

F# Training

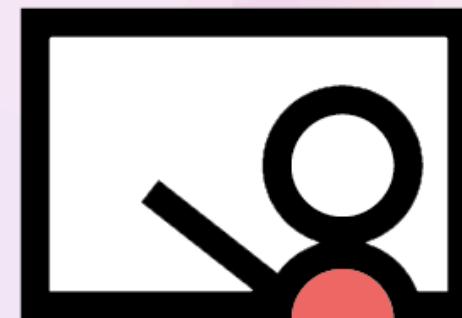
Asynchronous programming

2025 April

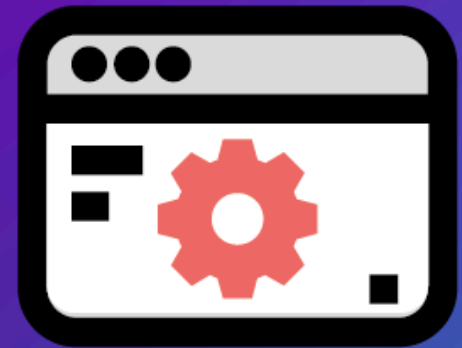


Table of contents

- Asynchronous workflow
- Interop with .NET TPL

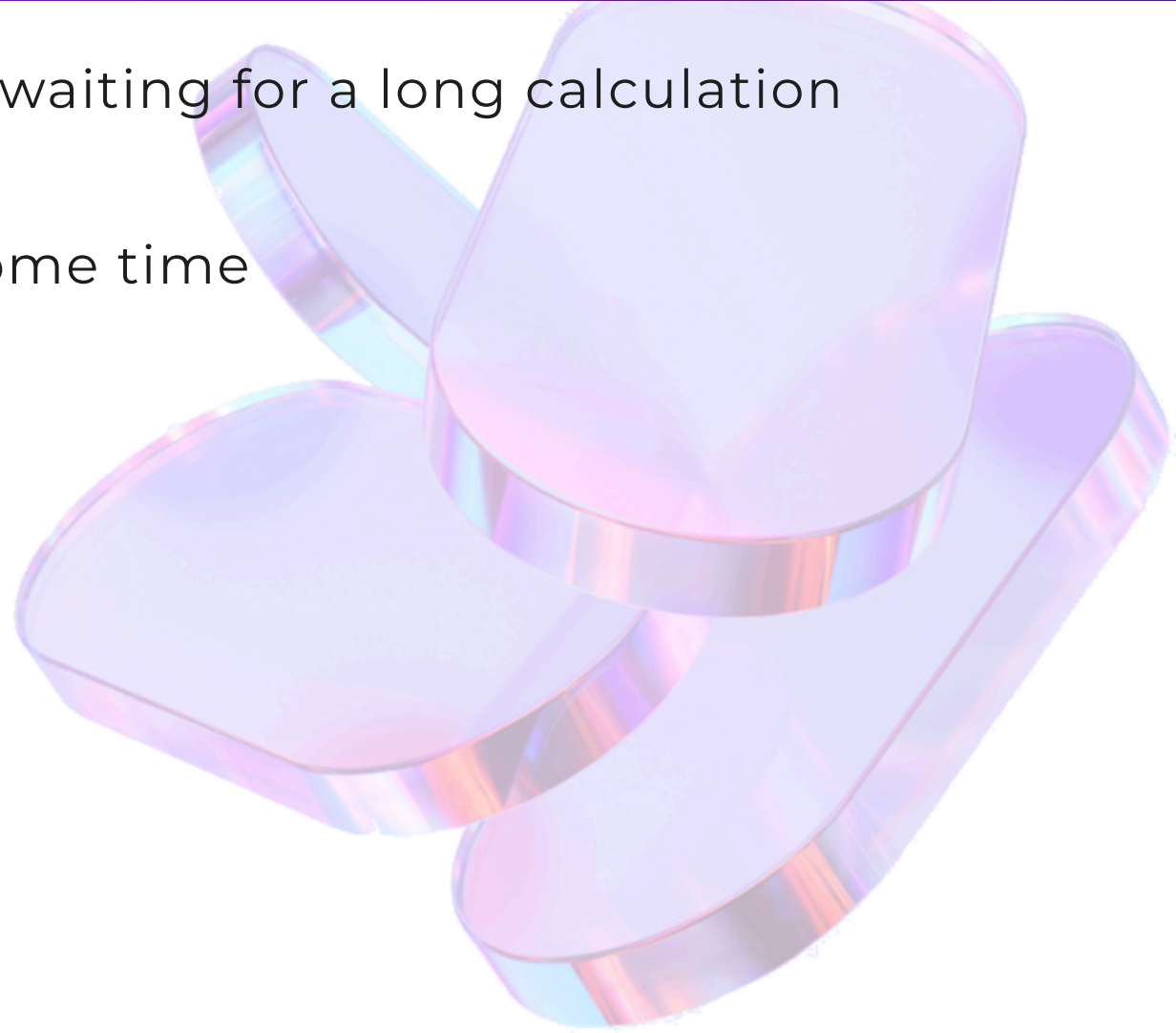


1. Asynchronous workflow



Asynchronous Workflow: Purpose

1. Do not block the current thread while waiting for a long calculation
2. Allow parallel calculations
3. Indicate that a calculation may take some time



Async<'T> type

Represents an asynchronous calculation

 Similar to the `async/await` pattern, way before C# and JS

- 2007: `Async<'T>` F#
- 2012: `Task<T>` .NET and pattern `async/await`
- 2017: `Promise` JavaScript and pattern `async/await`



Methods returning an Async object

`Async.AwaitTask(task : Task or Task<'T>) : Async<'T>`

→ Convert a `Task` (.NET) to `Async` (F#)

`Async.Sleep(milliseconds or TimeSpan) : Async<unit>`

≈ `await Task.Delay()` ≠ `Thread.Sleep` → does not block current thread

FSharp.Control `CommonExtensions` module: extends the `System.IO.Stream` type ([doc](#))

→ `AsyncRead(buffer: byte[], ?offset: int, ?count: int) : Async<int>`

→ `AsyncWrite(buffer: byte[], ?offset: int, ?count: int) : Async<unit>`

FSharp.Control `WebExtensions` module: extends type `System.Net.WebClient` ([doc](#))

→ `AsyncDownloadData(address : Uri) : Async<byte[]>`

→ `AsyncDownloadString(address : Uri) : Async<string>`

Run an async calculation

```
Async.RunSynchronously(calc: Async<'T>, ?timeoutMs: int, ?cancellationToken) : 'T
```

→ Waits for the calculation to end, blocking the calling thread! (≠ `await` C#) ⚠

```
Async.Start(operation: Async<unit>, ?cancellationToken) : unit
```

