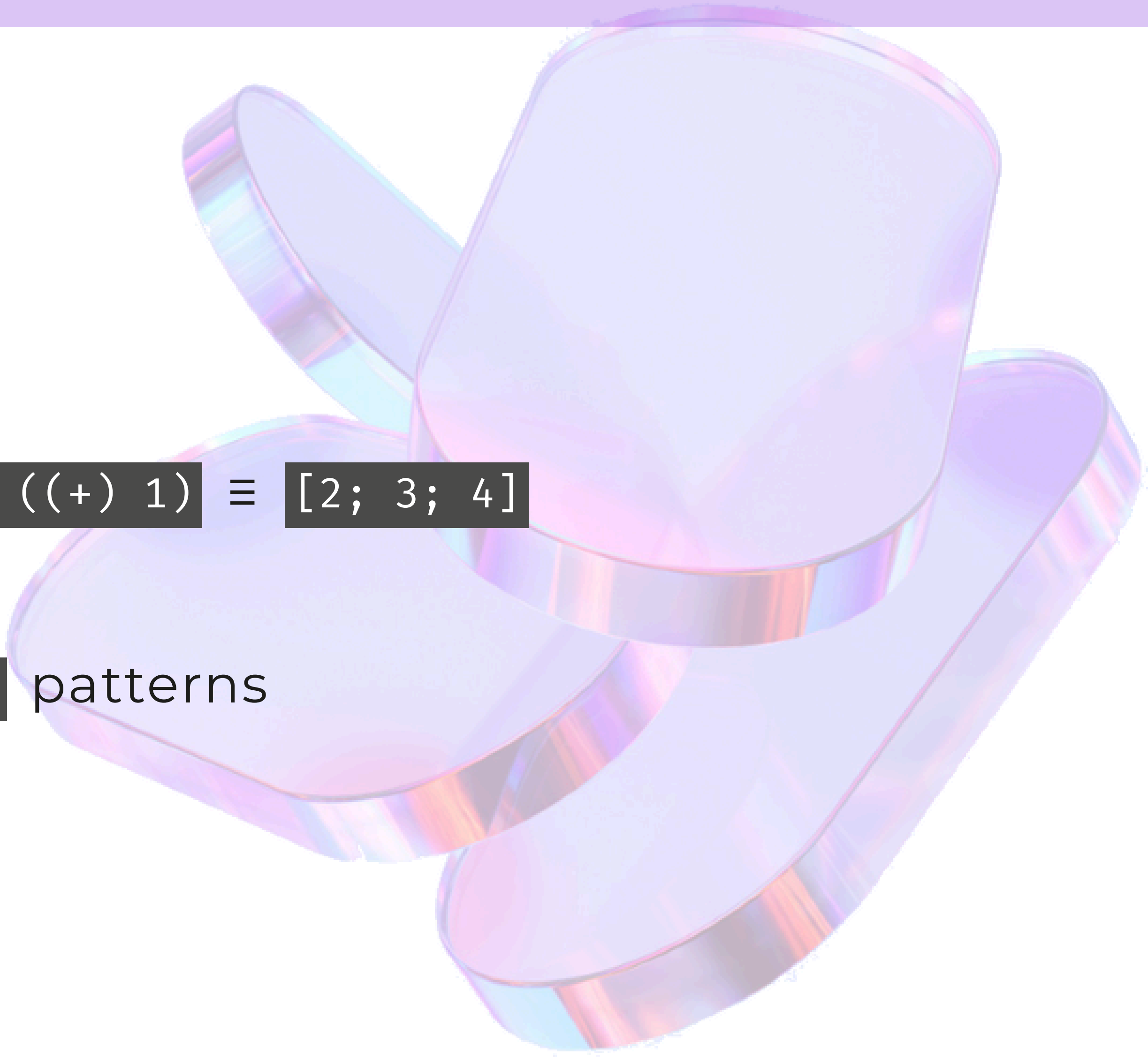
`List` - Exercices 🕹️

1. Implement the `rev` function

Inverts a list: `rev [1; 2; 3]` ≡ `[3; 2; 1]`

2. Implement the `map` function

Transforms each element: `[1; 2; 3] ▷ map ((+) 1)` ≡ `[2; 3; 4]`



💡 Hints

→ Use empty list `[]` or *Cons* `head :: tail` patterns
→ Write a recursive function

⏱️ 5'

`List` - Exercices 🎲

```fsharp
let rev list =
    let rec loop acc rest =
        match rest with
        | [] -> acc
        | x :: xs -> loop (x :: acc) xs
    loop [] list

let map f list =
    let rec loop acc rest =
        match rest with
        | [] -> acc
        | x :: xs -> loop (f x :: acc) xs
    list |> loop [] |> rev
```

💡 **Bonus:** verify the tail recursion with sharplab.io

List - Exercices ✅

Tests can be done in FSI console:

```
let (=!) actual expected =
    if actual = expected
    then printfn $"✅ {actual}"
    else printfn $"❌ {actual} ≠ {expected}"

[1..3] |> rev =! [3; 2; 1];;
// ✅ [3; 2; 1]

[1..3] |> map ((+) 1) =! [2; 3; 4];;
// ✅ [2; 3; 4]
```

Type `Array`

Signature: `'T array` *(recommended)* or `'T[]` or `'T []`

Main differences compared to the `List`:

- ➔ Fixed-size
- ➔ Fat square brackets `[| |]` for literals
- ➔ Mutable ❗
- ➔ Access by index in `O(1)` 👍

`Array` : Syntax

```fsharp
// Literal
[| 1; 2; 3; 4; 5 |]  // val it : int [] = [|1; 2; 3; 4; 5|]

// Range
[| 1 .. 5 |] = [| 1; 2; 3; 4; 5 |]  // true
[| 1 .. 3 .. 10 |] = [| 1; 4; 7; 10 |] // true

// Comprehension
[| for i in 1 .. 5 -> i, i * 2 |]
// [|(1, 2); (2, 4); (3, 6); (4, 8); (5, 10)|]

// Mutation
let names = [| "Juliet"; "Tony" |]
names[1] <- "Bob"
names;;  // [| "Juliet"; "Bob" |]
```

`Array` : Slicing

Returns a sub-array between the given `(start)..(end)` indices

```
let names =     [|"0: Alice"; "1: Jim"; "2: Rachel"; "3: Sophia"; "4: Tony"|]

names[1..3] // [|            "1: Jim"; "2: Rachel"; "3: Sophia"           |]
names[2..]  // [|                      "2: Rachel"; "3: Sophia"; "4: Tony"|]
names[..3]  // [|"0: Alice"; "1: Jim"; "2: Rachel"; "3: Sophia"           |]
```

💡 Works also with `string`: `"012345"[1..3]` ≡ `"123"`

TODO RDE: note / pas confondre avec range - syntaxe similarire

Alias `ResizeArray`

Alias for BCL `System.Collections.Generic.List<T>`

```fsharp
let rev items = items |> Seq.rev |> ResizeArray
let initial = ResizeArray [ 1..5 ]
let reversed = rev initial // ResizeArray [ 5..-1..0 ]
```

