This cheatsheet aims to succinctly cover the most important aspects of F# 8.0. The Microsoft F# Documentation is complete and authoritative and has received a lot of love in recent years; it’s well worth the time investment to read. Only after you’ve got the lowdown here of course :)

If you have any comments, corrections, or suggested additions, please open an issue or send a pull request to https://github.com/fsprojects/fsharp-cheatsheet. Questions are best addressed via the F# slack or the F# discord.

Contents

- Comments
- Strings
- Basic Types and Literals
- Functions
- Collections
  - Lists
  - Arrays
  - Sequences
- Data Types
  - Tuples
  - Records
  - Anonymous Records
  - Discriminated Unions
- Pattern Matching
- Exceptions
- Classes and Inheritance
- Interfaces and Object Expressions
- Active Patterns
- Asynchronous Programming
- Code Organization
- Compiler Directives

Comments

Block comments are placed between (* and *). Line comments start from // and continue until the end of the line.

(* This is block comment *)

// And this is line comment

XML doc comments come after /// allowing us to use XML tags to generate documentation.

/// The `let` keyword defines an (immutable) value
let result = 1 + 1 = 2
Strings

F# string type is an alias for System.String type.

// Create a string using string concatenation
let hello = "Hello" + " World"

Use verbatim strings preceded by @ symbol to avoid escaping control characters (except escaping " by ").

let verbatimXml = @"<book title="Paradise Lost">

We don’t even have to escape " with triple-quoted strings.

let tripleXml = """"<book title="Paradise Lost"">"

Backslash strings indent string contents by stripping leading spaces.

let poem = "The lesser world was daubed
By a colorist of modest skill
A master limned you in the finest inks
And with a fresh-cut quill."

String Slicing is supported by using [start..end] syntax.

let str = "Hello World"
let firstWord = str[0..4] // "Hello"
let lastWord = str[6..] // "World"

String Interpolation is supported by prefixing the string with $ symbol. All of these will output "Hello \ World!":

let expr = "Hello"
printfn " \%s\n" \ World!" expr
printfn $" \%s\n" \ World!" $expr
printfn $" \%s\n" \ World!" $expr
printfn $" \%s\n" \ World!" $expr
printf $" \%s\n" \ World!" $expr

See Strings (MS Learn) for more on escape characters, byte arrays, and format specifiers.

Basic Types and Literals

Use the let keyword to define values. Values are immutable by default, but can be modified if specified with the mutable keyword.

let myStringValue = "my string"
let myIntValue = 10
let myExplicitlyTypedIntValue: int = 10
let mutable myMutableInt = 10
myMutableInt <- 11 // use <- arrow to assign a new value

**Integer Prefixes** for hexadecimal, octal, or binary

```plaintext
let numbers = (0x9F, 0o77, 0b1010)  // (159, 63, 10)
```

**Literal Type Suffixes** for integers, floats, decimals, and ascii arrays

```plaintext
let ( sbyte, byte ) = ( 55y, 55uy )  // 8-bit integer
let ( short, ushort ) = ( 50s, 50us )  // 16-bit integer
let ( int, uint ) = ( 50, 50u )  // 32-bit integer
let ( long, ulong ) = ( 50L, 50uL )  // 64-bit integer

let bigInt = 9999999999999I  // System.Numerics.BigInteger
let float = 50.0f  // signed 32-bit float
let double = 50.0  // signed 64-bit float
let scientific = 2.3E+32  // signed 64-bit float
let decimal = 50.0m  // signed 128-bit decimal
```

```plaintext
let byte = 'a'B  // ascii character; 97uy
let byteArray = "text"B  // ascii string; [|116uy; 101uy; 120uy; 116uy|]
```

**Primes** (or a tick ' at the end of a label name) are idiomatic to functional languages and are included in F#. They are part of the identifier’s name and simply indicate to the developer a variation of an existing value or function. For example:

```plaintext
let x = 5
let x' = x + 1
let x'' = x' + 1
```

See Literals (MS Learn) for complete reference.