→ Performs the operation in background (*without blocking calling thread*)

⚠ If an exception occurs, it is "swallowed"!

```
Async.StartImmediate(calc: Async<'T>, ?cancellationToken) : unit
```

→ Performs the calculation in the calling thread!

💡 Useful in a GUI to update it: progress bar...

```
Async.StartWithContinuations(calc, continuations ... , ?cancellationToken)
```

→ Like `Async.RunSynchronously` ⚠ ... with 3 continuation *callbacks*:

→ on success ✅, exception 💣 and cancellation 🛑

`async { expression } block`

A.k.a. Async workflow

Syntax for sequentially writing an asynchronous calculation
→ The result of the calculation is wrapped in an `Async` object

Key words

- `return` → final value of calculation • `unit` if omitted
- `let!` → access to the result of an async sub-calculation (\approx `await` in C#)
- `use!` → like `use` (*management of an `IDisposable`*) + `let!`
- `do!` → like `let!` for async calculation without return (`Async<unit>`)

async block - Examples

```
let repeat (computeAsync: int → Async<string>) times = async {  
    for i in [ 1..times ] do  
        printf $"Start operation #{i} ... "  
        let! result = computeAsync i  
        printfn $"Result: {result}"  
}
```

```
let basicOp (num: int) = async {  
    do! Async.Sleep 150  
    return $" {num} * ({num} - 1) = {num * (num - 1)}"  
}
```

```
repeat basicOp 5 ▷ Async.RunSynchronously
```

```
// Start operation #1... Result: 1 * (1 - 1) = 0  
// Start operation #2... Result: 2 * (2 - 1) = 2  
// Start operation #3... Result: 3 * (3 - 1) = 6  
// Start operation #4... Result: 4 * (4 - 1) = 12  
// Start operation #5... Result: 5 * (5 - 1) = 20
```