Advantages 👍

- No need for `open System.Collections.Generic`
- No name conflicts on `List`

Notes ☝️

- Do not confuse the alias `ResizeArray` with the `Array` F♯ type.
- `ResizeArray` is in F♯ a better name for the BCL generic `List<T>`
  → Closer semantically and in usages to an array than a list

Type `Seq`

Definition: Series of elements of the same type

`'t seq` ≡ `Seq<'T>` ≡ `IEnumerable<'T>`

Lazy: sequence built gradually as it is iterated
≠ All other collections built entirely from their declaration

Seq - Syntax

`seq { items | range | comprehension }`

```
seq { yield 1; yield 2 }   // 'yield' explicit 😕
seq { 1; 2; 3; 5; 8; 13 }  // 'yield' omitted 👍

// Range
seq { 1 .. 10 }         // seq [1; 2; 3; 4; ...]
seq { 1 .. 2 .. 10 }  // seq [1; 3; 5; 7; ...]

// Comprehension
seq { for i in 1 .. 5 do i, i * 2 }
// seq [(1, 2); (2, 4); (3, 6); (4, 8); ...]
```

`Seq` - Infinite sequence

2 options to write an infinite sequence

- ➔ Use `Seq.initInfinite` function
- ➔ Write a recursive function to generate the sequence

Option 1: `Seq.initInfinite` function

- Signature: `(initializer: (index: int) -> 'T) -> seq<'T>`
- Parameter: `initializer` is used to create the specified index element (>= 0)

```
let seqOfSquares = Seq.initInfinite (fun i -> i * i)

seqOfSquares |> Seq.take 5 |> List.ofSeq;;
// val it: int list = [0; 1; 4; 9; 16]
```

Seq - Infinite sequence (2)

Option 2: recursive function to generate the sequence

```fsharp
[<TailCall>]
let rec private squaresStartingAt n =
    seq {
        yield n * n
        yield! squaresStartingAt (n + 1) // 🔄
    }

let squares = squaresStartingAt 0

squares |> Seq.take 10 |> List.ofSeq;;
// val it: int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

Type `Set`

- ➔ Self-ordering collection of unique elements *(without duplicates)*
- ➔ Implemented as a binary tree

```
// Construct
set [ 2; 9; 4; 2 ]          // set [2; 4; 9]  // ☝ Only one '2' in the set
Set.ofArray [| 1; 3 |]      // set [1; 3]
Set.ofList [ 1; 3 ]         // set [1; 3]
seq { 1; 3 } |> Set.ofSeq   // set [1; 3]

// Add/remove element
Set.empty         // set []
|> Set.add 2      // set [2]
|> Set.remove 9   // set [2]    // ☝ No exception
|> Set.add 9      // set [2; 9]
|> Set.remove 9   // set [2]
```

`Set` : Informations

→ `count`, `minElement`, `maxElement`

```
let oneToFive = set [1..5]          // set [1; 2; 3; 4; 5]

// Number of elements: `Count` property or `Set.count` function - ⚠️ O(N)
// ☝️ Do not confuse with `Xxx.length` for Array, List, Seq
let nb = Set.count oneToFive // 5

// Element min, max
let min = oneToFive |> Set.minElement   // 1
let max = oneToFive |> Set.maxElement   // 5
```

Set : Operations

	Operation	Operator	Function for 2 sets	Function for N sets
⊖	Difference	`-`	`Set.difference`	`Set.differenceMany`
∪	Union	`+`	`Set.union`	`Set.unionMany`
∩	Intersection	×	`Set.intersect`	`Set.intersectMany`

Set : Operations examples

```
│ Union               │ Difference          │ Intersection        │
├─────────────────────┼─────────────────────┼─────────────────────┤
│   [ 1 2 3 4 5   ]   │   [ 1 2 3 4 5   ]   │   [ 1 2 3 4 5   ]   │
│ + [   2   4   6 ]   │ - [   2   4   6 ]   │ ∩ [   2   4   6 ]   │
│ = [ 1 2 3 4 5 6 ]   │ = [ 1   3   5   ]   │ = [   2   4     ]   │
```

Type `Map`

Associative array { *Key* → *Value* } ≃ C♯ immutable dictionary

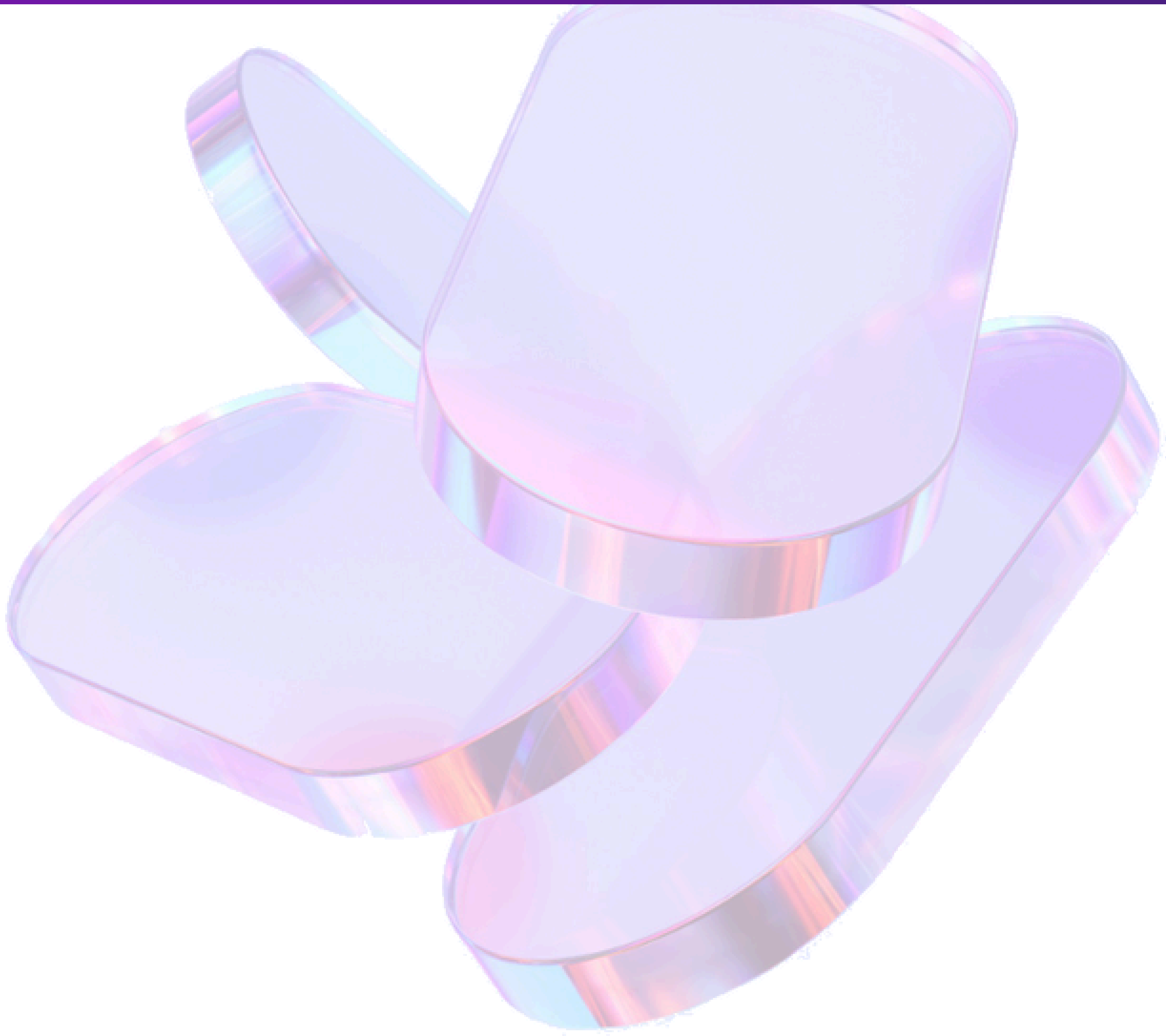
```
// Construct: from collection of (key, val) pairs
// → `Map.ofXxx` function • Xxx = Array, List, Seq
let map1 = seq { (2, "A"); (1, "B") } ▷ Map.ofSeq
// → `Map(tuples)` constructor
let map2 = Map [ (2, "A"); (1, "B"); (3, "C"); (3, "D") ]
// map [(1, "B"); (2, "A"); (3, "D")]
// 👉 Ordered by key (1, 2, 3) and deduplicated in last win - see '(3, "D")'

