

# F# Cheatsheet

## 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.

```
/// Double-backticks are placed between a pair of ``  
let ``1 + 1 should be equal to 2``() =  
    1 + 1 = 2
```

## Strings

In F# `string` is the alias for `System.String` type.

```
/// Create a string using concatenation operator  
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* in F# 3.0.

```
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."
```

## Basic Types and Literals

Most of numeric types have associated suffixes e.g. `uy` for unsigned 8-bit integers and `L` for signed 64-bit integer.

```
let b, i, l = 86uy, 86, 86L
```

```
val l : int64 = 86L  
val i : int = 86  
val b : byte = 86uy
```

Other common examples are `F` or `f` for 32-bit floating-point numbers, `M` or `m` for decimals and `I` for big integers.

```
let s, f, d, bi = 4.14F, 4.14, 0.7833M, 9999I
```

```
val s : float32 = 4.14f  
val f : float = 4.14  
val d : decimal = 0.7833M  
val bi : System.Numerics.BigInteger = 9999
```

See Literals page on MSDN for complete reference.

## Arrays, Lists and Sequences

The same list `[ 1; 3; 5; 7; 9 ]` or array `[| 1; 3; 5; 7; 9 |]` can be generated in various ways.

- Using range operator `..`

```
let xs = [ 1..2..9 ]
```
- Using list or array comprehension

```
let ys = [| for i in 0..4 -> 2 * i + 1 |]
```
- Using `init` function

```
let zs = List.init 5 (fun i -> 2 * i + 1)
```

Lists and arrays have comprehensive sets of high-order functions for manipulation.

- `fold` starts from the left of the list (or array) and `foldBack` does the opposite

```
let xs = Array.fold (fun str n ->  
    sprintf "%s,%i" str n) "" [| 1..10 |]
```
- `reduce` doesn't require an initial accumulator

```
let last xs = List.reduce (fun acc x -> x) xs
```
- `map` an array by squaring all elements

```
let ys = Array.map (fun x -> x * x) [| 1.. 10 |]
```
- iterate through a list and produce side effects

```
List.iter (fun x -> printfn "%i" x) [ 0..9 ]
```

All the operations above are also available for sequences. The added values of sequences are laziness and uniform treatments for all collections implementing `IEnumerable<'T>`.

```
let zs =  
    seq { for i in 0..10 do  
        printfn "Adding %d" i  
        yield i }
```

## Pattern Matching

## Function Composition and Pipelining

## Tuples and Records

## Discriminated Unions

## 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 this.X = x // (2)  
    member this.Y = y  
    member this.Mag = mag  
    member this.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 casting* (`?:>`) might throw an exception if the cast doesn't succeed at runtime.

```
let probablyADog = 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 }
```

## Namespaces and Modules

## Async Workflows

## Active Patterns

*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
```

*Parameterized 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"  
    | _ -> ""
```

*Partial active patterns* have the same syntax as the parameterized one above but their active recognizers accept

only one argument.

## Compiler Directives

Load another F# source file into FSI.

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

Reference an .NET assembly (/ symbol is recommended for Mono compatibility).

```
#r "../lib/FSharp.Markdown.dll"
```

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
```