Inappropriate use of `Async.RunSynchronously`

`Async.RunSynchronously` runs the calculation and returns the result BUT blocks the calling thread! Use it only at the "end of the chain" and not to *unwrap* intermediate asynchronous calculations! Use an `async` block instead.

```
// ❌ Avoid
let a = calcA ▷ Async.RunSynchronously
let b = calcB a ▷ Async.RunSynchronously
calcC b
```

```
// ✅ Favor
async {
    let! a = calcA
    let! b = calcB a
    return calcC b
}
▷ Async.RunSynchronously
```

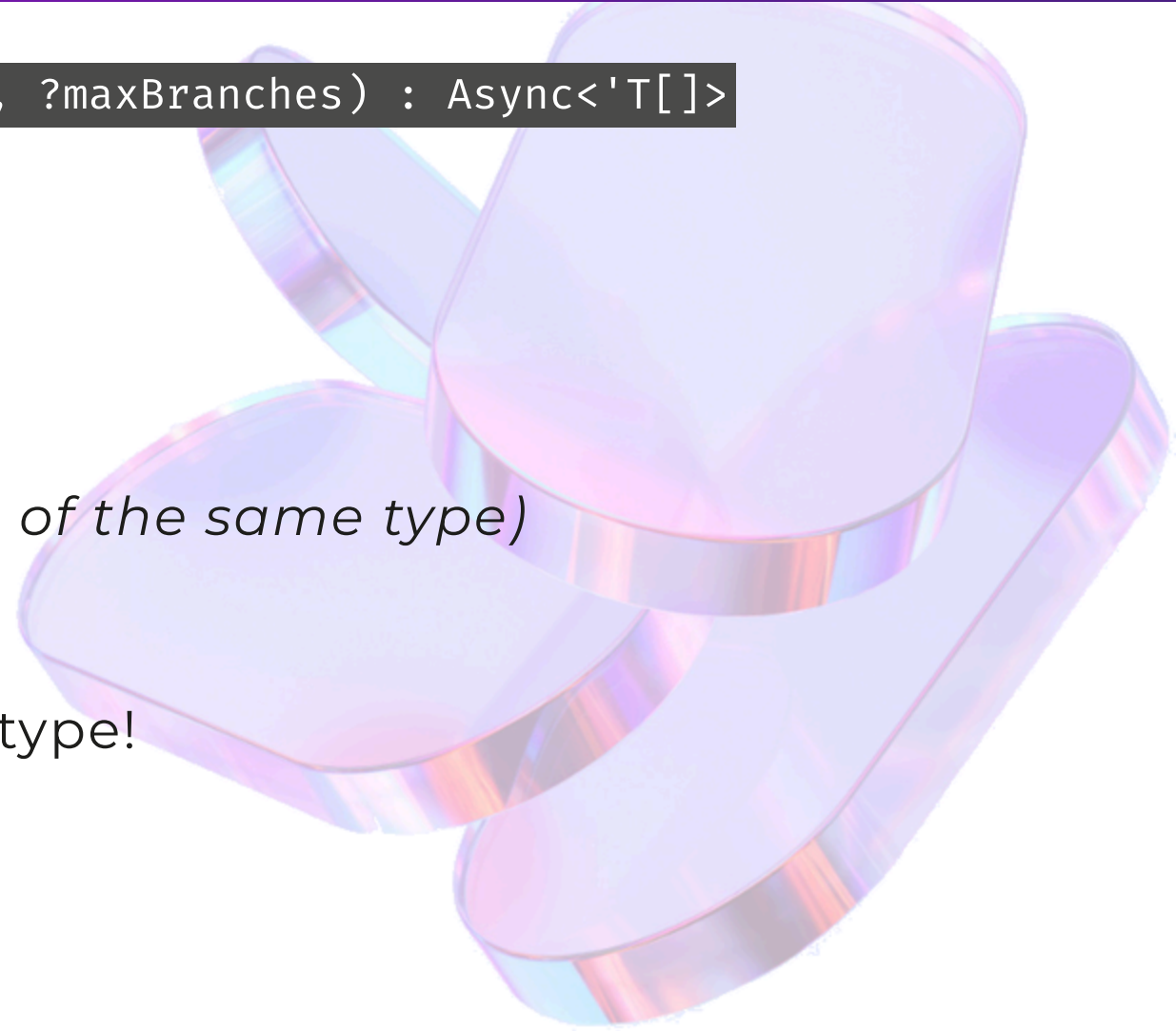
Parallel calculations

1. `Async.Parallel(computations: seq<Async<'T>>, ?maxBranches) : Async<'T[]>`

≈ `Task.WhenAll` : [Fork-Join model](#)

- *Fork*: calculations run in parallel
- Wait for all calculations to finish
- *Join*: aggregation of results (*which are of the same type*)
 - in the same order as calculations

⚠ All calculations must return the same type!



