

F# Cheatsheet

This cheatsheet aims to succinctly cover the most important aspects of F# 10.0.

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 ;)

This guide is a community effort. 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](#).

- [Comments](#)
- [Strings](#)
- [Basic Types and Literals](#)
- [Functions](#)
 - [let bindings](#)
 - [Pipe and Composition Operators](#)
 - [Anonymous functions](#)
 - [_.Property shorthand](#)
 - [unit Type](#)
 - [Signatures and Explicit Typing](#)
 - [Recursive Functions](#)
 - [TailCallAttribute](#)
 - [Mutually Recursive Functions](#)
 - [Statically Resolved Type Parameters](#)
- [Collections](#)
 - [Lists](#)
 - [Arrays](#)
 - [Sequences](#)
 - [Collection comprehension](#)
- [Data Types](#)
 - [Tuples](#)
 - [Records](#)
 - [Anonymous Records](#)
 - [Discriminated Unions](#)
- [Pattern Matching](#)
 - [when Guard clauses](#)
 - [Pattern matching function](#)
- [Exceptions](#)
 - [Try..With](#)
 - [Try..Finally](#)

- [Classes and Inheritance](#)
- [Interfaces and Object Expressions](#)
- [Active Patterns](#)
 - [Single-case active patterns](#)
 - [Complete active patterns](#)
 - [Partial active patterns](#)
- [Asynchronous Programming](#)
 - [.NET Tasks](#)
 - [Async Computations](#)
 - [Creation / Composition](#)
 - [Executing](#)
 - [Cancellation](#)
 - [.NET Tasks](#)
 - [Async](#)
 - [More to Explore](#)
- [Code Organization](#)
 - [Modules](#)
 - [Namespaces](#)
 - [Open](#)
 - [RequireQualifiedAccess Attribute](#)
 - [AutoOpen Attribute](#)
 - [Use AutoOpen and/or ignore RequireQualifiedAccess best practice with caution!](#)
 - [Accessibility Modifiers](#)
 - [Smart Constructors](#)
 - [Recursive Reference](#)
- [Compiler Directives](#)
 - [time](#)
 - [load](#)
 - [Referencing packages or assemblies in a script](#)
 - [Other important 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 a line comment
```

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

```
/// Double a number and add 1
let myFunction n = n * 2 + 1
```

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\n\
    By a colorist of modest skill\n\
    A master limned you in the finest inks\n\
    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" expr
printfn $" {expr}"
printfn $" {expr}" // using a format specifier
printfn $" {expr}"
printfn $" {expr}"
printfn $" {expr}" // no newline
```

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

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

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

```
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          = 9999999999999999999I // 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

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:

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

See [Literals \(MS Learn\)](#) for complete reference.

Functions

let bindings

Use the `let` keyword to define named functions.

```
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"
```

Anonymous functions

Anonymous, or "lambda" functions, are denoted by the `fun` keyword and the arrow operator `->`.

```
let isDescending xs =
    xs
    |> List.pairwise
    |> List.forAll (fun (x, y) -> x > y)

let suspiciousRecords =
```

```
records
|> Seq.filter (fun x -> x.Age >= 150)
```

_.Property shorthand

If the lambda function has a single argument that is used in an atomic expression, the following shorthand has been available since F# 8:

```
let names =
    people
    |> List.map (fun person -> person.Name)    // regular lambda expression

let names' =
    people
    |> List.map _.Name                        // _.Property shorthand
```

You may chain properties and methods together, so long as there is no "space" in the expression. E.g.:

```
let uppercaseNames =
    people |> List.map _.Name.ToUpperInvariant()
```

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:

```
// Without unit, DateTime.Now is only evaluated once. The return value will never change.
let getCurrentDateTime = DateTime.Now

// This version evaluates DateTime.Now every time you call it with a `unit` argument.
let getCurrentDateTime2 () = DateTime.Now

// How to call the function:
let startTime = getCurrentDateTime2 ()
```

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;
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; re
```

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 ' '
    |> Seq.where (fun (word: string) -> word.StartsWith theLetter) // explicit typing in
    |> Seq.length
```

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 Functions

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)
```

TailCallAttribute

In *tail recursive* functions, the recursive call is the final operation in the function, with its result directly returned without a nested function call (and the stack usage that implies). This pattern allows the compiler to instead generate a loop equivalent of the nested invocation by reusing the current stack frame instead of allocating a new one for each call.

As a guardrail, you can use the "TailCall" attribute (since F# 8).

By default, the compiler will emit a warning if this attribute is used with a function that is not properly tail recursive. It is typically a good idea to elevate this warning to an error, either in your project file, or by using a [compiler option](#).

