Contents

What is F#

8/5/2022 • 2 minutes to read • Edit Online

F# is a universal programming language for writing succinct, robust and performant code.

F# allows you to write uncluttered, self-documenting code, where your focus remains on your problem domain, rather than the details of programming.

It does this without compromising on speed and compatibility - it is open-source, cross-platform and interoperable.

```
open System // Gets access to functionality in System namespace.

// Defines a list of names
let names = [ "Peter"; "Julia"; "Xi" ]

// Defines a function that takes a name and produces a greeting.
let getGreeting name = $"Hello, {name}"

// Prints a greeting for each name!
names
|> List.map getGreeting
|> List.iter (fun greeting -> printfn $"{greeting}! Enjoy your F#")
```

F# has numerous features, including:

- Lightweight syntax
- Immutable by default
- Type inference and automatic generalization
- First-class functions
- Powerful data types
- Pattern matching
- Async programming

A full set of features are documented in the F# language guide.

Rich data types

Types such as Records and Discriminated Unions let you represent your data.

```
// Group data with Records
type SuccessfulWithdrawal =
    { Amount: decimal
      Balance: decimal }

type FailedWithdrawal =
    { Amount: decimal
      Balance: decimal
      IsOverdraft: bool }

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure
```

F# records and discriminated unions are non-null, immutable, and comparable by default, making them very easy to use.

Correctness with functions and pattern matching

F# functions are easy to define. When combined with pattern matching, they allow you to define behavior whose correctness is enforced by the compiler.

```
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn $"Successfully withdrew %f{s.Amount}"
    | InsufficientFunds f -> printfn $"Failed: balance is %f{f.Balance}"
    | CardExpired d -> printfn $"Failed: card expired on {d}"
    | UndisclosedFailure -> printfn "Failed: unknown :("
```

F# functions are also first-class, meaning they can be passed as parameters and returned from other functions.

Functions to define operations on objects

F# has full support for objects, which are useful when you need to blend data and functionality. F# members and functions can be defined to manipulate objects.

```
type Set<'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) =// Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<‘T>
    interface IReadOnlyCollection<‘T>

module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)
```

In F#, you will often write code that treats objects as a type for functions to manipulate. Features such as generic interfaces, object expressions, and judicious use of members are common in larger F# programs.

Next steps

To learn more about a larger set of F# features, check out the F# Tour.

Get Started with F#

8/5/2022 • 2 minutes to read • Edit Online

You can get started with F# on your machine or online.

Get started on your machine

There are multiple guides on how to install and use F# for the first time on your machine. You can use the following table to help in making a decision:

OS	PREFER VISUAL STUDIO	PREFER VISUAL STUDIO CODE	PREFER COMMAND LINE
Windows	Get started with Visual Studio	Get started with Visual Studio Code	Get started with the .NET CLI
macOS	Get started with VS for Mac	Get started with Visual Studio Code	Get started with the .NET CLI
Linux	N/A	Get started with Visual Studio Code	Get started with the .NET CLI

In general, there is no specific way that is better than the rest. We recommend trying all ways to use F# on your machine to see what you like the best!

Get started online

If you'd rather not install F# and .NET on your machine, you can also get started with F# in the browser:

- Introduction to F# on Binder is a Jupyter notebook hosted via the free Binder service. No sign-up needed!
- The Fable REPL is an interactive, in-browser REPL that uses Fable to translate F# code into JavaScript. Check out the numerous samples that range from F# basics to a fully fledged video game all executing in your browser!

Install F#

8/5/2022 • 2 minutes to read • Edit Online

You can install F# in multiple ways, depending on your environment.

Install F# with Visual Studio

1. If you're downloading Visual Studio for the first time, it will first install Visual Studio Installer. Install the appropriate edition of Visual Studio from the installer.

   If you already have Visual Studio installed, choose **Modify** next to the edition you want to add F# to.

2. On the Workloads page, select the **ASP.NET and web development** workload, which includes F# and .NET Core support for ASP.NET Core projects.

3. Choose **Modify** in the lower right-hand corner to install everything you've selected.

   You can then open Visual Studio with F# by choosing **Launch** in Visual Studio Installer.

Install F# with Visual Studio Code

1. Ensure you have git installed and available on your PATH. You can verify that it's installed correctly by entering `git --version` at a command prompt and pressing **Enter**.

2. Install the .NET SDK and Visual Studio Code.

3. Select the Extensions icon and search for "Ionide":

   The only plugin required for F# support in Visual Studio Code is Ionide-fsharp. However, you can also install Ionide-FAKE to get FAKE support and Ionide-Paket to get Paket support. FAKE and Paket are additional F# community tools for building projects and managing dependencies, respectively.

Install F# with Visual Studio for Mac

F# is installed by default in Visual Studio for Mac, no matter which configuration you choose.

After the install completes, choose **Start Visual Studio**. You can also open Visual Studio through Finder on macOS.

Install F# on a build server

If you're using .NET Core or .NET Framework via the .NET SDK, you simply need to install the .NET SDK on your build server. It has everything you need.

If you're using .NET Framework and you are **not** using the .NET SDK, then you'll need to install the Visual Studio Build Tools SKU onto your Windows Server. In the installer, select **.NET desktop build tools**, and then select the **F# compiler** component on the right-hand side of the installer menu.

Get started with F# in Visual Studio

8/5/2022 • 2 minutes to read • Edit Online

F# is supported in the Visual Studio integrated development environment (IDE).

To begin, ensure that you have Visual Studio installed with F# support.

Create a console application

One of the most basic projects in Visual Studio is the console app. Here's how to create one:

1. Open Visual Studio 2019.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** page, choose **F#** from the Language list.
4. Choose the **Console App (.NET Core)** template, and then choose **Next**.
5. On the **Configure your new project** page, enter a name in the **Project name** box. Then, choose **Create**.

   Visual Studio creates the new F# project. You can see it in the Solution Explorer window.

Write the code

Let's get started by writing some code. Make sure that the `Program.fs` file is open, and then replace its contents with the following:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

The previous code sample defines a function called `square` that takes an input named `x` and multiplies it by itself. Because F# uses Type inference, the type of `x` doesn't need to be specified. The F# compiler understands the types where multiplication is valid and assigns a type to `x` based on how `square` is called. If you hover over `square`, you should see the following:

```
val square: x: int -> int
```

This is what is known as the function's type signature. It can be read like this: "Square is a function that takes an integer named x and produces an integer". The compiler gave `square` the `int` type for now.

Another function, `main`, is defined, which is decorated with the `EntryPoint` attribute. This attribute tells the F# compiler that program execution should start there. It follows the same convention as other C-style programming languages, where command-line arguments can be passed to this function, and an integer code is returned (typically `0`).

It is in the entry point function, `main`, that you call the `square` function with an argument of `12`. The F# compiler then assigns the type of `square` to be `int -> int` (that is, a function that takes an `int` and produces an `int`). The call to `printfn` is a formatted printing function that uses a format string and prints the result (and a new line). The format string, similar to C-style programming languages, has parameters (`%d`) that correspond to the arguments that are passed to it, in this case, `12` and `(square 12)`.

Run the code

You can run the code and see the results by pressing **Ctrl+F5**. Alternatively, you can choose the **Debug** > **Start Without Debugging** from the top-level menu bar. This runs the program without debugging.

The following output prints to the console window that Visual Studio opened:

```
12 squared is: 144!
```

Congratulations! You've created your first F# project in Visual Studio, written an F# function that calculates and prints a value, and run the project to see the results.

Next steps

If you haven't already, check out the Tour of F#, which covers some of the core features of F#. It provides an overview of some of the capabilities of F# and ample code samples that you can copy into Visual Studio and run.

Tour of F#

See also

- F# language guide
- Type inference
- Symbol and operator reference

Get Started with F# in Visual Studio Code

8/5/2022 • 6 minutes to read • Edit Online

You can write F# in Visual Studio Code with the Ionide plugin to get a great cross-platform, lightweight Integrated Development Environment (IDE) experience with IntelliSense and code refactorings. Visit Ionide.io to learn more about the plugin.

To begin, ensure that you have F# and the Ionide plugin correctly installed.

Create your first project with Ionide

To create a new F# project, open a command line and create a new project with the .NET CLI:

```
dotnet new console -lang "F#" -o FirstIonideProject
```

Once it completes, change directory to the project and open Visual Studio Code:

```
cd FirstIonideProject
code .
```

After the project loads in Visual Studio Code, you should see the F# Solution Explorer pane on the left-hand side of your window open. This means Ionide has successfully loaded the project you just created. You can write code in the editor before this point in time, but once this happens, everything has finished loading.

Write your first script

Once you've configured Visual Studio Code to use .NET Core scripting, navigate to the Explorer view in Visual Studio Code and create a new file. Name it *MyFirstScript.fsx*.

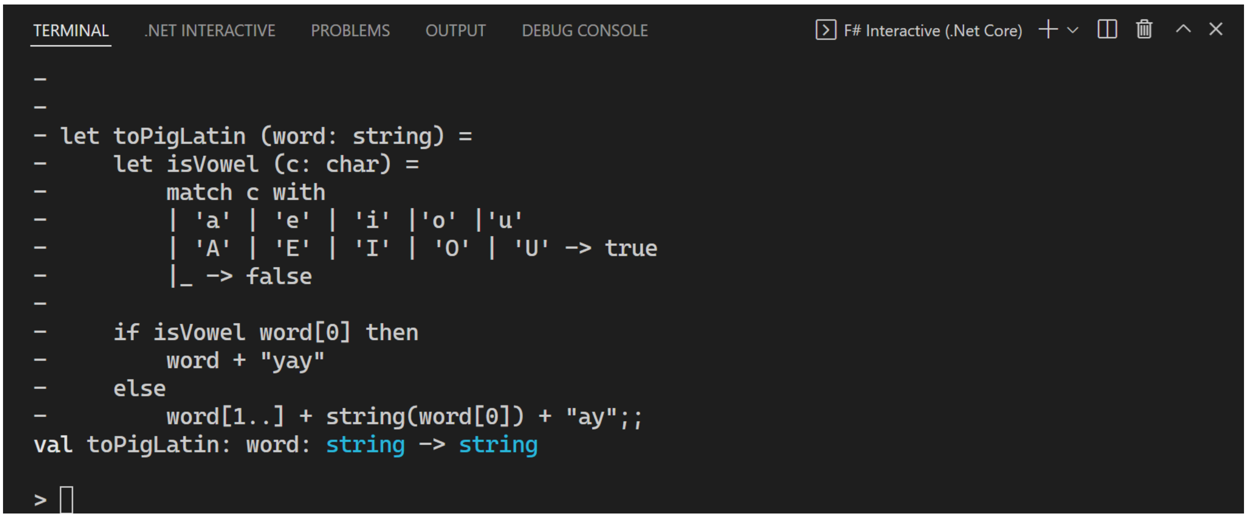
Now add the following code to it:

```
let toPigLatin (word: string) =
    let isVowel (c: char) =
        match c with
        | 'a' | 'e' | 'i' |'o' |'u'
        | 'A' | 'E' | 'I' | 'O' | 'U' -> true
        |_ -> false

    if isVowel word[0] then
        word + "yay"
    else
        word[1..] + string(word[0]) + "ay"
```

This function converts a word to a form of Pig Latin. The next step is to evaluate it using F# Interactive (FSI).

Highlight the entire function (it should be 11 lines long). Once it's highlighted, hold the `Alt` key and hit `Enter`. You'll notice a terminal window pop up on the bottom of the screen, and it should look similar to this:



This did three things:

1. It started the FSI process.
2. It sent the code you highlighted over to the FSI process.
3. The FSI process evaluated the code you sent over.

Because what you sent over was a function, you can now call that function with FSI! In the interactive window, type the following:

```
toPigLatin "banana";;
```

You should see the following result:

```
val it: string = "ananabay"
```

Now, let's try with a vowel as the first letter. Enter the following:

```
toPigLatin "apple";;
```

You should see the following result:

```
val it: string = "appleyay"
```

The function appears to be working as expected. Congratulations, you just wrote your first F# function in Visual Studio Code and evaluated it with FSI!

> **NOTE**
>
> As you may have noticed, the lines in FSI are terminated with `;;`. This is because FSI allows you to enter multiple lines. The `;;` at the end lets FSI know when the code is finished.

Explaining the code

If you're not sure about what the code is actually doing, here's a step-by-step.

As you can see, `toPigLatin` is a function that takes a word as its input and converts it to a Pig-Latin representation of that word. The rules for this are as follows:

If the first character in a word starts with a vowel, add "yay" to the end of the word. If it doesn't start with a vowel, move that first character to the end of the word and add "ay" to it.

You may have noticed the following in FSI:

```
val toPigLatin: word: string -> string
```

This states that `toPigLatin` is a function that takes in a `string` as input (called `word`), and returns another `string`. This is known as the type signature of the function, a fundamental piece of F# that's key to understanding F# code. You'll also notice this if you hover over the function in Visual Studio Code.

In the body of the function, you'll notice two distinct parts:

1. An inner function, called `isVowel`, that determines if a given character (`c`) is a vowel by checking if it matches one of the provided patterns via Pattern Matching:

   ```
   let isVowel (c: char) =
       match c with
       | 'a' | 'e' | 'i' |'o' |'u'
       | 'A' | 'E' | 'I' | 'O' | 'U' -> true
       |_ -> false
   ```

2. An `if..then..else` expression that checks if the first character is a vowel, and constructs a return value out of the input characters based on if the first character was a vowel or not:

   ```
   if isVowel word[0] then
       word + "yay"
   else
       word[1..] + string(word[0]) + "ay"
   ```

The flow of `toPigLatin` is thus:

Check if the first character of the input word is a vowel. If it is, attach "yay" to the end of the word. Otherwise, move that first character to the end of the word and add "ay" to it.

There's one final thing to notice about this: in F#, there's no explicit instruction to return from the function. This is because F# is expression-based, and the last expression evaluated in the body of a function determines the return value of that function. Because `if..then..else` is itself an expression, evaluation of the body of the `then` block or the body of the `else` block determines the value returned by the `toPigLatin` function.

Turn the console app into a Pig Latin generator

The previous sections in this article demonstrated a common first step in writing F# code: writing an initial function and executing it interactively with FSI. This is known as REPL-driven development, where REPL stands for "Read-Evaluate-Print Loop". It's a great way to experiment with functionality until you have something working.

The next step in REPL-driven development is to move working code into an F# implementation file. It can then be compiled by the F# compiler into an assembly that can be executed.

To begin, open the *Program.fs* file that you created earlier with the .NET CLI. You'll notice that some code is already in there.

Next, create a new `module` called `PigLatin` and copy the `toPigLatin` function you created earlier into it as such:

```
module PigLatin =
    let toPigLatin (word: string) =
        let isVowel (c: char) =
            match c with
            | 'a' | 'e' | 'i' |'o' |'u'
            | 'A' | 'E' | 'I' | 'O' | 'U' -> true
            |_ -> false

        if isVowel word[0] then
            word + "yay"
        else
            word[1..] + string word[0] + "ay"
```

This module should be above the `main` function and below the `open System` declaration. Order of declarations matters in F#, so you'll need to define the function before you call it in a file.

Now, in the `main` function, call your Pig Latin generator function on the arguments:

```
[<EntryPoint>]
let main args =
    for arg in args do
        let newArg = PigLatin.toPigLatin arg
        printfn "%s in Pig Latin is: %s" arg newArg

    0
```

Now you can run your console app from the command line:

```
dotnet run apple banana
```

And you'll see that it outputs the same result as your script file, but this time as a running program!

Troubleshooting Ionide

Here are a few ways you can troubleshoot certain problems that you might run into:

1. To get the code editing features of Ionide, your F# files need to be saved to disk and inside of a folder that is open in the Visual Studio Code workspace.
2. If you've made changes to your system or installed Ionide prerequisites with Visual Studio Code open, restart Visual Studio Code.
3. If you have invalid characters in your project directories, Ionide might not work. Rename your project directories if this is the case.
4. If none of the Ionide commands are working, check your Visual Studio Code Key Bindings to see if you're overriding them by accident.
5. If Ionide is broken on your machine and none of the above has fixed your problem, try removing the `ionide-fsharp` directory on your machine and reinstall the plugin suite.
6. If a project failed to load (the F# Solution Explorer will show this), right-click on that project and click **See details** to get more diagnostic info.

Ionide is an open-source project built and maintained by members of the F# community. Report issues and feel free to contribute at the ionide-vscode-fsharp GitHub repository.

You can also ask for further help from the Ionide developers and F# community in the Ionide Gitter channel.

Next steps

To learn more about F# and the features of the language, check out Tour of F#.

Get started with F# with the .NET CLI

8/5/2022 • 2 minutes to read • Edit Online

This article covers how you can get started with F# on any operating system (Windows, macOS, or Linux) with the .NET CLI. It goes through building a multi-project solution with a class library that is called by a console application.

Prerequisites

To begin, you must install the latest .NET SDK.

This article assumes that you know how to use a command line and have a preferred text editor. If you don't already use it, Visual Studio Code is a great option as a text editor for F#.

Build a simple multi-project solution

Open a command prompt/terminal and use the dotnet new command to create a new solution file called `FSharpSample`:

```
dotnet new sln -o FSharpSample
```

The following directory structure is produced after running the previous command:

```
FSharpSample
    ├── FSharpSample.sln
```

Write a class library

Change directories to *FSharpSample*.

Use the `dotnet new` command to create a class library project in the **src** folder named Library.

```
dotnet new classlib -lang "F#" -o src/Library
```

The following directory structure is produced after running the previous command:

```
└── FSharpSample
    ├── FSharpSample.sln
    └── src
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Replace the contents of `Library.fs` with the following code:

```
module Library

open System.Text.Json

let getJson value =
    let json = JsonSerializer.Serialize(value)
    value, json
```

Add the `Library` project to the `FSharpSample` solution using the dotnet sln add command:

```
dotnet sln add src/Library/Library.fsproj
```

Run `dotnet build` to build the project. Unresolved dependencies will be restored when building.

Write a console application that consumes the class library

Use the `dotnet new` command to create a console application in the `src` folder named App.

```
dotnet new console -lang "F#" -o src/App
```

The following directory structure is produced after running the previous command:

```
└── FSharpSample
    ├── FSharpSample.sln
    └── src
        ├── App
        │   ├── App.fsproj
        │   ├── Program.fs
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Replace the contents of the `Program.fs` file with the following code:

```
open System
open Library

[<EntryPoint>]
let main args =
    printfn "Nice command-line arguments! Here's what System.Text.Json has to say about them:"

    let value, json = getJson {| args=args; year=System.DateTime.Now.Year |}
    printfn $"Input: %0A{value}"
    printfn $"Output: %s{json}"

    0 // return an integer exit code
```

Add a reference to the `Library` project using dotnet add reference.

```
dotnet add src/App/App.fsproj reference src/Library/Library.fsproj
```

Add the `App` project to the `FSharpSample` solution using the `dotnet sln add` command:

```
dotnet sln add src/App/App.fsproj
```

Restore the NuGet dependencies with `dotnet restore` and run `dotnet build` to build the project.

Change directory to the `src/App` console project and run the project passing `Hello World` as arguments:

```
cd src/App
dotnet run Hello World
```

You should see the following results:

```
Nice command-line arguments! Here's what System.Text.Json has to say about them:
Input: { args = [|"Hello"; "World"|] year = 2021 }
Output: {"args":["Hello","World"],"year":2021}
```

Next steps

Next, check out the Tour of F# to learn more about different F# features.

Get started with F# in Visual Studio for Mac

8/5/2022 • 5 minutes to read • Edit Online

F# is supported in the Visual Studio for Mac IDE. Ensure that you have Visual Studio for Mac installed.

Creating a console application

One of the most basic projects in Visual Studio for Mac is the Console Application. Here's how to do it. Once Visual Studio for Mac is open:

1. On the **File** menu, point to **New Solution**.
2. In the New Project dialog, there are 2 different templates for Console Application. There is one under Other -> .NET which targets the .NET Framework. The other template is under .NET Core -> App which targets .NET Core. Either template should work for the purpose of this article.
3. Under console app, change C# to F# if needed. Choose the **Next** button to move forward!
4. Give your project a name, and choose the options you want for the app. Notice, the preview pane to the side of the screen that will show the directory structure that will be created based on the options selected.
5. Click **Create**. You should now see an F# project in the Solution Explorer.

Writing your code

Let's get started by writing some code first. Make sure that the `Program.fs` file is open, and then replace its contents with the following:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

In the previous code sample, a function `square` has been defined which takes an input named `x` and multiplies it by itself. Because F# uses Type Inference, the type of `x` doesn't need to be specified. The F# compiler understands the types where multiplication is valid, and will assign a type to `x` based on how `square` is called. If you hover over `square`, you should see the following:

```
val square: x:int -> int
```

This is what is known as the function's type signature. It can be read like this: "Square is a function which takes an integer named x and produces an integer". Note that the compiler gave `square` the `int` type for now - this is because multiplication is not generic across *all* types, but rather is generic across a closed set of types. The F# compiler picked `int` at this point, but it will adjust the type signature if you call `square` with a different input type, such as a `float`.

Another function, `main`, is defined, which is decorated with the `EntryPoint` attribute to tell the F# compiler that program execution should start there. It follows the same convention as other C-style programming languages, where command-line arguments can be passed to this function, and an integer code is returned (typically `0`).

It is in this function that we call the `square` function with an argument of `12`. The F# compiler then assigns the type of `square` to be `int -> int` (that is, a function which takes an `int` and produces an `int`). The call to `printfn` is a formatted printing function which uses a format string, similar to C-style programming languages, parameters which correspond to those specified in the format string, and then prints the result and a new line.

Running your code

You can run the code and see results by clicking on **Run** from the top level menu and then **Start Without Debugging**. This will run the program without debugging and allows you to see the results.

You should now see the following printed to the console window that Visual Studio for Mac popped up:

```
12 squared is 144!
```

Congratulations! You've created your first F# project in Visual Studio for Mac, written an F# function printed the results of calling that function, and run the project to see some results.

Using F# Interactive

One of the best features of F# tooling in Visual Studio for Mac is the F# Interactive Window. It allows you to send code over to a process where you can call that code and see the result interactively.

To begin using it, highlight the `square` function defined in your code. Next, click on **Edit** from the top level menu. Next select **Send selection to F# Interactive**. This executes the code in the F# Interactive Window. Alternatively, you can right click on the selection and choose **Send selection to F# Interactive**. You should see the F# Interactive Window appear with the following in it:

```
>

val square: x: int -> int

>
```

This shows the same function signature for the `square` function, which you saw earlier when you hovered over the function. Because `square` is now defined in the F# Interactive process, you can call it with different values:

```
> square 12;;
val it: int = 144
> square 13;;
val it: int = 169
```

This executes the function, binds the result to a new name `it`, and displays the type and value of `it`. Note that you must terminate each line with `;;`. This is how F# Interactive knows when your function call is finished. You can also define new functions in F# Interactive:

```
> let isOdd x = x % 2 <> 0;;

val isOdd: x: int -> bool

> isOdd 12;;
val it: bool = false
```

The above defines a new function, `isOdd`, which takes an `int` and checks to see if it's odd! You can call this function to see what it returns with different inputs. You can call functions within function calls:

```
> isOdd (square 15);;
val it: bool = true
```

You can also use the pipe-forward operator to pipeline the value into the two functions:

```
> 15 |> square |> isOdd;;
val it: bool = true
```

The pipe-forward operator, and more, are covered in later tutorials.

This is only a glimpse into what you can do with F# Interactive. To learn more, check out Interactive Programming with F#.

Next steps

If you haven't already, check out the Tour of F#, which covers some of the core features of F#. It will give you an overview of some of the capabilities of F#, and provide ample code samples that you can copy into Visual Studio for Mac and run. There are also some great external resources you can use, showcased in the F# Guide.

See also

- F# guide
- Tour of F#
- F# language guide
- Type inference
- Symbol and operator reference

F# Language Reference

8/5/2022 • 10 minutes to read • Edit Online

This section is a reference for F#, a multi-paradigm programming language targeting .NET. F# supports functional, object-oriented, and imperative programming models.

Organizing F# Code

The following table shows reference articles related to organizing your F# code.

TITLE	DESCRIPTION
Namespaces	Learn about namespace support in F#. A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.
Modules	Learn about modules. An F# module is like a namespace and can also include values and functions. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.
`open` Declarations	Learn about how `open` works. An `open` declaration specifies a module, namespace, or type whose elements you can reference without using a fully qualified name.
Signatures	Learn about signatures and signature files. A signature file contains information about the public signatures of a set of F# program elements, such as types, namespaces, and modules. It can be used to specify the accessibility of these program elements.
Access Control	Learn about access control in F#. Access control means declaring what clients are able to use certain program elements, such as types, methods, functions, and so on.
XML Documentation	Learn about support for generating documentation files from XML doc comments, also known as triple slash comments. You can produce documentation from code comments in F# as in other .NET languages.

Literals and Strings

The following table shows reference articles that describe literals and strings in F#.

TITLE	DESCRIPTION
Literals	Learn about the syntax for literal values in F# and how to specify type information for F# literals.

TITLE	DESCRIPTION
Strings	Learn about strings in F#. The `string` type represents immutable text, as a sequence of Unicode characters. `string` is an alias for `System.String` in .NET.
Interpolated strings	Learn about interpolated strings, a special form of string that allows you to embed F# expressions directly inside them.

Values and Functions

The following table shows reference articles that describe language concepts related to values, `let`-bindings, and functions.

TITLE	DESCRIPTION
Values	Learn about values, which are immutable quantities that have a specific type; values can be integral or floating point numbers, characters or text, lists, sequences, arrays, tuples, discriminated unions, records, class types, or function values.
Functions	Functions are the fundamental unit of program execution in any programming language. An F# function has a name, can have parameters and take arguments, and has a body. F# also supports functional programming constructs such as treating functions as values, using unnamed functions in expressions, composition of functions to form new functions, curried functions, and the implicit definition of functions by way of the partial application of function arguments.
Function Expressions	Learn how to use the F# 'fun' keyword to define a lambda expression, which is an anonymous function.

Loops and Conditionals

The following table lists articles that describe F# loops and conditionals.

TITLE	DESCRIPTION
Conditional Expressions: `if...then...else`	Learn about the `if...then...else` expression, which runs different branches of code and also evaluates to a different value depending on the Boolean expression given.
Loops: `for...in` Expression	Learn about the `for...in` expression, a looping construct that is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.
Loops: `for...to` Expression	Learn about the `for...to` expression, which is used to iterate in a loop over a range of values of a loop variable.
Loops: `while...do` Expression	Learn about the `while...do` expression, which is used to perform iterative execution (looping) while a specified test condition is true.

Pattern Matching

The following table shows reference articles that describe language concepts.

TITLE	DESCRIPTION
Pattern Matching	Learn about patterns, which are rules for transforming input data and are used throughout F#. You can compare data with a pattern, decompose data into constituent parts, or extract information from data in various ways.
Match Expressions	Learn about the `match` expression, which provides branching control that is based on the comparison of an expression with a set of patterns.
Active Patterns	Learn about active patterns. Active patterns enable you to define named partitions that subdivide input data. You can use active patterns to decompose data in a customized manner for each partition.

Exception Handling

The following table shows reference articles that describe language concepts related to exception handling.

TITLE	DESCRIPTION
Exception Handling	Contains information about exception handling support in F#.
The `try...with` Expression	Learn about how to use the `try...with` expression for exception handling.
The `try...finally` Expression	Learn about how the F# `try...finally` expression enables you to execute clean-up code even if a block of code throws an exception.
The `use` Keyword	Learn about the keywords `use` and `using`, which can control the initialization and release of resources.
Assertions	Learn about the `assert` expression, which is a debugging feature that you can use to test an expression. Upon failure in Debug mode, an assertion generates a system error dialog box.