Async.Parallel - Example

```
let downloadSite (site: string) = async {  
    do! Async.Sleep (100 * site.Length)  
    printfn $"{site} ✓"  
    return site.Length  
}  
  
[ "google"; "msn"; "yahoo" ]  
▷ List.map downloadSite // string list  
▷ Async.Parallel        // Async<string[]>  
▷ Async.RunSynchronously // string[]  
▷ printfn "%A"  
  
// msn ✓  
// yahoo ✓  
// google ✓  
// [6; 3; 5]
```

Parallel calculations (2)

2. `Async.StartChild(calc: Async<'T>, ?timeoutMs: int) : Async<Async<'T>>`

Allows several calculations to be run in parallel

→ ... whose results are of different types (\neq `Async.Parallel`)

Used in `async` block with 2 `let!` per child calculation (cf. `Async<Async<'T>>`)

Shared cancellation

→ Child calculation shares cancellation token with its parent calculation

Async.StartChild - Example part 1

Let's first define a function `delay`
→ which returns the specified value `x`
→ after `ms` milliseconds

```
let delay (ms: int) x = async {  
    do! Async.Sleep ms  
    return x  
}  
  
// 💡 Timing with FSI directive `#time` - https://kutt.it/Zbp6ot  
#time "on" // → Timer start  
"a" ▷ delay 100 ▷ Async.RunSynchronously // Real: 00:00:00.111, CPU ...  
#time "off" // → Timer stop
```

Async.StartChild - Example part 2

```
let inSeries = async {
    let! result1 = "a" ▷ delay 100
    let! result2 = 123 ▷ delay 200
    return (result1, result2)
}

let inParallel = async {
    let! child1 = "a" ▷ delay 100 ▷ Async.StartChild
    let! child2 = 123 ▷ delay 200 ▷ Async.StartChild
    let! result1 = child1
    let! result2 = child2
    return (result1, result2)
}


#time "on"
inSeries ▷ Async.RunSynchronously // Real: 00:00:00.317, ...
#time "off"
#time "on"
inParallel ▷ Async.RunSynchronously // Real: 00:00:00.205, ...
#time "off"
```


Cancelling a task

Based on a default or explicit `CancellationToken/Source`:

- `Async.RunSynchronously(computation, ?timeout, ?cancellationToken)`
- `Async.Start(computation, ?cancellationToken)`

Trigger cancellation

- Explicit token + `cancellationTokenSource.Cancel()`
- Explicit token with timeout `new CancellationTokenSource(timeout)`
- Default token: `Async.CancelDefaultToken()` → `OperationCanceledException` 

Check cancellation

- Implicit: at each keyword in async block: `let`, `let!`, `for ...`
- Explicit local: `let! ct = Async.CancellationToken` then `ct.IsCancellationRequested`
- Explicit global: `Async.OnCancel(callback)`

Cancelling a task - Example Part 1

```
let sleepLoop = async {  
    let stopwatch = System.Diagnostics.Stopwatch()  
    stopwatch.Start()  
    let log message = printfn $" "    [{stopwatch.Elapsed.ToString("s\\.fff")}] {message}" "  
  
    use! __ = Async.OnCancel (fun () →  
        log "  Cancelled ✗")  
  
    for i in [ 1..5 ] do  
        log $"Step #{i} ... "  
        do! Async.Sleep 500  
        log $"  Completed ✓"  
    }  
  
    // ...
```

Cancelling a task - Example Part 2

```
// ...  
  
open System.Threading  
  
printfn "1. RunSynchronously:"  
Async.RunSynchronously(sleepLoop)  
  
printfn "2. Start with CancellationTokenSource + Sleep + Cancel"  
use manualCancellationSource = new CancellationTokenSource()  
Async.Start(sleepLoop, manualCancellationSource.Token)  
Thread.Sleep(1200)  
manualCancellationSource.Cancel()  
  
printfn "3. Start with CancellationTokenSource with timeout"  
use cancellationByTimeoutSource = new CancellationTokenSource(1200)  
Async.Start(sleepLoop, cancellationByTimeoutSource.Token)
```