// Add/remove entry
Map.empty          // map []
 ▷ Map.add 2 "A"   // map [(2, "A")]
 ▷ Map.remove 5    // map [(2, "A")] // ☝️ No exception if key not found
 ▷ Map.add 9 "B"   // map [(2, "A"); (9, "B")]
 ▷ Map.remove 2    // map [(9, "B")]
```

`Map` : Access/Lookup by key

```
let table = Map [ ("A", "Abc"); ("G", "Ghi"); ("Z", "Zzz") ]

// Indexer by key
table["A"];;  // val it: string = "Abc"
table["-"];;  // 💣 KeyNotFoundException

// `Map.find`: return the matching value or 💣 if the key is not found
table ▷ Map.find "G";; // val it: string = "Ghi"

// `Map.tryFind`: return the matching value in an option
table ▷ Map.tryFind "Z";;  // val it: string option = Some "Zzz"
table ▷ Map.tryFind "-";;  // val it: string option = None
```

Dictionaries

- ➔ `dict`
- ➔ `readOnlyDict`



Dictionaries: `dict` function

- ➔ Builds an `IDictionary<'k, 'v>` from a sequence of key/value pairs
- ➔ The interface is not honest: the dictionary is **immutable** ❗

```
let table = dict [ (1, 100); (2, 200) ] // System.Collections.Generic.IDictionary<int,int>

table[1];;           // val it: int = 100

table[99];;          // 💣 KeyNotFoundException

table[1] <- 111;;    // 💣 NotSupportedException: This value cannot be mutated
table.Add(3, 300);; // 💣 NotSupportedException: This value cannot be mutated
```

Dictionaries: `readOnlyDict` function

- ➔ Builds an `IReadOnlyDictionary<'k, 'v>` from a sequence of key/value pairs
- ➔ The interface is honest: the dictionary is **immutable**

```fsharp
let table = readOnlyDict [ (1, 100); (2, 200) ]
// val table: System.Collections.Generic.IReadOnlyDictionary<int,int>

table[1];;          // val it: int = 100

table[99];;         // 💣 KeyNotFoundException

do table[1] ← 111;;
// ~~~~~~~~~ 💥 Error FS0810: Property 'Item' cannot be set

do table.Add(3, 300);;
//       ~~~ 💥 Error FS0039: The type 'IReadOnlyDictionary<_,_>'
//              does not define the field, constructor or member 'Add'.
```

Dictionaries: recommendation

`dict` returns an object that does not implement fully `IDictionary<'k, 'v>`
→ Violate the Liskov's substitution principle ❗

`readOnlyDict` returns an object that respects `IReadOnlyDictionary<'k, 'v>`

👉 Prefer `readOnlyDict` to `dict` when possible

Dictionaries: `KeyValue` active pattern

Used to deconstruct a `KeyValuePair` dictionary entry to a `(key, value)` pair

```fsharp
// FSharp.Core / prim-types.fs#4983
let (|KeyValue|) (kvp: KeyValuePair<'k, 'v>) : 'k * 'v =
    kvp.Key, kvp.Value

let table =
    readOnlyDict
        [ (1, 100)
          (2, 200)
          (3, 300) ]

// Iterate through the dictionary
for kv in table do // kv: KeyValuePair<int,int>
    printfn $"{kv.Key}, {kv.Value}"

// Same with the active pattern
for KeyValue (key, value) in table do
    printfn $"{key}, {value}"
```

Lookup performance

🔗 *High Performance Collections in F#* • https://kutt.it/dxDOi7 • Jan 2021

`Dictionary` *vs* `Map`

`readOnlyDict` creates **high-performance** dictionaries
→ 10x faster than `Map` for lookups

`Dictionary` *vs* `Array`

~ Rough heuristics

→ The `Array` type is OK for few lookups (< 100) and few elements (< 100)
→ Use a `Dictionary` otherwise

`Map` and `Set` *vs* `IComparable`

Only work if elements (of a `Set`) or keys (of a `Map`) are **comparable**!

Examples:

```
// Classes are not comparable by default, so you cannot use them in a set or a map
type NameClass(name: string) =
    member val Value = name

let namesClass = set [NameClass("Alice"); NameClass("Bob")]
//                     ~~~~~~~~~~~~~~~~~~
// 💥 Error FS0193: The type 'NameClass' does not support the 'comparison' constraint.
//      For example, it does not support the 'System.IComparable' interface
```

`Map` and `Set` : `IComparable` types

F# functional type: tuple, record, union

```fsharp
// Example: single-case union
type Name = Name of string

let names = set [Name "Alice"; Name "Bob"]
```

Structs:

```fsharp
[<Struct>]
type NameStruct(name: string) =
    member this.Name = name

let namesStruct = set [NameStruct("Alice"); NameStruct("Bob")]
```

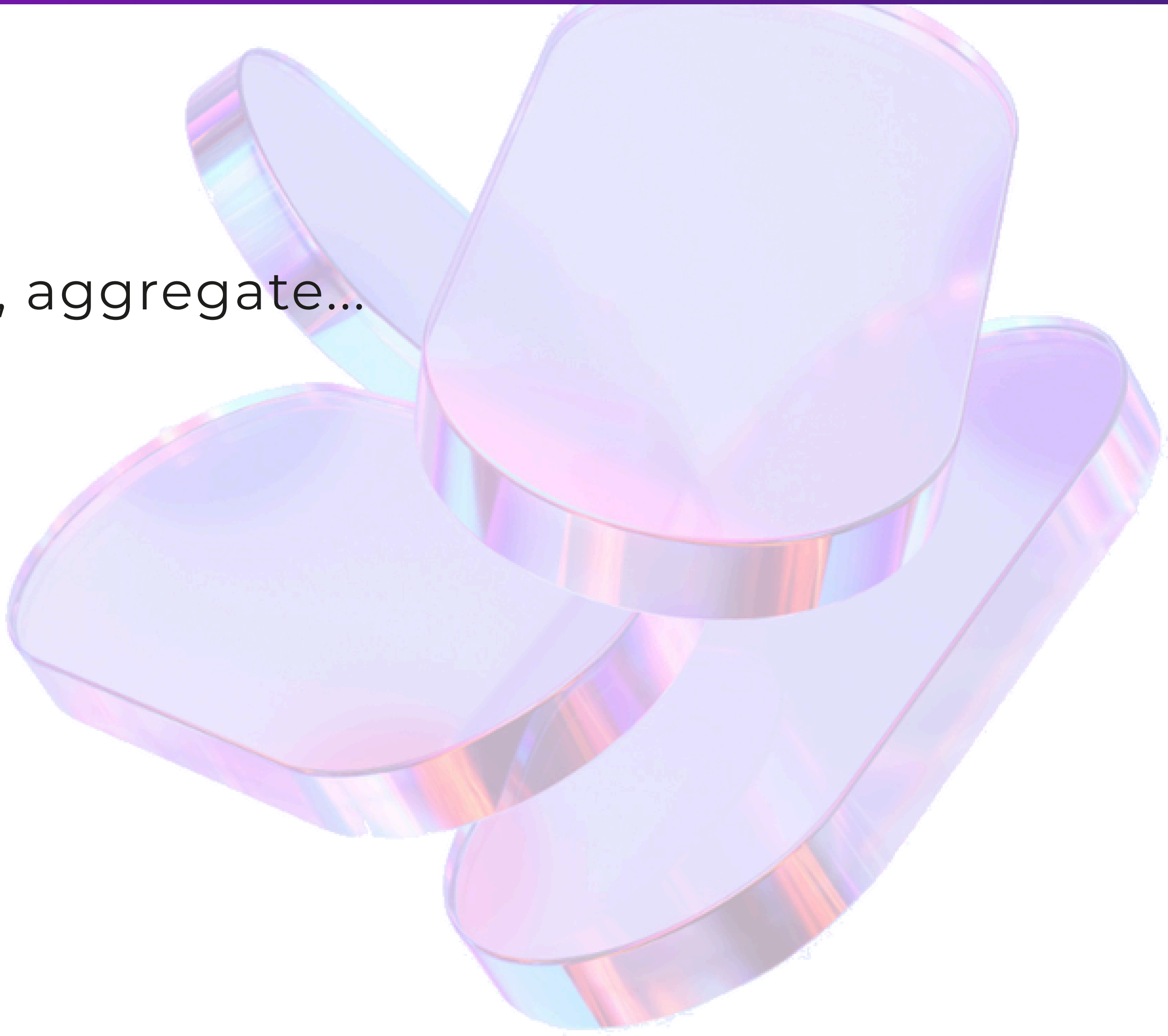
Classes implementing `IComparable`… *but not* *`IComparable<'T>`* 🤷

3. Common functions

Common functions

Functions available in different modules
→ Customized for the target type

Operations: access, construct, find, select, aggregate…



Convention

☝️ **Convention** used here:

1. Functions are given by their name
   ➔ To use them, we need to qualify them by the module.
2. The last parameter is omitted for brevity
   ➔ It's always the collection.

Example: `head` *vs* `List.head list`

Access to an element

↓ Access \ Returns →	`'T` or 💣	`'T option`
By index	`x[index]`	
By index	`item index`	`tryItem index`
First element	`head`	`tryHead`
Last element	`last`	`tryLast`

💣 `ArgumentException` or `IndexOutOfRangeException`

```
[1; 2] ▷ List.tryHead     // Some 1
[1; 2] ▷ List.tryItem 2   // None
```

Access to an element : Cost ⚠️

Function \ Module	`Array`	`List`	`Seq`
`head`	O(1)	O(1)	O(1)
`item`	O(1)	O(n) ❗	O(n) ❗
`last`	O(1)	O(n) ❗	O(n) ❗
`length`	O(1)	O(n) ❗	O(n) ❗

Combine collections

Function	Parameters	Final size
`append` / `@`	2 collections of sizes N1 et N2	N1 + N2
`concat`	K collections of sizes N1..Nk	N1 + N2 + ... + Nk
`zip`	2 collections of same size N ❗	N pairs
`allPairs`	2 collections of sizes N1 et N2	N1 * N2 pairs

```
List.concat [ [1]; [2; 3] ];;  // [1; 2; 3]
List.append [1;2;3] [4;5;6];;  // [1; 2; 3; 4; 5; 6]

// @ operator: alias of `List.append` only • not working with Array, Seq
[1;2;3] @ [4;5;6];;            // [1; 2; 3; 4; 5; 6]

List.zip      [1; 2] ['a'; 'b'];;  // [(1, 'a'); (2, 'b')]
List.allPairs [1; 2] ['a'; 'b'];;  // [(1, 'a'); (1, 'b'); (2, 'a'); (2, 'b')]
```

Find an element

Using a predicate `f : 'T -> bool`:

Which element \ Returns	`'T` or 💣	`'T option`
First found	`find`	`tryFind`
Last found	`findBack`	`tryFindBack`
Index of first found	`findIndex`	`tryFindIndex`
Index of last found	`findIndexBack`	`tryFindIndexBack`

```fsharp
[1; 2] |> List.find (fun x -> x < 2)       // 1
[1; 2] |> List.tryFind (fun x -> x >= 2)   // Some 2
[1; 2] |> List.tryFind (fun x -> x > 2)    // None
```

Search elements

Search	How many items	Function
By value	At least 1	`contains value`
By predicate `f`	At least 1	`exists f`
"	All	`forall f`

```
[1; 2] ▷ List.contains 0       // false
[1; 2] ▷ List.contains 1       // true
[1; 2] ▷ List.exists (fun x → x ≥ 2)  // true
[1; 2] ▷ List.forall (fun x → x ≥ 2)  // false
```

Select elements

What elements	By size	By predicate `f`
All those found		`filter f`
First ignored	`skip n`	`skipWhile f`
First found	`take n`	`takeWhile f`
	`truncate n`	

☝ **Notes:**

- `skip`, `take` *vs* `truncate` when `n` > collection's size
  → `skip`, `take`: 💣 exception
  → `truncate`: empty collections w/o exception
- Alternative for `Array`: *Range* `arr[2..5]`

Map elements

Functions taking as input:
→ A mapping function `f` (a.k.a. *mapper*)
→ A collection of type `~~~~ 'T`, `~~~~` meaning `Array`, `List`, or `Seq`

Function	Mapping `f`	Returns	How many elements?
`map`	`'T → 'U`	`'U ~~~~`	As many in than out
`mapi`	`int → 'T → 'U`	`'U ~~~~`	As many in than out
`collect`	`'T → 'U ~~~~`	`'U ~~~~`	*flatMap*
`choose`	`'T → 'U option`	`'U ~~~~`	Less
`pick`	`'T → 'U option`	`'U`	1 (the first matching) or 💣
`tryPick`	`'T → 'U option`	`'U option`	1 (the first matching)

`map` *vs* `mapi`

`mapi` ≡ `map` *with index*

The difference is on the `f` mapping parameter:

- `map`: `'T → 'U`
- `mapi`: `int → 'T → 'U` → the additional `int` parameter is the item index