**Functions**

Use the `let` keyword to define named functions.

```plaintext
let add n1 n2 = n1 + n2
let subtract n1 n2 = n1 - n2
let negate num = -1 * num
```
let print num = printfn $"The number is: {num}"

Pipe and Composition Operators

Pipe operator |> is used to chain functions and arguments together.

let addTwoSubtractTwoNegateAndPrint num =
    num |> add 2 |> subtract 2 |> negate |> print

Composition operator >> is used to compose functions:

let addTwoSubtractTwoNegateAndPrint' =
    add 2 >> subtract 2 >> negate >> print

Caution: The output is the last argument to the next function.

// `addTwoSubtractTwoNegateAndPrint 10` becomes:
10
    |> add 2    // 2 + 10 = 12
    |> subtract 2 // 2 - 12 = -10
    |> negate   // -1 * -10 = 10
    |> print    // "The number is 10"

unit Type

The unit type is a type that indicates the absence of a specific value. It is represented by (). The most common use is when you have a function that receives no parameters, but you need it to evaluate on every call:

let appendSomeTextToFile () = // without unit, only one line would be appended to the file
    System.IO.File.AppendAllText($"{__SOURCE_DIRECTORY__}/file.txt", "New line")

Signatures and Explicit Typing

Function signatures are useful for quickly learning the input and output of functions. The last type is the return type and all preceding types are the input types.

int -> string    // this defines a function that receives an integer; returns a string
int -> int -> string    // two integer inputs; returns a string
unit -> string    // unit; returns a string
string -> unit    // accepts a string; no return
(int * string) -> string -> string    // a tuple of int and string, and a string inputs; return a string

Most of the time, the compiler can determine the type of a parameter, but there are cases may you wish to be explicit or the compiler needs a hand. Here is a function with a signature string -> char -> int and the input and return types are explicit:

let countWordsStartingWithLetter (theString: string) (theLetter: char) : int =
    theString.Split ' ',
Examples of functions that take `unit` as arguments and return different Collection types.

let getList (): int list = ... // unit -> int list
let getArray (): int[] = ...
let getSeq (): seq<int> = ...

A complex declaration with an Anonymous Record:

let anonRecordFunc (record: { Count: int; LeftAndRight: bigint * bigint |}) = ...

Recursive

The `rec` keyword is used together with the `let` keyword to define a recursive function:

let rec fact x =
  if x < 1 then 1
  else x * fact (x - 1)

Mutually recursive functions (those functions which call each other) are indicated by and keyword:

let rec even x =
  if x = 0 then true
  else odd (x - 1)

and odd x =
  if x = 0 then false
  else even (x - 1)

Statically Resolved Type Parameters

A *statically resolved type parameter* is a type parameter that is replaced with an actual type at compile time instead of at run time. They are primarily useful in conjunction with member constraints.

let inline add x y = x + y
let integerAdd = add 1 2
let floatAdd = add 1.0f 2.0f // without `inline` on `add` function, this would cause a type error

type RequestA = { Id: string; StringValue: string }
type RequestB = { Id: string; IntValue: int }

let requestA: RequestA = { Id = "A"; StringValue = "Value" }
let requestB: RequestB = { Id = "B"; IntValue = 42 }

let inline getId<T when T : (member Id: string)> (x: 'T) = x.Id

let idA = getId requestA // "A"
let idB = getId requestB // "B"

See Statically Resolved Type Parameters (MS Learn) and Constraints (MS Learn) for more examples.

Collections

Lists

A list is an immutable collection of elements of the same type. Implemented internally as a linked list.

// Create
let list1 = [ "a"; "b" ]
let list2 = [ 1
2 ]
let list3 = "c" :: list1 // prepending; [ "c"; "a"; "b" ]
let list4 = list1 @ list3 // concat; [ "a"; "b"; "c"; "a"; "b" ]
let list5 = [ 1..2..9 ] // start..increment..last; [ 1; 3; 5; 7; 9 ]

// Slicing is inclusive
let firstTwo = list5[0..1] // [ 1; 3 ]

// Pattern matching
match myList with
| [] -> ...
| [ 3 ] -> ...
| [ _; 4 ] -> ...
| head :: tail -> ...