Cancelling a task - Example Outputs

```
1. RunSynchronously:
  [0.009] Step #1 ...
  [0.532]   Completed ✓
  [0.535] Step #2 ...
  [1.037]   Completed ✓
  [1.039] Step #3 ...
  [1.543]   Completed ✓
  [1.545] Step #4 ...
  [2.063]   Completed ✓
  [2.064] Step #5 ...
  [2.570]   Completed ✓

2. Start with CancellationTokenSource + Sleep + Cancel
  [0.000] Step #1 ...
  [0.505]   Completed ✓
  [0.505] Step #2 ...
  [1.011]   Completed ✓
  [1.013] Step #3 ...
  [1.234]   Cancelled ✗

3. Start with CancellationTokenSource with timeout
... idem 2.
```

2. Interop with .NET TPL



TPL: Task Parallel Library

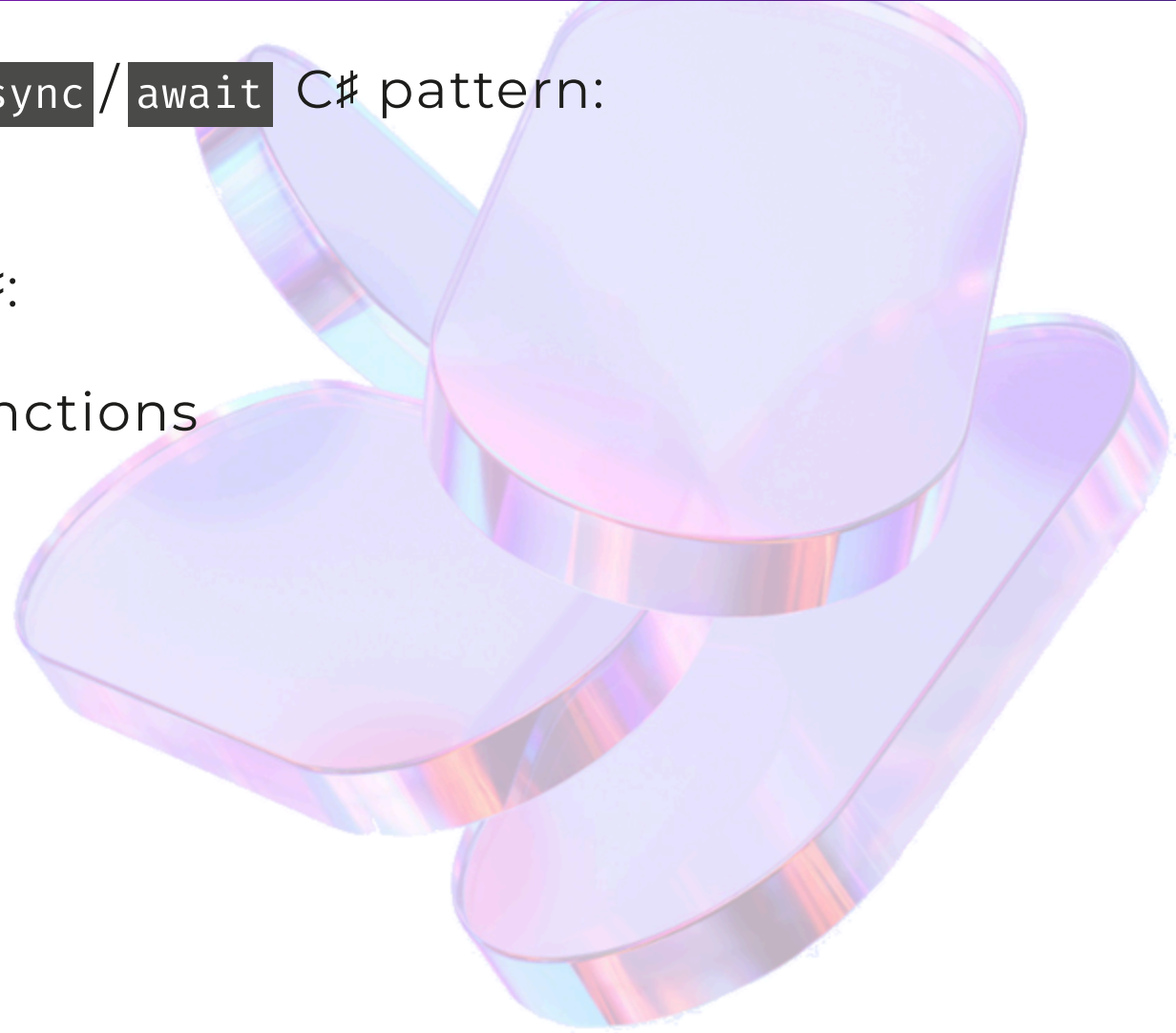
Interaction with .NET libraries

Asynchronous libraries in .NET and the `async` / `await` C# pattern:

→ Based on **TPL** and the `Task` type

Gateways with asynchronous workflow F#:

- `Async.AwaitTask` and `Async.StartAsTask` functions
- `task {}` block



Gateway functions

`Async.AwaitTask: Task<'T> → Async<'T>`

→ Consume an asynchronous .NET library in `async` block

`Async.StartAsTask: Async<'T> → Task<'T>`

→ Launch an async calculation as a `Task`

```
let getValueFromLibrary param = async {  
    let! value = DotNetLibrary.GetValueAsync param ▷ Async.AwaitTask  
    return value  
}  
  
let computationForCaller param =  
    async {  
        let! result = getAsyncResult param  
        return result  
    }  
    ▷ Async.StartAsTask
```


task {} block

“ Allows consuming an asynchronous .NET library directly, using a single `Async.AwaitTask` rather than one for each async method called. ”

💡 Available since F# 6 (before, we needed the [Ply](#) NuGet package)

```
task {  
    use client = new System.Net.Http.HttpClient()  
    let! response = client.GetStringAsync("https://www.google.fr/")  
    response.Substring(0, 300) ▷ printfn "%s"  
}  
// Task<unit>  
▷ Async.AwaitTask // Async<unit>  
▷ Async.RunSynchronously
```

Async VS Task

1. Calculation start mode

Task = *hot tasks* → calculations started immediately !

Async = *task generators* = calculation specification, independent of startup
→ Functional approach: no side-effects or mutations, composability
→ Control of startup mode: when and how 👍

2. Cancellation support

Task: by adding a **CancellationToken** parameter to async methods
→ Forces manual testing if token is canceled = tedious + *error-prone* !

Async: automatic support in calculations - token to be provided at startup 👍

Recommendation for async functions in F#

C# `async` applied at a method level

≠ F# `async` defines an async block, not an async function

👉 Recommendation:

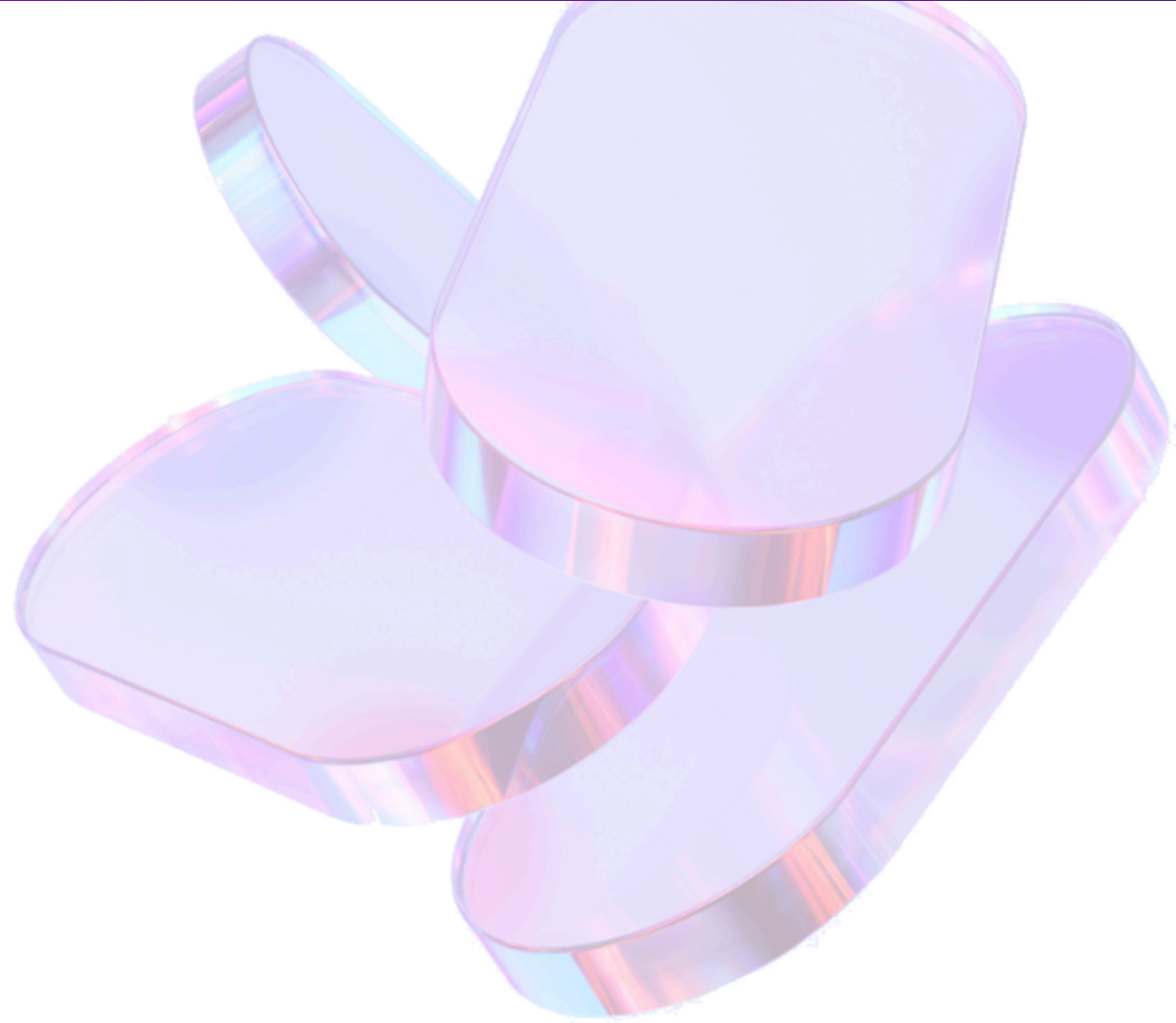
» Put the entire body of the async function in an `async` block.

```
// ❌ Avoid
let workThenWait () =
    Thread.Sleep(1000)
    async { do! Async.Sleep(1000) } // Async only in this block 🤔

// ✅ Prefer
let workThenWait () = async {
    Thread.Sleep(1000)
    printfn "work"
    do! Async.Sleep(1000)
}
```

Pitfalls of the `async` / `await` C# pattern

1. Really asynchronous?
2. Omit the `await`



Pitfall 1 - Really asynchronous?

In C#: method `async` remains on the calling thread until the 1st `await`
→ Misleading feeling of being asynchronous throughout the method

```
async Task WorkThenWait() {  
    Thread.Sleep(1000); // ⚠ Blocks calling thread !  
    await Task.Delay(1000); // Really async from here 🤔  
}
```

Pitfall 2 - Omit the `await`

```
async Task PrintAfterOneSecond(string message) {  
    await Task.Delay(1000);  
    Console.WriteLine($"[{DateTime.Now:T}] {message}");  
}  
  
async Task Main() {  
    PrintAfterOneSecond("Before"); // ⚠ Missing `await` → warning CS4014  
    Console.WriteLine($"[{DateTime.Now:T}] After");  
    await Task.CompletedTask;  
}
```

Compiles but returns unexpected result: *After* before *Before* !

```
[11:45:27] After  
[11:45:28] Before
```

Pitfall 2 - In F# too 😓

```
let printAfterOneSecond message = async {  
    do! Async.Sleep 1000  
    printfn $"{DateTime.Now:T} {message}"  
}  
  
async {  
    printAfterOneSecond "Before" // ⚠️ Missing `do!` → warning FS0020  
    printfn $"{DateTime.Now:T} After"  
} ▷ Async.RunSynchronously
```

Compiles but returns another unexpected result: no *Before* at all **!?**

```
[11:45:27] After
```


Pitfall 2 - Compilation warnings

The previous examples compile but with big *warnings*!

C# [warning CS4014](#) message:

“ *Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the `await` operator...* ”

F# [warning FS0020](#) message:

“ *The result of this expression has type `Async<unit>` and is implicitly ignored. Consider using `ignore` to discard this value explicitly...* ”

👉 **Recommendation:** be sure to **always** handle this type of warnings!
This is even more crucial in F# where compilation is tricky.