```fsharp
["A"; "B"]
▷ List.mapi (fun i x → $"{i+1}. {x}")
// ["1. A"; "2. B"]
```

Alternative to `mapi`

Apart from `map` and `iter`, no `xxx` function has a `xxxi` variant.

💡 Use `indexed` to obtain elements with their index

```
let isOk (i, x) = i ≥ 1 && x ≤ "C"

["A"; "B"; "C"; "D"]
▷ List.indexed       // [ (0, "A"); (1, "B"); (2, "C"); (3, "D") ]
▷ List.filter isOk   //             [ (1, "B"); (2, "C") ]
▷ List.map snd       //                 [ "B" ; "C" ]
```

`map` *vs* `iter`

`iter` looks like `map` with
- no mapping: `'T -> unit` *vs* `'T -> 'U`
- no output: `unit` *vs* `'U list`

But `iter` is in fact used for a different use case:
→ to trigger an action, a side-effect, for each item

Example: print the item to the console

```
["A"; "B"; "C"] |> List.iteri (fun i x -> printfn $"Item #{i}: {x}")
// Item #0: A
// Item #1: B
// Item #2: C
```

`choose`, `pick`, `tryPick`

Signatures:

- `choose  : mapper: ('T → 'U option) → items: 'T ~~~~ → 'U ~~~~`
- `pick    : mapper: ('T → 'U option) → items: 'T ~~~~ → 'U`
- `tryPick : mapper: ('T → 'U option) → items: 'T ~~~~ → 'U option`

The mapping may return `None` for some items not mappable (or just ignored)

Different use cases:

- `choose` to get all the mappable items mapped
- `pick` or `tryPick` to get the first one

When no items are mappable:

- `choose` returns an empty collection
- `tryPick` returns `None`
- `pick` raises a `KeyNotFoundException` 💣

`choose`, `pick`, `tryPick` - Examples

```fsharp
let tryParseInt (s: string) =
    match System.Int32.TryParse(s) with
    | true,  i -> Some i
    | false, _ -> None

["1"; "2"; "?"] |> List.choose tryParseInt   // [1; 2]
["1"; "2"; "?"] |> List.pick tryParseInt     // 1
["1"; "2"; "?"] |> List.tryPick tryParseInt  // Some 1
```

Analogies

`choose f` ≃

- `collect (f >> Option.to{Collection})`
- `(filter (f >> Option.isSome)) >> (map (f >> Option.value))`

`(try)pick f` ≃

- `(try)find(f >> Option.isSome)) >> f`
- `choose f >> (try)head`

Aggregate : specialized functions

Given a projection `f: 'T → 'U`

Direct	Projection	Mapping	Constraint
`max`	`maxBy f`	❌ No	`comparison`
`min`	`minBy f`	❌ No	`comparison`
`sum`	`sumBy f`	✅ Yes	Monoid 📍
`average`	`averageBy f`	✅ Yes	Monoid 📍

💣 `ArgumentException` if the collection is empty.

Aggregate : `max(By)` examples

```fsharp
type N = One | Two | Three | Four | Five

let max = List.max [ One; Two; Three; Four; Five ]
// val max: N = Five

let maxText = List.maxBy string [ One; Two; Three; Four; Five ]
// val maxText: N = Two
```

☝️ **Notes:**

➔ `comparison` constraint followed by `N`: unions are `IComparable` by default

➔ `maxBy` performs no mapping: see the returned value: `Two`, ≠ `"Two"`

➔ Same for `min` and `minBy`

Aggregate : `sum(By)` examples

```fsharp
let sumKO = List.sum [ (1,"a"); (2,"b"); (3,"c") ]
//                     ~~~~~~ 💥 Error FS0001
// Expecting a type supporting the operator 'get_Zero' but given a tuple type

let sum = List.sumBy fst [ (1,"a"); (2,"b"); (3,"c") ]
// val sum: int = 6
```

☝️ **Notes:**

- ➔ The *Monoid* constraint is revealed with the error `FS0001`
  - ➔ Items type must have: `Zero` static prop, overload of the `+` operator
- ➔ `sumBy` performs a mapping: see the returned type: `int`, ≠ `int * string`
- ➔ Same for `average` and `averageBy`

Monoidal custom type example

```fsharp
type Point = Point of X:int * Y:int with
    static member Zero = Point (0, 0)
    static member (+) (Point (ax, ay), Point (bx, by)) = Point (ax + bx, ay + by)

let addition = (Point (1, 1)) + (Point (2, 2))
// val addition : Point = Point (3, 3)

let sum = [1..3] |> List.sumBy (fun i -> Point (i, -i))
// val sum : Point = Point (6, -6)
```

Aggregate : `countBy`

Uses a projection `f: 'T → 'U` to compute a "key" for each item
Returns all the keys with the number of items having this key

```
let words = [ "hello"; "world"; "fsharp"; "is"; "awesome" ]
let wordCountByLength = words ▷ List.countBy String.length ▷ List.sortBy fst
//val wordCountByLength: (int * int) list = [(2, 1); (5, 2); (6, 1); (7, 1)]
```

💡 Useful to determine duplicates:

```
let findDuplicates items =
    items
    ▷ List.countBy id
    ▷ List.choose (fun (item, count) → if count > 1 then Some item else None)

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

Aggregate : versatile functions

- `fold       (f: 'U → 'T → 'U) (seed: 'U) items`
- `foldBack   (f: 'T → 'U → 'U) items (seed: 'U)`
- `reduce     (f: 'T → 'T → 'T) items`
- `reduceBack (f: 'T → 'T → 'T) items`

folder `f` takes 2 parameters: an "accumulator" `acc` and the current element `x`

`xxxBack` *vs* `xxx`:
- Iterates from last to first element
- Parameters `seed` and `items` reversed (for `foldBack`)
- Folder `f` parameters reversed (`x acc`)

`reduceXxx` *vs* `foldXxx`:
- `reduceXxx` uses the first item as the *seed* and performs no mapping (`'T → 'T`)
- `reduceXxx` fails if the collection is empty 💣

Aggregate : versatile functions (2)

Examples:

```fsharp
["a";"b";"c"] |> List.reduce (+)  // "abc"
[ 1; 2; 3 ] |> List.reduce ( * )  // 6

[1;2;3;4] |> List.reduce     (fun acc x -> 10 * acc + x)  // 1234
[1;2;3;4] |> List.reduceBack (fun x acc -> 10 * acc + x)  // 4321

("all:", [1;2;3;4]) ||> List.fold     (fun acc x -> $"{acc}{x}")  // "all:1234"
([1;2;3;4], "rev:") ||> List.foldBack (fun x acc -> $"{acc}{x}")  // "rev:4321"
```

Aggregate : `fold(Back)` versatility

We could write almost all functions with `fold` or `foldBack` *(performance aside)*

```fsharp
// collect function using fold
let collect f list = List.fold (fun acc x -> acc @ f x) [] list

let test1 = [1; 2; 3] |> collect (fun x -> [x; x])  // [1; 1; 2; 2; 3; 3]

// filter function using foldBack
let filter f list = List.foldBack (fun x acc -> if f x then x :: acc else acc) list []

let test2 = [1; 2; 3; 4; 5] |> filter ((=) 2)  // [2]

// map function using foldBack
let map f list = List.foldBack (fun x acc -> f x :: acc) list []

let test3 = [1; 2; 3; 4; 5] |> map (~-)  // [-1; -2; -3; -4; -5]
```

Change the order of elements

Operation	Direct	Mapping
Inversion	`rev list`	×
Sort asc	`sort list`	`sortBy f list`
Sort desc	`sortDescending list`	`sortDescendingBy f list`
Sort custom	`sortWith comparer list`	×

```fsharp
[1..5] |> List.rev // [5; 4; 3; 2; 1]
[2; 4; 1; 3; 5] |> List.sort // [1..5]
["b1"; "c3"; "a2"] |> List.sortBy (fun x -> x[0]) // ["a2"; "b1"; "c3"] because a < b < c
["b1"; "c3"; "a2"] |> List.sortBy (fun x -> x[1]) // ["b1"; "a2"; "c3"] because 1 < 2 < 3
```

Separate

💡 Elements are divided into groups.