Types and Type Inference

The following table shows reference articles that describe how types and type inference work in F#.

TITLE	DESCRIPTION
Types	Learn about the types that are used in F# and how F# types are named and described.

TITLE	DESCRIPTION
Basic Types	Learn about the fundamental types that are used in F#. It also provides the corresponding .NET types and the minimum and maximum values for each type.
Unit Type	Learn about the `unit` type, which is a type that indicates the absence of a specific value; the `unit` type has only a single value, which acts as a placeholder when no other value exists or is needed.
Type Abbreviations	Learn about type abbreviations, which are alternate names for types.
Type Inference	Learn about how the F# compiler infers the types of values, variables, parameters, and return values.
Casting and Conversions	Learn about support for type conversions in F#.
Generics	Learn about generic constructs in F#.
Automatic Generalization	Learn about how F# automatically generalizes the arguments and types of functions so that they work with multiple types when possible.
Constraints	Learn about constraints that apply to generic type parameters to specify the requirements for a type argument in a generic type or function.
Flexible Types	Learn about flexible types. A flexible type annotation is an indication that a parameter, variable, or value has a type that is compatible with type specified, where compatibility is determined by position in an object-oriented hierarchy of classes or interfaces.
Units of Measure	Learn about units of measure. Floating point values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on.
Byrefs	Learn about byref and byref-like types in F#, which are used for low-level programming.

Tuples, Lists, Collections, Options

The following table shows reference articles that describe types supported by F#.

TITLE	DESCRIPTION
Tuples	Learn about tuples, which are groupings of unnamed but ordered values of possibly different types.
Collections	An overview of the F# functional collection types, including types for arrays, lists, sequences (seq), maps, and sets.
Lists	Learn about lists. A list in F# is an ordered, immutable series of elements all of the same type.

TITLE	DESCRIPTION
Options	Learn about the option type. An option in F# is used when a value may or may not exist. An option has an underlying type and may either hold a value of that type or it may not have a value.
Arrays	Learn about arrays. Arrays are fixed-size, zero-based, mutable sequences of consecutive data elements, all of the same type.
Sequences	Learn about sequences. A sequence is a logical series of elements all of one type. Individual sequence elements are only computed if necessary, so the representation may be smaller than a literal element count indicates.
Sequence Expressions	Learn about sequence expressions, which let you generate sequences of data on-demand.
Reference Cells	Learn about reference cells, which are storage locations that enable you to create mutable variables with reference semantics.

Records and Discriminated Unions

The following table shows reference articles that describe record and discriminated union type definitions supported by F#.

TITLE	DESCRIPTION
Records	Learn about records. Records represent simple aggregates of named values, optionally with members.
Anonymous Records	Learn how to construct and use anonymous records, a language feature that helps with the manipulation of data.
Discriminated Unions	Learn about discriminated unions, which provide support for values that may be one of a variety of named cases, each with possibly different values and types.
Structs	Learn about structs, which are compact object types that can be more efficient than a class for types that have a small amount of data and simple behavior.
Enumerations	Enumerations are types that have a defined set of named values. You can use them in place of literals to make code more readable and maintainable.

Object Programming

The following table shows reference articles that describe F# object programming.

TITLE	DESCRIPTION

TITLE	DESCRIPTION
Classes	Learn about classes, which are types that represent objects that can have properties, methods, and events.
Interfaces	Learn about interfaces, which specify sets of related members that other classes implement.
Abstract Classes	Learn about abstract classes, which are classes that leave some or all members unimplemented, so that implementations can be provided by derived classes.
Type Extensions	Learn about type extensions, which let you add new members to a previously defined object type.
Delegates	Learn about delegates, which represent a function call as an object.
Inheritance	Learn about inheritance, which is used to model the "is-a" relationship, or subtyping, in object-oriented programming.
Members	Learn about members of F# object types.
Parameters and Arguments	Learn about language support for defining parameters and passing arguments to functions, methods, and properties. It includes information about how to pass by reference.
Operator Overloading	Learn about how to overload arithmetic operators in a class or record type, and at the global level.
Object Expressions	Learn about object expressions, which are expressions that create new instances of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.

Async, Tasks and Lazy

The following table lists topics that describe F# async, task and lazy expressions.

TITLE	DESCRIPTION
Async Expressions	Learn about async expressions, which let you write asynchronous code in a way that is very close to the way you would naturally write synchronous code.
Task Expressions	Learn about task expressions, which are an alternative way of writing asynchronous code used when interoperating with .NET code that consumes or produces .NET tasks.
Lazy Expressions	Learn about lazy expressions, which are computations that are not evaluated immediately, but are instead evaluated when the result is actually needed.

Computation expressions and Queries

The following table lists topics that describe F# computation expressions and queries.

TITLE	DESCRIPTION
Computation Expressions	Learn about computation expressions in F#, which provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. They can be used to manage data, control, and side effects in functional programs.
Query Expressions	Learn about query expressions, a language feature that implements LINQ for F# and enables you to write queries against a data source or enumerable collection.

Attributes, Reflection, Quotations and Plain Text Formatting

The following table lists articles that describe F# reflective features, including attributes, quotations, `nameof`, and plain text formatting.

TITLE	DESCRIPTION
Attributes	Learn how F# Attributes enable metadata to be applied to a programming construct.
nameof	Learn about the `nameof` operator, a metaprogramming feature that allows you to produce the name of any symbol in your source code.
Caller Information	Learn about how to use Caller Info Argument Attributes to obtain caller information from a method.
Source Line, File, and Path Identifiers	Learn about the identifiers `__LINE__`, `__SOURCE_DIRECTORY__`, and `__SOURCE_FILE__`, which are built-in values that enable you to access the source line number, directory, and file name in your code.
Code Quotations	Learn about code quotations, a language feature that enables you to generate and work with F# code expressions programmatically.
Plain Text Formatting	Learn how to use sprintf and other plain text formatting in F# applications and scripts.

Type Providers

The following table lists articles that describe F# type providers.

TITLE	DESCRIPTION
Type Providers	Learn about type providers and find links to walkthroughs on using the built-in type providers to access databases and web services.
Create a Type Provider	Learn how to create your own F# type providers by examining several simple type providers that illustrate the basic concepts.

F# Core Library API reference

F# Core Library (FSharp.Core) API reference is the reference for all F# Core Library namespaces, modules, types, and functions.

Reference Tables

The following table shows reference articles that provide tables of keywords, symbols, and literals that are used as tokens in F#.

TITLE	DESCRIPTION
Keyword Reference	Contains links to information about all F# language keywords.
Symbol and Operator Reference	Contains a table of symbols and operators that are used in F#.

Compiler-supported Constructs

The following table lists topics that describe special compiler-supported constructs.

TOPIC	DESCRIPTION
Compiler Options	Describes the command-line options for the F# compiler.
Compiler Directives	Describes the processor directives and compiler directives supported by the F# compiler.

Literals

8/5/2022 • 2 minutes to read • Edit Online

This article provides a table that shows how to specify the type of a literal in F#.

Literal types

The following table shows the literal types in F#. Characters that represent digits in hexadecimal notation are not case-sensitive; characters that identify the type are case-sensitive.

TYPE	DESCRIPTION	SUFFIX OR PREFIX	EXAMPLES
sbyte	signed 8-bit integer	y	`86y` `0b00000101y`
byte	unsigned 8-bit natural number	uy	`86uy` `0b00000101uy`
int16	signed 16-bit integer	s	`86s`
uint16	unsigned 16-bit natural number	us	`86us`
int int32	signed 32-bit integer	l or none	`86` `86l`
uint uint32	unsigned 32-bit natural number	u or ul	`86u` `86ul`
nativeint	native pointer to a signed natural number	n	`123n`
unativeint	native pointer as an unsigned natural number	un	`0x00002D3Fun`
int64	signed 64-bit integer	L	`86L`
uint64	unsigned 64-bit natural number	UL	`86UL`
single, float32	32-bit floating point number	F or f	`4.14F` or `4.14f`
		lf	`0x00000000lf`

TYPE	DESCRIPTION	SUFFIX OR PREFIX	EXAMPLES
float; double	64-bit floating point number	none	`4.14` or `2.3E+32` or `2.3e+32`
		LF	`0x0000000000000000LF`
bigint	integer not limited to 64-bit representation	I	`9999999999999999999999999999I`
decimal	fractional number represented as a fixed point or rational number	M or m	`0.7833M` or `0.7833m`
Char	Unicode character	none	`'a'` or `'\u0061'`

String	Unicode string	none	`"text\n"`
			
			or
			
			`@"c:\filename"`
			
			or
			
			`"""<book title="Paradise Lost">"""`
			
			or
			
			`"string1" + "string2"`
			
			See also Strings.

byte	ASCII character	B	`'a'B`
byte[]	ASCII string	B	`"text"B`
String or byte[]	verbatim string	@ prefix	`@"\\server\share"` (Unicode) `@"\\server\share"B` (ASCII)

Named literals

Values that are intended to be constants can be marked with the Literal attribute. This attribute has the effect of causing a value to be compiled as a constant.

Named literals are useful for:

- Pattern matching without a `when` clause.
- Attribute arguments.
- Static type provider arguments.

In pattern matching expressions, identifiers that begin with lowercase characters are always treated as variables to be bound, rather than as literals, so you should generally use initial capitals when you define literals.

```
[<Literal>]
let SomeJson = """{"numbers":[1,2,3,4,5]}"""

[<Literal>]
let Literal1 = "a" + "b"

[<Literal>]
let FileLocation =   __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let Literal2 = 1 ||| 64

[<Literal>]
let Literal3 = System.IO.FileAccess.Read ||| System.IO.FileAccess.Write
```

Remarks

Unicode strings can contain explicit encodings that you can specify by using `\u` followed by a 16-bit hexadecimal code (0000 - FFFF), or UTF-32 encodings that you can specify by using `\U` followed by a 32-bit hexadecimal code that represents any Unicode code point (00000000 - 0010FFFF).

The use of bitwise operators other than `|||` isn't allowed.

Integers in other bases

Signed 32-bit integers can also be specified in hexadecimal, octal, or binary using a `0x`, `0o` or `0b` prefix, respectively.

```
let numbers = (0x9F, 0o77, 0b1010)
// Result: numbers : int * int * int = (159, 63, 10)
```

Underscores in numeric literals

You can separate digits with the underscore character (`_`).

```
let value = 0xDEAD_BEEF

let valueAsBits = 0b1101_1110_1010_1101_1011_1110_1110_1111

let exampleSSN = 123_456_7890
```

Strings

8/5/2022 • 4 minutes to read • Edit Online

The `string` type represents immutable text as a sequence of Unicode characters. `string` is an alias for `System.String` in .NET.

Remarks

String literals are delimited by the quotation mark (") character. The backslash character (\ ) is used to encode certain special characters. The backslash and the next character together are known as an *escape sequence*. Escape sequences supported in F# string literals are shown in the following table.

CHARACTER	ESCAPE SEQUENCE
Alert	`\a`
Backspace	`\b`
Form feed	`\f`
Newline	`\n`
Carriage return	`\r`
Tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Quotation mark	`\"`
Apostrophe	`\'`
Unicode character	`\DDD` (where `D` indicates a decimal digit; range of 000 - 255; for example, `\231` = "ç")
Unicode character	`\xHH` (where `H` indicates a hexadecimal digit; range of 00 - FF; for example, `\xE7` = "ç")
Unicode character	`\uHHHH` (UTF-16) (where `H` indicates a hexadecimal digit; range of 0000 - FFFF; for example, `\u00E7` = "ç")
Unicode character	`\U00HHHHHH` (UTF-32) (where `H` indicates a hexadecimal digit; range of 000000 - 10FFFF; for example, `\U0001F47D` = "👽")

IMPORTANT

The `\DDD` escape sequence is decimal notation, not octal notation like in most other languages. Therefore, digits `8` and `9` are valid, and a sequence of `\032` represents a space (U+0020), whereas that same code point in octal notation would be `\040`.

NOTE

Being constrained to a range of 0 - 255 (0xFF), the `\DDD` and `\x` escape sequences are effectively the ISO-8859-1 character set, since that matches the first 256 Unicode code points.

Verbatim Strings

If preceded by the @ symbol, the literal is a verbatim string. Declaring a verbatim string means that any escape sequences are ignored, except that two quotation mark characters are interpreted as one quotation mark character.

Triple Quoted Strings

Additionally, a string may be enclosed by triple quotes. In this case, all escape sequences are ignored, including double quotation mark characters. To specify a string that contains an embedded quoted string, you can either use a verbatim string or a triple-quoted string. If you use a verbatim string, you must specify two quotation mark characters to indicate a single quotation mark character. If you use a triple-quoted string, you can use the single quotation mark characters without them being parsed as the end of the string. This technique can be useful when you work with XML or other structures that include embedded quotation marks.

```
// Using a verbatim string
let xmlFragment1 = @"<book author=""Milton, John"" title=""Paradise Lost"">"

// Using a triple-quoted string
let xmlFragment2 = """<book author="Milton, John" title="Paradise Lost">"""
```

In code, strings that have line breaks are accepted and the line breaks are interpreted literally as newlines, unless a backslash character is the last character before the line break. Leading white space on the next line is ignored when the backslash character is used. The following code produces a string `str1` that has value `"abc\ndef"` and a string `str2` that has value `"abcdef"`.

```
let str1 = "abc
def"
let str2 = "abc\
def"
```

String Indexing and Slicing

You can access individual characters in a string by using array-like syntax. The following examples use `[]` to index strings. This syntax was introduced in F# 6.0. You can also use `.[]` to index strings in all versions. The new syntax is preferred.

```
printfn "%c" str1[1]
```

The output is `b`.

Or you can extract substrings by using array slice syntax, as shown in the following code.

```
printfn "%s" str1[0..2]
printfn "%s" str2[3..5]
```

The output is as follows.

```
abc
def
```

You can represent ASCII strings by arrays of unsigned bytes, type `byte[]`. You add the suffix `B` to a string literal to indicate that it's an ASCII string. ASCII string literals used with byte arrays support the same escape sequences as Unicode strings, except for the Unicode escape sequences.

```
// "abc" interpreted as a Unicode string.
let str1 : string = "abc"
// "abc" interpreted as an ASCII byte array.
let bytearray : byte[] = "abc"B
```

String Operators

The `+` operator can be used to concatenate strings, maintaining compatibility with the .NET Framework string handling features. The following example illustrates string concatenation.

```
let string1 = "Hello, " + "world"
```

String Class

Because the string type in F# is actually a .NET Framework `System.String` type, all the `System.String` members are available. `System.String` includes the `+` operator, which is used to concatenate strings, the `Length` property, and the `Chars` property, which returns the string as an array of Unicode characters. For more information about strings, see `System.String`.

By using the `Chars` property of `System.String`, you can access the individual characters in a string by specifying an index, as is shown in the following code.

```
let printChar (str : string) (index : int) =
    printfn "First character: %c" (str.Chars(index))
```

String Module

Additional functionality for string handling is included in the `String` module in the `FSharp.Core` namespace. For more information, see String Module.

See also

- Interpolated Strings
- F# Language Reference

Interpolated strings

8/5/2022 • 2 minutes to read • Edit Online

Interpolated strings are strings that allow you to embed F# expressions into them. They are helpful in a wide range of scenarios where the value of a string may change based on the result of a value or expression.

Syntax

```
$"string-text {expr}"
$"string-text %format-specifier{expr}"
$"""string-text {"embedded string literal"}"""
```

Remarks

Interpolated strings let you write code in "holes" inside of a string literal. Here's a basic example:

```
let name = "Phillip"
let age = 30
printfn $"Name: {name}, Age: {age}"

printfn $"I think {3.0 + 0.14} is close to {System.Math.PI}!"
```

The contents in between each `{}` brace pair can be any F# expression.

To escape a `{}` brace pair, write two of them like so:

```
let str = $"A pair of braces: {{}}"
// "A pair of braces: {}"
```

Typed interpolated strings

Interpolated strings can also have F# format specifiers to enforce type safety.

```
let name = "Phillip"
let age = 30

printfn $"Name: %s{name}, Age: %d{age}"

// Error: type mismatch
printfn $"Name: %s{age}, Age: %d{name}"
```

In the previous example, the code mistakenly passes the `age` value where `name` should be, and vice/versa. Because the interpolated strings use format specifiers, this is a compile error instead of a subtle runtime bug.

Verbatim interpolated strings

F# supports verbatim interpolated strings with triple quotes so that you can embed string literals.

```
let age = 30

printfn $"""Name: {"Phillip"}, Age: %d{age}"""
```

Format specifiers

Format specifiers can either be printf-style or .NET-style. Printf-style specifiers are those covered in plaintext formatting, placed before the braces. For example:

```
let pi = $"%0.3f{System.Math.PI}"  // "3.142"
let code = $"0x%08x{43962}"  // "0x0000abba"
```

The format specifier `%A` is particularly useful for producing diagnostic output of structured F# data.

```
let data = [0..4]
let output = $"The data is %A{data}"  // "The data is [0; 1; 2; 3; 4]"
```

.NET-style specifiers are those usable with String.Format, placed after a `:` within the braces. For example:

```
let pi = $"{System.Math.PI:N4}"  // "3.1416"
let now = $"{System.DateTime.UtcNow:``yyyyMMdd``}" // e.g. "20220210"
```

If a .NET-style specifier contains an unusual character, then it can be escaped using double-backticks:

```
let nowDashes = $"{System.DateTime.UtcNow:``yyyy-MM-dd``}" // e.g. "2022-02-10"
```

Aligning expressions in interpolated strings

You can left-align or right-align expressions inside interpolated strings with `|` and a specification of how many spaces. The following interpolated string aligns the left and right expressions to the left and right, respectively, by seven spaces.

```
printfn $"""|{"Left",-7}|{"Right",7}|"""
// |Left   |  Right|
```

Interpolated strings and `FormattableString` formatting

You can also apply formatting that adheres to the rules for FormattableString:

```
let speedOfLight = 299792.458
printfn $"The speed of light is {speedOfLight:N3} km/s."
// "The speed of light is 299,792.458 km/s."
```

Additionally, an interpolated string can also be type checked as a FormattableString via a type annotation:

```
let frmtStr = $"The speed of light is {speedOfLight:N3} km/s." : FormattableString
// Type: FormattableString
// The speed of light is 299,792.458 km/s.
```

Note that the type annotation must be on the interpolated string expression itself. F# does not implicitly convert an interpolated string into a FormattableString.

See also

- Strings
- F# RFC FS-1001 - Interpolated strings

Values

8/5/2022 • 3 minutes to read • Edit Online

Values in F# are quantities that have a specific type; values can be integral or floating point numbers, characters or text, lists, sequences, arrays, tuples, discriminated unions, records, class types, or function values.

Binding a Value

The term *binding* means associating a name with a definition. The `let` keyword binds a value, as in the following examples:

```
let a = 1
let b = 100u
let str = "text"

// A function value binding.

let f x = x + 1
```

The type of a value is inferred from the definition. For a primitive type, such as an integral or floating point number, the type is determined from the type of the literal. Therefore, in the previous example, the compiler infers the type of `b` to be `unsigned int`, whereas the compiler infers the type of `a` to be `int`. The type of a function value is determined from the return value in the function body. For more information about function value types, see Functions. For more information about literal types, see Literals.

The compiler does not issue diagnostics about unused bindings by default. To receive these messages, enable warning 1182 in your project file or when invoking the compiler (see `--warnon` under Compiler Options).

Why Immutable?

Immutable values are values that cannot be changed throughout the course of a program's execution. If you are used to languages such as C++, Visual Basic, or C#, you might find it surprising that F# puts primacy over immutable values rather than variables that can be assigned new values during the execution of a program. Immutable data is an important element of functional programming. In a multithreaded environment, shared mutable variables that can be changed by many different threads are difficult to manage. Also, with mutable variables, it can sometimes be hard to tell if a variable might be changed when it is passed to another function.

In pure functional languages, there are no variables, and functions behave strictly as mathematical functions. Where code in a procedural language uses a variable assignment to alter a value, the equivalent code in a functional language has an immutable value that is the input, an immutable function, and different immutable values as the output. This mathematical strictness allows for tighter reasoning about the behavior of the program. This tighter reasoning is what enables compilers to check code more stringently and to optimize more effectively, and helps make it easier for developers to understand and write correct code. Functional code is therefore likely to be easier to debug than ordinary procedural code.

F# is not a pure functional language, yet it fully supports functional programming. Using immutable values is a good practice because doing this allows your code to benefit from an important aspect of functional programming.

Mutable Variables

You can use the keyword `mutable` to specify a variable that can be changed. Mutable variables in F# should generally have a limited scope, either as a field of a type or as a local value. Mutable variables with a limited scope are easier to control and are less likely to be modified in incorrect ways.

You can assign an initial value to a mutable variable by using the `let` keyword in the same way as you would define a value. However, the difference is that you can subsequently assign new values to mutable variables by using the `<-` operator, as in the following example.

```
let mutable x = 1
x <- x + 1
```

Values marked `mutable` may be automatically promoted to `'a ref` if captured by a closure, including forms that create closures, such as `seq` builders. If you wish to be notified when this occurs, enable warning 3180 in your project file or when invoking the compiler.

Related Topics

TITLE	DESCRIPTION
let Bindings	Provides information about using the `let` keyword to bind names to values and functions.
Functions	Provides an overview of functions in F#.

See also

- Null Values
- F# Language Reference

let Bindings

8/5/2022 • 5 minutes to read • Edit Online

A *binding* associates an identifier with a value or function. You use the `let` keyword to bind a name to a value or function.

Syntax

```
// Binding a value:
let identifier-or-pattern [: type] =expressionbody-expression
// Binding a function value:
let identifier parameter-list [: return-type ] =expressionbody-expression
```

Remarks

The `let` keyword is used in binding expressions to define values or function values for one or more names. The simplest form of the `let` expression binds a name to a simple value, as follows.

```
let i = 1
```

If you separate the expression from the identifier by using a new line, you must indent each line of the expression, as in the following code.

```
let someVeryLongIdentifier =
    // Note indentation below.
    3 * 4 + 5 * 6
```

Instead of just a name, a pattern that contains names can be specified, for example, a tuple, as shown in the following code.

```
let i, j, k = (1, 2, 3)
```

The *body-expression* is the expression in which the names are used. The body expression appears on its own line, indented to line up exactly with the first character in the `let` keyword:

```
let result =

    let i, j, k = (1, 2, 3)

    // Body expression:
    i + 2*j + 3*k
```

A `let` binding can appear at the module level, in the definition of a class type, or in local scopes, such as in a function definition. A `let` binding at the top level in a module or in a class type does not need to have a body expression, but at other scope levels, the body expression is required. The bound names are usable after the point of definition, but not at any point before the `let` binding appears, as is illustrated in the following code.

```
// Error:
printfn "%d" x
let x = 100
// OK:
printfn "%d" x
```

Function Bindings

Function bindings follow the rules for value bindings, except that function bindings include the function name and the parameters, as shown in the following code.

```
let function1 a =
    a + 1
```

In general, parameters are patterns, such as a tuple pattern:

```
let function2 (a, b) = a + b
```

A `let` binding expression evaluates to the value of the last expression. Therefore, in the following code example, the value of `result` is computed from `100 * function3 (1, 2)`, which evaluates to `300`.

```
let result =
    let function3 (a, b) = a + b
    100 * function3 (1, 2)
```

For more information, see Functions.

Type Annotations

You can specify types for parameters by including a colon (:) followed by a type name, all enclosed in parentheses. You can also specify the type of the return value by appending the colon and type after the last parameter. The full type annotations for `function1`, with integers as the parameter types, would be as follows.

```
let function1 (a: int) : int = a + 1
```

When there are no explicit type parameters, type inference is used to determine the types of parameters of functions. This can include automatically generalizing the type of a parameter to be generic.

For more information, see Automatic Generalization and Type Inference.

let Bindings in Classes

A `let` binding can appear in a class type but not in a structure or record type. To use a let binding in a class type, the class must have a primary constructor. Constructor parameters must appear after the type name in the class definition. A `let` binding in a class type defines private fields and members for that class type and, together with `do` bindings in the type, forms the code for the primary constructor for the type. The following code examples show a class `MyClass` with private fields `field1` and `field2`.

```
type MyClass(a) =
    let field1 = a
    let field2 = "text"
    do printfn "%d %s" field1 field2
    member this.F input =
        printfn "Field1 %d Field2 %s Input %A" field1 field2 input
```

The scopes of `field1` and `field2` are limited to the type in which they are declared. For more information, see `let` Bindings in Classes and Classes.

Type Parameters in let Bindings

A `let` binding at the module level, in a type, or in a computation expression can have explicit type parameters. A let binding in an expression, such as within a function definition, cannot have type parameters. For more information, see Generics.

Attributes on let Bindings

Attributes can be applied to top-level `let` bindings in a module, as shown in the following code.

```
[<Obsolete>]
let function1 x y = x + y
```

Scope and Accessibility of Let Bindings

The scope of an entity declared with a let binding is limited to the portion of the containing scope (such as a function, module, file or class) after the binding appears. Therefore, it can be said that a let binding introduces a name into a scope. In a module, a let-bound value or function is accessible to clients of a module as long as the module is accessible, since the let bindings in a module are compiled into public functions of the module. By contrast, let bindings in a class are private to the class.

Normally, functions in modules must be qualified by the name of the module when used by client code. For example, if a module `Module1` has a function `function1`, users would specify `Module1.function1` to refer to the function.

Users of a module may use an import declaration to make the functions within that module available for use without being qualified by the module name. In the example just mentioned, users of the module can in that case open the module by using the import declaration `open Module1` and thereafter refer to `function1` directly.

```
module Module1 =
    let function1 x = x + 1.0

module Module2 =
    let function2 x =
        Module1.function1 x

open Module1

let function3 x =
    function1 x
```

Some modules have the attribute RequireQualifiedAccess, which means that the functions that they expose must be qualified with the name of the module. For example, the F# List module has this attribute.

For more information on modules and access control, see Modules and Access Control.

See also

- Functions
- `let` Bindings in Classes

do Bindings

8/5/2022 • 2 minutes to read • Edit Online

A `do` binding is used to execute code without defining a function or value. Also, do bindings can be used in classes, see `do` Bindings in Classes.

Syntax

```
[ attributes ]
[ do ]expression
```

Remarks

Use a `do` binding when you want to execute code independently of a function or value definition. The expression in a `do` binding must return `unit`. Code in a top-level `do` binding is executed when the module is initialized. The keyword `do` is optional.

Attributes can be applied to a top-level `do` binding. For example, if your program uses COM interop, you might want to apply the `STAThread` attribute to your program. You can do this by using an attribute on a `do` binding, as shown in the following code.

```
open System
open System.Windows.Forms

let form1 = new Form()
form1.Text <- "XYZ"

[<STAThread>]
do
   Application.Run(form1)
```

See also

- F# Language Reference
- Functions

The fixed keyword

8/5/2022 • 2 minutes to read • Edit Online

The `fixed` keyword allows you to "pin" a local onto the stack to prevent it from being collected or moved during garbage-collection. It is used for low-level programming scenarios.

Syntax

```
use ptr = fixed expression
```

Remarks

This extends the syntax of expressions to allow extracting a pointer and binding it to a name which is prevented from being collected or moved during garbage-collection.

A pointer from an expression is fixed via the `fixed` keyword and is bound to an identifier via the `use` keyword. The semantics of this are similar to resource management via the `use` keyword. The pointer is fixed while it is in scope, and once it is out of scope, it is no longer fixed. `fixed` cannot be used outside the context of a `use` binding. You must bind the pointer to a name with `use`.

Use of `fixed` must occur within an expression in a function or a method. It cannot be used at a script-level or module-level scope.

Like all pointer code, this is an unsafe feature and will emit a warning when used.