3. The Recap



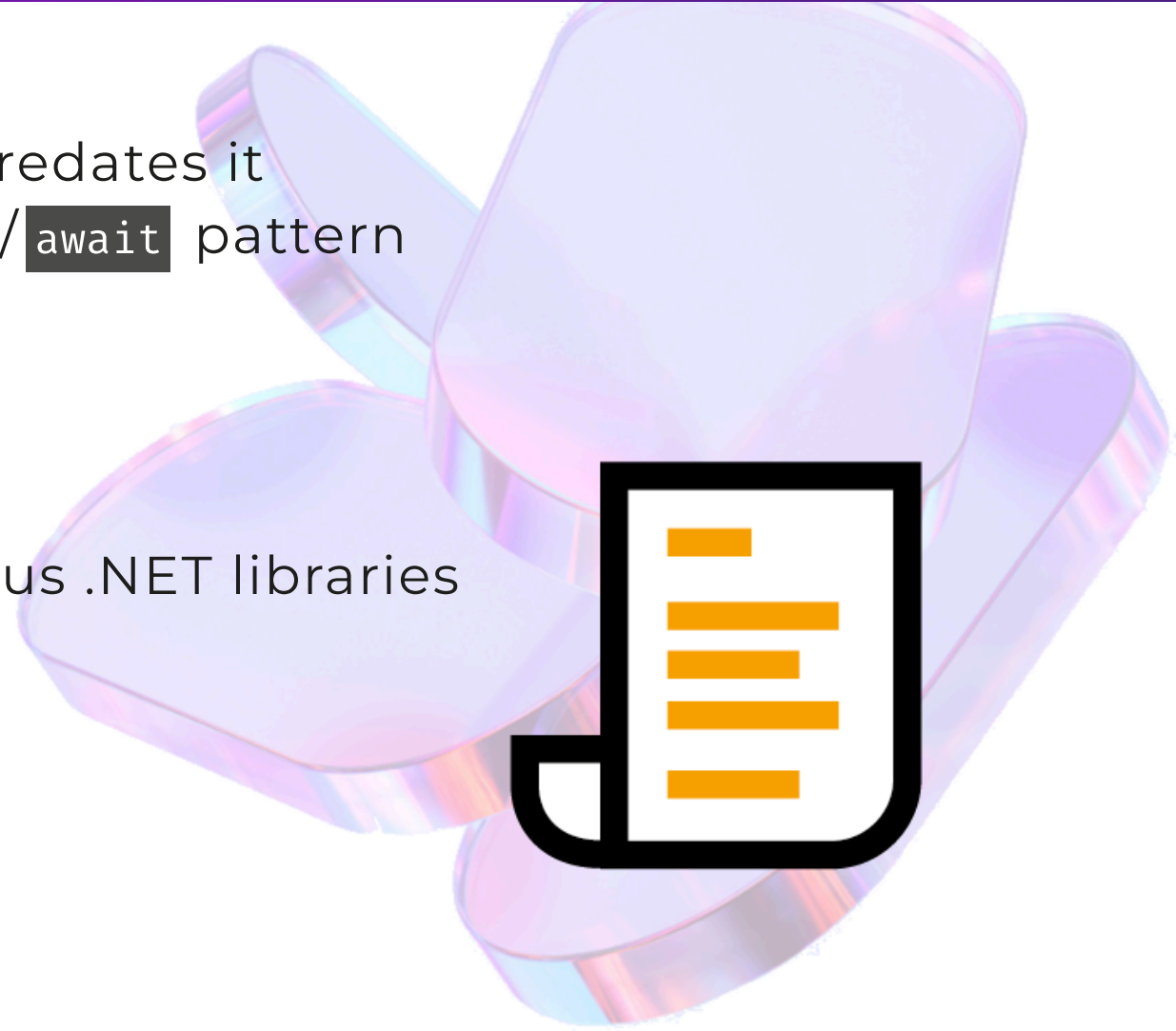
Asynchronous programming in F#

Via `async {}` block in pure F#

- Similar to C# `async/await` pattern but predates it
- Avoids some of the pitfalls of the `async/await` pattern
- Requires manual start of calculation
- But compilation prevents forgetting it

Via `task {}` block

- Facilitates interaction with asynchronous .NET libraries



Additional resources

<https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/async>



Thanks 🙏