Operation	Result (`;` *omitted)*	Remark
`[1..10]`	`[ 1 2 3 4 5 6 7 8 9 10 ]`	`length = 10`
`chunkBySize 3`	`[[1 2 3] [4 5 6] [7 8 9] [10]]`	`forall: length ⩽ 3`
`splitInto 3`	`[[1 2 3 4] [5 6 7] [8 9 10]]`	`length ⩽ 3`
`splitAt 3`	`([1 2 3],[4 5 6 7 8 9 10])`	Tuple ❗

Group items - By size

💡 Items can be **duplicated** into different groups.

Operation	Result (`'` *and* `;` *omitted)*	Remark
`[1..5]`	`[ 1 2 3 4 5 ]`	
`pairwise`	`[(1,2) (2,3) (3,4) (4,5)]`	Tuple ❗
`windowed 2`	`[[1 2] [2 3] [3 4] [4 5]]`	Array of arrays of 2 items
`windowed 3`	`[[1 2 3] [2 3 4] [3 4 5]]`	Array of arrays of 3 items

Group items - By criteria

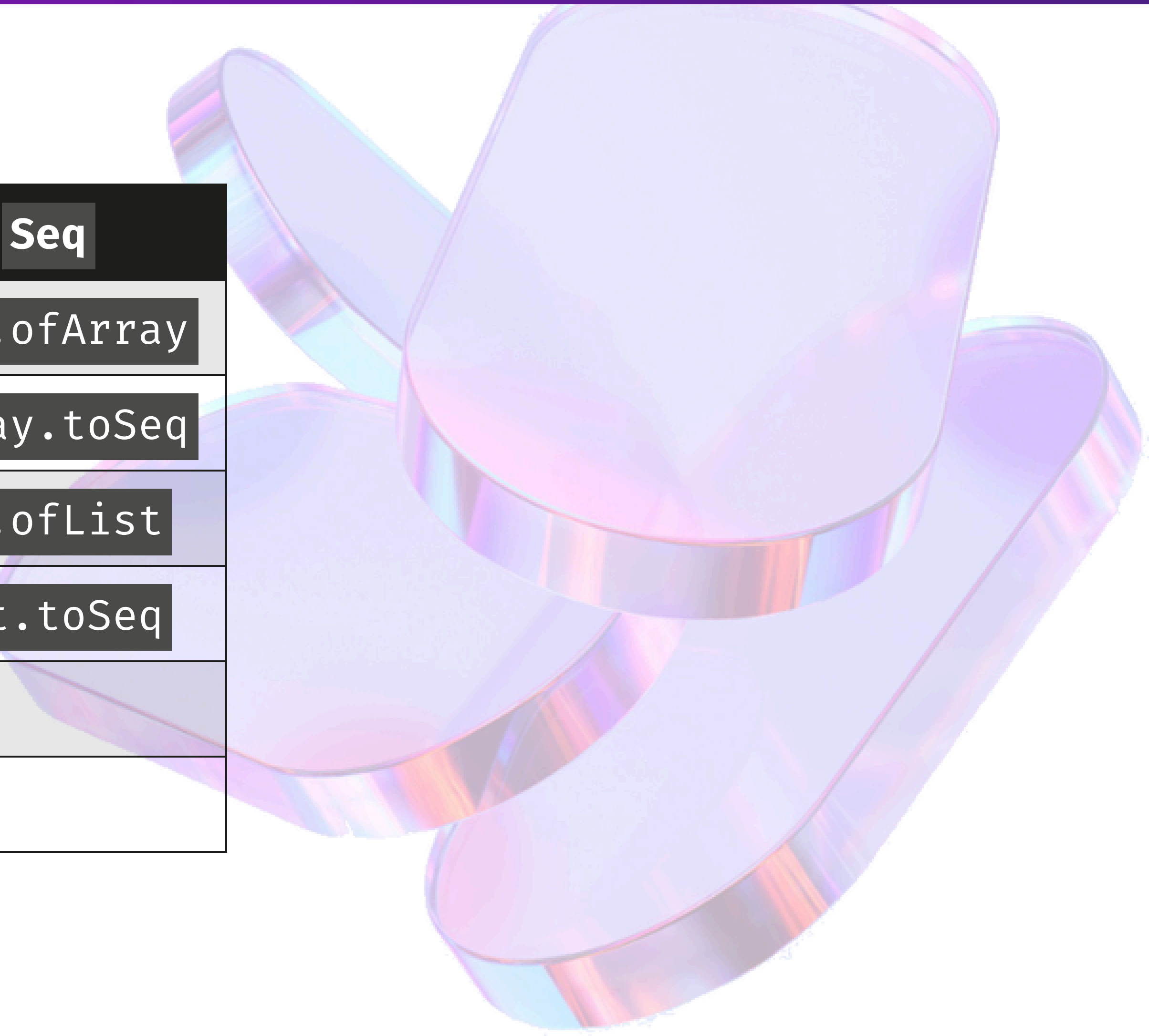
Operation	Criteria	Result
`partition`	`predicate: 'T → bool`	`('T list * 'T list)`
		→ 1 pair `([OKs], [KOs])`
`groupBy`	`projection: 'T → 'K`	`('K * 'T list) list`
		→ N tuples `[(key, [related items])]`

```fsharp
let isOdd i = (i % 2 = 1)
[1..10] |> List.partition isOdd // (        [1; 3; 5; 7; 9] ,          [2; 4; 6; 8; 10]  )
[1..10] |> List.groupBy isOdd   // [ (true, [1; 3; 5; 7; 9]); (false, [2; 4; 6; 8; 10]) ]

let firstLetter (s: string) = s.[0]
["apple"; "alice"; "bob"; "carrot"] |> List.groupBy firstLetter
// [('a', ["apple"; "alice"]); ('b', ["bob"]); ('c', ["carrot"])]
```

Change collection type

Your choice: `Dest.ofSource` or `Source.toDest`

From \ to	`Array`	`List`	`Seq`
`Array`	×	`List.ofArray`	`Seq.ofArray`
	×	`Array.toList`	`Array.toSeq`
`List`	`Array.ofList`	×	`Seq.ofList`
	`List.toArray`	×	`List.toSeq`
`Seq`	`Array.ofSeq`	`List.ofSeq`	×
	`Seq.toArray`	`Seq.toList`	×