Example

```
open Microsoft.FSharp.NativeInterop

type Point = { mutable X: int; mutable Y: int}

let squareWithPointer (p: nativeptr<int>) =
    // Dereference the pointer at the 0th address.
    let mutable value = NativePtr.get p 0

    // Perform some work
    value <- value * value

    // Set the value in the pointer at the 0th address.
    NativePtr.set p 0 value

let pnt = { X = 1; Y = 2 }
printfn $"pnt before - X: %d{pnt.X} Y: %d{pnt.Y}" // prints 1 and 2

// Note that the use of 'fixed' is inside a function.
// You cannot fix a pointer at a script-level or module-level scope.
let doPointerWork() =
    use ptr = fixed &pnt.Y

    // Square the Y value
    squareWithPointer ptr
    printfn $"pnt after - X: %d{pnt.X} Y: %d{pnt.Y}" // prints 1 and 4

doPointerWork()
```

See also

- NativePtr Module

Functions

8/5/2022 • 9 minutes to read • Edit Online

Functions are the fundamental unit of program execution in any programming language. As in other languages, an F# function has a name, can have parameters and take arguments, and has a body. F# also supports functional programming constructs such as treating functions as values, using unnamed functions in expressions, composition of functions to form new functions, curried functions, and the implicit definition of functions by way of the partial application of function arguments.

You define functions by using the `let` keyword, or, if the function is recursive, the `let rec` keyword combination.

Syntax

```
// Non-recursive function definition.
let [inline] function-name parameter-list [ : return-type ] = function-body
// Recursive function definition.
let rec function-name parameter-list = recursive-function-body
```

Remarks

The *function-name* is an identifier that represents the function. The *parameter-list* consists of successive parameters that are separated by spaces. You can specify an explicit type for each parameter, as described in the Parameters section. If you do not specify a specific argument type, the compiler attempts to infer the type from the function body. The *function-body* consists of an expression. The expression that makes up the function body is typically a compound expression consisting of a number of expressions that culminate in a final expression that is the return value. The *return-type* is a colon followed by a type and is optional. If you do not specify the type of the return value explicitly, the compiler determines the return type from the final expression.

A simple function definition resembles the following:

```
let f x = x + 1
```

In the previous example, the function name is `f`, the argument is `x`, which has type `int`, the function body is `x + 1`, and the return value is of type `int`.

Functions can be marked `inline`. For information about `inline`, see Inline Functions.

Scope

At any level of scope other than module scope, it is not an error to reuse a value or function name. If you reuse a name, the name declared later shadows the name declared earlier. However, at the top level scope in a module, names must be unique. For example, the following code produces an error when it appears at module scope, but not when it appears inside a function:

```
let list1 = [ 1; 2; 3]
// Error: duplicate definition.
let list1 = []
let function1 () =
   let list1 = [1; 2; 3]
   let list1 = []
   list1
```

But the following code is acceptable at any level of scope:

```
let list1 = [ 1; 2; 3]
let sumPlus x =
// OK: inner list1 hides the outer list1.
   let list1 = [1; 5; 10]
   x + List.sum list1
```

Parameters

Names of parameters are listed after the function name. You can specify a type for a parameter, as shown in the following example:

```
let f (x : int) = x + 1
```

If you specify a type, it follows the name of the parameter and is separated from the name by a colon. If you omit the type for the parameter, the parameter type is inferred by the compiler. For example, in the following function definition, the argument `x` is inferred to be of type `int` because 1 is of type `int`.

```
let f x = x + 1
```

However, the compiler will attempt to make the function as generic as possible. For example, note the following code:

```
let f x = (x, x)
```

The function creates a tuple from one argument of any type. Because the type is not specified, the function can be used with any argument type. For more information, see Automatic Generalization.

Function Bodies

A function body can contain definitions of local variables and functions. Such variables and functions are in scope in the body of the current function but not outside it. You must use indentation to indicate that a definition is in a function body, as shown in the following example:

```
let cylinderVolume radius length =
    // Define a local value pi.
    let pi = 3.14159
    length * pi * radius * radius
```

For more information, see Code Formatting Guidelines and Verbose Syntax.

Return Values

The compiler uses the final expression in a function body to determine the return value and type. The compiler might infer the type of the final expression from previous expressions. In the function `cylinderVolume`, shown in the previous section, the type of `pi` is determined from the type of the literal `3.14159` to be `float`. The compiler uses the type of `pi` to determine the type of the expression `length * pi * radius * radius` to be `float`. Therefore, the overall return type of the function is `float`.

To specify the return type explicitly, write the code as follows:

```
let cylinderVolume radius length : float =
   // Define a local value pi.
   let pi = 3.14159
   length * pi * radius * radius
```

As the code is written above, the compiler applies **float** to the entire function; if you mean to apply it to the parameter types as well, use the following code:

```
let cylinderVolume (radius : float) (length : float) : float
```

Calling a Function

You call functions by specifying the function name followed by a space and then any arguments separated by spaces. For example, to call the function **cylinderVolume** and assign the result to the value **vol**, you write the following code:

```
let vol = cylinderVolume 2.0 3.0
```

Partial Application of Arguments

If you supply fewer than the specified number of arguments, you create a new function that expects the remaining arguments. This method of handling arguments is referred to as *currying* and is a characteristic of functional programming languages like F#. For example, suppose you are working with two sizes of pipe: one has a radius of **2.0** and the other has a radius of **3.0**. You could create functions that determine the volume of pipe as follows:

```
let smallPipeRadius = 2.0
let bigPipeRadius = 3.0

// These define functions that take the length as a remaining
// argument:

let smallPipeVolume = cylinderVolume smallPipeRadius
let bigPipeVolume = cylinderVolume bigPipeRadius
```

You would then supply the final argument as needed for various lengths of pipe of the two different sizes:

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

Recursive Functions

Recursive functions are functions that call themselves. They require that you specify the **rec** keyword following the **let** keyword. Invoke the recursive function from within the body of the function just as you would invoke any function call. The following recursive function computes the *n*th Fibonacci number. The Fibonacci number sequence has been known since antiquity and is a sequence in which each successive number is the sum of the previous two numbers in the sequence.

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

Some recursive functions might overflow the program stack or perform inefficiently if you do not write them with care and with awareness of special techniques, such as the use of tail recursion, accumulators, and continuations.

Function Values

In F#, all functions are considered values; in fact, they are known as *function values*. Because functions are values, they can be used as arguments to other functions or in other contexts where values are used. Following is an example of a function that takes a function value as an argument:

```
let apply1 (transform : int -> int ) y = transform y
```

You specify the type of a function value by using the `->` token. On the left side of this token is the type of the argument, and on the right side is the return value. In the previous example, `apply1` is a function that takes a function `transform` as an argument, where `transform` is a function that takes an integer and returns another integer. The following code shows how to use `apply1`:

```
let increment x = x + 1

let result1 = apply1 increment 100
```

The value of `result` will be 101 after the previous code runs.

Multiple arguments are separated by successive `->` tokens, as shown in the following example:

```
let apply2 ( f: int -> int -> int) x y = f x y

let mul x y = x * y

let result2 = apply2 mul 10 20
```

The result is 200.

Lambda Expressions

A *lambda expression* is an unnamed function. In the previous examples, instead of defining named functions **increment** and **mul**, you could use lambda expressions as follows:

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

You define lambda expressions by using the `fun` keyword. A lambda expression resembles a function definition, except that instead of the `=` token, the `->` token is used to separate the argument list from the function body.

As in a regular function definition, the argument types can be inferred or specified explicitly, and the return type of the lambda expression is inferred from the type of the last expression in the body. For more information, see Lambda Expressions: The `fun` Keyword.

Pipelines

The pipe operator `|>` is used extensively when processing data in F#. This operator allows you to establish "pipelines" of functions in a flexible manner. Pipelining enables function calls to be chained together as successive operations:

```
let result = 100 |> function1 |> function2
```

The following sample walks through how you can use these operators to build a simple functional pipeline:

```
/// Square the odd values of the input and add one, using F# pipe operators.
let squareAndAddOdd values =
    values
    |> List.filter (fun x -> x % 2 <> 0)
    |> List.map (fun x -> x * x + 1)

let numbers = [ 1; 2; 3; 4; 5 ]

let result = squareAndAddOdd numbers
```

The result is `[2; 10; 26]`. The previous sample uses list processing functions, demonstrating how functions can be used to process data when building pipelines. The pipeline operator itself is defined in the F# core library as follows:

```
let (|>) x f = f x
```

Function composition

Functions in F# can be composed from other functions. The composition of two functions **function1** and **function2** is another function that represents the application of **function1** followed by the application of **function2**:

```
let function1 x = x + 1
let function2 x = x * 2
let h = function1 >> function2
let result5 = h 100
```

The result is 202.

The composition operator `>>` takes two functions and returns a function; by contrast, the pipeline operator `|>` takes a value and a function and returns a value. The following code example shows the difference between the pipeline and composition operators by showing the differences in the function signatures and usage.

```fsharp
// Function composition and pipeline operators compared.

let addOne x = x + 1
let timesTwo x = 2 * x

// Composition operator
// ( >> ) : ('T1 -> 'T2) -> ('T2 -> 'T3) -> 'T1 -> 'T3
let Compose2 = addOne >> timesTwo

// Backward composition operator
// ( << ) : ('T2 -> 'T3) -> ('T1 -> 'T2) -> 'T1 -> 'T3
let Compose1 = addOne << timesTwo

// Result is 5
let result1 = Compose1 2

// Result is 6
let result2 = Compose2 2

// Pipelining
// Pipeline operator
// ( |> ) : 'T1 -> ('T1 -> 'U) -> 'U
let Pipeline2 x = addOne x |> timesTwo

// Backward pipeline operator
// ( <| ) : ('T -> 'U) -> 'T -> 'U
let Pipeline1 x = addOne <| timesTwo x

// Result is 5
let result3 = Pipeline1 2

// Result is 6
let result4 = Pipeline2 2
```

Overloading Functions

You can overload methods of a type but not functions. For more information, see Methods.

See also

- Values
- F# Language Reference

Recursive Functions: The rec Keyword

8/5/2022 • 3 minutes to read • Edit Online

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

Syntax

```
// Recursive function:
let rec function-name parameter-list =
    function-body

// Mutually recursive functions:
let rec function1-name parameter-list =
    function1-body

and function2-name parameter-list =
    function2-body
...
```

Remarks

Recursive functions - functions that call themselves - are identified explicitly in the F# language with the `rec` keyword. The `rec` keyword makes the name of the `let` binding available in its body.

The following example shows a recursive function that computes the n^{th} Fibonacci number using the mathematical definition.

```
let rec fib n =
    match n with
    | 0 | 1 -> n
    | n -> fib (n-1) + fib (n-2)
```

NOTE

In practice, code like the previous sample is not ideal because it unnecessarily recomputes values that have already been computed. This is because it is not tail recursive, which is explained further in this article.

Methods are implicitly recursive within the type they are defined in, meaning there is no need to add the `rec` keyword. For example:

```
type MyClass() =
    member this.Fib(n) =
        match n with
        | 0 | 1 -> n
        | n -> this.Fib(n-1) + this.Fib(n-2)
```

`let` bindings within classes are not implicitly recursive, though. All `let`-bound functions require the `rec` keyword.

Tail recursion

For some recursive functions, it is necessary to refactor a more "pure" definition to one that is tail recursive. This prevents unnecessary recomputations. For example, the previous Fibonacci number generator can be rewritten like this:

```
let fib n =
    let rec loop acc1 acc2 n =
        match n with
        | 0 -> acc1
        | 1 -> acc2
        | _ ->
            loop acc2 (acc1 + acc2) (n - 1)
    loop 0 1 n
```

Generating a Fibonacci number is a great example of a "naive" algorithm that's mathematically pure but inefficient in practice. While this is a more complicated implementation, several aspects make it efficient in F# while still remaining recursively defined:

- A recursive inner function named `loop`, which is an idiomatic F# pattern.
- Two accumulator parameters, which pass accumulated values to recursive calls.
- A check on the value of `n` to return a specific accumulator.

If this example were written iteratively with a loop, the code would look similar with two different values accumulating numbers until a particular condition was met.

The reason why this is tail-recursive is because the recursive call does not need to save any values on the call stack. All intermediate values being calculated are accumulated via inputs to the inner function. This also allows the F# compiler to optimize the code to be just as fast as if you had written something like a `while` loop.

It's common to write F# code that recursively processes something with an inner and outer function, as the previous example shows. The inner function uses tail recursion, while the outer function has a better interface for callers.

Mutually Recursive Functions

Sometimes functions are *mutually recursive*, meaning that calls form a circle, where one function calls another which in turn calls the first, with any number of calls in between. You must define such functions together in one `let` binding, using the `and` keyword to link them together.

The following example shows two mutually recursive functions.

```
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)
```

Recursive values

You can also define a `let`-bound value to be recursive. This is sometimes done for logging. With F# 5 and the `nameof` function, you can do this:

```
let rec nameDoubles = nameof nameDoubles + nameof nameDoubles
```

See also

- Functions

Inline Functions

8/5/2022 • 3 minutes to read • Edit Online

Inline functions are functions that are integrated directly into the calling code.

Using Inline Functions

When you use static type parameters, any functions that are parameterized by type parameters must be inline. This guarantees that the compiler can resolve these type parameters. When you use ordinary generic type parameters, there is no such restriction.

Other than enabling the use of member constraints, inline functions can be helpful in optimizing code. However, overuse of inline functions can cause your code to be less resistant to changes in compiler optimizations and the implementation of library functions. For this reason, you should avoid using inline functions for optimization unless you have tried all other optimization techniques. Making a function or method inline can sometimes improve performance, but that is not always the case. Therefore, you should also use performance measurements to verify that making any given function inline does in fact have a positive effect.

The `inline` modifier can be applied to functions at the top level, at the module level, or at the method level in a class.

The following code example illustrates an inline function at the top level, an inline instance method, and an inline static method.

```
let inline increment x = x + 1
type WrapInt32() =
    member inline this.incrementByOne(x) = x + 1
    static member inline Increment(x) = x + 1
```

Inline Functions and Type Inference

The presence of `inline` affects type inference. This is because inline functions can have statically resolved type parameters, whereas non-inline functions cannot. The following code example shows a case where `inline` is helpful because you are using a function that has a statically resolved type parameter, the `float` conversion operator.

```
let inline printAsFloatingPoint number =
    printfn "%f" (float number)
```

Without the `inline` modifier, type inference forces the function to take a specific type, in this case `int`. But with the `inline` modifier, the function is also inferred to have a statically resolved type parameter. With the `inline` modifier, the type is inferred to be the following:

```
^a -> unit when ^a : (static member op_Explicit : ^a -> float)
```

This means that the function accepts any type that supports a conversion to **float**.

InlineIfLambda

The F# compiler includes an optimizer that performs inlining of code. The `InlineIfLambda` attribute allows code to optionally indicate that, if an argument is determined to be a lambda function, then that argument should itself always be inlined at call sites. For more information, see F# RFC FS-1098.

For example, consider the following `iterateTwice` function to traverse an array:

```
let inline iterateTwice ([<InlineIfLambda>] action) (array: 'T[]) =
    for i = 0 to array.Length-1 do
        action array[i]
    for i = 0 to array.Length-1 do
        action array[i]
```

If the call site is:

```
let arr = [| 1.. 100 |]
let mutable sum = 0
arr  |> iterateTwice (fun x ->
    sum <- sum + x)
```

Then after inlining and other optimizations, the code becomes:

```
let arr = [| 1..100 |]
let mutable sum = 0
for i = 0 to arr.Length - 1 do
    sum <- sum + arr[i]
for i = 0 to arr.Length - 1 do
    sum <- sum + arr[i]
```

This optimization is applied regardless of the size of the lambda expression involved. This feature can also be used to implement loop unrolling and similar transformations more reliably.

An opt-in warning (`/warnon:3517` or property `<WarnOn>3517</WarnOn>`) can be turned on to indicate places in your code where `InlineIfLambda` arguments are not bound to lambda expressions at call sites. In normal situations, this warning should not be enabled. However, in certain kinds of high-performance programming, it can be useful to ensure all code is inlined and flattened.

See also

- Functions
- Constraints
- Statically Resolved Type Parameters

Lambda Expressions: The fun Keyword (F#)

8/5/2022 • 2 minutes to read • Edit Online

The `fun` keyword is used to define a lambda expression, that is, an anonymous function.

Syntax

```
fun parameter-list -> expression
```

Remarks

The *parameter-list* typically consists of names and, optionally, types of parameters. More generally, the *parameter-list* can be composed of any F# patterns. For a full list of possible patterns, see Pattern Matching. Lists of valid parameters include the following examples.

```
// Lambda expressions with parameter lists.
fun a b c -> ...
fun (a: int) b c -> ...
fun (a : int) (b : string) (c:float) -> ...

// A lambda expression with a tuple pattern.
fun (a, b) -> …

// A lambda expression with a list pattern.
fun head :: tail -> …
```

The *expression* is the body of the function, the last expression of which generates a return value. Examples of valid lambda expressions include the following:

```
fun x -> x + 1
fun a b c -> printfn "%A %A %A" a b c
fun (a: int) (b: int) (c: int) -> a + b * c
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

Using Lambda Expressions

Lambda expressions are especially useful when you want to perform operations on a list or other collection and want to avoid the extra work of defining a function. Many F# library functions take function values as arguments, and it can be especially convenient to use a lambda expression in those cases. The following code applies a lambda expression to elements of a list. In this case, the anonymous function adds 1 to every element of a list.

```
let list = List.map (fun i -> i + 1) [1;2;3]
printfn "%A" list
```

See also

- Functions

Conditional Expressions: `if...then...else`

8/5/2022 • 2 minutes to read • Edit Online

The `if...then...else` expression runs different branches of code and also evaluates to a different value depending on the Boolean expression given.

Syntax

```
if boolean-expression then expression1 [ else expression2 ]
```

Remarks

In the previous syntax, *expression1* runs when the Boolean expression evaluates to `true`; otherwise, *expression2* runs.

Like other languages, the `if...then...else` construct can be used to conditionally execute code. In F#, `if...then...else` is an expression and produces a value by the branch that executes. The types of the expressions in each branch must match.

If there is no explicit `else` branch, the overall type is `unit`, and the type of the `then` branch must also be `unit`.

When chaining `if...then...else` expressions together, you can use the keyword `elif` instead of `else if`; they are equivalent.

Example

The following example illustrates how to use the `if...then...else` expression.

```
let test x y =
  if x = y then "equals"
  elif x < y then "is less than"
  else "is greater than"

printfn "%d %s %d." 10 (test 10 20) 20

printfn "What is your name? "
let nameString = System.Console.ReadLine()

printfn "What is your age? "
let ageString = System.Console.ReadLine()
let age = System.Int32.Parse(ageString)

if age < 10 then
    printfn "You are only %d years old and already learning F#? Wow!" age
```

```
10 is less than 20
What is your name? John
How old are you? 9
You are only 9 years old and already learning F#? Wow!
```

See also

- F# Language Reference

Loops: for...in Expression

8/5/2022 • 3 minutes to read • Edit Online

This looping construct is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.

Syntax

```
for pattern in enumerable-expression do
    body-expression
```

Remarks

The `for...in` expression can be compared to the `for each` statement in other .NET languages because it is used to loop over the values in an enumerable collection. However, `for...in` also supports pattern matching over the collection instead of just iteration over the whole collection.

The enumerable expression can be specified as an enumerable collection or, by using the `..` operator, as a range on an integral type. Enumerable collections include lists, sequences, arrays, sets, maps, and so on. Any type that implements `System.Collections.IEnumerable` can be used.

When you express a range by using the `..` operator, you can use the following syntax.

start .. *finish*

You can also use a version that includes an increment called the *skip*, as in the following code.

start .. *skip* .. *finish*

When you use integral ranges and a simple counter variable as a pattern, the typical behavior is to increment the counter variable by 1 on each iteration, but if the range includes a skip value, the counter is incremented by the skip value instead.

Values matched in the pattern can also be used in the body expression.

The following code examples illustrate the use of the `for...in` expression.

```
// Looping over a list.
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
   printfn "%d" i
```

The output is as follows.

```
1
5
100
450
788
```

The following example shows how to loop over a sequence, and how to use a tuple pattern instead of a simple variable.

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
  printfn "%d squared is %d" a asqr
```

The output is as follows.

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

The following example shows how to loop over a simple integer range.

```
let function1() =
  for i in 1 .. 10 do
    printf "%d " i
  printfn ""
function1()
```

The output of function1 is as follows.

```
1 2 3 4 5 6 7 8 9 10
```

The following example shows how to loop over a range with a skip of 2, which includes every other element of the range.

```
let function2() =
  for i in 1 .. 2 .. 10 do
     printf "%d " i
  printfn ""
function2()
```

The output of `function2` is as follows.

```
1 3 5 7 9
```

The following example shows how to use a character range.

```
let function3() =
  for c in 'a' .. 'z' do
    printf "%c " c
  printfn ""
function3()
```

The output of `function3` is as follows.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

The following example shows how to use a negative skip value for a reverse iteration.

```
let function4() =
    for i in 10 .. -1 .. 1 do
        printf "%d " i
    printfn " ... Lift off!"
function4()
```

The output of `function4` is as follows.

```
10 9 8 7 6 5 4 3 2 1 ... Lift off!
```

The beginning and ending of the range can also be expressions, such as functions, as in the following code.

```
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function5 x y =
  for i in (beginning x y) .. (ending x y) do
     printf "%d " i
  printfn ""

function5 10 4
```

The output of `function5` with this input is as follows.

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

The next example shows the use of a wildcard character (_) when the element is not needed in the loop.

```
let mutable count = 0
for _ in list1 do
   count <- count + 1
printfn "Number of elements in list1: %d" count
```

The output is as follows.

```
Number of elements in list1: 5
```

`Note` You can use `for...in` in sequence expressions and other computation expressions, in which case a customized version of the `for...in` expression is used. For more information, see Sequences, Async expressions, Task expressions, and Computation Expressions.

See also

- F# Language Reference
- Loops: `for...to` Expression
- Loops: `while...do` Expression

Loops: for...to Expression

8/5/2022 • 2 minutes to read • Edit Online

The `for...to` expression is used to iterate in a loop over a range of values of a loop variable.

Syntax

```
for identifier = start [ to | downto ] finish do
    body-expression
```

Remarks

The type of the identifier is inferred from the type of the *start* and *finish* expressions. Types for these expressions must be 32-bit integers.

Although technically an expression, `for...to` is more like a traditional statement in an imperative programming language. The return type for the *body-expression* must be `unit`. The following examples show various uses of the `for...to` expression.

```
// A simple for...to loop.
let function1() =
  for i = 1 to 10 do
    printf "%d " i
  printfn ""

// A for...to loop that counts in reverse.
let function2() =
  for i = 10 downto 1 do
    printf "%d " i
  printfn ""

function1()
function2()

// A for...to loop that uses functions as the start and finish expressions.
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function3 x y =
  for i = (beginning x y) to (ending x y) do
     printf "%d " i
  printfn ""

function3 10 4
```

The output of the previous code is as follows.

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

See also

- F# Language Reference
- Loops: `for...in` Expression
- Loops: `while...do` Expression

Loops: while...do Expression

8/5/2022 • 2 minutes to read • Edit Online

The `while...do` expression is used to perform iterative execution (looping) while a specified test condition is true.

Syntax

```
while test-expression do
    body-expression
```

Remarks

The *test-expression* is evaluated; if it is `true`, the *body-expression* is executed and the test expression is evaluated again. The *body-expression* must have type `unit`. If the test expression is `false`, the iteration ends.

The following example illustrates the use of the `while...do` expression.

```
open System

let lookForValue value maxValue =
  let mutable continueLooping = true
  let randomNumberGenerator = new Random()
  while continueLooping do
    // Generate a random number between 1 and maxValue.
    let rand = randomNumberGenerator.Next(maxValue)
    printf "%d " rand
    if rand = value then
       printfn "\nFound a %d!" value
       continueLooping <- false

lookForValue 10 20
```

The output of the previous code is a stream of random numbers between 1 and 20, the last of which is 10.

```
13 19 8 18 16 2 10
Found a 10!
```

> **NOTE**
>
> You can use `while...do` in sequence expressions and other computation expressions, in which case a customized version of the `while...do` expression is used. For more information, see Sequences, Async expressions, Task expressions, and Computation Expressions.

See also

- F# Language Reference
- Loops: `for...in` Expression
- Loops: `for...to` Expression

Pattern Matching

8/5/2022 • 13 minutes to read • Edit Online

Patterns are rules for transforming input data. They are used throughout F# to compare data with a logical structure or structures, decompose data into constituent parts, or extract information from data in various ways.

Remarks

Patterns are used in many language constructs, such as the `match` expression. They are used when you are processing arguments for functions in `let` bindings, lambda expressions, and in the exception handlers associated with the `try...with` expression. For more information, see Match Expressions, let Bindings, Lambda Expressions: The `fun` Keyword, and Exceptions: The `try...with` Expression.

For example, in the `match` expression, the *pattern* is what follows the pipe symbol.

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

Each pattern acts as a rule for transforming input in some way. In the `match` expression, each pattern is examined in turn to see if the input data is compatible with the pattern. If a match is found, the result expression is executed. If a match is not found, the next pattern rule is tested. The optional when *condition* part is explained in Match Expressions.

Supported patterns are shown in the following table. At run time, the input is tested against each of the following patterns in the order listed in the table, and patterns are applied recursively, from first to last as they appear in your code, and from left to right for the patterns on each line.

NAME	DESCRIPTION	EXAMPLE
Constant pattern	Any numeric, character, or string literal, an enumeration constant, or a defined literal identifier	`1.0`, `"test"`, `30`, `Color.Red`

Identifier pattern	A case value of a discriminated union, an exception label, or an active pattern case	`Some(x)`
		
		`Failure(msg)`

Variable pattern	*identifier*	`a`
`as` pattern	*pattern* as *identifier*	`(a, b) as tuple1`
OR pattern	*pattern1* | *pattern2*	`([h] | [h; _])`
AND pattern	*pattern1* & *pattern2*	`(a, b) & (_, "test")`
Cons pattern	*identifier* :: *list-identifier*	`h :: t`
List pattern	[*pattern_1*; ... ; *pattern_n*]	`[ a; b; c ]`

NAME	DESCRIPTION	EXAMPLE
Array pattern	[| *pattern_1*; ..; *pattern_n* |]	`[| a; b; c |]`
Parenthesized pattern	(*pattern*)	`( a )`
Tuple pattern	(*pattern_1*, ... , *pattern_n*)	`( a, b )`
Record pattern	{ *identifier1* = *pattern_1*; ... ; *identifier_n* = *pattern_n* }	`{ Name = name; }`
Wildcard pattern	_	`_`
Pattern together with type annotation	*pattern* : *type*	`a : int`
Type test pattern	:? *type* [as *identifier*]	`:? System.DateTime as dt`
Null pattern	null	`null`
Nameof pattern	*nameof expr*	`nameof str`

Constant Patterns

Constant patterns are numeric, character, and string literals, enumeration constants (with the enumeration type name included). A `match` expression that has only constant patterns can be compared to a case statement in other languages. The input is compared with the literal value and the pattern matches if the values are equal. The type of the literal must be compatible with the type of the input.

The following example demonstrates the use of literal patterns, and also uses a variable pattern and an OR pattern.

```
[<Literal>]
let Three = 3

let filter123 x =
    match x with
    // The following line contains literal patterns combined with an OR pattern.
    | 1 | 2 | Three -> printfn "Found 1, 2, or 3!"
    // The following line contains a variable pattern.
    | var1 -> printfn "%d" var1

for x in 1..10 do filter123 x
```

Another example of a literal pattern is a pattern based on enumeration constants. You must specify the enumeration type name when you use enumeration constants.