If we add this attribute to the previous example:

```
[<TailCall>]
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)
```

...the compiler gives us this warning:

```
Warning FS3569 : The member or function 'fact' has the 'TailCallAttribute' attribute, but
```

However, when refactored to be properly tail recursive by using an accumulator parameter, the warning goes away:

```
[<TailCall>]
let rec factTail acc x =
    if x < 1 then acc
    else factTail (acc * x) (x - 1)
```

Mutually Recursive Functions

Pairs or groups of functions that call each other are indicated by both `rec` and `and` keywords:

```
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 ty
```

```
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

Lists are immutable collections 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
| [] -> ... // empty list
| [ 3 ] -> ... // a single item, which is '3'
| [ _; 4 ] -> ... // two items, second item is '4'
| head :: tail -> ... // cons pattern; matches non-empty. `head` is the first item, `tail` is the rest

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

See [Lists \(MS Learn\)](#) to learn more. 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 [Arrays \(MS Learn\)](#) to learn more. See the [Array Module](#) for built-in functions.

Sequences

A *sequence* is a logical series of elements of the same type. Sequences are lazily evaluated, and individual elements are computed only as required. `seq<'t>` is an alias for `System.Collections.Generic.IEnumerable<'t>`.

```

let emptySeq : int seq = seq { } // empty works as of F# 9
let seq1 = { 1; 2 }
let seq2 = seq {
    1
    2 }
let seq3 = seq { 1..2..9 } // start..increment..last; 1,3,5,7,9

```

See [Sequences \(MS Learn\)](#) to learn more. 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):

```

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! comprehendedList
    |]
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:

```
// 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 $"One: %s{word}"  
    | (2, word) -> printfn $"Two: %s{word}"  
    | (_, word) -> printfn $"Number: %s{word}"  
  
// Function parameter deconstruction  
let printNumberAndWord' (number, word) = printfn $"%d{number}: %s{word}"
```

In C#, if a method has an `out` parameter (e.g. `DateTime.TryParse`) the `out` result will be part of a tuple.

```
let (success, outParsedDateTime) = System.DateTime.TryParse("2001/02/06")
```

See [Tuples \(MS Learn\)](#) to 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; Diamet
    printfn $"Circle with R=%f{circle.Radius}; D=%f{circle.Diameter}; A=%f{circle.Area};

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

```

F# Core has built-in discriminated unions for error handling, e.g., [option](#) and [Result](#).

```

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

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

```

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

```

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:

```
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

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

when Guard clauses

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

```
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:

```
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:

```
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
| :? ArgumentNullException -> printfn "NullException"; false // (3)
| :? ArgumentException as ex -> printfn $"{ex.Message}"; false // (4)
| :? InvalidOperationException as ex when guard -> printfn $"{ex.Message}"; reraise () //
| MyException(num, str) when guard -> printfn $"{num}, {str}"; false // (5)
| MyException(num, str) -> printfn $"{num}, {str}"; reraise () //
| ex when guard -> printfn $"{ex.Message}"; false
| ex -> printfn $"{ex.Message}"; false
```

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;
5. support for `when` guards;
6. use `reraise ()` to re-throw the exception being handled (retaining the original throw site in the stack trace); 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)
    Vector(a.X + b.X, a.Y + b.Y)

```

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

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

```
// 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"
```

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

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

A partial active pattern typically returns an `Option<'T>`. However, as of [F# 9](#), where no value is being returned, and there is only one success case, you may return a `bool` instead.

- `Option<T>`

```
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
```

- `bool`

```
let (|DivisibleBy|_|) by n = n % by = 0
```

```
let (|EqualsIgnoreCase|_|) (pattern: string) (value: string) =
    String.Equals(value, pattern, StringComparison.OrdinalIgnoreCase)
```

See [Active Patterns \(MS Learn\)](#) to learn more.

Asynchronous Programming

F# asynchronous programming support consists of two complementary mechanisms::

- .NET's Tasks (via `task { }` expressions). This provides semantics very close to that of C#'s `async / await` mechanism, requiring explicit direct management of `CancellationToken`s.
- F# native `Async` computations (via `async { }` expressions). Predates `Task`. Provides intrinsic `CancellationToken` propagation.

