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 follow `///`, that allow developers 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 shortcut for `System.String` type.

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

Use *verbatim strings* preceding 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

Arrays, Lists and Sequences

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 executing in FSI (`INTERACTIVE`) and querying current directory (`__SOURCE_DIRECTORY__`).

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