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2

let printColorName (color:Color) =
    match color with
    | Color.Red -> printfn "Red"
    | Color.Green -> printfn "Green"
    | Color.Blue -> printfn "Blue"
    | _ -> ()

printColorName Color.Red
printColorName Color.Green
printColorName Color.Blue
```

Identifier Patterns

If the pattern is a string of characters that forms a valid identifier, the form of the identifier determines how the pattern is matched. If the identifier is longer than a single character and starts with an uppercase character, the compiler tries to make a match to the identifier pattern. The identifier for this pattern could be a value marked with the Literal attribute, a discriminated union case, an exception identifier, or an active pattern case. If no matching identifier is found, the match fails and the next pattern rule, the variable pattern, is compared to the input.

Discriminated union patterns can be simple named cases or they can have a value, or a tuple containing multiple values. If there is a value, you must specify an identifier for the value. In the case of a tuple, you must supply a tuple pattern with an identifier for each element of the tuple or an identifier with a field name for one or more named union fields. See the code examples in this section for examples.

The `option` type is a discriminated union that has two cases, `Some` and `None`. One case (`Some`) has a value, but the other (`None`) is just a named case. Therefore, `Some` needs to have a variable for the value associated with the `Some` case, but `None` must appear by itself. In the following code, the variable `var1` is given the value that is obtained by matching to the `Some` case.

```
let printOption (data : int option) =
    match data with
    | Some var1  -> printfn "%d" var1
    | None -> ()
```

In the following example, the `PersonName` discriminated union contains a mixture of strings and characters that represent possible forms of names. The cases of the discriminated union are `FirstOnly`, `LastOnly`, and `FirstLast`.

```
type PersonName =
    | FirstOnly of string
    | LastOnly of string
    | FirstLast of string * string

let constructQuery personName =
    match personName with
    | FirstOnly(firstName) -> printf "May I call you %s?" firstName
    | LastOnly(lastName) -> printf "Are you Mr. or Ms. %s?" lastName
    | FirstLast(firstName, lastName) -> printf "Are you %s %s?" firstName lastName
```

For discriminated unions that have named fields, you use the equals sign (=) to extract the value of a named field. For example, consider a discriminated union with a declaration like the following.

```
type Shape =
    | Rectangle of height : float * width : float
    | Circle of radius : float
```

You can use the named fields in a pattern matching expression as follows.

```
let matchShape shape =
    match shape with
    | Rectangle(height = h) -> printfn $"Rectangle with length %f{h}"
    | Circle(r) -> printfn $"Circle with radius %f{r}"
```

The use of the named field is optional, so in the previous example, both `Circle(r)` and `Circle(radius = r)` have the same effect.

When you specify multiple fields, use the semicolon (;) as a separator.

```
match shape with
| Rectangle(height = h; width = w) -> printfn $"Rectangle with height %f{h} and width %f{w}"
| _ -> ()
```

Active patterns enable you to define more complex custom pattern matching. For more information about active patterns, see Active Patterns.

The case in which the identifier is an exception is used in pattern matching in the context of exception handlers. For information about pattern matching in exception handling, see Exceptions: The `try...with` Expression.

Variable Patterns

The variable pattern assigns the value being matched to a variable name, which is then available for use in the execution expression to the right of the `->` symbol. A variable pattern alone matches any input, but variable patterns often appear within other patterns, therefore enabling more complex structures such as tuples and arrays to be decomposed into variables.

The following example demonstrates a variable pattern within a tuple pattern.

```
let function1 x =
    match x with
    | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2
    | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2
    | (var1, var2) -> printfn "%d equals %d" var1 var2

function1 (1,2)
function1 (2, 1)
function1 (0, 0)
```

as Pattern

The `as` pattern is a pattern that has an `as` clause appended to it. The `as` clause binds the matched value to a name that can be used in the execution expression of a `match` expression, or, in the case where this pattern is used in a `let` binding, the name is added as a binding to the local scope.

The following example uses an `as` pattern.

```
let (var1, var2) as tuple1 = (1, 2)
printfn "%d %d %A" var1 var2 tuple1
```

OR Pattern

The OR pattern is used when input data can match multiple patterns, and you want to execute the same code as a result. The types of both sides of the OR pattern must be compatible.

The following example demonstrates the OR pattern.

```
let detectZeroOR point =
    match point with
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."
    | _ -> printfn "Both nonzero."
detectZeroOR (0, 0)
detectZeroOR (1, 0)
detectZeroOR (0, 10)
detectZeroOR (10, 15)
```

AND Pattern

The AND pattern requires that the input match two patterns. The types of both sides of the AND pattern must be compatible.

The following example is like `detectZeroTuple` shown in the Tuple Pattern section later in this topic, but here both `var1` and `var2` are obtained as values by using the AND pattern.

```
let detectZeroAND point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (var1, var2) & (0, _) -> printfn "First value is 0 in (%d, %d)" var1 var2
    | (var1, var2)  & (_, 0) -> printfn "Second value is 0 in (%d, %d)" var1 var2
    | _ -> printfn "Both nonzero."
detectZeroAND (0, 0)
detectZeroAND (1, 0)
detectZeroAND (0, 10)
detectZeroAND (10, 15)
```

Cons Pattern

The cons pattern is used to decompose a list into the first element, the *head*, and a list that contains the remaining elements, the *tail*.

```
let list1 = [ 1; 2; 3; 4 ]

// This example uses a cons pattern and a list pattern.
let rec printList l =
    match l with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1
```

List Pattern

The list pattern enables lists to be decomposed into a number of elements. The list pattern itself can match only lists of a specific number of elements.

```
// This example uses a list pattern.
let listLength list =
    match list with
    | [] -> 0
    | [ _ ] -> 1
    | [ _; _ ] -> 2
    | [ _; _; _ ] -> 3
    | _ -> List.length list

printfn "%d" (listLength [ 1 ])
printfn "%d" (listLength [ 1; 1 ])
printfn "%d" (listLength [ 1; 1; 1; ])
printfn "%d" (listLength [ ] )
```

Array Pattern

The array pattern resembles the list pattern and can be used to decompose arrays of a specific length.

```
// This example uses array patterns.
let vectorLength vec =
    match vec with
    | [| var1 |] -> var1
    | [| var1; var2 |] -> sqrt (var1*var1 + var2*var2)
    | [| var1; var2; var3 |] -> sqrt (var1*var1 + var2*var2 + var3*var3)
    | _ -> failwith (sprintf "vectorLength called with an unsupported array size of %d." (vec.Length))

printfn "%f" (vectorLength [| 1. |])
printfn "%f" (vectorLength [| 1.; 1. |])
printfn "%f" (vectorLength [| 1.; 1.; 1.; |])
printfn "%f" (vectorLength [| |] )
```

Parenthesized Pattern

Parentheses can be grouped around patterns to achieve the desired associativity. In the following example, parentheses are used to control associativity between an AND pattern and a cons pattern.

```
let countValues list value =
    let rec checkList list acc =
       match list with
       | (elem1 & head) :: tail when elem1 = value -> checkList tail (acc + 1)
       | head :: tail -> checkList tail acc
       | [] -> acc
    checkList list 0

let result = countValues [ for x in -10..10 -> x*x - 4 ] 0
printfn "%d" result
```

Tuple Pattern

The tuple pattern matches input in tuple form and enables the tuple to be decomposed into its constituent elements by using pattern matching variables for each position in the tuple.

The following example demonstrates the tuple pattern and also uses literal patterns, variable patterns, and the wildcard pattern.

```
let detectZeroTuple point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (0, var2) -> printfn "First value is 0 in (0, %d)" var2
    | (var1, 0) -> printfn "Second value is 0 in (%d, 0)" var1
    | _ -> printfn "Both nonzero."
detectZeroTuple (0, 0)
detectZeroTuple (1, 0)
detectZeroTuple (0, 10)
detectZeroTuple (10, 15)
```

Record Pattern

The record pattern is used to decompose records to extract the values of fields. The pattern does not have to reference all fields of the record; any omitted fields just do not participate in matching and are not extracted.

```
// This example uses a record pattern.

type MyRecord = { Name: string; ID: int }

let IsMatchByName record1 (name: string) =
    match record1 with
    | { MyRecord.Name = nameFound; MyRecord.ID = _; } when nameFound = name -> true
    | _ -> false

let recordX = { Name = "Parker"; ID = 10 }
let isMatched1 = IsMatchByName recordX "Parker"
let isMatched2 = IsMatchByName recordX "Hartono"
```

Wildcard Pattern

The wildcard pattern is represented by the underscore (`_`) character and matches any input, just like the variable pattern, except that the input is discarded instead of assigned to a variable. The wildcard pattern is often used within other patterns as a placeholder for values that are not needed in the expression to the right of the `->` symbol. The wildcard pattern is also frequently used at the end of a list of patterns to match any unmatched input. The wildcard pattern is demonstrated in many code examples in this topic. See the preceding code for one example.

Patterns That Have Type Annotations

Patterns can have type annotations. These behave like other type annotations and guide inference like other type annotations. Parentheses are required around type annotations in patterns. The following code shows a pattern that has a type annotation.

```
let detect1 x =
    match x with
    | 1 -> printfn "Found a 1!"
    | (var1 : int) -> printfn "%d" var1
detect1 0
detect1 1
```

Type Test Pattern

The type test pattern is used to match the input against a type. If the input type is a match to (or a derived type of) the type specified in the pattern, the match succeeds.

The following example demonstrates the type test pattern.

```
open System.Windows.Forms

let RegisterControl(control:Control) =
    match control with
    | :? Button as button -> button.Text <- "Registered."
    | :? CheckBox as checkbox -> checkbox.Text <- "Registered."
    | _ -> ()
```

If you're only checking if an identifier is of a particular derived type, you don't need the `as identifier` part of the pattern, as shown in the following example:

```
type A() = class end
type B() = inherit A()
type C() = inherit A()

let m (a: A) =
    match a with
    | :? B -> printfn "It's a B"
    | :? C -> printfn "It's a C"
    | _ -> ()
```

Null Pattern

The null pattern matches the null value that can appear when you are working with types that allow a null value. Null patterns are frequently used when interoperating with .NET Framework code. For example, the return value of a .NET API might be the input to a `match` expression. You can control program flow based on whether the return value is null, and also on other characteristics of the returned value. You can use the null pattern to prevent null values from propagating to the rest of your program.

The following example uses the null pattern and the variable pattern.

```
let ReadFromFile (reader : System.IO.StreamReader) =
    match reader.ReadLine() with
    | null -> printfn "\n"; false
    | line -> printfn "%s" line; true

let fs = System.IO.File.Open("..\..\Program.fs", System.IO.FileMode.Open)
let sr = new System.IO.StreamReader(fs)
while ReadFromFile(sr) = true do ()
sr.Close()
```

Nameof pattern

The `nameof` pattern matches against a string when its value is equal to the expression that follows the `nameof` keyword. for example:

```
let f (str: string) =
    match str with
    | nameof str -> "It's 'str'!"
    | _ -> "It is not 'str'!"

f "str" // matches
f "asdf" // does not match
```

See the `nameof` operator for information on what you can take a name of.

See also

- Match Expressions
- Active Patterns
- F# Language Reference

Match expressions

8/5/2022 • 3 minutes to read • Edit Online

The `match` expression provides branching control that is based on the comparison of an expression with a set of patterns.

Syntax

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Remarks

The pattern matching expressions allow for complex branching based on the comparison of a test expression with a set of patterns. In the `match` expression, the *test-expression* is compared with each pattern in turn, and when a match is found, the corresponding *result-expression* is evaluated and the resulting value is returned as the value of the match expression.

The pattern matching function shown in the previous syntax is a lambda expression in which pattern matching is performed immediately on the argument. The pattern matching function shown in the previous syntax is equivalent to the following.

```
fun arg ->
    match arg with
    | pattern1 [ when condition ] -> result-expression1
    | pattern2 [ when condition ] -> result-expression2
    | ...
```

For more information about lambda expressions, see Lambda Expressions: The `fun` Keyword.

The whole set of patterns should cover all the possible matches of the input variable. Frequently, you use the wildcard pattern (`_`) as the last pattern to match any previously unmatched input values.

The following code illustrates some of the ways in which the `match` expression is used. For a reference and examples of all the possible patterns that can be used, see Pattern Matching.

```
let list1 = [ 1; 5; 100; 450; 788 ]

// Pattern matching by using the cons pattern and a list
// pattern that tests for an empty list.
let rec printList listx =
    match listx with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1

// Pattern matching with multiple alternatives on the same line.
let filter123 x =
    match x with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
    | a -> printfn "%d" a

// The same function written with the pattern matching
// function syntax.
let filterNumbers =
    function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
             | a -> printfn "%d" a
```

Guards on patterns

You can use a `when` clause to specify an additional condition that the variable must satisfy to match a pattern. Such a clause is referred to as a *guard*. The expression following the `when` keyword is not evaluated unless a match is made to the pattern associated with that guard.

The following example illustrates the use of a guard to specify a numeric range for a variable pattern. Note that multiple conditions are combined by using Boolean operators.

```
let rangeTest testValue mid size =
    match testValue with
    | var1 when var1 >= mid - size/2 && var1 <= mid + size/2 -> printfn "The test value is in range."
    | _ -> printfn "The test value is out of range."

rangeTest 10 20 5
rangeTest 10 20 10
rangeTest 10 20 40
```

Note that because values other than literals cannot be used in the pattern, you must use a `when` clause if you have to compare some part of the input against a value. This is shown in the following code:

```
// This example uses patterns that have when guards.
let detectValue point target =
    match point with
    | (a, b) when a = target && b = target -> printfn "Both values match target %d." target
    | (a, b) when a = target -> printfn "First value matched target in (%d, %d)" target b
    | (a, b) when b = target -> printfn "Second value matched target in (%d, %d)" a target
    | _ -> printfn "Neither value matches target."
detectValue (0, 0) 0
detectValue (1, 0) 0
detectValue (0, 10) 0
detectValue (10, 15) 0
```

Note that when a union pattern is covered by a guard, the guard applies to **all** of the patterns, not just the last one. For example, given the following code, the guard `when a > 12` applies to both `A a` and `B a`:

```
type Union =
    | A of int
    | B of int

let foo() =
    let test = A 42
    match test with
    | A a
    | B a when a > 41 -> a // the guard applies to both patterns
    | _ -> 1

foo() // returns 42
```

See also

- F# Language Reference
- Active Patterns
- Pattern Matching

Active Patterns

8/5/2022 • 7 minutes to read • Edit Online

Active patterns enable you to define named partitions that subdivide input data, so that you can use these names in a pattern matching expression just as you would for a discriminated union. You can use active patterns to decompose data in a customized manner for each partition.

Syntax

```
// Active pattern of one choice.
let (|identifier|) [arguments] valueToMatch = expression

// Active Pattern with multiple choices.
// Uses a FSharp.Core.Choice<_,...,_> based on the number of case names. In F#, the limitation n <= 7
applies.
let (|identifier1|identifier2|...|) valueToMatch = expression

// Partial active pattern definition.
// Uses a FSharp.Core.option<_> to represent if the type is satisfied at the call site.
let (|identifier|_|) [arguments] valueToMatch = expression
```

Remarks

In the previous syntax, the identifiers are names for partitions of the input data that is represented by *arguments*, or, in other words, names for subsets of the set of all values of the arguments. There can be up to seven partitions in an active pattern definition. The *expression* describes the form into which to decompose the data. You can use an active pattern definition to define the rules for determining which of the named partitions the values given as arguments belong to. The (| and |) symbols are referred to as *banana clips* and the function created by this type of let binding is called an *active recognizer*.

As an example, consider the following active pattern with an argument.

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

You can use the active pattern in a pattern matching expression, as in the following example.

```
let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

The output of this program is as follows:

```
7 is odd
11 is odd
32 is even
```

Another use of active patterns is to decompose data types in multiple ways, such as when the same underlying data has various possible representations. For example, a `Color` object could be decomposed into an RGB representation or an HSB representation.

```
open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
   ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: System.Drawing.Color) =
   match col with
   | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: System.Drawing.Color) =
   match col with
   | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
  printfn "%s" colorString
  printRGB col
  printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

The output of the above program is as follows:

```
Red
 Red: 255 Green: 0 Blue: 0
 Hue: 360.000000 Saturation: 1.000000 Brightness: 0.500000
Black
 Red: 0 Green: 0 Blue: 0
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
 Red: 255 Green: 255 Blue: 255
 Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
 Red: 128 Green: 128 Blue: 128
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
 Red: 255 Green: 235 Blue: 205
 Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

In combination, these two ways of using active patterns enable you to partition and decompose data into just the appropriate form and perform the appropriate computations on the appropriate data in the form most convenient for the computation.

The resulting pattern matching expressions enable data to be written in a convenient way that is very readable, greatly simplifying potentially complex branching and data analysis code.

Partial Active Patterns

Sometimes, you need to partition only part of the input space. In that case, you write a set of partial patterns each of which match some inputs but fail to match other inputs. Active patterns that do not always produce a value are called *partial active patterns*; they have a return value that is an option type. To define a partial active pattern, you use a wildcard character (_) at the end of the list of patterns inside the banana clips. The following code illustrates the use of a partial active pattern.

```
let (|Integer|_|) (str: string) =
   let mutable intvalue = 0
   if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
   else None

let (|Float|_|) (str: string) =
   let mutable floatvalue = 0.0
   if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
   else None

let parseNumeric str =
   match str with
     | Integer i -> printfn "%d : Integer" i
     | Float f -> printfn "%f : Floating point" f
     | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

The output of the previous example is as follows:

```
1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.
```

When using partial active patterns, sometimes the individual choices can be disjoint or mutually exclusive, but they need not be. In the following example, the pattern Square and the pattern Cube are not disjoint, because some numbers are both squares and cubes, such as 64. The following program uses the AND pattern to combine the Square and Cube patterns. It prints out all integers up to 1000 that are both squares and cubes, as well as those which are only cubes.

```
let err = 1.e-10

let isNearlyIntegral (x:float) = abs (x - round(x)) < err

let (|Square|_|) (x : int) =
  if isNearlyIntegral (sqrt (float x)) then Some(x)
  else None

let (|Cube|_|) (x : int) =
  if isNearlyIntegral ((float x) ** ( 1.0 / 3.0)) then Some(x)
  else None

let findSquareCubes x =
   match x with
       | Cube x & Square _ -> printfn "%d is a cube and a square" x
       | Cube x -> printfn "%d is a cube" x
       | _ -> ()


[ 1 .. 1000 ] |> List.iter (fun elem -> findSquareCubes elem)
```

The output is as follows:

```
1 is a cube and a square
8 is a cube
27 is a cube
64 is a cube and a square
125 is a cube
216 is a cube
343 is a cube
512 is a cube
729 is a cube and a square
1000 is a cube
```

Parameterized Active Patterns

Active patterns always take at least one argument for the item being matched, but they may take additional arguments as well, in which case the name *parameterized active pattern* applies. Additional arguments allow a general pattern to be specialized. For example, active patterns that use regular expressions to parse strings often include the regular expression as an extra parameter, as in the following code, which also uses the partial active pattern `Integer` defined in the previous code example. In this example, strings that use regular expressions for various date formats are given to customize the general ParseRegex active pattern. The Integer active pattern is used to convert the matched strings into integers that can be passed to the DateTime constructor.

```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
   let m = Regex(regex).Match(str)
   if m.Success
   then Some (List.tail [ for x in m.Groups -> x.Value ])
   else None

// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
   match str with
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y + 2000, m, d)
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y, m, d)
     | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Integer d]
          -> new System.DateTime(y, m, d)
     | _ -> new System.DateTime()

let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.ToString())
```

The output of the previous code is as follows:

```
12/22/2008 12:00:00 AM 1/1/2009 12:00:00 AM 1/15/2008 12:00:00 AM 12/28/1995 12:00:00 AM
```

Active patterns are not restricted only to pattern matching expressions, you can also use them on let-bindings.

```
let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e

let greet (Default "random citizen" name) =
    printfn "Hello, %s!" name

greet None
greet (Some "George")
```

The output of the previous code is as follows:

```
Hello, random citizen!
Hello, George!
```

Note however that only single-case active patterns can be parameterized.

```
// A single-case partial active pattern can be parameterized
let (| Foo|_|) s x = if x = s then Some Foo else None
// A multi-case active patterns cannot be parameterized
// let (| Even|Odd|Special |) (s: int) (x: int) = if x = s then Special elif x % 2 = 0 then Even else Odd
```

Struct Representations for Partial Active Patterns

By default, partial active patterns return an `option` value, which will involve an allocation for the `Some` value on a successful match. Alternatively, you can use a value option as a return value through the use of the `Struct` attribute:

```
open System

[<return: Struct>]
let (|Int|_|) str =
   match Int32.TryParse(str) with
   | (true, n) -> ValueSome n
   | _ -> ValueNone
```

The attribute must be specified, because the use of a struct return is not inferred from simply changing the return type to `ValueOption`. For more information, see RFC FS-1039.

See also

- F# Language Reference
- Match Expressions

Exception Handling

8/5/2022 • 2 minutes to read • Edit Online

This section contains information about exception handling support in F#.

Exception Handling Basics

Exception handling is the standard way of handling error conditions in the .NET Framework. Thus, any .NET language must support this mechanism, including F#. An *exception* is an object that encapsulates information about an error. When errors occur, exceptions are raised and regular execution stops. Instead, the runtime searches for an appropriate handler for the exception. The search starts in the current function, and proceeds up the stack through the layers of callers until a matching handler is found. Then the handler is executed.

In addition, as the stack is unwound, the runtime executes any code in `finally` blocks, to guarantee that objects are cleaned up correctly during the unwinding process.

Related Topics

TITLE	DESCRIPTION
Exception Types	Describes how to declare an exception type.
Exceptions: The `try...with` Expression	Describes the language construct that supports exception handling.
Exceptions: The `try...finally` Expression	Describes the language construct that enables you to execute clean-up code as the stack unwinds when an exception is thrown.
Exceptions: the `raise` Function	Describes how to throw an exception object.
Exceptions: The `failwith` Function	Describes how to generate a general F# exception.
Exceptions: The `invalidArg` Function	Describes how to generate an invalid argument exception.

Exception Types

8/5/2022 • 2 minutes to read • Edit Online

There are two categories of exceptions in F#: .NET exception types and F# exception types. This topic describes how to define and use F# exception types.

Syntax

```
exception exception-type of argument-type
```

Remarks

In the previous syntax, *exception-type* is the name of a new F# exception type, and *argument-type* represents the type of an argument that can be supplied when you raise an exception of this type. You can specify multiple arguments by using a tuple type for *argument-type*.

A typical definition for an F# exception resembles the following.

```
exception MyError of string
```

You can generate an exception of this type by using the `raise` function, as follows.

```
raise (MyError("Error message"))
```

You can use an F# exception type directly in the filters in a `try...with` expression, as shown in the following example.

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

The exception type that you define with the `exception` keyword in F# is a new type that inherits from `System.Exception`.

See also

- Exception Handling
- Exceptions: the `raise` Function
- Exception Hierarchy

Exceptions: The try...with Expression

8/5/2022 • 3 minutes to read • Edit Online

This topic describes the `try...with` expression, the expression that is used for exception handling in F#.

Syntax

```
try
    expression1
with
| pattern1 -> expression2
| pattern2 -> expression3
...
```

Remarks

The `try...with` expression is used to handle exceptions in F#. It is similar to the `try...catch` statement in C#. In the preceding syntax, the code in *expression1* might generate an exception. The `try...with` expression returns a value. If no exception is thrown, the whole expression returns the value of *expression1*. If an exception is thrown, each *pattern* is compared in turn with the exception, and for the first matching pattern, the corresponding *expression*, known as the *exception handler*, for that branch is executed, and the overall expression returns the value of the expression in that exception handler. If no pattern matches, the exception propagates up the call stack until a matching handler is found. The types of the values returned from each expression in the exception handlers must match the type returned from the expression in the `try` block.

Frequently, the fact that an error occurred also means that there is no valid value that can be returned from the expressions in each exception handler. A frequent pattern is to have the type of the expression be an option type. The following code example illustrates this pattern.

```
let divide1 x y =
   try
      Some (x / y)
   with
      | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divide1 100 0
```

Exceptions can be .NET exceptions, or they can be F# exceptions. You can define F# exceptions by using the `exception` keyword.

You can use a variety of patterns to filter on the exception type and other conditions; the options are summarized in the following table.

PATTERN	DESCRIPTION
:? *exception-type*	Matches the specified .NET exception type.
:? *exception-type* as *identifier*	Matches the specified .NET exception type, but gives the exception a named value.

PATTERN	DESCRIPTION
exception-name(*arguments*)	Matches an F# exception type and binds the arguments.
identifier	Matches any exception and binds the name to the exception object. Equivalent to **:? System.Exception as** *identifier*
identifier when *condition*	Matches any exception if the condition is true.

Examples

The following code examples illustrate the use of the various exception handler patterns.

```
// This example shows the use of the as keyword to assign a name to a
// .NET exception.
let divide2 x y =
  try
    Some( x / y )
  with
    | :? System.DivideByZeroException as ex -> printfn "Exception! %s " (ex.Message); None

// This version shows the use of a condition to branch to multiple paths
// with the same exception.
let divide3 x y flag =
  try
     x / y
  with
     | ex when flag -> printfn "TRUE: %s" (ex.ToString()); 0
     | ex when not flag -> printfn "FALSE: %s" (ex.ToString()); 1

let result2 = divide3 100 0 true

// This version shows the use of F# exceptions.
exception Error1 of string
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

NOTE

The `try...with` construct is a separate expression from the `try...finally` expression. Therefore, if your code requires both a `with` block and a `finally` block, you will have to nest the two expressions.

NOTE

You can use `try...with` in async expressions, task expressions, and other computation expressions, in which case a customized version of the `try...with` expression is used. For more information, see Async Expressions, Task Expressions, and Computation Expressions.

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...finally` Expression

Exceptions: The try...finally Expression

8/5/2022 • 2 minutes to read • Edit Online

The `try...finally` expression enables you to execute clean-up code even if a block of code throws an exception.

Syntax

```
try
    expression1
finally
    expression2
```

Remarks

The `try...finally` expression can be used to execute the code in *expression2* in the preceding syntax regardless of whether an exception is generated during the execution of *expression1*.

The type of *expression2* does not contribute to the value of the whole expression; the type returned when an exception does not occur is the last value in *expression1*. When an exception does occur, no value is returned and the flow of control transfers to the next matching exception handler up the call stack. If no exception handler is found, the program terminates. Before the code in a matching handler is executed or the program terminates, the code in the `finally` branch is executed.

The following code demonstrates the use of the `try...finally` expression.

```
let divide x y =
   let stream : System.IO.FileStream = System.IO.File.Create("test.txt")
   let writer : System.IO.StreamWriter = new System.IO.StreamWriter(stream)
   try
      writer.WriteLine("test1")
      Some( x / y )
   finally
      writer.Flush()
      printfn "Closing stream"
      stream.Close()

let result =
  try
     divide 100 0
  with
     | :? System.DivideByZeroException -> printfn "Exception handled."; None
```

The output to the console is as follows.

```
Closing stream
Exception handled.
```

As you can see from the output, the stream was closed before the outer exception was handled, and the file `test.txt` contains the text `test1`, which indicates that the buffers were flushed and written to disk even though the exception transferred control to the outer exception handler.