.NET Tasks

In F#, .NET Tasks can be constructed using the `task { }` computational expression. .NET Tasks are "hot" - they immediately start running. At the first `let!` or `do!`, the `Task<T>` is returned and execution continues on the ThreadPool.

```
open System
open System.Threading
open System.Threading.Tasks
open System.IO

let readFile filename ct = task {
    printfn "Started Reading Task"
    do! Task.Delay((TimeSpan.FromSeconds 5), cancellationToken = ct) // use do! when awaiting a Task<T>
    let! text = File.ReadAllTextAsync(filename, ct) // use let! when awaiting a Task<T>
    return text
}

let readFileTask: Task<string> = readFile "myfile.txt" CancellationToken.None // (before
// (readFileTask continues execution on the ThreadPool)

let fileContent = readFileTask.Result // Blocks thread and waits for content. (1)
let fileContent' = readFileTask.Result // Task is already completed, returns same value
```

(1) `.Result` used for demonstration only. Read about [async/await Best Practices](#)

Async Computations

Async computations were invented before .NET Tasks existed, which is why F# has two core methods for asynchronous programming. However, async computations did not become obsolete. They offer another, but different, approach: dataflow. Async computations are constructed using `async { }` expressions, and the `Async` module is used to compose and execute them. In contrast to .NET Tasks, async expressions are "cold" (need to be explicitly started) and every execution [propagates a `CancellationToken` implicitly](#).

```

open System
open System.Threading
open System.IO

let readFile filename = async {
    do! Async.Sleep(TimeSpan.FromSeconds 5) // use do! when awaiting an Async
    let! text = File.ReadAllTextAsync(filename) |> Async.AwaitTask // (1)
    printfn "Finished Reading File"
    return text
}

// compose a new async computation from existing async computations
let readFiles = [ readFile "A"; readFile "B" ] |> Async.Parallel

// execute async computation
let textOfFiles: string[] = readFiles |> Async.RunSynchronously
// Out: Finished Reading File
// Out: Finished Reading File

// re-execute async computation again
let textOfFiles': string[] = readFiles |> Async.RunSynchronously
// Out: Finished Reading File
// Out: Finished Reading File

```

(1) As .NET Tasks became the central component of task-based asynchronous programming after F# Async were introduced, F#'s Async has `Async.AwaitTask` to map from `Task<'T>` to `Async<'T>`. Note that cancellation and exception handling require [special considerations](#).

Creation / Composition

The `Async` module has a number of functions to compose and start computations. The full list with explanations can be found in the [Async Type Reference](#).

Function	Description
<code>Async.Ignore</code>	Creates an <code>Async<unit></code> computation from an <code>Async<'T></code>
<code>Async.Parallel</code>	Composes a new computation from multiple computations, <code>Async<'T> seq</code> , and runs them in parallel; it returns all the results in an array <code>Async<'T[]></code>
<code>Async.Sequential</code>	Composes a new computation from multiple computations, <code>Async<'T> seq</code> , and runs them in series; it returns all the results in an array <code>Async<'T[]></code>
<code>Async.Choice</code>	Composes a new computation from multiple computations, <code>Async<'T option> seq</code> , and returns the first where <code>'T</code> is <code>Some value</code> (all others running are canceled). If all computations return <code>None</code> then the result is <code>None</code>

For all functions that compose a new computation from children, if any child computations raise an exception, then the overall computation will trigger an exception. The `CancellationToken` passed to the child computations will be triggered, and execution continues when all running children have cancelled execution.

Executing

Function	Description
<code>Async.RunSynchronously</code>	Runs an async computation and awaits its result.
<code>Async.StartAsTask</code>	Runs an async computation on the <code>ThreadPool</code> and wraps the result in a <code>Task<'T></code> .
<code>Async.StartImmediateAsTask</code>	Runs an async computation, starting immediately on the current operating system thread, and wraps the result in a <code>Task<'T></code> .
<code>Async.Start</code>	Runs an <code>Async<unit></code> computation on the <code>ThreadPool</code> (without observing any exceptions).
<code>Async.StartImmediate</code>	Runs a computation, starting immediately on the current thread and continuations completing in the <code>ThreadPool</code> .

Cancellation

.NET Tasks

.NET Tasks do not have any intrinsic handling of `CancellationToken`s; you are responsible for passing `CancellationToken`s down the call hierarchy to all sub-Tasks.

```
open System
open System.Threading
open System.Threading.Tasks

let loop (token: CancellationToken) = task {
    for cnt in [ 0 .. 9 ] do
        printf $"{cnt}: And..."
        do! Task.Delay((TimeSpan.FromSeconds 2), token) // token is required for Task.Delay
        printfn "Done"
    }

let cts = new CancellationTokenSource (TimeSpan.FromSeconds 5)
let runningLoop = loop cts.Token
try
    runningLoop.GetAwaiter().GetResult() // (1)
with :? OperationCanceledException -> printfn "Canceled"
```