// Tail-recursion with a list, using cons pattern
let sumEachItem (myList:int list) = 
match myList with
| [] -> 0
| head :: tail -> head + sumEachItem tail

See the List Module for built-in functions.

Arrays

Arrays are fixed-size, zero-based, collections of consecutive data elements maintained as one block of memory. They are mutable; individual elements can be
changed.

// Create
let array1 = [| "a"; "b"; "c" |]
let array2 =
  [| 1
     2 |]
let array3 = [| 1..2..9 |] // start..increment..last; [| 1; 3; 5; 7; 9 |]

// Indexed access
let first = array1[0] // "a"

// Slicing is inclusive; [| "a"; "b" |]
let firstTwo = array1[0..1]

// Assignment using `<-`
array1[1] <- "d" // [| "a"; "d"; "c" |]

// Pattern matching
match myArray with
| [] -> ... // match an empty array
| [ | 3 |] -> ... // match array with single 3 item
| [ | _; 4 |] -> ... // match array with 2 items, second item = 4

See the Array Module for built-in functions.

Sequences

A sequence is a logical series of elements of the same type. seq<'t> is an alias for System.Collections.Generic.IEnumerable<'t>.

// Create
let seq1 = { 1; 2 }
let seq2 = seq {
  1
  2 }
let seq3 = seq { 1..2..9 } // start..increment..last; 1,3,5,7,9

See the Seq Module for built-in functions.

Collection comprehension

-Computed expressions with ->. Results in 1, 3, 5, 7, 9

  let listComp = [ for i in 0..4 -> 2 * i + 1 ]
  let arrayComp = [| for i in 0..4 -> 2 * i + 1 |]
  let seqComp = seq { for i in 0..4 -> 2 * i + 1 }
• Using computed expressions with `yield` and `yield!`. (`yield` is optional in a `do`, but is being used explicitly here):

```fsharp
let comprehendedList = [ // [ 1;3;5;7;9 ]
    for i in 0..4 do
        yield 2 * i + 1
]
let comprehendedArray = [\ // [\ 1;3;5;7;9;1;3;5;7;9 ]\]
    for i in 0..4 do
        yield 2 * i + 1
    yield! listWithYield
]
let comprehendedSequence = seq { // seq { 1;3;5;7;9;1;3;5;7;9;.... }
    while true do
        yield! listWithYield
}
```