Note that the `try...with` construct is a separate construct from the `try...finally` construct. Therefore, if your code requires both a `with` block and a `finally` block, you have to nest the two constructs, as in the following code example.

```
exception InnerError of string
exception OuterError of string

let function1 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."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

In the context of computation expressions, including sequence expressions and async expressions, **try...finally** expressions can have a custom implementation. For more information, see Computation Expressions.

See also

- Exception Handling
- Exceptions: The `try...with` Expression

Resource Management: The use Keyword

8/5/2022 • 3 minutes to read • Edit Online

This topic describes the keyword `use` and the `using` function, which can control the initialization and release of resources.

Resources

The term *resource* is used in more than one way. Yes, resources can be data that an application uses, such as strings, graphics, and the like, but in this context, *resources* refers to software or operating system resources, such as graphics device contexts, file handles, network and database connections, concurrency objects such as wait handles, and so on. The use of these resources by applications involves the acquisition of the resource from the operating system or other resource provider, followed by the later release of the resource to the pool so that it can be provided to another application. Problems occur when applications do not release resources back to the common pool.

Managing Resources

To efficiently and responsibly manage resources in an application, you must release resources promptly and in a predictable manner. The .NET Framework helps you do this by providing the `System.IDisposable` interface. A type that implements `System.IDisposable` has the `System.IDisposable.Dispose` method, which correctly frees resources. Well-written applications guarantee that `System.IDisposable.Dispose` is called promptly when any object that holds a limited resource is no longer needed. Fortunately, most .NET languages provide support to make this easier, and F# is no exception. There are two useful language constructs that support the dispose pattern: the `use` binding and the `using` function.

use Binding

The `use` keyword has a form that resembles that of the `let` binding:

use *value* = *expression*

It provides the same functionality as a `let` binding but adds a call to `Dispose` on the value when the value goes out of scope. Note that the compiler inserts a null check on the value, so that if the value is `null`, the call to `Dispose` is not attempted.

The following example shows how to close a file automatically by using the `use` keyword.

```
open System.IO

let writetofile filename obj =
   use file1 = File.CreateText(filename)
   file1.WriteLine("{0}", obj.ToString() )
   // file1.Dispose() is called implicitly here.

writetofile "abc.txt" "Humpty Dumpty sat on a wall."
```

> **NOTE**
>
> You can use `use` in computation expressions, in which case a customized version of the `use` expression is used. For more information, see Sequences, Async expressions, Task expressions, and Computation Expressions.

using Function

The `using` function has the following form:

`using` (*expression1*) *function-or-lambda*

In a `using` expression, *expression1* creates the object that must be disposed. The result of *expression1* (the object that must be disposed) becomes an argument, *value*, to *function-or-lambda*, which is either a function that expects a single remaining argument of a type that matches the value produced by *expression1*, or a lambda expression that expects an argument of that type. At the end of the execution of the function, the runtime calls `Dispose` and frees the resources (unless the value is `null`, in which case the call to Dispose is not attempted).

The following example demonstrates the `using` expression with a lambda expression.

```
open System.IO

let writetofile2 filename obj =
    using (System.IO.File.CreateText(filename)) ( fun file1 ->
        file1.WriteLine("{0}", obj.ToString() )
    )

writetofile2 "abc2.txt" "The quick sly fox jumps over the lazy brown dog."
```

The next example shows the `using` expression with a function.

```
let printToFile (file1 : System.IO.StreamWriter) =
    file1.WriteLine("Test output");

using (System.IO.File.CreateText("test.txt")) printToFile
```

Note that the function could be a function that has some arguments applied already. The following code example demonstrates this. It creates a file that contains the string `XYZ`.

```
let printToFile2 obj (file1 : System.IO.StreamWriter) =
    file1.WriteLine(obj.ToString())

using (System.IO.File.CreateText("test.txt")) (printToFile2 "XYZ")
```

The `using` function and the `use` binding are nearly equivalent ways to accomplish the same thing. The `using` keyword provides more control over when `Dispose` is called. When you use `using`, `Dispose` is called at the end of the function or lambda expression; when you use the `use` keyword, `Dispose` is called at the end of the containing code block. In general, you should prefer to use `use` instead of the `using` function.

See also

- F# Language Reference

Exceptions: raise and reraise functions

8/5/2022 • 2 minutes to read • Edit Online

- The `raise` function is used to indicate that an error or exceptional condition has occurred. Information about the error is captured in an exception object.
- The `reraise` function is used to propagate a handled exception up the call chain.

Syntax

```
raise (expression)
```

Remarks

The `raise` function generates an exception object and initiates a stack unwinding process. The stack unwinding process is managed by the common language runtime (CLR), so the behavior of this process is the same as it is in any other .NET language. The stack unwinding process is a search for an exception handler that matches the generated exception. The search starts in the current `try...with` expression, if there is one. Each pattern in the `with` block is checked, in order. When a matching exception handler is found, the exception is considered handled; otherwise, the stack is unwound and `with` blocks up the call chain are checked until a matching handler is found. Any `finally` blocks that are encountered in the call chain are also executed in sequence as the stack unwinds.

The `raise` function is the equivalent of `throw` in C# or C++.

The following code examples illustrate the use of the `raise` function to generate an exception.

```
exception InnerError of string
exception OuterError of string

let function1 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."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

The `raise` function can also be used to raise .NET exceptions, as shown in the following example.

```
let divide x y =
  if (y = 0) then raise (System.ArgumentException("Divisor cannot be zero!"))
  else
     x / y
```

Reraising an exception

The `reraise` function can be used in a `with` block to propagate a handled exception up the call chain. `reraise` does not take an exception operand. It's most useful when a method passes on an argument from a caller to some other library method, and the library method raises an exception that must be passed on to the caller.

The `reraise` function may not be used on the `with` block of `try`/`with` constructs in computed lists, arrays, sequences, or computation expressions including `task { .. }` or `async { .. }`.

```
open System

let getFirstCharacter(value: string) =
    try
        value[0]
    with :? IndexOutOfRangeException as e ->
        reraise()

let s = getFirstCharacter("")
Console.WriteLine($"The first character is {s}")

// The example displays the following output:
//   System.IndexOutOfRangeException: Index was outside the bounds of the array.
//      at System.String.get_Chars(Int32 index)
//      at getFirstCharacter(String value)
//      at <StartupCode>.main@()
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: The `failwith` Function
- Exceptions: The `invalidArg` Function

Exceptions: The failwith Function

8/5/2022 • 2 minutes to read • Edit Online

The `failwith` function generates an F# exception.

Syntax

```
failwith error-message-string
```

Remarks

The *error-message-string* in the previous syntax is a literal string or a value of type `string`. It becomes the `Message` property of the exception.

The exception that is generated by `failwith` is a `System.Exception` exception, which is a reference that has the name `Failure` in F# code. The following code illustrates the use of `failwith` to throw an exception.

```
let divideFailwith x y =
  if (y = 0) then failwith "Divisor cannot be zero."
  else
    x / y

let testDivideFailwith x y =
  try
     divideFailwith x y
  with
     | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: the `raise` Function

Exceptions: The invalidArg Function

8/5/2022 • 2 minutes to read • Edit Online

The `invalidArg` function generates an argument exception.

Syntax

```
invalidArg parameter-name error-message-string
```

Remarks

The parameter-name in the previous syntax is a string with the name of the parameter whose argument was invalid. The *error-message-string* is a literal string or a value of type `string`. It becomes the `Message` property of the exception object.

The exception generated by `invalidArg` is a `System.ArgumentException` exception. The following code illustrates the use of `invalidArg` to throw an exception.

```
let months = [| "January"; "February"; "March"; "April";
                "May"; "June"; "July"; "August"; "September";
                "October"; "November"; "December" |]

let lookupMonth month =
   if (month > 12 || month < 1)
     then invalidArg (nameof month) (sprintf "Value passed in was %d." month)
   months[month - 1]

printfn "%s" (lookupMonth 12)
printfn "%s" (lookupMonth 1)
printfn "%s" (lookupMonth 13)
```

The output is the following, followed by a stack trace (not shown).

```
December
January
System.ArgumentException: Value passed in was 13. (Parameter 'month')
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: the `raise` Function
- Exceptions: The `failwith` Function

Assertions

8/5/2022 • 2 minutes to read • Edit Online

The `assert` expression is a debugging feature that you can use to test an expression. Upon failure in Debug mode, an assertion generates a system error dialog box.

Syntax

```
assert condition
```

Remarks

The `assert` expression has type `bool -> unit`.

The `assert` function resolves to Debug.Assert. This means its behavior is identical to having called Debug.Assert directly.

Assertion checking is enabled only when you compile in Debug mode; that is, if the constant `DEBUG` is defined. In the project system, by default, the `DEBUG` constant is defined in the Debug configuration but not in the Release configuration.

The assertion failure error cannot be caught by using F# exception handling.

Example

The following code example illustrates the use of the `assert` expression.

```
let subtractUnsigned (x : uint32) (y : uint32) =
    assert (x > y)
    let z = x - y
    z
// This code does not generate an assertion failure.
let result1 = subtractUnsigned 2u 1u
// This code generates an assertion failure.
let result2 = subtractUnsigned 1u 2u
```

See also

- F# Language Reference

F# Types

8/5/2022 • 4 minutes to read • Edit Online

This topic describes the types that are used in F# and how F# types are named and described.

Summary of F# Types

Some types are considered *primitive types*, such as the Boolean type `bool` and integral and floating point types of various sizes, which include types for bytes and characters. These types are described in Primitive Types.

Other types that are built into the language include tuples, lists, arrays, sequences, records, and discriminated unions. If you have experience with other .NET languages and are learning F#, you should read the topics for each of these types. These F#-specific types support styles of programming that are common to functional programming languages. Many of these types have associated modules in the F# library that support common operations on these types.

The type of a function includes information about the parameter types and return type.

The .NET Framework is the source of object types, interface types, delegate types, and others. You can define your own object types just as you can in any other .NET language.

Also, F# code can define aliases, which are named *type abbreviations*, that are alternative names for types. You might use type abbreviations when the type might change in the future and you want to avoid changing the code that depends on the type. Or, you might use a type abbreviation as a friendly name for a type that can make code easier to read and understand.

F# provides useful collection types that are designed with functional programming in mind. Using these collection types helps you write code that is more functional in style. For more information, see F# Collection Types.

Syntax for Types

In F# code, you often have to write out the names of types. Every type has a syntactic form, and you use these syntactic forms in type annotations, abstract method declarations, delegate declarations, signatures, and other constructs. Whenever you declare a new program construct in the interpreter, the interpreter prints the name of the construct and the syntax for its type. This syntax might be just an identifier for a user-defined type or a built-in identifier such as for `int` or `string`, but for more complex types, the syntax is more complex.

The following table shows aspects of the type syntax for F# types.

TYPE	TYPE SYNTAX	EXAMPLES

primitive type	*type-name*	`int`
		`float`
		`string`

aggregate type (class, structure, union, record, enum, and so on)	*type-name*	`System.DateTime` `Color`

TYPE	TYPE SYNTAX	EXAMPLES
type abbreviation	*type-abbreviation-name*	`bigint`
fully qualified type	*namespaces.type-name* or *modules.type-name* or *namespaces.modules.type-name*	`System.IO.StreamWriter`

array	*type-name*[] or
	
	type-name array
		`int[]`
		
		`array<int>`
		
		`int array`

two-dimensional array	*type-name*[,]	`int[,]`
		
		`float[,]`

three-dimensional array	*type-name*[,,]	`float[,,]`
tuple	*type-name1* * *type-name2* ...	For example, `(1,'b',3)` has type `int * char * int`

generic type	*type-parameter generic-type-name*
	
	or
	
	generic-type-name< *type-parameter-list*>
		`'a list`
		
		`list<'a>`
		
		`Dictionary<'key, 'value>`

constructed type (a generic type that has a specific type argument supplied)	*type-argument generic-type-name*
	
	or
	
	generic-type-name< *type-argument-list*>
		`int option`
		
		`string list`
		
		`int ref`
		
		`option<int>`
		
		`list<string>`
		
		`ref<int>`
		
		`Dictionary<int, string>`

function type that has a single parameter	*parameter-type1* -> *return-type*	A function that takes an `int` and returns a `string` has type `int -> string`
function type that has multiple parameters	*parameter-type1* -> *parameter-type2* -> ... -> *return-type*	A function that takes an `int` and a `float` and returns a `string` has type `int -> float -> string`

TYPE	TYPE SYNTAX	EXAMPLES
higher order function as a parameter	(*function-type*)	`List.map` has type `('a -> 'b) -> 'a list -> 'b list`
delegate	delegate of *function-type*	`delegate of unit -> int`
flexible type	#*type-name*	`#System.Windows.Forms.Control` `#seq<int>`

Related Topics

TOPIC	DESCRIPTION
Primitive Types	Describes built-in simple types such as integral types, the Boolean type, and character types.
Unit Type	Describes the `unit` type, a type that has one value and that is indicated by (); equivalent to `void` in C# and `Nothing` in Visual Basic.
Tuples	Describes the tuple type, a type that consists of associated values of any type grouped in pairs, triples, quadruples, and so on.
Options	Describes the option type, a type that may either have a value or be empty.
Lists	Describes lists, which are ordered, immutable series of elements all of the same type.
Arrays	Describes arrays, which are ordered sets of mutable elements of the same type that occupy a contiguous block of memory and are of fixed size.
Sequences	Describes the sequence type, which represents a logical series of values; individual values are computed only as necessary.
Records	Describes the record type, a small aggregate of named values.
Discriminated Unions	Describes the discriminated union type, a type whose values can be any one of a set of possible types.
Functions	Describes function values.
Classes	Describes the class type, an object type that corresponds to a .NET reference type. Class types can contain members, properties, implemented interfaces, and a base type.

TOPIC	DESCRIPTION
Structs	Describes the `struct` type, an object type that corresponds to a .NET value type. The `struct` type usually represents a small aggregate of data.
Interfaces	Describes interface types, which are types that represent a set of members that provide certain functionality but that contain no data. An interface type must be implemented by an object type to be useful.
Delegates	Describes the delegate type, which represents a function as an object.
Enumerations	Describes enumeration types, whose values belong to a set of named values.
Attributes	Describes attributes, which are used to specify metadata for another type.
Exception Types	Describes exceptions, which specify error information.

Basic types

8/5/2022 • 2 minutes to read • Edit Online

This topic lists the basic types that are defined in F#. These types are the most fundamental in F#, forming the basis of nearly every F# program. They are a superset of .NET primitive types.

TYPE	.NET TYPE	DESCRIPTION	EXAMPLE
`bool`	Boolean	Possible values are `true` and `false`.	`true` / `false`
`byte`	Byte	Values from 0 to 255.	`1uy`
`sbyte`	SByte	Values from -128 to 127.	`1y`
`int16`	Int16	Values from -32768 to 32767.	`1s`
`uint16`	UInt16	Values from 0 to 65535.	`1us`
`int`	Int32	Values from -2,147,483,648 to 2,147,483,647.	`1`
`uint`	UInt32	Values from 0 to 4,294,967,295.	`1u`
`int64`	Int64	Values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	`1L`
`uint64`	UInt64	Values from 0 to 18,446,744,073,709,551,615.	`1UL`
`nativeint`	IntPtr	A native pointer as a signed integer.	`nativeint 1`
`unativeint`	UIntPtr	A native pointer as an unsigned integer.	`unativeint 1`
`decimal`	Decimal	A floating point data type that has at least 28 significant digits.	`1.0m`
`float`, `double`	Double	A 64-bit floating point type.	`1.0`
`float32`, `single`	Single	A 32-bit floating point type.	`1.0f`
`char`	Char	Unicode character values.	`'c'`

TYPE	.NET TYPE	DESCRIPTION	EXAMPLE
`string`	String	Unicode text.	`"str"`
`unit`	not applicable	Indicates the absence of an actual value. The type has only one formal value, which is denoted `()`. The unit value, `()`, is often used as a placeholder where a value is needed but no real value is available or makes sense.	`()`

> **NOTE**
> You can perform computations with integers too big for the 64-bit integer type by using the `bigint` type. `bigint` is not considered a basic type; it is an abbreviation for `System.Numerics.BigInteger`.

See also

- F# Language Reference

Unit Type

8/5/2022 • 2 minutes to read • Edit Online

The `unit` type is a type that indicates the absence of a specific value; the `unit` type has only a single value, which acts as a placeholder when no other value exists or is needed.

Syntax

```
// The value of the unit type.
()
```

Remarks

Every F# expression must evaluate to a value. For expressions that do not generate a value that is of interest, the value of type `unit` is used. The `unit` type resembles the `void` type in languages such as C# and C++.

The `unit` type has a single value, and that value is indicated by the token `()`.

The value of the `unit` type is often used in F# programming to hold the place where a value is required by the language syntax, but when no value is needed or desired. An example might be the return value of a `printf` function. Because the important actions of the `printf` operation occur in the function, the function does not have to return an actual value. Therefore, the return value is of type `unit`.

Some constructs expect a `unit` value. For example, a `do` binding or any code at the top level of a module is expected to evaluate to a `unit` value. The compiler reports a warning when a `do` binding or code at the top level of a module produces a result other than the `unit` value that is not used, as shown in the following example.

```
let function1 x y = x + y
// The next line results in a compiler warning.
function1 10 20
// Changing the code to one of the following eliminates the warning.
// Use this when you do want the return value.
let result = function1 10 20
// Use this if you are only calling the function for its side effects,
// and do not want the return value.
function1 10 20 |> ignore
```

This warning is a characteristic of functional programming; it does not appear in other .NET programming languages. In a purely functional program, in which functions do not have any side effects, the final return value is the only result of a function call. Therefore, when the result is ignored, it is a possible programming error. Although F# is not a purely functional programming language, it is a good practice to follow functional programming style whenever possible.

See also

- Primitive
- F# Language Reference

Type Inference

8/5/2022 • 2 minutes to read • Edit Online

This topic describes how the F# compiler infers the types of values, variables, parameters and return values.

Type Inference in General

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type. Omitting explicit type information does not mean that F# is a dynamically typed language or that values in F# are weakly typed. F# is a statically typed language, which means that the compiler deduces an exact type for each construct during compilation. If there is not enough information for the compiler to deduce the types of each construct, you must supply additional type information, typically by adding explicit type annotations somewhere in the code.

Inference of Parameter and Return Types

In a parameter list, you do not have to specify the type of each parameter. And yet, F# is a statically typed language, and therefore every value and expression has a definite type at compile time. For those types that you do not specify explicitly, the compiler infers the type based on the context. If the type is not otherwise specified, it is inferred to be generic. If the code uses a value inconsistently, in such a way that there is no single inferred type that satisfies all the uses of a value, the compiler reports an error.

The return type of a function is determined by the type of the last expression in the function.

For example, in the following code, the parameter types `a` and `b` and the return type are all inferred to be `int` because the literal `100` is of type `int`.

```
let f a b = a + b + 100
```

You can influence type inference by changing the literals. If you make the `100` a `uint32` by appending the suffix `u`, the types of `a`, `b`, and the return value are inferred to be `uint32`.

You can also influence type inference by using other constructs that imply restrictions on the type, such as functions and methods that work with only a particular type.

Also, you can apply explicit type annotations to function or method parameters or to variables in expressions, as shown in the following examples. Errors result if conflicts occur between different constraints.

```
// Type annotations on a parameter.
let addu1 (x : uint32) y =
    x + y

// Type annotations on an expression.
let addu2 x y =
    (x : uint32) + y
```

You can also explicitly specify the return value of a function by providing a type annotation after all the parameters.

```
let addu1 x y : uint32 =
   x + y
```

A common case where a type annotation is useful on a parameter is when the parameter is an object type and you want to use a member.

```
let replace(str: string) =
    str.Replace("A", "a")
```

Automatic Generalization

If the function code is not dependent on the type of a parameter, the compiler considers the parameter to be generic. This is called *automatic generalization*, and it can be a powerful aid to writing generic code without increasing complexity.

For example, the following function combines two parameters of any type into a tuple.

```
let makeTuple a b = (a, b)
```

The type is inferred to be

```
'a -> 'b -> 'a * 'b
```

Additional Information

Type inference is described in more detail in the F# Language Specification.

See also

- Automatic Generalization

Type Abbreviations

8/5/2022 • 2 minutes to read • Edit Online

A *type abbreviation* is an alias or alternate name for a type.

Syntax

```
type [accessibility-modifier] type-abbreviation = type-name
```

Remarks

You can use type abbreviations to give a type a more meaningful name, in order to make code easier to read. You can also use them to create an easy to use name for a type that is otherwise cumbersome to write out. Additionally, you can use type abbreviations to make it easier to change an underlying type without changing all the code that uses the type. The following is a simple type abbreviation.

Accessibility of type abbreviations defaults to `public`.

```
type SizeType = uint32
```

Type abbreviations can include generic parameters, as in the following code.

```
type Transform<'a> = 'a -> 'a
```

In the previous code, `Transform` is a type abbreviation that represents a function that takes a single argument of any type and that returns a single value of that same type.

Type abbreviations are not preserved in the .NET Framework MSIL code. Therefore, when you use an F# assembly from another .NET Framework language, you must use the underlying type name for a type abbreviation.

Type abbreviations can also be used on units of measure. For more information, see Units of Measure.

See also

- F# Language Reference

Casting and conversions (F#)

8/5/2022 • 9 minutes to read • Edit Online

This article describes support for type conversions in F#.

Arithmetic Types

F# provides conversion operators for arithmetic conversions between various primitive types, such as between integer and floating point types. The integral and char conversion operators have checked and unchecked forms; the floating point operators and the `enum` conversion operator do not. The unchecked forms are defined in `FSharp.Core.Operators` and the checked forms are defined in `FSharp.Core.Operators.Checked`. The checked forms check for overflow and generate a runtime exception if the resulting value exceeds the limits of the target type.

Each of these operators has the same name as the name of the destination type. For example, in the following code, in which the types are explicitly annotated, `byte` appears with two different meanings. The first occurrence is the type and the second is the conversion operator.

```
let x : int = 5

let b : byte = byte x
```

The following table shows conversion operators defined in F#.

OPERATOR	DESCRIPTION
`byte`	Convert to byte, an 8-bit unsigned type.
`sbyte`	Convert to signed byte.
`int16`	Convert to a 16-bit signed integer.
`uint16`	Convert to a 16-bit unsigned integer.
`int32, int`	Convert to a 32-bit signed integer.
`uint32`	Convert to a 32-bit unsigned integer.
`int64`	Convert to a 64-bit signed integer.
`uint64`	Convert to a 64-bit unsigned integer.
`nativeint`	Convert to a native integer.
`unativeint`	Convert to an unsigned native integer.
`float, double`	Convert to a 64-bit double-precision IEEE floating point number.

OPERATOR	DESCRIPTION
`float32, single`	Convert to a 32-bit single-precision IEEE floating point number.
`decimal`	Convert to `System.Decimal`.
`char`	Convert to `System.Char`, a Unicode character.
`enum`	Convert to an enumerated type.

In addition to built-in primitive types, you can use these operators with types that implement `op_Explicit` or `op_Implicit` methods with appropriate signatures. For example, the `int` conversion operator works with any type that provides a static method `op_Explicit` that takes the type as a parameter and returns `int`. As a special exception to the general rule that methods cannot be overloaded by return type, you can do this for `op_Explicit` and `op_Implicit`.

Enumerated Types

The `enum` operator is a generic operator that takes one type parameter that represents the type of the `enum` to convert to. When it converts to an enumerated type, type inference attempts to determine the type of the `enum` that you want to convert to. In the following example, the variable `col1` is not explicitly annotated, but its type is inferred from the later equality test. Therefore, the compiler can deduce that you are converting to a `Color` enumeration. Alternatively, you can supply a type annotation, as with `col2` in the following example.

```
type Color =
    | Red = 1
    | Green = 2
    | Blue = 3

// The target type of the conversion cannot be determined by type inference, so the type parameter must be
explicit.
let col1 = enum<Color> 1

// The target type is supplied by a type annotation.
let col2 : Color = enum 2
```

You can also specify the target enumeration type explicitly as a type parameter, as in the following code:

```
let col3 = enum<Color> 3
```

Note that the enumeration casts work only if the underlying type of the enumeration is compatible with the type being converted. In the following code, the conversion fails to compile because of the mismatch between `int32` and `uint32`.

```
// Error: types are incompatible
let col4 : Color = enum 2u
```

For more information, see Enumerations.

Casting Object Types

Conversion between types in an object hierarchy is fundamental to object-oriented programming. There are two basic types of conversions: casting up (upcasting) and casting down (downcasting). Casting up a hierarchy means casting from a derived object reference to a base object reference. Such a cast is guaranteed to work as long as the base class is in the inheritance hierarchy of the derived class. Casting down a hierarchy, from a base object reference to a derived object reference, succeeds only if the object actually is an instance of the correct destination (derived) type or a type derived from the destination type.

F# provides operators for these types of conversions. The `:>` operator casts up the hierarchy, and the `:?>` operator casts down the hierarchy.

Upcasting

In many object-oriented languages, upcasting is implicit; in F#, the rules are slightly different. Upcasting is applied automatically when you pass arguments to methods on an object type. However, for let-bound functions in a module, upcasting is not automatic, unless the parameter type is declared as a flexible type. For more information, see Flexible Types.

The `:>` operator performs a static cast, which means that the success of the cast is determined at compile time. If a cast that uses `:>` compiles successfully, it is a valid cast and has no chance of failure at run time.

You can also use the `upcast` operator to perform such a conversion. The following expression specifies a conversion up the hierarchy:

```
upcast expression
```

When you use the upcast operator, the compiler attempts to infer the type you are converting to from the context. If the compiler is unable to determine the target type, the compiler reports an error. A type annotation may be required.

Downcasting

The `:?>` operator performs a dynamic cast, which means that the success of the cast is determined at run time. A cast that uses the `:?>` operator is not checked at compile time; but at run time, an attempt is made to cast to the specified type. If the object is compatible with the target type, the cast succeeds. If the object is not compatible with the target type, the runtime raises an `InvalidCastException`.

You can also use the `downcast` operator to perform a dynamic type conversion. The following expression specifies a conversion down the hierarchy to a type that is inferred from program context:

```
downcast expression
```

As for the `upcast` operator, if the compiler cannot infer a specific target type from the context, it reports an error. A type annotation may be required.

The following code illustrates the use of the `:>` and `:?>` operators. The code illustrates that the `:?>` operator is best used when you know that conversion will succeed, because it throws `InvalidCastException` if the conversion fails. If you do not know that a conversion will succeed, a type test that uses a `match` expression is better because it avoids the overhead of generating an exception.