Functions *vs* comprehension

The functions of `List`/`Array`/`Seq` can often be replaced by a comprehension, more versatile:

```
let list = [ 0..99 ]

list ▷ List.map f                  ⟷  [ for x in list do f x ]
list ▷ List.filter p               ⟷  [ for x in list do if p x then x ]
list ▷ List.filter p ▷ List.map f  ⟷  [ for x in list do if p x then f x ]
list ▷ List.collect g              ⟷  [ for x in list do yield! g x ]
```

4. Dedicated functions

`List` module

Cons `1 :: [2; 3]`

- ➔ Item added to top of list
- ➔ List appears in reverse order 😕
- ➔ But operation efficient: in **O(1)** *(`Tail` preserved)* 👍

Append `[1] @ [2; 3]`

- ➔ List in normal order
- ➔ But operation in **O(n)** ❗ *(New `Tail` at each iteration)*

`Map` module

`Map.add key value`

→ **Safe add:** replace existing value of existing key

→ Parameters `key value` curried, not a pair `(key, value)`

`Map.remove key`

→ **Safe remove:** just return the given `Map` if key not found

```
let input = Map [ (1, "a"); (2, "b") ]

input
▷ Map.add 3 "c"   // map [(1, "a"); (2, "b"); (3, "c")]
▷ Map.add 1 "@"   // map [(1, "@"); (2, "b"); (3, "c")]
▷ Map.remove 2    // map [(1, "@"); (3, "c")]
```

`Map.change`

`Map.change key (f: 'T option → 'T option)`
→ All-in-one function to add, modify or remove the element of a given key
→ Depends on the `f` function passed as an argument

Key	Input	`f` returns `None`	`f` returns `Some newVal`
-	-	≡ `Map.remove key`	≡ `Map.add key newVal`
Found	`Some value`	Remove the entry	Change the value to *newVal*
Not found	`None`	Ignore this key	Add the item *(key, newVal)*

Map.change example

Lexicon: Build a Map to classify words by their first letter capitalized

```fsharp
let firstLetter (word: string) = System.Char.ToUpperInvariant(word[0])

let classifyWordsByLetter words =
    (Map.empty, words)
    ||> Seq.fold (fun map word ->
        map |> Map.change (word |> firstLetter) (fun wordsWithThisLetter ->
            wordsWithThisLetter
            |> Option.defaultValue Set.empty
            |> Set.add word
            |> Some)
        )

let t = classifyWordsByLetter ["apple"; "blueberry"; "banana"; "apricot"; "cherry"; "avocado"]
// map [ 'A', set ["apple"; "apricot"; "avocado"]
//       'B', set ["banana"; "blueberry"]
//       'C', set ["cherry"] ]
```

`Map.change` example (2)

Lexicon → *Better implementation*

```fsharp
let firstLetter (word: string) = System.Char.ToUpperInvariant(word[0])

let classifyWordsByLetter words =
    words
    |> Seq.groupBy firstLetter
    |> Seq.map (fun (letter, wordsWithThisLetter) -> letter, set wordsWithThisLetter)
    |> Map.ofSeq
```

`Map.containsKey` *vs* `Map.exists`

`Map.containsKey (key: 'K)`
→ Indicates whether the key is present

`Map.exists (predicate: 'K → 'V → bool)`
→ Indicates whether an entry (as `key value` parameters) satisfies the predicate
→ Parameters `key value` curried, not a pair `(key, value)`

```
let table = Map [ (2, "A"); (1, "B"); (3, "D") ]

table |> Map.containsKey 0;;  // false
table |> Map.containsKey 2;;  // true

let isEven i = i % 2 = 0
let isVowel (s: string) = "AEIOUY".Contains(s)

table |> Map.exists (fun k v -> (isEven k) && (isVowel v));;  // true
```

`Seq` module

`Seq.cache (source: 'T seq) → 'T seq`

Sequences are **lazy**: elements (re)evaluated at each time iteration
→ Can be costly 💸

Invariant sequences iterated multiple times
→ Iterations can be optimized by caching the sequence using `Seq.cache`
→ Caching is optimized by being deferred and performed only on first iteration

⚠️ **Recommendation:** Caching is hidden, not reflected on the type (`'T seq`)
→ Only apply caching on a sequence used in a very small scope
→ Prefer another collection type otherwise

`String` module

`string` ≡ `Seq<char>` functions + `String` *FSharp.Core* module / `System` class

```
String.concat (separator: string) (strings: seq<string>) : string

String.init      (count: int) (f: (index: int) → string) : string
String.replicate (count: int) (s: string) : string

String.exists (predicate: char → bool) (s: string) : bool
String.forall (predicate: char → bool) (s: string) : bool
String.filter (predicate: char → bool) (s: string) : string

String.collect (mapping:        char → string) (s: string) : string
String.map     (mapping:        char → char)   (s: string) : string
String.mapi    (mapping: int → char → char)   (s: string) : string
// Idem iter/iteri which returns unit
```

`String` module - Examples

```fsharp
let a = String.concat "-" ["a"; "b"; "c"]  // "a-b-c"
let b = String.init 3 (fun i -> $"#{i}")   // "#0#1#2"
let c = String.replicate 3 "0"             // "000"

let d = "abcd" |> String.exists (fun c -> c >= 'b')  // true
let e = "abcd" |> String.forall (fun c -> c >= 'b')  // false
let f = "abcd" |> String.filter (fun c -> c >= 'b')  // "bcd"

let g = "abcd" |> String.collect (fun c -> $"{c}{c}")  // "aabbccdd"

let h = "abcd" |> String.map (fun c -> (int c) + 1 |> char)  // "bcde"
```

5. 🍔 Quiz

Question 1

What function should you use to `format` all `addresses`?

```fsharp
type Address = { City: string; Country: string }

let format address = $"{address.City}, {address.Country}"

let addresses: Address list = [ … ]

let formattedAddresses = addresses |> List.??? format // ❓
```

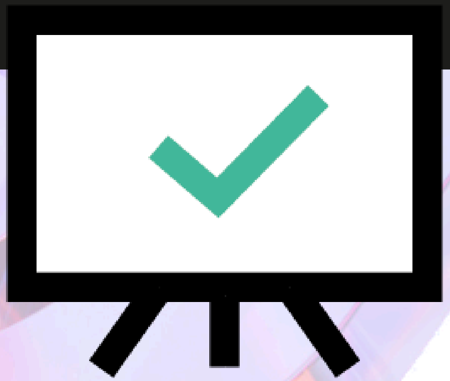
A. `List.iter()`
B. `List.map()`
C. `List.sum()`

⏱ 10’’

Question 1 » Answer

What function should you use to `format` all `addresses`?

```fsharp
type Address = { City: string; Country: string }

let format address = $"{address.City}, {address.Country}"

let addresses: Address list = [ … ]

let formattedAddresses = addresses ▷ List.map format
```



A. `List.iter()` ❌
B. `List.map()` ✅
C. `List.sum()` ❌

Question 2

What is the returned value of `[1..4] |> List.head`?

A. `[2; 3; 4]`

B. `1`

C. `4`

⏱️ 10’’

Question 2 » Answer

What is the returned value of `[1..4] ▷ List.head`?