Data Types

Tuples

A tuple is a grouping of unnamed but ordered values, possibly of different types:

```fsharp
// Construction
let numberAndWord = (1, "Hello")
let numberAndWordAndNow = (1, "Hello", System.DateTime.Now)

// Deconstruction
let (number, word) = numberAndWord
let (_, _, now) = numberAndWordAndNow

// fst and snd functions for two-item tuples:
let number = fst numberAndWord
let word = snd numberAndWord

// Pattern matching
let printNumberAndWord numberAndWord =
    match numberAndWord with
    | (1, word) -> printfn "$1: %s{word}"
    | (2, word) -> printfn "$2: %s{word}"
    | (_, word) -> printfn "$Number: %s{word}"

// Function parameter deconstruction
let printNumberAndWord' (number, word) = printfn "$\%(number): %s{word}"`
let (success, outParsedDateTime) = System.DateTime.TryParse("2001/02/06")
See Tuples (MS Learn) for learn more.

Records

*Records* represent aggregates of named values. They are sealed classes with extra toppings: default immutability, structural equality, and pattern matching support.

// Declare
type Person = { Name: string; Age: int }
type Car =
  { Make: string
    Model: string
    Year: int }

// Create
let paul = { Name = "Paul"; Age = 28 }

// Copy and Update
let paulsTwin = { paul with Name = "Jim" }

// Built-in equality
let evilPaul = { Name = "Paul"; Age = 28 }
paul = evilPaul // true

// Pattern matching
let isPaul person =
  match person with
  | { Name = "Paul" } -> true
  | _ -> false
See Records (MS Learn) to learn more; including *struct*-based records.

Anonymous Records

*Anonymous Records* represent aggregates of named values, but do not need declaring before use.

// Create
let anonRecord1 = { Name = "Don Syme"; Language = "F#"; Age = 999 }

// Copy and Update
let anonRecord2 = { anonRecord1 with Name = "Mads Torgersen"; Language = "C#" }

let getCircleStats (radius: float) =
  { Radius = radius }
Diameter = radius * 2.0
Area = System.Math.PI * (radius ** 2.0)
Circumference = 2.0 * System.Math.PI * radius |

// Signature
let printCircleStats (circle: {| Radius: float; Area: float; Circumference: float; Diameter: float |}) =
    printfn $"Circle with R=%f{circle.Radius}; D=%f{circle.Diameter}; A=%f{circle.Area}; C=%f{circle.Circumference}" |

let cc = getCircleStats 2.0
printCircleStats cc

See Anonymous Records (MS Learn) to learn more; including struct-based anonymous records.

**Discriminated Unions**

Discriminated unions (DU) provide support for values that can be one of a number of named cases, each possibly with different values and types.

// Declaration
type Interaction =
    | Keyboard of char
    | KeyboardWithModifier of char * modifier: System.ConsoleModifiers
    | MouseClick of countOfClicks: int

// Create
let interaction1 = MouseClick 1
let interaction2 = MouseClick (countOfClicks = 2)
let interaction3 = KeyboardWithModifier ('c', System.ConsoleModifiers.Control)

// Pattern matching
match interaction3 with
    | Keyboard chr -> $"Character: {chr}"$
    | KeyboardWithModifier (chr, modifier) -> $"Character: {modifier}+{chr}"$
    | MouseClick (countOfClicks = 1) -> "Click"
    | MouseClick (countOfClicks = x) -> $"Clicked: {x}"$

**Generics**

type Tree<'T> =
    | Node of Tree<'T> * 'T * Tree<'T>
    | Leaf

let rec depth =
    match depth with
    | Node (l, _, r) -> 1 + max (depth l) (depth r)
    | Leaf -> 0

10
F# Core has built-in discriminated unions for error handling, e.g., `option` and `Result`.

```fsharp
let optionPatternMatch input =  
    match input with  
    | Some value -> printfn $"input is %d{value}"  
    | None -> printfn "input is missing"

let resultPatternMatch input =  
    match input with  
    | Ok value -> $"Input: %d{value}"  
    | Error value -> $"Error: %d{value}"  
```

Single-case discriminated unions are often used to create type-safe abstractions with pattern matching support:

```fsharp
type OrderId = Order of string  
// Create a DU value  
let orderId = Order "12"  
// Use pattern matching to deconstruct single-case DU  
let (Order id) = orderId  // id = "12"
```

See Discriminated Unions to learn more.

**Pattern Matching**

Patterns are a core concept that makes the F# language and other MLs very powerful. They are found in `let` bindings, `match` expressions, lambda expressions, and exceptions.

The matches are evaluated top-to-bottom, left-to-right; and the first one to match is selected.

Examples of pattern matching in Collections and Data Types can be found in their corresponding sections. Here are some additional patterns:

```fsharp
match intValue with  
| 0 -> "Zero"  // constant pattern  
| 1 | 2 -> "One or Two"  // OR pattern with constants  
| x -> $"Something else: {x}"  // variable pattern; assign value to x
```

```fsharp
match tupleValue with  
| (_,3) & (x, y) -> $"{x}, 3"  // AND pattern with a constant and variable; matches 3 and a  
| _ -> "Wildcard"  // underscore matches anything
```
**when Guard clauses**

In order to match sophisticated inputs, one can use `when` to create filters, or guards, on patterns:

```fsharp
class MatchPatterns {
    public static void CheckNumbers(int num) {
        match num with
        | 0 -> 0
        | x when x < 0 -> -1
        | x -> 1
    }
}
```

**Pattern matching function**

The `let..match..with` statement can be simplified using just the `function` statement:

```fsharp
let filterNumbers num =
    match num with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
    | a -> printfn "%d" a

let filterNumbers' = // the parameter and `match num with` are combined
    function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
           | a -> printfn "%d" a
```

See Pattern Matching (MS Learn) to learn more.

**Exceptions**

**Try..With**

An illustrative example with: custom F# exception creation, all exception aliases, `raise()` usage, and an exhaustive demonstration of the exception handler patterns:

```fsharp
open System

exception MyException of int * string // (1)

let guard = true

try
    failwith "Message" // throws a System.Exception (aka exn)
    nullArg "ArgumentName" // throws a System.ArgumentNullException
    invalidArg "ArgumentName" "Message" // throws a System.ArgumentException
    invalidOp "Message" // throws a System.InvalidOperationException

    raise(NotImplementedException("Message")) // throws a .NET exception (2)
    raise(MyException(0, "Message")) // throws an F# exception (2)

    true // (3)
with
```
\[ \begin{align*}
| \text{ArgumentNullException} & \rightarrow \text{printfn "NullException"}; \text{false} // (3) \\
| \text{ArgumentException} \text{ as ex} & \rightarrow \text{printfn "$\{\text{ex.Message}\}$"}; \text{false} // (4) \\
| \text{InvalidOperationException} \text{ as ex when guard} & \rightarrow \text{printfn "$\{\text{ex.Message}\}$"}; \text{reraise()} // (5,6) \\
| \text{MyException} (\text{num, str}) \text{ when guard} & \rightarrow \text{printfn "$\{\text{num}, \{\text{str}\}\}$"}; \text{false} // (5) \\
| \text{MyException} (\text{num, str}) & \rightarrow \text{printfn "$\{\text{num}, \{\text{str}\}\}$"}; \text{reraise()} // (6) \\
| \text{ex when guard} & \rightarrow \text{printfn "$\{\text{ex.Message}\}$"}; \text{false} \\
| \text{ex} & \rightarrow \text{printfn "$\{\text{ex.Message}\}$"}; \text{false}
\end{align*} \]

(1) define your own F# exception types with exception, a new type that will inherit from System.Exception;
(2) use raise() to throw an F# or .NET exception;
(3) the entire try..with expression must evaluate to the same type, in this example: bool; (4)ArgumentNullException inherits from ArgumentException, so ArgumentException must follow after;
(4) support for when guards;
(5) use reraise() to re-throw an exception; works with both .NET and F# exceptions

The difference between F# and .NET exceptions is how they are created and how they can be handled.

Try..Finally

The try..finally expression enables you to execute clean-up code even if a block of code throws an exception. Here’s an example that also defines custom exceptions.

exception InnerError of string
exception OuterError of string

let handleErrors x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
            | InnerError str -> printfn "Error1 %s" str
        finally
            printfn "Always print this."
    Note that finally does not follow with. try..with and try..finally are separate expressions.

Classes and Inheritance

This example is a basic class with (1) local let bindings, (2) properties, (3) methods, and (4) static members.

type Vector(x: float, y: float) =
let mag = sqrt(x * x + y * y) // (1)
member _._X = x // (2)
member _._Y = y
member _._Mag = mag
member _._Scale(s) = // (3)
    Vector(x * s, y * s)
static member (+) (a : Vector, b : Vector) = // (4)

Call a base class from a derived one.
type Animal() =
    member _._Rest() = ()
type Dog() =
    inherit Animal()
    member _._Run() =
        base.Rest()

*Upcasting* is denoted by `:>` operator.

let dog = Dog()
let animal = dog :> Animal

*Dynamic downcasting* (`?:>`) might throw an `InvalidCastException` if the cast doesn’t succeed at runtime.

let shouldBeADog = animal :?> Dog

### Interfaces and Object Expressions

Declare `IVector` interface and implement it in `Vector`.

type IVector =
    abstract Scale : float -> IVector

type Vector(x, y) =
    interface IVector with
    member _._Scale(s) =
        Vector(x * s, y * s) :> IVector
    member _._X = x
    member _._Y = y

Another way of implementing interfaces is to use *object expressions*.

type ICustomer =
    abstract Name : string
    abstract Age : int

let createCustomer name age =
{ new ICustomer with
    member __.Name = name
    member __.Age = age }

Active Patterns
Single-case active patterns

// Basic
let (|EmailDomain|) email =
    let match = Regex.Match(email, "@(.*)$")
    if match.Success
    then match.Groups[1].ToString()
    else ""
let (EmailDomain emailDomain) = "yennefer@aretuza.org"  // emailDomain = 'aretuza.org'

// As Parameters
open System.Numerics
let (|Real|) (x: Complex) =
    (x.Real, x.Imaginary)
let addReal (Real (real1, _)) (Real (real2, _)) = // conversion done in the parameters
    real1 + real2
let addRealOut = addReal Complex.ImaginaryOne Complex.ImaginaryOne

// Parameterized
let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e
let (Default "random citizen" name) = None  // name = "random citizen"
let (Default "random citizen" name) = Some "Steve"  // name = "Steve"

Single-case active patterns can be thought of as a simple way to convert data to a new form.

Complete active patterns