```
type Base1() =
    abstract member F : unit -> unit
    default u.F() =
     printfn "F Base1"

type Derived1() =
    inherit Base1()
    override u.F() =
      printfn "F Derived1"


let d1 : Derived1 = Derived1()

// Upcast to Base1.
let base1 = d1 :> Base1

// This might throw an exception, unless
// you are sure that base1 is really a Derived1 object, as
// is the case here.
let derived1 = base1 :?> Derived1

// If you cannot be sure that b1 is a Derived1 object,
// use a type test, as follows:
let downcastBase1 (b1 : Base1) =
   match b1 with
   | :? Derived1 as derived1 -> derived1.F()
   | _ -> ()

downcastBase1 base1
```

Because the generic operators `downcast` and `upcast` rely on type inference to determine the argument and return type, you can replace `let base1 = d1 :> Base1` in the previous code example with `let base1: Base1 = upcast d1`.

A type annotation is required, because `upcast` by itself could not determine the base class.

Implicit upcast conversions

Implicit upcasts are inserted in the following situations:

- When providing a parameter to a function or method with a known named type. This includes when a construct such as computation expressions or slicing becomes a method call.
- When assigning to or mutating a record field or property that has a known named type.
- When a branch of an `if/then/else` or `match` expression has a known target type arising from another branch or overall known type.
- When an element of a list, array, or sequence expression has a known target type.

For example, consider the following code:

```
open System
open System.IO

let findInputSource () : TextReader =
    if DateTime.Now.DayOfWeek = DayOfWeek.Monday then
        // On Monday a TextReader
        Console.In
    else
        // On other days a StreamReader
        File.OpenText("path.txt")
```

Here the branches of the conditional compute a `TextReader` and `StreamReader` respectively. On the second branch, the known target type is `TextReader` from the type annotation on the method, and from the first branch. This means no upcast is needed on the second branch.

To show a warning at every point an additional implicit upcast is used, you can enable warning 3388 (`/warnon:3388` or property `<WarnOn>3388</WarnOn>`).

Implicit numeric conversions

F# uses explicit widening of numeric types in most cases via conversion operators. For example, explicit widening is needed for most numeric types, such as `int8` or `int16`, or from `float32` to `float64`, or when either source or destination type is unknown.

However, implicit widening is allowed for 32-bit integers widened to 64-bit integers, in the same situations as implicit upcasts. For example, consider a typical API shape:

```
type Tensor(…) =
    static member Create(sizes: seq<int64>) = Tensor(…)
```

Integer literals for int64 may be used:

```
Tensor.Create([100L; 10L; 10L])
```

Or integer literals for int32:

```
Tensor.Create([int64 100; int64 10; int64 10])
```

Widening happens automatically for `int32` to `int64`, `int32` to `nativeint`, and `int32` to `double`, when both source and destination type are known during type inference. So in cases such as the previous examples, `int32` literals can be used:

```
Tensor.Create([100; 10; 10])
```

You can also optionally enable the warning 3389 (`/warnon:3389` or property `<WarnOn>3389</WarnOn>`) to show a warning at every point implicit numeric widening is used.

.NET-style implicit conversions

.NET APIs allow the definition of `op_Implicit` static methods to provide implicit conversions between types. These are applied automatically in F# code when passing arguments to methods. For example, consider the following code making explicit calls to `op_Implicit` methods:

```
open System.Xml.Linq

let purchaseOrder = XElement.Load("PurchaseOrder.xml")
let partNos = purchaseOrder.Descendants(XName.op_Implicit "Item")
```

.NET-style `op_Implicit` conversions are applied automatically for argument expressions when types are available for source expression and target type:

```
open System.Xml.Linq

let purchaseOrder = XElement.Load("PurchaseOrder.xml")
let partNos = purchaseOrder.Descendants("Item")
```

You can also optionally enable the warning 3395 (`/warnon:3395` or property `<WarnOn>3395</WarnOn>`) to show a warning at every point a .NET-style implicit conversion is used.

.NET-style `op_Implicit` conversions are also applied automatically for non-method-argument expressions in the same situations as implicit upcasts. However, when used widely or inappropriately, implicit conversions can interact poorly with type inference and lead to code that's harder to understand. For this reason, these always generate warnings when used in non-argument positions.

To show a warning at every point that a .NET-style implicit conversion is used for a non-method argument, you can enable warning 3391 (`/warnon:3391` or property `<WarnOn>3391</WarnOn>`).

Summary of warnings related to conversions

The following optional warnings are provided for uses of implicit conversions:

- `/warnon:3388` (additional implicit upcast)
- `/warnon:3389` (implicit numeric widening)
- `/warnon:3391` (`op_Implicit` at non-method arguments, on by default)
- `/warnon:3395` (`op_Implicit` at method arguments)

See also

- F# Language Reference

Generics

8/5/2022 • 6 minutes to read • Edit Online

F# function values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be *generic*. Generic constructs contain at least one type parameter, which is usually supplied by the user of the generic construct. Generic functions and types enable you to write code that works with a variety of types without repeating the code for each type. Making your code generic can be simple in F#, because often your code is implicitly inferred to be generic by the compiler's type inference and automatic generalization mechanisms.

Syntax

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
function-body

// Explicitly generic method.
[ static ] member object-identifier.method-name<type-parameters> parameter-list [ return-type ] =
method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

Remarks

The declaration of an explicitly generic function or type is much like that of a non-generic function or type, except for the specification (and use) of the type parameters, in angle brackets after the function or type name.

Declarations are often implicitly generic. If you do not fully specify the type of every parameter that is used to compose a function or type, the compiler attempts to infer the type of each parameter, value, and variable from the code you write. For more information, see Type Inference. If the code for your type or function does not otherwise constrain the types of parameters, the function or type is implicitly generic. This process is named *automatic generalization*. There are some limits on automatic generalization. For example, if the F# compiler is unable to infer the types for a generic construct, the compiler reports an error that refers to a restriction called the *value restriction*. In that case, you may have to add some type annotations. For more information about automatic generalization and the value restriction, and how to change your code to address the problem, see Automatic Generalization.

In the previous syntax, *type-parameters* is a comma-separated list of parameters that represent unknown types, each of which starts with a single quotation mark, optionally with a constraint clause that further limits what types may be used for that type parameter. For the syntax for constraint clauses of various kinds and other information about constraints, see Constraints.

The *type-definition* in the syntax is the same as the type definition for a non-generic type. It includes the constructor parameters for a class type, an optional `as` clause, the equal symbol, the record fields, the `inherit` clause, the choices for a discriminated union, `let` and `do` bindings, member definitions, and anything else permitted in a non-generic type definition.

The other syntax elements are the same as those for non-generic functions and types. For example, *object-identifier* is an identifier that represents the containing object itself.

Properties, fields, and constructors cannot be more generic than the enclosing type. Also, values in a module cannot be generic.

Implicitly Generic Constructs

When the F# compiler infers the types in your code, it automatically treats any function that can be generic as generic. If you specify a type explicitly, such as a parameter type, you prevent automatic generalization.

In the following code example, `makeList` is generic, even though neither it nor its parameters are explicitly declared as generic.

```
let makeList a b =
    [a; b]
```

The signature of the function is inferred to be `'a -> 'a -> 'a list`. Note that `a` and `b` in this example are inferred to have the same type. This is because they are included in a list together, and all elements of a list must be of the same type.

You can also make a function generic by using the single quotation mark syntax in a type annotation to indicate that a parameter type is a generic type parameter. In the following code, `function1` is generic because its parameters are declared in this manner, as type parameters.

```
let function1 (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

Explicitly Generic Constructs

You can also make a function generic by explicitly declaring its type parameters in angle brackets (`<type-parameter>`). The following code illustrates this.

```
let function2<'T> (x: 'T) (y: 'T) =
    printfn "%A, %A" x y
```

Using Generic Constructs

When you use generic functions or methods, you might not have to specify the type arguments. The compiler uses type inference to infer the appropriate type arguments. If there is still an ambiguity, you can supply type arguments in angle brackets, separating multiple type arguments with commas.

The following code shows the use of the functions that are defined in the previous sections.

```
// In this case, the type argument is inferred to be int.
function1 10 20
// In this case, the type argument is float.
function1 10.0 20.0
// Type arguments can be specified, but should only be specified
// if the type parameters are declared explicitly. If specified,
// they have an effect on type inference, so in this example,
// a and b are inferred to have type int.
let function3 a b =
    // The compiler reports a warning:
    function1<int> a b
    // No warning.
    function2<int> a b
```

NOTE

There are two ways to refer to a generic type by name. For example, `list<int>` and `int list` are two ways to refer to a generic type `list` that has a single type argument `int`. The latter form is conventionally used only with built-in F# types such as `list` and `option`. If there are multiple type arguments, you normally use the syntax `Dictionary<int, string>` but you can also use the syntax `(int, string) Dictionary`.

Wildcards as Type Arguments

To specify that a type argument should be inferred by the compiler, you can use the underscore, or wildcard symbol (`_`), instead of a named type argument. This is shown in the following code.

```
let printSequence (sequence1: Collections.seq<_>) =
   Seq.iter (fun elem -> printf "%s " (elem.ToString())) sequence1
```

Constraints in Generic Types and Functions

In a generic type or function definition, you can use only those constructs that are known to be available on the generic type parameter. This is required to enable the verification of function and method calls at compile time. If you declare your type parameters explicitly, you can apply an explicit constraint to a generic type parameter to notify the compiler that certain methods and functions are available. However, if you allow the F# compiler to infer your generic parameter types, it will determine the appropriate constraints for you. For more information, see Constraints.

Statically Resolved Type Parameters

There are two kinds of type parameters that can be used in F# programs. The first are generic type parameters of the kind described in the previous sections. This first kind of type parameter is equivalent to the generic type parameters that are used in languages such as Visual Basic and C#. Another kind of type parameter is specific to F# and is referred to as a *statically resolved type parameter*. For information about these constructs, see Statically Resolved Type Parameters.

Examples

```
// A generic function.
// In this example, the generic type parameter 'a makes function3 generic.
let function3 (x : 'a) (y : 'a) =
    printf "%A %A" x y

// A generic record, with the type parameter in angle brackets.
type GR<'a> =
    {
        Field1: 'a;
        Field2: 'a;
    }

// A generic class.
type C<'a>(a : 'a, b : 'a) =
    let z = a
    let y = b
    member this.GenericMethod(x : 'a) =
        printfn "%A %A %A" x y z

// A generic discriminated union.
type U<'a> =
    | Choice1 of 'a
    | Choice2 of 'a * 'a

type Test() =
    // A generic member
    member this.Function1<'a>(x, y) =
        printfn "%A, %A" x y

    // A generic abstract method.
    abstract abstractMethod<'a, 'b> : 'a * 'b -> unit
    override this.abstractMethod<'a, 'b>(x:'a, y:'b) =
         printfn "%A, %A" x y
```

See also

- Language Reference
- Types
- Statically Resolved Type Parameters
- Generics
- Automatic Generalization
- Constraints

Automatic Generalization

8/5/2022 • 3 minutes to read • Edit Online

F# uses type inference to evaluate the types of functions and expressions. This topic describes how F# automatically generalizes the arguments and types of functions so that they work with multiple types when this is possible.

Automatic Generalization

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic. The compiler examines each parameter and determines whether the function has a dependency on the specific type of that parameter. If it does not, the type is inferred to be generic.

The following code example illustrates a function that the compiler infers to be generic.

```
let max a b = if a > b then a else b
```

The type is inferred to be `'a -> 'a -> 'a`.

The type indicates that this is a function that takes two arguments of the same unknown type and returns a value of that same type. One of the reasons that the previous function can be generic is that the greater-than operator (`>`) is itself generic. The greater-than operator has the signature `'a -> 'a -> bool`. Not all operators are generic, and if the code in a function uses a parameter type together with a non-generic function or operator, that parameter type cannot be generalized.

Because `max` is generic, it can be used with types such as `int`, `float`, and so on, as shown in the following examples.

```
let biggestFloat = max 2.0 3.0
let biggestInt = max 2 3
```

However, the two arguments must be of the same type. The signature is `'a -> 'a -> 'a`, not `'a -> 'b -> 'a`. Therefore, the following code produces an error because the types do not match.

```
// Error: type mismatch.
let biggestIntFloat = max 2.0 3
```

The `max` function also works with any type that supports the greater-than operator. Therefore, you could also use it on a string, as shown in the following code.

```
let testString = max "cab" "cat"
```

Value Restriction

The compiler performs automatic generalization only on complete function definitions that have explicit arguments, and on simple immutable values.

This means that the compiler issues an error if you try to compile code that is not sufficiently constrained to be a specific type, but is also not generalizable. The error message for this problem refers to this restriction on automatic generalization for values as the *value restriction*.

Typically, the value restriction error occurs either when you want a construct to be generic but the compiler has insufficient information to generalize it, or when you unintentionally omit sufficient type information in a nongeneric construct. The solution to the value restriction error is to provide more explicit information to more fully constrain the type inference problem, in one of the following ways:

- Constrain a type to be nongeneric by adding an explicit type annotation to a value or parameter.
- If the problem is using a nongeneralizable construct to define a generic function, such as a function composition or incompletely applied curried function arguments, try to rewrite the function as an ordinary function definition.
- If the problem is an expression that is too complex to be generalized, make it into a function by adding an extra, unused parameter.
- Add explicit generic type parameters. This option is rarely used.
- The following code examples illustrate each of these scenarios.

Case 1: Too complex an expression. In this example, the list `counter` is intended to be `int option ref`, but it is not defined as a simple immutable value.

```
let counter = ref None
// Adding a type annotation fixes the problem:
let counter : int option ref = ref None
```

Case 2: Using a nongeneralizable construct to define a generic function. In this example, the construct is nongeneralizable because it involves partial application of function arguments.

```
let maxhash = max << hash
// The following is acceptable because the argument for maxhash is explicit:
let maxhash obj = (max << hash) obj
```

Case 3: Adding an extra, unused parameter. Because this expression is not simple enough for generalization, the compiler issues the value restriction error.

```
let emptyList10 = Array.create 10 []
// Adding an extra (unused) parameter makes it a function, which is generalizable.
let emptyList10 () = Array.create 10 []
```

Case 4: Adding type parameters.

```
let arrayOf10Lists = Array.create 10 []
// Adding a type parameter and type annotation lets you write a generic value.
let arrayOf10Lists<'T> = Array.create 10 ([]:'T list)
```

In the last case, the value becomes a type function, which may be used to create values of many different types, for example as follows:

```
let intLists = arrayOf10Lists<int>
let floatLists = arrayOf10Lists<float>
```

See also

- Type Inference
- Generics
- Statically Resolved Type Parameters
- Constraints

Constraints

8/5/2022 • 4 minutes to read • Edit Online

This topic describes constraints that you can apply to generic type parameters to specify the requirements for a type argument in a generic type or function.

Syntax

```
type-parameter-list when constraint1 [ and constraint2]
```

Remarks

There are several different constraints you can apply to limit the types that can be used in a generic type. The following table lists and describes these constraints.

CONSTRAINT	SYNTAX	DESCRIPTION
Type Constraint	*type-parameter* :> *type*	The provided type must be equal to or derived from the type specified, or, if the type is an interface, the provided type must implement the interface.
Null Constraint	*type-parameter* : null	The provided type must support the null literal. This includes all .NET object types but not F# list, tuple, function, class, record, or union types.
Explicit Member Constraint	[(]*type-parameter* [or ... or *type-parameter*)] : (*member-signature*)	At least one of the type arguments provided must have a member that has the specified signature; not intended for common use. Members must be either explicitly defined on the type or part of an implicit type extension to be valid targets for an Explicit Member Constraint.
Constructor Constraint	*type-parameter* : (new : unit -> 'a)	The provided type must have a parameterless constructor.
Value Type Constraint	*type-parameter* : struct	The provided type must be a .NET value type.
Reference Type Constraint	*type-parameter* : not struct	The provided type must be a .NET reference type.
Enumeration Type Constraint	*type-parameter* : enum<*underlying-type*>	The provided type must be an enumerated type that has the specified underlying type; not intended for common use.

CONSTRAINT	SYNTAX	DESCRIPTION
Delegate Constraint	*type-parameter* : delegate< *tuple-parameter-type*, *return-type*>	The provided type must be a delegate type that has the specified arguments and return value; not intended for common use.
Comparison Constraint	*type-parameter* : comparison	The provided type must support comparison.
Equality Constraint	*type-parameter* : equality	The provided type must support equality.
Unmanaged Constraint	*type-parameter* : unmanaged	The provided type must be an unmanaged type. Unmanaged types are either certain primitive types (`sbyte` , `byte` , `char` , `nativeint` , `unativeint` , `float32` , `float` , `int16` , `uint16` , `int32` , `uint32` , `int64` , `uint64` , or `decimal`), enumeration types, `nativeptr<_>` , or a non-generic structure whose fields are all unmanaged types.

You have to add a constraint when your code has to use a feature that is available on the constraint type but not on types in general. For example, if you use the type constraint to specify a class type, you can use any one of the methods of that class in the generic function or type.

Specifying constraints is sometimes required when writing type parameters explicitly, because without a constraint, the compiler has no way of verifying that the features that you are using will be available on any type that might be supplied at run time for the type parameter.

The most common constraints you use in F# code are type constraints that specify base classes or interfaces. The other constraints are either used by the F# library to implement certain functionality, such as the explicit member constraint, which is used to implement operator overloading for arithmetic operators, or are provided mainly because F# supports the complete set of constraints that is supported by the common language runtime.

During the type inference process, some constraints are inferred automatically by the compiler. For example, if you use the `+` operator in a function, the compiler infers an explicit member constraint on variable types that are used in the expression.

The following code illustrates some constraint declarations:

```
// Base Type Constraint
type Class1<'T when 'T :> System.Exception> =
class end

// Interface Type Constraint
type Class2<'T when 'T :> System.IComparable> =
class end

// Null constraint
type Class3<'T when 'T : null> =
class end

// Member constraint with instance member
type Class5<'T when 'T : (member Method1 : 'T -> int)> =
class end

// Member constraint with property
type Class6<'T when 'T : (member Property1 : int)> =
class end

// Constructor constraint
type Class7<'T when 'T : (new : unit -> 'T)>() =
member val Field = new 'T()

// Reference type constraint
type Class8<'T when 'T : not struct> =
class end

// Enumeration constraint with underlying value specified
type Class9<'T when 'T : enum<uint32>> =
class end

// 'T must implement IComparable, or be an array type with comparable
// elements, or be System.IntPtr or System.UIntPtr. Also, 'T must not have
// the NoComparison attribute.
type Class10<'T when 'T : comparison> =
class end

// 'T must support equality. This is true for any type that does not
// have the NoEquality attribute.
type Class11<'T when 'T : equality> =
class end

type Class12<'T when 'T : delegate<obj * System.EventArgs, unit>> =
class end

type Class13<'T when 'T : unmanaged> =
class end

// Member constraints with two type parameters
// Most often used with static type parameters in inline functions
let inline add(value1 : ^T when ^T : (static member (+) : ^T * ^T -> ^T), value2: ^T) =
value1 + value2

// ^T and ^U must support operator +
let inline heterogenousAdd(value1 : ^T when (^T or ^U) : (static member (+) : ^T * ^U -> ^T), value2 : ^U) =
value1 + value2

// If there are multiple constraints, use the and keyword to separate them.
type Class14<'T,'U when 'T : equality and 'U : equality> =
class end
```

See also

- Generics

Statically Resolved Type Parameters

8/5/2022 • 3 minutes to read • Edit Online

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 preceded by a caret (^) symbol.

Syntax

```
^type-parameter
```

Remarks

In F#, there are two distinct kinds of type parameters. The first kind is the standard generic type parameter. These are indicated by an apostrophe ('), as in `'T` and `'U`. They are equivalent to generic type parameters in other .NET Framework languages. The other kind is statically resolved and is indicated by a caret symbol, as in `^T` and `^U`.

Statically resolved type parameters are primarily useful in conjunction with member constraints, which are constraints that allow you to specify that a type argument must have a particular member or members in order to be used. There is no way to create this kind of constraint by using a regular generic type parameter.

The following table summarizes the similarities and differences between the two kinds of type parameters.

FEATURE	GENERIC	STATICALLY RESOLVED
Syntax	`'T`, `'U`	`^T`, `^U`
Resolution time	Run time	Compile time
Member constraints	Cannot be used with member constraints.	Can be used with member constraints.
Code generation	A type (or method) with standard generic type parameters results in the generation of a single generic type or method.	Multiple instantiations of types and methods are generated, one for each type that is needed.
Use with types	Can be used on types.	Cannot be used on types.
Use with inline functions	No. An inline function cannot be parameterized with a standard generic type parameter.	Yes. Statically resolved type parameters cannot be used on functions or methods that are not inline.

Many F# core library functions, especially operators, have statically resolved type parameters. These functions and operators are inline, and result in efficient code generation for numeric computations.

Inline methods and functions that use operators, or use other functions that have statically resolved type parameters, can also use statically resolved type parameters themselves. Often, type inference infers such inline functions to have statically resolved type parameters. The following example illustrates an operator definition that is inferred to have a statically resolved type parameter.

```
let inline (+@) x y = x + x * y
// Call that uses int.
printfn "%d" (1 +@ 1)
// Call that uses float.
printfn "%f" (1.0 +@ 0.5)
```

The resolved type of `(+@)` is based on the use of both `(+)` and `(*)`, both of which cause type inference to infer member constraints on the statically resolved type parameters. The resolved type, as shown in the F# interpreter, is as follows.

```
^a -> ^c -> ^d
when (^a or ^b) : (static member ( + ) : ^a * ^b -> ^d) and
(^a or ^c) : (static member ( * ) : ^a * ^c -> ^b)
```

The output is as follows.

```
2
1.500000
```

Starting with F# 4.1, you can also specify concrete type names in statically resolved type parameter signatures. In previous versions of the language, the type name was inferred by the compiler, but could not be specified in the signature. As of F# 4.1, you may also specify concrete type names in statically resolved type parameter signatures. Here's an example:

```
let inline konst x _ = x

type CFunctor() =
    static member inline fmap (f: ^a -> ^b, a: ^a list) = List.map f a
    static member inline fmap (f: ^a -> ^b, a: ^a option) =
        match a with
        | None -> None
        | Some x -> Some (f x)

    // default implementation of replace
    static member inline replace< ^a, ^b, ^c, ^d, ^e when ^a :> CFunctor and (^a or ^d): (static member
fmap: (^b -> ^c) * ^d -> ^e) > (a, f) =
        ((^a or ^d) : (static member fmap : (^b -> ^c) * ^d -> ^e) (konst a, f))

    // call overridden replace if present
    static member inline replace< ^a, ^b, ^c when ^b: (static member replace: ^a * ^b -> ^c)>(a: ^a, f: ^b)
=
        (^b : (static member replace: ^a * ^b -> ^c) (a, f))

let inline replace_instance< ^a, ^b, ^c, ^d when (^a or ^c): (static member replace: ^b * ^c -> ^d)> (a: ^b,
f: ^c) =
        ((^a or ^c): (static member replace: ^b * ^c -> ^d) (a, f))

// Note the concrete type 'CFunctor' specified in the signature
let inline replace (a: ^a) (f: ^b): ^a0 when (CFunctor or  ^b): (static member replace: ^a *  ^b ->  ^a0) =
    replace_instance<CFunctor, _, _, _> (a, f)
```

See also

- Generics
- Type Inference
- Automatic Generalization
- Constraints

- Inline Functions

Flexible Types

8/5/2022 • 2 minutes to read • Edit Online

A *flexible type annotation* indicates that a parameter, variable, or value has a type that is compatible with a specified type, where compatibility is determined by position in an object-oriented hierarchy of classes or interfaces. Flexible types are useful specifically when the automatic conversion to types higher in the type hierarchy does not occur but you still want to enable your functionality to work with any type in the hierarchy or any type that implements an interface.

Syntax

```
#type
```

Remarks

In the previous syntax, *type* represents a base type or an interface.

A flexible type is equivalent to a generic type that has a constraint that limits the allowed types to types that are compatible with the base or interface type. That is, the following two lines of code are equivalent.

```
#SomeType

'T when 'T :> SomeType
```

Flexible types are useful in several types of situations. For example, when you have a higher order function (a function that takes a function as an argument), it is often useful to have the function return a flexible type. In the following example, the use of a flexible type with a sequence argument in `iterate2` enables the higher order function to work with functions that generate sequences, arrays, lists, and any other enumerable type.

Consider the following two functions, one of which returns a sequence, the other of which returns a flexible type.

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

As another example, consider the Seq.concat library function:

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

You can pass any of the following enumerable sequences to this function:

- A list of lists

- A list of arrays
- An array of lists
- An array of sequences
- Any other combination of enumerable sequences

The following code uses `Seq.concat` to demonstrate the scenarios that you can support by using flexible types.

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

The output is as follows.

```
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
```

In F#, as in other object-oriented languages, there are contexts in which derived types or types that implement interfaces are automatically converted to a base type or interface type. These automatic conversions occur in direct arguments, but not when the type is in a subordinate position, as part of a more complex type such as a return type of a function type, or as a type argument. Thus, the flexible type notation is primarily useful when the type you are applying it to is part of a more complex type.

See also

- F# Language Reference
- Generics

Units of Measure

8/5/2022 • 8 minutes to read • Edit Online

Floating point and signed integer values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on. By using quantities with units, you enable the compiler to verify that arithmetic relationships have the correct units, which helps prevent programming errors.

Syntax

```
[<Measure>] type unit-name [ = measure ]
```

Remarks

The previous syntax defines *unit-name* as a unit of measure. The optional part is used to define a new measure in terms of previously defined units. For example, the following line defines the measure `cm` (centimeter).

```
[<Measure>] type cm
```

The following line defines the measure `ml` (milliliter) as a cubic centimeter (`cm^3`).

```
[<Measure>] type ml = cm^3
```

In the previous syntax, *measure* is a formula that involves units. In formulas that involve units, integral powers are supported (positive and negative), spaces between units indicate a product of the two units, `*` also indicates a product of units, and `/` indicates a quotient of units. For a reciprocal unit, you can either use a negative integer power or a `/` that indicates a separation between the numerator and denominator of a unit formula. Multiple units in the denominator should be surrounded by parentheses. Units separated by spaces after a `/` are interpreted as being part of the denominator, but any units following a `*` are interpreted as being part of the numerator.

You can use 1 in unit expressions, either alone to indicate a dimensionless quantity, or together with other units, such as in the numerator. For example, the units for a rate would be written as `1/s`, where `s` indicates seconds. Parentheses are not used in unit formulas. You do not specify numeric conversion constants in the unit formulas; however, you can define conversion constants with units separately and use them in unit-checked computations.

Unit formulas that mean the same thing can be written in various equivalent ways. Therefore, the compiler converts unit formulas into a consistent form, which converts negative powers to reciprocals, groups units into a single numerator and a denominator, and alphabetizes the units in the numerator and denominator.

For example, the unit formulas `kg m s^-2` and `m /s s * kg` are both converted to `kg m/s^2`.

You use units of measure in floating point expressions. Using floating point numbers together with associated units of measure adds another level of type safety and helps avoid the unit mismatch errors that can occur in formulas when you use weakly typed floating point numbers. If you write a floating point expression that uses units, the units in the expression must match.

You can annotate literals with a unit formula in angle brackets, as shown in the following examples.

```
1.0<cm>
55.0<miles/hour>
```

You do not put a space between the number and the angle bracket; however, you can include a literal suffix such as `f`, as in the following example.

```
// The f indicates single-precision floating point.
55.0f<miles/hour>
```

Such an annotation changes the type of the literal from its primitive type (such as `float`) to a dimensioned type, such as `float<cm>` or, in this case, `float<miles/hour>`. A unit annotation of `<1>` indicates a dimensionless quantity, and its type is equivalent to the primitive type without a unit parameter.

The type of a unit of measure is a floating point or signed integral type together with an extra unit annotation, indicated in brackets. Thus, when you write the type of a conversion from `g` (grams) to `kg` (kilograms), you describe the types as follows.

```
let convertg2kg (x : float<g>) = x / 1000.0<g/kg>
```

Units of measure are used for compile-time unit checking but are not persisted in the run-time environment. Therefore, they do not affect performance.

Units of measure can be applied to any type, not just floating point types; however, only floating point types, signed integral types, and decimal types support dimensioned quantities. Therefore, it only makes sense to use units of measure on the primitive types and on aggregates that contain these primitive types.

The following example illustrates the use of units of measure.

```fsharp
// Mass, grams.
[<Measure>] type g
// Mass, kilograms.
[<Measure>] type kg
// Weight, pounds.
[<Measure>] type lb

// Distance, meters.
[<Measure>] type m
// Distance, cm
[<Measure>] type cm

// Distance, inches.
[<Measure>] type inch
// Distance, feet
[<Measure>] type ft

// Time, seconds.
[<Measure>] type s

// Force, Newtons.
[<Measure>] type N = kg m / s^2

// Pressure, bar.
[<Measure>] type bar
// Pressure, Pascals
[<Measure>] type Pa = N / m^2

// Volume, milliliters.
[<Measure>] type ml
// Volume, liters.
[<Measure>] type L

// Define conversion constants.
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>
let cmPerMeter : float<cm/m> = 100.0<cm/m>
let cmPerInch : float<cm/inch> = 2.54<cm/inch>

let mlPerCubicCentimeter : float<ml/cm^3> = 1.0<ml/cm^3>
let mlPerLiter : float<ml/L> = 1000.0<ml/L>

// Define conversion functions.
let convertGramsToKilograms (x : float<g>) = x / gramsPerKilogram
let convertCentimetersToInches (x : float<cm>) = x / cmPerInch
```

The following code example illustrates how to convert from a dimensionless floating point number to a dimensioned floating point value. You just multiply by 1.0, applying the dimensions to the 1.0. You can abstract this into a function like `degreesFahrenheit`.