Output:

```
0: And...Done
1: And...Done
2: And...Canceled
```

(1) `.GetAwaiter().GetResult()` used for demonstration only. Read about [async/await Best Practices](#)

Async

Asynchronous computations have the benefit of implicit `CancellationToken` passing and checking.

```
open System
open System.Threading
open System.Threading.Tasks

let loop = async {
    for cnt in [ 0 .. 9 ] do
        printf $"{cnt}: And..."
        do! Async.Sleep(TimeSpan.FromSeconds 1) // Async.Sleep implicitly receives and c

        let! ct = Async.CancellationToken // when interoperating with Tasks, cancellatio
        do! Task.Delay((TimeSpan.FromSeconds 1), cancellationToken = ct) |> Async.AwaitTa

        printfn "Done"
}

let cts = new CancellationTokenSource(TimeSpan.FromSeconds 5)
try
    Async.RunSynchronously (loop, Timeout.Infinite, cts.Token)
with :? OperationCanceledException -> printfn "Canceled"
```

Output:

```
0: And...Done
1: And...Done
2: And...Canceled
```

All methods for cancellation can be found in the [Core Library Documentation](#)

More to Explore

Asynchronous programming is a vast topic. Here are some other resources worth exploring:

- [Asynchronous Programming in F#](#) - Microsoft's tutorial guide. Recommended as it is up-to-date and expands on some of the topics here.
- [Iced Tasks](#) - .NET Tasks start immediately. The IcedTasks library provide additional [computational expressions](#) such as `cancellableTask`, which combines the benefits of .NET Tasks (natural interoperation with Task APIs and the performance benefits of the `task`'s State-Machine based implementation) with asynchronous expressions (composability, implicit `CancellationToken` passing, and the fact that you can invoke (or retry) a given computation multiple times).
- [Asynchronous Programming Best Practices](#) by David Fowler - offers a fantastic list of good practices for .NET Task usage.

Code Organization

Modules

Modules are key building blocks for grouping related code; they can contain `types` , `let` bindings, or (nested) sub `module` s. 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 =  
    | Card of CardInfo  
    | Cash of int  
  
  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 `module` s 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

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

```

RequireQualifiedAccess Attribute

This attribute can be used on modules, records, and discriminated unions to avoid namespace collisions and unexpected "shadowing". It can also make your code more explicit and readable.

```

[<RequireQualifiedAccess>]
module Calcs =
  let add x y = x + y
  let subtract x y = x - y

open Calcs // Error! Although you may open the containing namespace, if allowed.

// As you can't open the module, you must use qualified names for module members.
let sum = Calcs.add 5 3 // Works
let diff = subtract 5 3 // Error!

```

AutoOpen Attribute

Applicable to `module` declarations only, the `AutoOpen` attribute alleviates the need for an `open`.

```

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

let fruit = Banana

```

Use `AutoOpen` and/or ignore `RequireQualifiedAccess` best practice with caution!

Functions with identical names from different modules will silently "shadow" each other, causing the most recently imported definition to be used instead of the one you might expect. **The compiler will not**

warn you that this has happened. Even if your code works, you may encounter compiler errors if the `open` order changes. A [coding convention \(MS Learn\)](#) exists for `open` statements to avoid pitfalls.

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`.

Element	Example with Modifier
Module	<code>module internal MyModule =</code>
Module .. <code>let</code>	<code>let private value =</code>
Record	<code>type internal MyRecord = { id: int }</code>
Record <code>ctor</code>	<code>type MyRecord = private { id: int }</code>
Discriminated Union	<code>type internal MyDiscUni = A B</code>
Discriminated Union <code>ctor</code>	<code>type MyDiscUni = private A B</code>
Class	<code>type internal MyClass() =</code>
Class <code>ctor</code>	<code>type MyClass private () =</code>
Class Additional <code>ctor</code>	<code>internal new() = MyClass("defaultValue")</code>
Class .. <code>let</code>	<i>Always private. Cannot be overridden</i>
<code>type</code> .. <code>member</code>	<code>member private _.TypeMember =</code>
<code>type</code> .. <code>val</code>	<code>val internal explicitInt : int</code>

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:

```
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.

```

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

time

The `dotnet fs!` directive, `#time` switches on basic metrics covering real time, CPU time, and garbage collection information.

```

#time
System.Threading.Thread.Sleep (System.TimeSpan.FromSeconds 1)
#time

```

Output:

```
--> Timing now on
Real: 00:00:01.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it: unit = ()
--> Timing now off
```

load

Load another F# source file into FSI.

```
#load "../lib/StringParsing.fs"
```

Referencing packages or assemblies in a script

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

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
```