A. `[2; 3; 4]` ❌ *(This is the result of `List.tail`)*

B. `1` ✅

C. `4` ❌ *(This is the result of `List.last`)*

Question 3

What's the right way to compute the average of a list?

A. `[2; 4] |> List.average`

B. `[2; 4] |> List.avg`

C. `[2.0; 4.0] |> List.average`

⏱️ 10’’

Question 3 » Answer

What's the right way to compute the average of a list?

A. `[2; 4] ▷ List.average` ❌

💥 **Error FS0001:** `List.average` does not support the type `int`, because the latter lacks the required (real or built-in) member `DivideByInt`

B. `[2; 4] ▷ List.avg`

💥 **Error FS0039:** The value, constructor, namespace or type `avg` is not defined.

C. `[2.0; 4.0] ▷ List.average` ✅

`val it : float = 3.0`

6. The Recap

Types

5 collections including 4 functional/immutable

`List`: default choice / *Versatile*, *Practical*
→ `[ ]`, Operators: *Cons* `::`, *Append* `@`, Pattern matching

`Array`: fixed-size, mutable, performance-oriented (e.g. indexer)
→ `[| |]` less handy to write than `[ ]`

`Seq`: deferred evaluation (*Lazy*), infinite sequence

`Set`: unique elements

`Map`: values by unique key

API

Rich
→ Hundreds of functions >> Fifty for LINQ

Consistency
→ Common syntax and functions
→ Functions preserve the collection type (≠ LINQ sticked to `IEnumerable<>`)

Semantic
→ Function names closer to JS

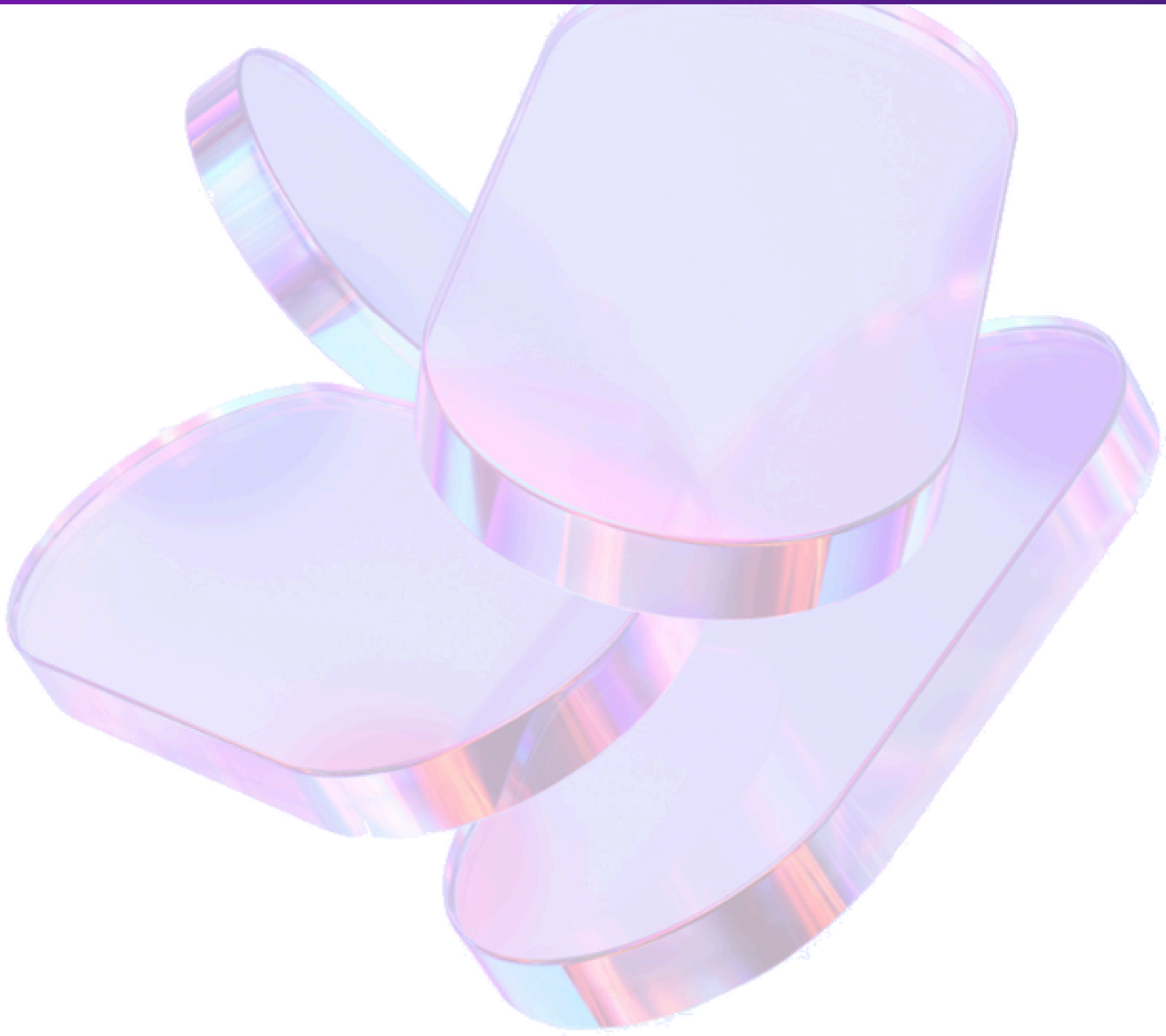
API - Comparison C♯ / F♯ / JS

C♯ LINQ	F♯	JS `Array`
`Select()`, `SelectMany()`	`map`, `collect`	`map()`, `flatMap()`
`Any(predicate)`, `All()`	`exists`, `forall`	`some()`, `every()`
`Where()`, ×	`filter`, `choose`	`filter()`, ×
`First()`, `FirstOrDefault()`	`find`, `tryFind`	×, `find()`
×	`pick`, `tryPick`	×
`Aggregate([seed]])`	`fold`, `reduce`	`reduce()`
`Average()`, `Sum()`	`average`, `sum`	×
`ToList()`, `AsEnumerable()`	`List.ofSeq`, `toSeq`	×
`Zip()`	`zip`	×

BCL types

- ➔ `Array`
- ➔ `ResizeArray` for C# List
- ➔ *Dictionaries:* `dict`, `readOnlyDict`

For interop or performance



🕹️ Exercises @ exercism.io

Exercise	Level	Topics
High Scores	Easy	`List`
Protein Translation	Medium+	`Seq`/`List` 💡
ETL	Medium	`Map` of `List`, Tuple
Grade School	Medium+	`Map` of `List`

☝️ Pre-requisites:

→ Create an account, with GitHub for instance
→ Solve the first exercises to unlock the access to the one above

💡 Tips:

→ `string` is a `Seq<char>`
→ What about `Seq.chunkBySize`?

Additional resources

All functions, with their cost in O(?)
https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/fsharp-collection-types#table-of-functions

Choosing between collection functions (2015)
https://fsharpforfunandprofit.com/posts/list-module-functions/

An introduction to F# for curious C# developers - Working with collections
https://laenas.github.io/posts/01-fs-primer.html#work-with-collections

Formatting collections
https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/formatting#formatting-lists-and-arrays

Thanks 🙏