Also, when you pass dimensioned values to functions that expect dimensionless floating point numbers, you must cancel out the units or cast to `float` by using the `float` operator. In this example, you divide by `1.0<degC>` for the arguments to `printf` because `printf` expects dimensionless quantities.

```
[<Measure>] type degC // temperature, Celsius/Centigrade
[<Measure>] type degF // temperature, Fahrenheit

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF>
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>)

// Define conversion functions from dimensionless floating point values.
let degreesFahrenheit temp = temp * 1.0<degF>
let degreesCelsius temp = temp * 1.0<degC>

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let parsedOk, floatValue = System.Double.TryParse(input)
if parsedOk
   then
      printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
   else
      printfn "Error parsing input."
```

The following example session shows the outputs from and inputs to this code.

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

Primitive Types supporting Units of Measure

The following types or type abbreviation aliases support unit-of-measure annotations:

F# ALIAS	CLR TYPE
`float32` / `single`	`System.Single`
`float` / `double`	`System.Double`
`decimal`	`System.Decimal`
`sbyte` / `int8`	`System.SByte`
`int16`	`System.Int16`
`int` / `int32`	`System.Int32`
`int64`	`System.Int64`
`byte` / `uint8`	`System.Byte`
`uint16`	`System.UInt16`
`uint` / `uint32`	`System.UInt32`
`uint64`	`System.UIn64`

F# ALIAS	CLR TYPE
`nativeint`	`System.IntPtr`
`unativeint`	`System.UIntPtr`

For example, you can annotate an unsigned integer as follows:

```
[<Measure>]
type days

let better_age = 3u<days>
```

The addition of unsigned integer types to this feature is documented in F# RFC FS-1091.

Pre-defined Units of Measure

A unit library is available in the `FSharp.Data.UnitSystems.SI` namespace. It includes SI units in both their symbol form (like `m` for meter) in the `UnitSymbols` subnamespace, and in their full name (like `meter` for meter) in the `UnitNames` subnamespace.

Using Generic Units

You can write generic functions that operate on data that has an associated unit of measure. You do this by specifying a type together with a generic unit as a type parameter, as shown in the following code example.

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

let genericSumUnits ( x : float<'u>) (y: float<'u>) = x + y

let v1 = 3.1<m/s>
let v2 = 2.7<m/s>
let x1 = 1.2<m>
let t1 = 1.0<s>

// OK: a function that has unit consistency checking.
let result1 = genericSumUnits v1 v2
// Error reported: mismatched units.
// Uncomment to see error.
// let result2 = genericSumUnits v1 x1
```

Creating Collection Types with Generic Units

The following code shows how to create an aggregate type that consists of individual floating point values that have units that are generic. This enables a single type to be created that works with a variety of units. Also, generic units preserve type safety by ensuring that a generic type that has one set of units is a different type than the same generic type with a different set of units. The basis of this technique is that the `Measure` attribute can be applied to the type parameter.

```
 // Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

// Define a vector together with a measure type parameter.
// Note the attribute applied to the type parameter.
type vector3D<[<Measure>] 'u> = { x : float<'u>; y : float<'u>; z : float<'u>}

// Create instances that have two different measures.
// Create a position vector.
let xvec : vector3D<m> = { x = 0.0<m>; y = 0.0<m>; z = 0.0<m> }
// Create a velocity vector.
let v1vec : vector3D<m/s> = { x = 1.0<m/s>; y = -1.0<m/s>; z = 0.0<m/s> }
```

Units at Runtime

Units of measure are used for static type checking. When floating point values are compiled, the units of measure are eliminated, so the units are lost at run time. Therefore, any attempt to implement functionality that depends on checking the units at run time is not possible. For example, implementing a `ToString` function to print out the units is not possible.

Conversions

To convert a type that has units (for example, `float<'u>`) to a type that does not have units, you can use the standard conversion function. For example, you can use `float` to convert to a `float` value that does not have units, as shown in the following code.

```
[<Measure>]
type cm
let length = 12.0<cm>
let x = float length
```

To convert a unitless value to a value that has units, you can multiply by a 1 or 1.0 value that is annotated with the appropriate units. However, for writing interoperability layers, there are also some explicit functions that you can use to convert unitless values to values with units. These are in the FSharp.Core.LanguagePrimitives module. For example, to convert from a unitless `float` to a `float<cm>`, use FloatWithMeasure, as shown in the following code.

```
open Microsoft.FSharp.Core
let height:float<cm> = LanguagePrimitives.FloatWithMeasure x
```

See also

- F# Language Reference

Byrefs

8/5/2022 • 6 minutes to read • Edit Online

F# has two major feature areas that deal in the space of low-level programming:

- The `byref` / `inref` / `outref` types, which are managed pointers. They have restrictions on usage so that you cannot compile a program that is invalid at run time.
- A `byref` -like struct, which is a struct that has similar semantics and the same compile-time restrictions as `byref<'T>`. One example is Span<T>.

Syntax

```
// Byref types as parameters
let f (x: byref<'T>) = ()
let g (x: inref<'T>) = ()
let h (x: outref<'T>) = ()

// Calling a function with a byref parameter
let mutable x = 3
f &x

// Declaring a byref-like struct
open System.Runtime.CompilerServices

[<Struct; IsByRefLike>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

Byref, inref, and outref

There are three forms of `byref`:

- `inref<'T>`, a managed pointer for reading the underlying value.
- `outref<'T>`, a managed pointer for writing to the underlying value.
- `byref<'T>`, a managed pointer for reading and writing the underlying value.

A `byref<'T>` can be passed where an `inref<'T>` is expected. Similarly, a `byref<'T>` can be passed where an `outref<'T>` is expected.

Using byrefs

To use a `inref<'T>`, you need to get a pointer value with `&`:

```
open System

let f (dt: inref<DateTime>) =
    printfn $"Now: %O{dt}"

let usage =
    let dt = DateTime.Now
    f &dt // Pass a pointer to 'dt'
```

To write to the pointer by using an `outref<'T>` or `byref<'T>`, you must also make the value you grab a pointer to `mutable`.

```
open System

let f (dt: byref<DateTime>) =
    printfn $"Now: %O{dt}"
    dt <- DateTime.Now

// Make 'dt' mutable
let mutable dt = DateTime.Now

// Now you can pass the pointer to 'dt'
f &dt
```

If you are only writing the pointer instead of reading it, consider using `outref<'T>` instead of `byref<'T>`.

Inref semantics

Consider the following code:

```
let f (x: inref<SomeStruct>) = x.SomeField
```

Semantically, this means the following:

- The holder of the `x` pointer may only use it to read the value.
- Any pointer acquired to `struct` fields nested within `SomeStruct` are given type `inref<_>`.

The following is also true:

- There is no implication that other threads or aliases do not have write access to `x`.
- There is no implication that `SomeStruct` is immutable by virtue of `x` being an `inref`.

However, for F# value types that **are** immutable, the `this` pointer is inferred to be an `inref`.

All of these rules together mean that the holder of an `inref` pointer may not modify the immediate contents of the memory being pointed to.

Outref semantics

The purpose of `outref<'T>` is to indicate that the pointer should only be written to. Unexpectedly, `outref<'T>` permits reading the underlying value despite its name. This is for compatibility purposes.

Semantically, `outref<'T>` is no different than `byref<'T>`, except for one difference: methods with `outref<'T>` parameters are implicitly constructed into a tuple return type, just like when calling a method with an `[<Out>]` parameter.

```
type C =
    static member M1(x, y: _ outref) =
        y <- x
        true

match C.M1 1 with
| true, 1 -> printfn "Expected" // Fine with outref, error with byref
| _ -> printfn "Never matched"
```

Interop with C#

C# supports the `in ref` and `out ref` keywords, in addition to `ref` returns. The following table shows how F# interprets what C# emits:

C# CONSTRUCT	F# INFERS
`ref` return value	`outref<'T>`
`ref readonly` return value	`inref<'T>`
`in ref` parameter	`inref<'T>`
`out ref` parameter	`outref<'T>`

The following table shows what F# emits:

F# CONSTRUCT	EMITTED CONSTRUCT
`inref<'T>` argument	`[In]` attribute on argument
`inref<'T>` return	`modreq` attribute on value
`inref<'T>` in abstract slot or implementation	`modreq` on argument or return
`outref<'T>` argument	`[Out]` attribute on argument

Type inference and overloading rules

An `inref<'T>` type is inferred by the F# compiler in the following cases:

1. A .NET parameter or return type that has an `IsReadOnly` attribute.
2. The `this` pointer on a struct type that has no mutable fields.
3. The address of a memory location derived from another `inref<_>` pointer.

When an implicit address of an `inref` is being taken, an overload with an argument of type `SomeType` is preferred to an overload with an argument of type `inref<SomeType>`. For example:

```
type C() =
    static member M(x: System.DateTime) = x.AddDays(1.0)
    static member M(x: inref<System.DateTime>) = x.AddDays(2.0)
    static member M2(x: System.DateTime, y: int) = x.AddDays(1.0)
    static member M2(x: inref<System.DateTime>, y: int) = x.AddDays(2.0)

let res = System.DateTime.Now
let v =  C.M(res)
let v2 =  C.M2(res, 4)
```

In both cases, the overloads taking `System.DateTime` are resolved rather than the overloads taking `inref<System.DateTime>`.

Byref-like structs

In addition to the `byref`/`inref`/`outref` trio, you can define your own structs that can adhere to `byref`-like semantics. This is done with the IsByRefLikeAttribute attribute:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` does not imply `Struct`. Both must be present on the type.

A "`byref`-like" struct in F# is a stack-bound value type. It is never allocated on the managed heap. A `byref`-like struct is useful for high-performance programming, as it is enforced with set of strong checks about lifetime and non-capture. The rules are:

- They can be used as function parameters, method parameters, local variables, method returns.
- They cannot be static or instance members of a class or normal struct.
- They cannot be captured by any closure construct (`async` methods or lambda expressions).
- They cannot be used as a generic parameter.

This last point is crucial for F# pipeline-style programming, as `|>` is a generic function that parameterizes its input types. This restriction may be relaxed for `|>` in the future, as it is inline and does not make any calls to non-inlined generic functions in its body.

Although these rules strongly restrict usage, they do so to fulfill the promise of high-performance computing in a safe manner.

Byref returns

Byref returns from F# functions or members can be produced and consumed. When consuming a `byref`-returning method, the value is implicitly dereferenced. For example:

```
let squareAndPrint (data : byref<int>) =
    let squared = data*data    // data is implicitly dereferenced
    printfn $"%d{squared}"
```

To return a value byref, the variable that contains the value must live longer than the current scope. Also, to return byref, use `&value` (where value is a variable that lives longer than the current scope).

```
let mutable sum = 0
let safeSum (bytes: Span<byte>) =
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes[i]
    &sum  // sum lives longer than the scope of this function.
```

To avoid the implicit dereference, such as passing a reference through multiple chained calls, use `&x` (where `x` is the value).

You can also directly assign to a return `byref`. Consider the following (highly imperative) program:

```
type C() =
    let mutable nums = [| 1; 3; 7; 15; 31; 63; 127; 255; 511; 1023 |]

    override _.ToString() = String.Join(' ', nums)

    member _.FindLargestSmallerThan(target: int) =
        let mutable ctr = nums.Length - 1

        while ctr > 0 && nums[ctr] >= target do ctr <- ctr - 1

        if ctr > 0 then &nums[ctr] else &nums[0]

[<EntryPoint>]
let main argv =
    let c = C()
    printfn $"Original sequence: %O{c}"

    let v = &c.FindLargestSmallerThan 16

    v <- v*2 // Directly assign to the byref return

    printfn $"New sequence:      %O{c}"

    0 // return an integer exit code
```

This is the output:

```
Original sequence: 1 3 7 15 31 63 127 255 511 1023
New sequence:      1 3 7 30 31 63 127 255 511 1023
```

Scoping for byrefs

A `let`-bound value cannot have its reference exceed the scope in which it was defined. For example, the following is disallowed:

```
let test2 () =
    let x = 12
    &x // Error: 'x' exceeds its defined scope!

let test () =
    let x =
        let y = 1
        &y // Error: `y` exceeds its defined scope!
    ()
```

This prevents you from getting different results depending on if you compile with optimizations or not.

Tuples

8/5/2022 • 5 minutes to read • Edit Online

A *tuple* is a grouping of unnamed but ordered values, possibly of different types. Tuples can either be reference types or structs.

Syntax

```
(element, ... , element)
struct(element, ... ,element )
```

Remarks

Each *element* in the previous syntax can be any valid F# expression.

Examples

Examples of tuples include pairs, triples, and so on, of the same or different types. Some examples are illustrated in the following code.

```
(1, 2)

// Triple of strings.
("one", "two", "three")

// Tuple of generic types.
(a, b)

// Tuple that has mixed types.
("one", 1, 2.0)

// Tuple of integer expressions.
(a + 1, b + 1)

// Struct Tuple of floats
struct (1.025f, 1.5f)
```

Obtaining Individual Values

You can use pattern matching to access and assign names for tuple elements, as shown in the following code.

```
let print tuple1 =
   match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

You can also deconstruct a tuple via pattern matching outside of a `match` expression via `let` binding:

```
let (a, b) = (1, 2)

// Or as a struct
let struct (c, d) = struct (1, 2)
```

Or you can pattern match on tuples as inputs to functions:

```
let getDistance ((x1,y1): float*float) ((x2,y2): float*float) =
    // Note the ability to work on individual elements
    (x1*x2 - y1*y2)
    |> abs
    |> sqrt
```

If you need only one element of the tuple, the wildcard character (the underscore) can be used to avoid creating a new name for a value that you do not need.

```
let (a, _) = (1, 2)
```

Copying elements from a reference tuple into a struct tuple is also simple:

```
// Create a reference tuple
let (a, b) = (1, 2)

// Construct a struct tuple from it
let struct (c, d) = struct (a, b)
```

The functions `fst` and `snd` (reference tuples only) return the first and second elements of a tuple, respectively.

```
let c = fst (1, 2)
let d = snd (1, 2)
```

There is no built-in function that returns the third element of a triple, but you can easily write one as follows.

```
let third (_, _, c) = c
```

Generally, it is better to use pattern matching to access individual tuple elements.

Using Tuples

Tuples provide a convenient way to return multiple values from a function, as shown in the following example. This example performs integer division and returns the rounded result of the operation as a first member of a tuple pair and the remainder as a second member of the pair.

```
let divRem a b =
   let x = a / b
   let y = a % b
   (x, y)
```

Tuples can also be used as function arguments when you want to avoid the implicit currying of function arguments that is implied by the usual function syntax.

```
let sumNoCurry (a, b) = a + b
```

The usual syntax for defining the function `let sum a b = a + b` enables you to define a function that is the partial application of the first argument of the function, as shown in the following code.

```
let sum a b = a + b

let addTen = sum 10
let result = addTen 95
// Result is 105.
```

Using a tuple as the parameter disables currying. For more information, see "Partial Application of Arguments" in Functions.

Names of Tuple Types

When you write out the name of a type that is a tuple, you use the `*` symbol to separate elements. For a tuple that consists of an `int`, a `float`, and a `string`, such as `(10, 10.0, "ten")`, the type would be written as follows.

```
int * float * string
```

Note that outer parentheses are mandatory when creating a type alias for a struct tuple type.

```
type TupleAlias = string * float
type StructTupleAlias = (struct (string * float))
```

Interoperation with C# Tuples

C# 7.0 introduced tuples to the language. Tuples in C# are structs, and are equivalent to struct tuples in F#. If you need to interoperate with C#, you must use struct tuples.

This is easy to do. For example, imagine you have to pass a tuple to a C# class and then consume its result, which is also a tuple:

```
namespace CSharpTupleInterop
{
    public static class Example
    {
        public static (int, int) AddOneToXAndY((int x, int y) a) =>
            (a.x + 1, a.y + 1);
    }
}
```

In your F# code, you can then pass a struct tuple as the parameter and consume the result as a struct tuple.

```
open TupleInterop

let struct (newX, newY) = Example.AddOneToXAndY(struct (1, 2))
// newX is now 2, and newY is now 3
```

Converting between Reference Tuples and Struct Tuples

Because Reference Tuples and Struct Tuples have a completely different underlying representation, they are not implicitly convertible. That is, code such as the following won't compile:

```
// Will not compile!
let (a, b) = struct (1, 2)

// Will not compile!
let struct (c, d) = (1, 2)

// Won't compile!
let f(t: struct(int*int)): int*int = t
```

You must pattern match on one tuple and construct the other with the constituent parts. For example:

```
// Pattern match on the result.
let (a, b) = (1, 2)

// Construct a new tuple from the parts you pattern matched on.
let struct (c, d) = struct (a, b)
```

Compiled Form of Reference Tuples

This section explains the form of tuples when they're compiled. The information here isn't necessary to read unless you are targeting .NET Framework 3.5 or lower.

Tuples are compiled into objects of one of several generic types, all named `System.Tuple`, that are overloaded on the arity, or number of type parameters. Tuple types appear in this form when you view them from another language, such as C# or Visual Basic, or when you are using a tool that is not aware of F# constructs. The `Tuple` types were introduced in .NET Framework 4. If you are targeting an earlier version of .NET Framework, the compiler uses versions of `System.Tuple` from the 2.0 version of the F# Core Library. The types in this library are used only for applications that target the 2.0, 3.0, and 3.5 versions of .NET Framework. Type forwarding is used to ensure binary compatibility between .NET Framework 2.0 and .NET Framework 4 F# components.

Compiled Form of Struct Tuples

Struct tuples (for example, `struct (x, y)`), are fundamentally different from reference tuples. They are compiled into the ValueTuple type, overloaded by arity, or the number of type parameters. They are equivalent to C# 7.0 Tuples and Visual Basic 2017 Tuples, and interoperate bidirectionally.

See also

- F# Language Reference
- F# Types

Options

8/5/2022 • 4 minutes to read • Edit Online

The option type in F# is used when an actual value might not exist for a named value or variable. An option has an underlying type and can hold a value of that type, or it might not have a value.

Remarks

The following code illustrates a function which generates an option type.

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

As you can see, if the input `a` is greater than 0, `Some(a)` is generated. Otherwise, `None` is generated.

The value `None` is used when an option does not have an actual value. Otherwise, the expression `Some( ... )` gives the option a value. The values `Some` and `None` are useful in pattern matching, as in the following function `exists`, which returns `true` if the option has a value and `false` if it does not.

```
let exists (x : int option) =
    match x with
    | Some(x) -> true
    | None -> false
```

Using Options

Options are commonly used when a search does not return a matching result, as shown in the following code.

```
let rec tryFindMatch pred list =
    match list with
    | head :: tail -> if pred(head)
                        then Some(head)
                        else tryFindMatch pred tail
    | [] -> None

// result1 is Some 100 and its type is int option.
let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ]

// result2 is None and its type is int option.
let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]
```

In the previous code, a list is searched recursively. The function `tryFindMatch` takes a predicate function `pred` that returns a Boolean value, and a list to search. If an element that satisfies the predicate is found, the recursion ends and the function returns the value as an option in the expression `Some(head)`. The recursion ends when the empty list is matched. At that point the value `head` has not been found, and `None` is returned.

Many F# library functions that search a collection for a value that may or may not exist return the `option` type. By convention, these functions begin with the `try` prefix, for example, `Seq.tryFindIndex`.

Options can also be useful when a value might not exist, for example if it is possible that an exception will be thrown when you try to construct a value. The following code example illustrates this.

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
                None
```

The `openFile` function in the previous example has type `string -> File option` because it returns a `File` object if the file opens successfully and `None` if an exception occurs. Depending on the situation, it may not be an appropriate design choice to catch an exception rather than allowing it to propagate.

Additionally, it is still possible to pass `null` or a value that is null to the `Some` case of an option. This is generally to be avoided, and typically is in routine F# programming, but is possible due to the nature of reference types in .NET.

Option Properties and Methods

The option type supports the following properties and methods.

PROPERTY OR METHOD	TYPE	DESCRIPTION
None	`'T option`	A static property that enables you to create an option value that has the `None` value.
IsNone	`bool`	Returns `true` if the option has the `None` value.
IsSome	`bool`	Returns `true` if the option has a value that is not `None`.
Some	`'T option`	A static member that creates an option that has a value that is not `None`.
Value	`'T`	Returns the underlying value, or throws a `System.NullReferenceException` if the value is `None`.

Option Module

There is a module, Option, that contains useful functions that perform operations on options. Some functions repeat the functionality of the properties but are useful in contexts where a function is needed. Option.isSome and Option.isNone are both module functions that test whether an option holds a value. Option.get obtains the value, if there is one. If there is no value, it throws `System.ArgumentException`.

The Option.bind function executes a function on the value, if there is a value. The function must take exactly one argument, and its parameter type must be the option type. The return value of the function is another option type.

The option module also includes functions that correspond to the functions that are available for lists, arrays, sequences, and other collection types. These functions include `Option.map`, `Option.iter`, `Option.forall`, `Option.exists`, `Option.foldBack`, `Option.fold`, and `Option.count`. These functions enable options to be used like a collection of zero or one elements. For more information and examples, see the discussion of collection functions in Lists.

Converting to Other Types

Options can be converted to lists or arrays. When an option is converted into either of these data structures, the resulting data structure has zero or one element. To convert an option to an array, use `Option.toArray`. To convert an option to a list, use `Option.toList`.

See also

- F# Language Reference
- F# Types

Value Options

8/5/2022 • 2 minutes to read • Edit Online

The Value Option type in F# is used when the following two circumstances hold:

1. A scenario is appropriate for an F# Option.
2. Using a struct provides a performance benefit in your scenario.

Not all performance-sensitive scenarios are "solved" by using structs. You must consider the additional cost of copying when using them instead of reference types. However, large F# programs commonly instantiate many optional types that flow through hot paths, and in such cases, structs can often yield better overall performance over the lifetime of a program.

Definition

Value Option is defined as a struct discriminated union that is similar to the reference option type. Its definition can be thought of this way:

```
[<StructuralEquality; StructuralComparison>]
[<Struct>]
type ValueOption<'T> =
    | ValueNone
    | ValueSome of 'T
```

Value Option conforms to structural equality and comparison. The main difference is that the compiled name, type name, and case names all indicate that it is a value type.

Using Value Options

Value Options are used just like Options. `ValueSome` is used to indicate that a value is present, and `ValueNone` is used when a value is not present:

```
let tryParseDateTime (s: string) =
    match System.DateTime.TryParse(s) with
    | (true, dt) -> ValueSome dt
    | (false, _) -> ValueNone

let possibleDateString1 = "1990-12-25"
let possibleDateString2 = "This is not a date"

let result1 = tryParseDateTime possibleDateString1
let result2 = tryParseDateTime possibleDateString2

match (result1, result2) with
| ValueSome d1, ValueSome d2 -> printfn "Both are dates!"
| ValueSome d1, ValueNone -> printfn "Only the first is a date!"
| ValueNone, ValueSome d2 -> printfn "Only the second is a date!"
| ValueNone, ValueNone -> printfn "None of them are dates!"
```

As with Options, the naming convention for a function that returns `ValueOption` is to prefix it with `try`.

Value Option properties and methods

There is one property for Value Options at this time: `Value`. An InvalidOperationException is raised if no value is present when this property is invoked.

Value Option functions

The `ValueOption` module in FSharp.Core contains equivalent functionality to the `Option` module. There are a few differences in name, such as `defaultValueArg`:

```
val defaultValueArg : arg:'T voption -> defaultValue:'T -> 'T
```

This acts just like `defaultArg` in the `Option` module, but operates on a Value Option instead.

See also

- Options

Results

8/5/2022 • 2 minutes to read • Edit Online

The `Result<'T,'TFailure>` type lets you write error-tolerant code that can be composed.

Syntax

```
// The definition of Result in FSharp.Core
[<StructuralEquality; StructuralComparison>]
[<CompiledName("FSharpResult`2")>]
[<Struct>]
type Result<'T,'TError> =
    | Ok of ResultValue:'T
    | Error of ErrorValue:'TError
```

Remarks

See the `Result` module for the built-in combinators for the `Result`. type.

Note that the result type is a struct discriminated union. Structural equality semantics apply here.

The `Result` type is typically used in monadic error-handling, which is often referred to as Railway-oriented Programming within the F# community. The following trivial example demonstrates this approach.

```fsharp
// Define a simple type which has fields that can be validated
type Request =
    { Name: string
      Email: string }

// Define some logic for what defines a valid name.
//
// Generates a Result which is an Ok if the name validates;
// otherwise, it generates a Result which is an Error.
let validateName req =
    match req.Name with
    | null -> Error "No name found."
    | "" -> Error "Name is empty."
    | "bananas" -> Error "Bananas is not a name."
    | _ -> Ok req

// Similarly, define some email validation logic.
let validateEmail req =
    match req.Email with
    | null -> Error "No email found."
    | "" -> Error "Email is empty."
    | s when s.EndsWith("bananas.com") -> Error "No email from bananas.com is allowed."
    | _ -> Ok req

let validateRequest reqResult =
    reqResult
    |> Result.bind validateName
    |> Result.bind validateEmail

let test() =
    // Now, create a Request and pattern match on the result.
    let req1 = { Name = "Phillip"; Email = "phillip@contoso.biz" }
    let res1 = validateRequest (Ok req1)
    match res1 with
    | Ok req -> printfn $"My request was valid! Name: {req.Name} Email {req.Email}"
    | Error e -> printfn $"Error: {e}"
    // Prints: "My request was valid!  Name: Phillip Email: phillip@consoto.biz"

    let req2 = { Name = "Phillip"; Email = "phillip@bananas.com" }
    let res2 = validateRequest (Ok req2)
    match res2 with
    | Ok req -> printfn $"My request was valid! Name: {req.Name} Email {req.Email}"
    | Error e -> printfn $"Error: {e}"
    // Prints: "Error: No email from bananas.com is allowed."

test()
```

As you can see, it's quite easy to chain together various validation functions if you force them all to return a `Result`. This lets you break up functionality like this into small pieces which are as composable as you need them to be. This also has the added value of *enforcing* the use of pattern matching at the end of a round of validation, which in turns enforces a higher degree of program correctness.

See also

- Discriminated Unions
- Pattern Matching

F# collection types

8/5/2022 • 15 minutes to read • Edit Online

By reviewing this topic, you can determine which F# collection type best suits a particular need. These collection types differ from the collection types in .NET, such as those in the `System.Collections.Generic` namespace, in that the F# collection types are designed from a functional programming perspective rather than an object-oriented perspective. More specifically, only the array collection has mutable elements. Therefore, when you modify a collection, you create an instance of the modified collection instead of altering the original collection.

Collection types also differ in the type of data structure in which objects are stored. Data structures such as hash tables, linked lists, and arrays have different performance characteristics and a different set of available operations.

Table of collection types

The following table shows F# collection types.