let (|Even|Odd|) i =
    if i % 2 = 0 then Even else Odd

let testNumber i =
    match i with
    | Even -> printfn "%d is even" i
    | Odd -> printfn "%d is odd" i

let (|Phone|Email|) (s:string) =
    if s.Contains '@' then Email $"Email: {s}" else Phone $"Phone: {s}"
match "yennefer@aretuza.org" with // output: "Email: yennefer@aretuza.org"
| Email email -> printfn $"{email}"  
| Phone phone -> printfn $"{phone}"  

Partial active patterns

let (|DivisibleBy|_|) by n = 
        if n % by = 0 then Some DivisibleBy else None

let fizzBuzz = function  
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz" 
    | DivisibleBy 3 -> "Fizz" 
    | DivisibleBy 5 -> "Buzz" 
    | i -> string i

Partial active patterns share the syntax of parameterized patterns but their active recognizers accept only one argument.

Asynchronous Programming

F# asynchronous programming is centered around two core concepts: async computations and tasks.

async {
    do! Async.Sleep (waitInSeconds * 1000) 
    let! asyncResult = asyncComputation 
    use! disposableResult = iDisposableAsyncComputation
}

Async vs Tasks

An async computation is a unit of work, and a task is a promise of a result. A subtle but important distinction. Async computations are composable and are not started until explicitly requested; Tasks (when created using the task expression) are hot:

let runAsync waitInSeconds = 
    async {
        printfn "Created Async" 
        do! Async.Sleep (waitInSeconds * 1000) 
        printfn "$"Completed Async"
    }

let runTask waitInSeconds = 
    task {
        printfn "Started Task" 
        do! System.Threading.Tasks.Task.Delay (waitInSeconds * 1000) 
    }

16
printfn "$\text{Completed Task}"
}

let asyncComputation = runAsync 5 // returns Async<unit> and does not print anything
let newTask = runTask 3 // returns System.Threading.Tasks.Task<unit> and outputs: "$\text{Started Task}\$ <3\text{ second delay}> \text{Completed Task}"

asyncComputation |> Async.RunSynchronously // this now runs the async computation
newTask.Wait() // this will have already completed by this point

// Output:
// $\text{Started Task}$
// $\text{Created Async}$
// $\text{Completed Task}$
// Completed Async

Async and Task Interop

As F# sits in .NET, and a lot of the codebase uses the C# async/await, the majority of actions are going to be executed and tracked using System.Threading.Tasks.Task<T>es.

Async.AwaitTask  This converts a Task into an async computation. It has the signature: Task<T> -> Async<T>
async {
  let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
  let fileName = Path.GetFileName(path)
  printfn "$\text{File }\{\text{fileName}\} \text{ has }\%\{\text{bytes.Length}\} \text{ bytes}$"
}

Async.StartAsTask  This converts an async computation into a Task. It has the signature: Async<T> -> Task<T>
async {
  do! Async.Sleep 5000
} |> Async.StartAsTask

Async and the ThreadPool

Below is a demonstration of the different ways to start an async computation and where it ends up in the dotnet runtime. In the comments; M, X, Y, and Z are used to represent differences in values.

#r "nuget:Nito.AsyncEx, 5.1.2"

let asyncTask from =
  async {


let run () =
    printfn "run() Thread Id = {System.Threading.Thread.CurrentThread.ManagedThreadId}"
    asyncTask "StartImmediate" |> Async.StartImmediate  // run and wait on same thread
    asyncTask "Start" |> Async.Start  // queue in ThreadPool and do not wait
    asyncTask "RunSynchronously" |> Async.RunSynchronously  // run and wait; depends
    printfn ""

run ()
    // run() Thread Id = M - Main, non-ThreadPool thread
    // StartImmediate Thread Id = M - started, waited, and completed on run() thread
    // Start Thread Id = X - queued and completed in ThreadPool
    // RunSynchronously Thread Id = Y - started, waited, and completed in ThreadPool
    // Important: As `Start` is queued in the ThreadPool, it might finish before `RunSynchronously`

    // Run in ThreadPool
    async { run () } |> Async.RunSynchronously
    // run() Thread Id = X - ThreadPool thread
    // StartImmediate Thread Id = X - started, waited, and completed on run() thread
    // Start Thread Id = Y - queued and completed in new ThreadPool thread
    // RunSynchronously Thread Id = X - started, waited, and completed on run() thread
    // Important: `RunSynchronously` behaves like `StartImmediate` when on a ThreadPool thread without a SynchronizationContext

    // Run in ThreadPool with a SynchronizationContext
    async { Nito.AsyncEx.AsyncContext.Run run } |> Async.RunSynchronously
    // run() Thread Id = X - ThreadPool thread
    // StartImmediate Thread Id = X - started, waited, and completed on run() thread
    // Start Thread Id = Y - queued and completed in new ThreadPool thread
    // RunSynchronously Thread Id = Z - started, waited, and completed in new ThreadPool thread

### Code Organization

#### Modules

Modules are key building blocks for grouping related code; they can contain types, let bindings, or (nested) sub modules. Identifiers within modules can be referenced using dot notation, or you can bring them into scope via the open keyword. Illustrative-only example:

module Money =
    type CardInfo =
        { number: string
          expiration: int * int }

    type Payment =
module Functions =
    let validCard (cardNumber: string) =
        cardNumber.Length = 16 && (cardNumber[0], ['3'; '4'; '5'; '6']) ||> List.contains

If there is only one module in a file, the module name can be declared at the top, and all code constructs within the file will be included in the modules definition (no indentation required).

module Functions // notice there is no '=' when at the top of a file

let sumOfSquares n = seq {1..n} |> Seq.sumBy (fun x -> x * x) // Functions.sumOfSquares

Namespaces

Namespaces are simply dotted names that prefix type and module declarations to allow for hierarchical scoping. The first namespace directives must be placed at the top of the file. Subsequent namespace directives either: (a) create a sub-namespace; or (b) create a new namespace.

namespace MyNamespace

module MyModule = // MyNamespace.MyModule
    let myLet = ... // MyNamespace.MyModule.myLet

namespace MyNamespace.SubNamespace

namespace MyNewNamespace // a new namespace

A top-level module’s namespace can be specified via a dotted prefix:

module MyNamespace.SubNamespace.Functions

Open and AutoOpen

The open keyword can be used on module, namespace, and type.

module Groceries =
    type Fruit =
        | Apple
        | Banana

let fruit1 = Groceries.Banana
open Groceries // module
let fruit2 = Apple
open System.Diagnostics  // namespace
let stopwatch = Stopwatch.StartNew()  // Stopwatch is accessible

open type System.Text.RegularExpressions.Regex  // type
let isHttp url = IsMatch("https?:", url)  // Regex.IsMatch directly accessible

Available to module declarations only, is the AutoOpen attribute, which alleviates the need for an open.

[<AutoOpen>]
module Groceries =
    type Fruit =
        | Apple
        | Banana

let fruit = Banana

However, AutoOpen should be used cautiously. When an open or AutoOpen is used, all declarations in the containing element will be brought into scope. This can lead to shadowing; where the last named declaration replaces all prior identically-named declarations. There is no error - or even a warning - in F#, when shadowing occurs. A coding convention (MS Learn) exists for open statements to avoid pitfalls; AutoOpen would sidestep this.

Accessibility Modifiers

F# supports public, private (limiting access to its containing type or module) and internal (limiting access to its containing assembly). They can be applied to module, let, member, type, new (MS Learn), and val (MS Learn).

With the exception of let bindings in a class type, everything defaults to public.

<table>
<thead>
<tr>
<th>Element</th>
<th>Example with Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>module internal MyModule =</td>
</tr>
<tr>
<td>Module .. let</td>
<td>let private value =</td>
</tr>
<tr>
<td>Record</td>
<td>type internal MyRecord = { id: int }</td>
</tr>
<tr>
<td>Record ctor</td>
<td>type MyRecord = private { id: int }</td>
</tr>
<tr>
<td>Discriminated Union</td>
<td>type internal MyDiscUni = A \ B</td>
</tr>
<tr>
<td>Discriminated Union ctor</td>
<td>type MyDiscUni = private A | B</td>
</tr>
<tr>
<td>Class</td>
<td>type internal MyClass() =</td>
</tr>
<tr>
<td>Class ctor</td>
<td>type MyClass private () =</td>
</tr>
<tr>
<td>Class Additional ctor</td>
<td>internal new() = MyClass(&quot;defaultValue&quot;)</td>
</tr>
<tr>
<td>Class .. let</td>
<td>Always private. Cannot be overridden</td>
</tr>
<tr>
<td>Element</td>
<td>Example with Modifier</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><code>type .. member</code></td>
<td><code>member private _..TypeMember =</code></td>
</tr>
<tr>
<td><code>type .. val</code></td>
<td><code>val internal explicitInt : int</code></td>
</tr>
</tbody>
</table>

**Smart Constructors**

Making a primary constructor (ctor) `private` or `internal` is a common convention for ensuring value integrity; otherwise known as “making illegal states unrepresentable” (YouTube:Effective ML).

Example of Single-case Discriminated Union with a `private` constructor that constrains a quantity between 0 and 100:

```fsharp
type UnitQuantity =
    private UnitQuantity of int

module UnitQuantity = // common idiom: type companion module
    let tryCreate qty =
        if qty < 1 || qty > 100 then None
        else Some (UnitQuantity qty)
    let value (UnitQuantity uQty) = uQty
    let zero = UnitQuantity 0
...
let unitQtyOpt = UnitQuantity.tryCreate 5

let validQty =
    unitQtyOpt
    |> Option.defaultValue UnitQuantity.zero
```

**Recursive Reference**

F#’s type inference and name resolution runs in file and line order. By default, any forward references are considered errors. This default provides a single benefit, which can be hard to appreciate initially: you never need to look beyond the current file for a dependency. In general this also nudges toward more careful design and organisation of codebases, which results in cleaner, maintainable code. However, in rare cases forward referencing might be needed. To do this we have `rec` for `module` and `namespace`; and `and` for `type` and `let` (Recursive Functions) functions.

```fsharp
module rec CarModule

exception OutOfGasException of Car // Car not defined yet; would be an error

type Car =
```
{ make: string; model: string; hasGas: bool }
member self.Drive destination =
  if not self.hasGas
  then raise (OutOfGasException self)
  else ...

type Person =
  { Name: string; Address: Address }
and Address =
  { Line1: string; Line2: string; Occupant: Person }

See Namespaces (MS Learn) and Modules (MS Learn) to learn more.

Compiler Directives

Load another F# source file into FSI.
#load "../lib/StringParsing.fs"

Reference a .NET assembly (/ symbol is recommended for Mono compatibility).
Reference a .NET assembly:
#r "../lib/FSharp.Markdown.dll"

Reference a nuget package
#r "nuget:Serilog.Sinks.Console" // latest production release
#r "nuget:FSharp.Data, 6.3.0" // specific version
#r "nuget:Equinox, *-*" // latest version, including `-alpha`, `-rc` version etc

Include a directory in assembly search paths.
#I "../lib"
#r "FSharp.Markdown.dll"

Other important directives are conditional execution in FSI (INTERACTIVE) and
querying current directory (__SOURCE_DIRECTORY__).

#if INTERACTIVE
let path = __SOURCE_DIRECTORY__ + "../lib"
#else
let path = ".././.././lib"
#endif