TYPE	DESCRIPTION	RELATED LINKS
List	An ordered, immutable series of elements of the same type. Implemented as a linked list.	Lists List Module
Array	A fixed-size, zero-based, mutable collection of consecutive data elements that are all of the same type.	Arrays Array Module Array2D Module Array3D Module
seq	A logical series of elements that are all of one type. Sequences are particularly useful when you have a large, ordered collection of data but don't necessarily expect to use all the elements. Individual sequence elements are computed only as required, so a sequence can perform better than a list if not all the elements are used. Sequences are represented by the `seq<'T>` type, which is an alias for `IEnumerable<T>`. Therefore, any .NET Framework type that implements `System.Collections.Generic.IEnumerable<'T>` can be used as a sequence.	Sequences Seq Module
Map	An immutable dictionary of elements. Elements are accessed by key.	Map Module
Set	An immutable set that's based on binary trees, where comparison is the F# structural comparison function, which potentially uses implementations of the `System.IComparable` interface on key values.	Set Module

Table of functions

This section compares the functions that are available on F# collection types. The computational complexity of the function is given, where N is the size of the first collection, and M is the size of the second collection, if any. A dash (-) indicates that this function isn't available on the collection. Because sequences are lazily evaluated, a function such as `Seq.distinct` may be O(1) because it returns immediately, although it still affects the performance of the sequence when enumerated.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
append	O(N)	O(N)	O(N)	-	-	Returns a new collection that contains the elements of the first collection followed by elements of the second collection.
add	-	-	-	O(log(N))	O(log(N))	Returns a new collection with the element added.
average	O(N)	O(N)	O(N)	-	-	Returns the average of the elements in the collection.
averageBy	O(N)	O(N)	O(N)	-	-	Returns the average of the results of the provided function applied to each element.
blit	O(N)	-	-	-	-	Copies a section of an array.
cache	-	-	O(N)	-	-	Computes and stores elements of a sequence.
cast	-	-	O(N)	-	-	Converts the elements to the specified type.
choose	O(N)	O(N)	O(N)	-	-	Applies the given function `f` to each element `x` of the list. Returns the list that contains the results for each element where the function returns `Some(f(x))`.
collect	O(N)	O(N)	O(N)	-	-	Applies the given function to each element of the collection, concatenates all the results, and returns the combined list.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
compareWith	-	-	O(N)	-	-	Compares two sequences by using the given comparison function, element by element.
concat	O(N)	O(N)	O(N)	-	-	Combines the given enumeration-of-enumerations as a single concatenated enumeration.
contains	-	-	-	-	O(log(N))	Returns true if the set contains the specified element.
containsKey	-	-	-	O(log(N))	-	Tests whether an element is in the domain of a map.
count	-	-	-	-	O(N)	Returns the number of elements in the set.
countBy	-	-	O(N)	-	-	Applies a key-generating function to each element of a sequence, and returns a sequence that yields unique keys and their number of occurrences in the original sequence.
copy	O(N)	-	O(N)	-	-	Copies the collection.
create	O(N)	-	-	-	-	Creates an array of whole elements that are all initially the given value.
delay	-	-	O(1)	-	-	Returns a sequence that's built from the given delayed specification of a sequence.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
difference	-	-	-	-	O(M*log(N))	Returns a new set with the elements of the second set removed from the first set.
distinct			O(1)*			Returns a sequence that contains no duplicate entries according to generic hash and equality comparisons on the entries. If an element occurs multiple times in the sequence, later occurrences are discarded.
distinctBy			O(1)*			Returns a sequence that contains no duplicate entries according to the generic hash and equality comparisons on the keys that the given key-generating function returns. If an element occurs multiple times in the sequence, later occurrences are discarded.
empty	O(1)	O(1)	O(1)	O(1)	O(1)	Creates an empty collection.
exists	O(N)	O(N)	O(N)	O(log(N))	O(log(N))	Tests whether any element of the sequence satisfies the given predicate.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
exists2	O(min(N,M))	-	O(min(N,M))			Tests whether any pair of corresponding elements of the input sequences satisfies the given predicate.
fill	O(N)					Sets a range of elements of the array to the given value.
filter	O(N)	O(N)	O(N)	O(N)	O(N)	Returns a new collection that contains only the elements of the collection for which the given predicate returns `true`.
find	O(N)	O(N)	O(N)	O(log(N))	-	Returns the first element for which the given function returns `true`. Returns `System.Collections.Generic.KeyNotF` if no such element exists.
findIndex	O(N)	O(N)	O(N)	-	-	Returns the index of the first element in the array that satisfies the given predicate. Raises `System.Collections.Generic.KeyNotF` if no element satisfies the predicate.
findKey	-	-	-	O(log(N))	-	Evaluates the function on each mapping in the collection, and returns the key for the first mapping where the function returns `true`. If no such element exists, this function raises `System.Collections.Generic.KeyNotF`.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
fold	O(N)	O(N)	O(N)	O(N)	O(N)	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f (...(f s i0)...) iN.
fold2	O(N)	O(N)	-	-	-	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, this function computes f (...(f s i0 j0)...) iN jN.
foldBack	O(N)	O(N)	-	O(N)	O(N)	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f i0 (...(f iN s)).

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
foldBack2	O(N)	O(N)	-	-	-	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, this function computes f i0 j0 (...(f iN jN s)).
forall	O(N)	O(N)	O(N)	O(N)	O(N)	Tests whether all elements of the collection satisfy the given predicate.
forall2	O(N)	O(N)	O(N)	-	-	Tests whether all corresponding elements of the collection satisfy the given predicate pairwise.
get / nth	O(1)	O(N)	O(N)	-	-	Returns an element from the collection given its index.
head	-	O(1)	O(1)	-	-	Returns the first element of the collection.
init	O(N)	O(N)	O(1)	-	-	Creates a collection given the dimension and a generator function to compute the elements.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
initInfinite	-	-	O(1)	-	-	Generates a sequence that, when iterated, returns successive elements by calling the given function.
intersect	-	-	-	-	O(log(N)*log(M))	Computes the intersection of two sets.
intersectMany	-	-	-	-	O(N1*N2...)	Computes the intersection of a sequence of sets. The sequence must not be empty.
isEmpty	O(1)	O(1)	O(1)	O(1)	-	Returns `true` if the collection is empty.
isProperSubset	-	-	-	-	O(M*log(N))	Returns `true` if all elements of the first set are in the second set, and at least one element of the second set isn't in the first set.
isProperSuperset	-	-	-	-	O(M*log(N))	Returns `true` if all elements of the second set are in the first set, and at least one element of the first set isn't in the second set.
isSubset	-	-	-	-	O(M*log(N))	Returns `true` if all elements of the first set are in the second set.
isSuperset	-	-	-	-	O(M*log(N))	Returns `true` if all elements of the second set are in the first set.
iter	O(N)	O(N)	O(N)	O(N)	O(N)	Applies the given function to each element of the collection.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
iteri	O(N)	O(N)	O(N)	-	-	Applies the given function to each element of the collection. The integer that's passed to the function indicates the index of the element.
iteri2	O(N)	O(N)	-	-	-	Applies the given function to a pair of elements that are drawn from matching indices in two arrays. The integer that's passed to the function indicates the index of the elements. The two arrays must have the same length.
iter2	O(N)	O(N)	O(N)	-	-	Applies the given function to a pair of elements that are drawn from matching indices in two arrays. The two arrays must have the same length.
last	O(1)	O(N)	O(N)	-	-	Returns the last item in the applicable collection.
length	O(1)	O(N)	O(N)	-	-	Returns the number of elements in the collection.
map	O(N)	O(N)	O(1)	-	-	Builds a collection whose elements are the results of applying the given function to each element of the array.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
map2	O(N)	O(N)	O(1)	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise. The two input arrays must have the same length.
map3	-	O(N)	-	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the three collections simultaneously.
mapi	O(N)	O(N)	O(N)	-	-	Builds an array whose elements are the results of applying the given function to each element of the array. The integer index that's passed to the function indicates the index of the element that's being transformed.
mapi2	O(N)	O(N)	-	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise, also passing the index of the elements. The two input arrays must have the same length.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
max	O(N)	O(N)	O(N)	-	-	Returns the greatest element in the collection, compared by using the max operator.
maxBy	O(N)	O(N)	O(N)	-	-	Returns the greatest element in the collection, compared by using max on the function result.
maxElement	-	-	-	-	O(log(N))	Returns the greatest element in the set according to the ordering that's used for the set.
min	O(N)	O(N)	O(N)	-	-	Returns the least element in the collection, compared by using the min operator.
minBy	O(N)	O(N)	O(N)	-	-	Returns the least element in the collection, compared by using the min operator on the function result.
minElement	-	-	-	-	O(log(N))	Returns the lowest element in the set according to the ordering that's used for the set.
ofArray	-	O(N)	O(1)	O(N)	O(N)	Creates a collection that contains the same elements as the given array.
ofList	O(N)	-	O(1)	O(N)	O(N)	Creates a collection that contains the same elements as the given list.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
ofSeq	O(N)	O(N)	-	O(N)	O(N)	Creates a collection that contains the same elements as the given sequence.
pairwise	-	-	O(N)	-	-	Returns a sequence of each element in the input sequence and its predecessor except for the first element, which is returned only as the predecessor of the second element.
partition	O(N)	O(N)	-	O(N)	O(N)	Splits the collection into two collections. The first collection contains the elements for which the given predicate returns `true`, and the second collection contains the elements for which the given predicate returns `false`.
permute	O(N)	O(N)	-	-	-	Returns an array with all elements permuted according to the specified permutation.
pick	O(N)	O(N)	O(N)	O(log(N))	-	Applies the given function to successive elements, returning the first result where the function returns Some. If the function never returns Some, `System.Collections.Generic.KeyNotF` is raised.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
readonly	-	-	O(N)	-	-	Creates a sequence object that delegates to the given sequence object. This operation ensures that a type cast can't rediscover and mutate the original sequence. For example, if given an array, the returned sequence will return the elements of the array, but you can't cast the returned sequence object to an array.
reduce	O(N)	O(N)	O(N)	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. This function starts by applying the function to the first two elements, passes this result into the function along with the third element, and so on. The function returns the final result.
reduceBack	O(N)	O(N)	-	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f i0 (...(f iN-1 iN)).

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
remove	-	-	-	O(log(N))	O(log(N))	Removes an element from the domain of the map. No exception is raised if the element isn't present.
replicate	-	O(N)	-	-	-	Creates a list of a specified length with every element set to the given value.
rev	O(N)	O(N)	-	-	-	Returns a new list with the elements in reverse order.
scan	O(N)	O(N)	O(N)	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. This operation applies the function to the second argument and the first element of the list. The operation then passes this result into the function along with the second element and so on. Finally, the operation returns the list of intermediate results and the final result.
scanBack	O(N)	O(N)	-	-	-	Resembles the foldBack operation but returns both the intermediate and final results.
singleton	-	-	O(1)	-	O(1)	Returns a sequence that yields only one item.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
set	O(1)	-	-	-	-	Sets an element of an array to the specified value.
skip	-	-	O(N)	-	-	Returns a sequence that skips N elements of the underlying sequence and then yields the remaining elements of the sequence.
skipWhile	-	-	O(N)	-	-	Returns a sequence that, when iterated, skips elements of the underlying sequence while the given predicate returns `true` and then yields the remaining elements of the sequence.
sort	O(N*log(N)) average O(N^2) worst case	O(N*log(N))	O(N*log(N))	-	-	Sorts the collection by element value. Elements are compared using compare.
sortBy	O(N*log(N)) average O(N^2) worst case	O(N*log(N))	O(N*log(N))	-	-	Sorts the given list by using keys that the given projection provides. Keys are compared using compare.
sortInPlace	O(N*log(N)) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given comparison function. Elements are compared by using compare.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
sortInPlaceBy	O(N*log(N)) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given projection for the keys. Elements are compared by using compare.
sortInPlaceWith	O(N*log(N)) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given comparison function as the order.
sortWith	O(N*log(N)) average O(N^2) worst case	O(N*log(N))	-	-	-	Sorts the elements of a collection, using the given comparison function as the order and returning a new collection.
sub	O(N)	-	-	-	-	Builds an array that contains the given subrange that's specified by starting index and length.
sum	O(N)	O(N)	O(N)	-	-	Returns the sum of the elements in the collection.
sumBy	O(N)	O(N)	O(N)	-	-	Returns the sum of the results that are generated by applying the function to each element of the collection.
tail	-	O(1)	-	-	-	Returns the list without its first element.
take	-	-	O(N)	-	-	Returns the elements of the sequence up to a specified count.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
takeWhile	-	-	O(1)	-	-	Returns a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns `true` and then returns no more elements.
toArray	-	O(N)	O(N)	O(N)	O(N)	Creates an array from the given collection.
toList	O(N)	-	O(N)	O(N)	O(N)	Creates a list from the given collection.
toSeq	O(1)	O(1)	-	O(1)	O(1)	Creates a sequence from the given collection.
truncate	-	-	O(1)	-	-	Returns a sequence that, when enumerated, returns no more than N elements.
tryFind	O(N)	O(N)	O(N)	O(log(N))	-	Searches for an element that satisfies a given predicate.
tryFindIndex	O(N)	O(N)	O(N)	-	-	Searches for the first element that satisfies a given predicate and returns the index of the matching element, or `None` if no such element exists.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
tryFindKey	-	-	-	O(log(N))	-	Returns the key of the first mapping in the collection that satisfies the given predicate, or returns `None` if no such element exists.
tryPick	O(N)	O(N)	O(N)	O(log(N))	-	Applies the given function to successive elements, returning the first result where the function returns `Some` for some value. If no such element exists, the operation returns `None`.
unfold	-	-	O(N)	-	-	Returns a sequence that contains the elements that the given computation generates.
union	-	-	-	-	O(M*log(N))	Computes the union of the two sets.
unionMany	-	-	-	-	O(N1*N2...)	Computes the union of a sequence of sets.
unzip	O(N)	O(N)	O(N)	-	-	Splits a list of pairs into two lists.
unzip3	O(N)	O(N)	O(N)	-	-	Splits a list of triples into three lists.
windowed	-	-	O(N)	-	-	Returns a sequence that yields sliding windows of containing elements that are drawn from the input sequence. Each window is returned as a fresh array.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
zip	O(N)	O(N)	O(N)	-	-	Combines the two collections into a list of pairs. The two lists must have equal lengths.
zip3	O(N)	O(N)	O(N)	-	-	Combines the three collections into a list of triples. The lists must have equal lengths.

See also

- F# Types
- F# Language Reference

Lists

8/5/2022 • 23 minutes to read • Edit Online

A list in F# is an ordered, immutable series of elements of the same type. To perform basic operations on lists, use the functions in the List module.

Creating and Initializing Lists

You can define a list by explicitly listing out the elements, separated by semicolons and enclosed in square brackets, as shown in the following line of code.

```
let list123 = [ 1; 2; 3 ]
```

You can also put line breaks between elements, in which case the semicolons are optional. The latter syntax can result in more readable code when the element initialization expressions are longer, or when you want to include a comment for each element.

```
let list123 = [
    1
    2
    3 ]
```

Normally, all list elements must be the same type. An exception is that a list in which the elements are specified to be a base type can have elements that are derived types. Thus the following is acceptable, because both `Button` and `CheckBox` derive from `Control`.

```
let myControlList : Control list = [ new Button(); new CheckBox() ]
```

You can also define list elements by using a range indicated by integers separated by the range operator (`..`), as shown in the following code.

```
let list1 = [ 1 .. 10 ]
```

An empty list is specified by a pair of square brackets with nothing in between them.

```
// An empty list.
let listEmpty = []
```

You can also use a sequence expression to create a list. See Sequence Expressions for more information. For example, the following code creates a list of squares of integers from 1 to 10.

```
let listOfSquares = [ for i in 1 .. 10 -> i*i ]
```

Operators for Working with Lists

You can attach elements to a list by using the `::` (cons) operator. If `list1` is `[2; 3; 4]`, the following code creates `list2` as `[100; 2; 3; 4]`.

```
let list2 = 100 :: list1
```

You can concatenate lists that have compatible types by using the `@` operator, as in the following code. If `list1` is `[2; 3; 4]` and `list2` is `[100; 2; 3; 4]`, this code creates `list3` as `[2; 3; 4; 100; 2; 3; 4]`.

```
let list3 = list1 @ list2
```

Functions for performing operations on lists are available in the List module.

Because lists in F# are immutable, any modifying operations generate new lists instead of modifying existing lists.

Lists in F# are implemented as singly linked lists, which means that operations that access only the head of the list are O(1), and element access is O(*n*).

Properties

The list type supports the following properties:

PROPERTY	TYPE	DESCRIPTION
Head	`'T`	The first element.
Empty	`'T list`	A static property that returns an empty list of the appropriate type.
IsEmpty	`bool`	`true` if the list has no elements.
Item	`'T`	The element at the specified index (zero-based).
Length	`int`	The number of elements.
Tail	`'T list`	The list without the first element.

Following are some examples of using these properties.

```
let list1 = [ 1; 2; 3 ]

// Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

Using Lists

Programming with lists enables you to perform complex operations with a small amount of code. This section describes common operations on lists that are important to functional programming.

Recursion with Lists

Lists are uniquely suited to recursive programming techniques. Consider an operation that must be performed on every element of a list. You can do this recursively by operating on the head of the list and then passing the tail of the list, which is a smaller list that consists of the original list without the first element, back again to the next level of recursion.

To write such a recursive function, you use the cons operator (`::`) in pattern matching, which enables you to separate the head of a list from the tail.

The following code example shows how to use pattern matching to implement a recursive function that performs operations on a list.

```
let rec sum list =
   match list with
   | head :: tail -> head + sum tail
   | [] -> 0
```

The previous code works well for small lists, but for larger lists, it could overflow the stack. The following code improves on this code by using an accumulator argument, a standard technique for working with recursive functions. The use of the accumulator argument makes the function tail recursive, which saves stack space.

```
let sum list =
   let rec loop list acc =
       match list with
       | head :: tail -> loop tail (acc + head)
       | [] -> acc
   loop list 0
```

The function `RemoveAllMultiples` is a recursive function that takes two lists. The first list contains the numbers whose multiples will be removed, and the second list is the list from which to remove the numbers. The code in the following example uses this recursive function to eliminate all the non-prime numbers from a list, leaving a list of prime numbers as the result.

```
let IsPrimeMultipleTest n x =
   x = n || x % n <> 0

let rec RemoveAllMultiples listn listx =
   match listn with
   | head :: tail -> RemoveAllMultiples tail (List.filter (IsPrimeMultipleTest head) listx)
   | [] -> listx


let GetPrimesUpTo n =
    let max = int (sqrt (float n))
    RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

The output is as follows:

```
Primes Up To 100:
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]
```

Module Functions

The List module provides functions that access the elements of a list. The head element is the fastest and easiest to access. Use the property Head or the module function List.head. You can access the tail of a list by using the Tail property or the List.tail function. To find an element by index, use the List.nth function. `List.nth` traverses the list. Therefore, it is O(*n*). If your code uses `List.nth` frequently, you might want to consider using an array instead of a list. Element access in arrays is O(1).

Boolean Operations on Lists

The List.isEmpty function determines whether a list has any elements.

The List.exists function applies a Boolean test to elements of a list and returns `true` if any element satisfies the test. List.exists2 is similar but operates on successive pairs of elements in two lists.

The following code demonstrates the use of `List.exists`.

```
// Use List.exists to determine whether there is an element of a list satisfies a given Boolean expression.
// containsNumber returns true if any of the elements of the supplied list match
// the supplied number.
let containsNumber number list = List.exists (fun elem -> elem = number) list
let list0to3 = [0 .. 3]
printfn "For list %A, contains zero is %b" list0to3 (containsNumber 0 list0to3)
```

The output is as follows:

```
For list [0; 1; 2; 3], contains zero is true
```

The following example demonstrates the use of `List.exists2`.

```
// Use List.exists2 to compare elements in two lists.
// isEqualElement returns true if any elements at the same position in two supplied
// lists match.
let isEqualElement list1 list2 = List.exists2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
let list1to5 = [ 1 .. 5 ]
let list5to1 = [ 5 .. -1 .. 1 ]
if (isEqualElement list1to5 list5to1) then
    printfn "Lists %A and %A have at least one equal element at the same position." list1to5 list5to1
else
    printfn "Lists %A and %A do not have an equal element at the same position." list1to5 list5to1
```

The output is as follows:

```
Lists [1; 2; 3; 4; 5] and [5; 4; 3; 2; 1] have at least one equal element at the same position.
```

You can use List.forall if you want to test whether all the elements of a list meet a condition.

```
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list
printfn "%b" (isAllZeroes [0.0; 0.0])
printfn "%b" (isAllZeroes [0.0; 1.0])
```

The output is as follows:

```
true
false
```

Similarly, List.forall2 determines whether all elements in the corresponding positions in two lists satisfy a Boolean expression that involves each pair of elements.

```
let listEqual list1 list2 = List.forall2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
printfn "%b" (listEqual [0; 1; 2] [0; 1; 2])
printfn "%b" (listEqual [0; 0; 0] [0; 1; 0])
```

The output is as follows:

```
true
false
```

Sort Operations on Lists

The List.sort, List.sortBy, and List.sortWith functions sort lists. The sorting function determines which of these three functions to use. `List.sort` uses default generic comparison. Generic comparison uses global operators based on the generic compare function to compare values. It works efficiently with a wide variety of element types, such as simple numeric types, tuples, records, discriminated unions, lists, arrays, and any type that implements `System.IComparable`. For types that implement `System.IComparable`, generic comparison uses the `System.IComparable.CompareTo()` function. Generic comparison also works with strings, but uses a culture-independent sorting order. Generic comparison should not be used on unsupported types, such as function types. Also, the performance of the default generic comparison is best for small structured types; for larger structured types that need to be compared and sorted frequently, consider implementing `System.IComparable` and providing an efficient implementation of the `System.IComparable.CompareTo()` method.

`List.sortBy` takes a function that returns a value that is used as the sort criterion, and `List.sortWith` takes a comparison function as an argument. These latter two functions are useful when you are working with types that do not support comparison, or when the comparison requires more complex comparison semantics, as in the case of culture-aware strings.

The following example demonstrates the use of `List.sort`.

```
let sortedList1 = List.sort [1; 4; 8; -2; 5]
printfn "%A" sortedList1
```

The output is as follows:

```
[-2; 1; 4; 5; 8]
```

The following example demonstrates the use of `List.sortBy`.

```
let sortedList2 = List.sortBy (fun elem -> abs elem) [1; 4; 8; -2; 5]
printfn "%A" sortedList2
```

The output is as follows:

```
[1; -2; 4; 5; 8]
```

The next example demonstrates the use of `List.sortWith`. In this example, the custom comparison function `compareWidgets` is used to first compare one field of a custom type, and then another when the values of the first field are equal.

```
type Widget = { ID: int; Rev: int }

let compareWidgets widget1 widget2 =
   if widget1.ID < widget2.ID then -1 else
   if widget1.ID > widget2.ID then 1 else
   if widget1.Rev < widget2.Rev then -1 else
   if widget1.Rev > widget2.Rev then 1 else
   0

let listToCompare = [
    { ID = 92; Rev = 1 }
    { ID = 110; Rev = 1 }
    { ID = 100; Rev = 5 }
    { ID = 100; Rev = 2 }
    { ID = 92; Rev = 1 }
    ]

let sortedWidgetList = List.sortWith compareWidgets listToCompare
printfn "%A" sortedWidgetList
```

The output is as follows:

```
[{ID = 92;
Rev = 1;}; {ID = 92;
Rev = 1;}; {ID = 100;
Rev = 2;}; {ID = 100;
Rev = 5;}; {ID = 110;
Rev = 1;}]
```

Search Operations on Lists

Numerous search operations are supported for lists. The simplest, List.find, enables you to find the first element that matches a given condition.

The following code example demonstrates the use of `List.find` to find the first number that is divisible by 5 in a list.

```
let isDivisibleBy number elem = elem % number = 0
let result = List.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

The result is 5.

If the elements must be transformed first, call List.pick, which takes a function that returns an option, and looks for the first option value that is `Some(x)`. Instead of returning the element, `List.pick` returns the result `x`. If no matching element is found, `List.pick` throws `System.Collections.Generic.KeyNotFoundException`. The following code shows the use of `List.pick`.

```
let valuesList = [ ("a", 1); ("b", 2); ("c", 3) ]

let resultPick = List.pick (fun elem ->
                    match elem with
                    | (value, 2) -> Some value
                    | _ -> None) valuesList
printfn "%A" resultPick
```

The output is as follows:

```
"b"
```

Another group of search operations, List.tryFind and related functions, return an option value. The `List.tryFind` function returns the first element of a list that satisfies a condition if such an element exists, but the option value `None` if not. The variation List.tryFindIndex returns the index of the element, if one is found, rather than the element itself. These functions are illustrated in the following code.

```
let list1d = [1; 3; 7; 9; 11; 13; 15; 19; 22; 29; 36]
let isEven x = x % 2 = 0
match List.tryFind isEven list1d with
| Some value -> printfn "The first even value is %d." value
| None -> printfn "There is no even value in the list."

match List.tryFindIndex isEven list1d with
| Some value -> printfn "The first even value is at position %d." value
| None -> printfn "There is no even value in the list."
```

The output is as follows:

```
The first even value is 22.
The first even value is at position 8.
```

Arithmetic Operations on Lists

Common arithmetic operations such as sum and average are built into the List module. To work with List.sum, the list element type must support the `+` operator and have a zero value. All built-in arithmetic types satisfy these conditions. To work with List.average, the element type must support division without a remainder, which excludes integral types but allows for floating point types. The List.sumBy and List.averageBy functions take a function as a parameter, and this function's results are used to calculate the values for the sum or average.

The following code demonstrates the use of `List.sum`, `List.sumBy`, and `List.average`.

```
// Compute the sum of the first 10 integers by using List.sum.
let sum1 = List.sum [1 .. 10]

// Compute the sum of the squares of the elements of a list by using List.sumBy.
let sum2 = List.sumBy (fun elem -> elem*elem) [1 .. 10]

// Compute the average of the elements of a list by using List.average.
let avg1 = List.average [0.0; 1.0; 1.0; 2.0]

printfn "%f" avg1
```

The output is `1.000000`.

The following code shows the use of `List.averageBy`.

```
let avg2 = List.averageBy (fun elem -> float elem) [1 .. 10]
printfn "%f" avg2
```

The output is `5.5`.

Lists and Tuples

Lists that contain tuples can be manipulated by zip and unzip functions. These functions combine two lists of single values into one list of tuples or separate one list of tuples into two lists of single values. The simplest List.zip function takes two lists of single elements and produces a single list of tuple pairs. Another version, List.zip3, takes three lists of single elements and produces a single list of tuples that have three elements. The following code example demonstrates the use of `List.zip`.

```
let list1 = [ 1; 2; 3 ]
let list2 = [ -1; -2; -3 ]
let listZip = List.zip list1 list2
printfn "%A" listZip
```

The output is as follows:

```
[(1, -1); (2, -2); (3; -3)]
```

The following code example demonstrates the use of `List.zip3`.

```
let list3 = [ 0; 0; 0]
let listZip3 = List.zip3 list1 list2 list3
printfn "%A" listZip3
```

The output is as follows:

```
[(1, -1, 0); (2, -2, 0); (3, -3, 0)]
```

The corresponding unzip versions, List.unzip and List.unzip3, take lists of tuples and return lists in a tuple, where the first list contains all the elements that were first in each tuple, and the second list contains the second element of each tuple, and so on.

The following code example demonstrates the use of List.unzip.

```
let lists = List.unzip [(1,2); (3,4)]
printfn "%A" lists
printfn "%A %A" (fst lists) (snd lists)
```

The output is as follows:

```
([1; 3], [2; 4])
[1; 3] [2; 4]
```

The following code example demonstrates the use of List.unzip3.

```
let listsUnzip3 = List.unzip3 [(1,2,3); (4,5,6)]
printfn "%A" listsUnzip3
```

The output is as follows:

```
([1; 4], [2; 5], [3; 6])
```

Operating on List Elements

F# supports a variety of operations on list elements. The simplest is List.iter, which enables you to call a function on every element of a list. Variations include List.iter2, which enables you to perform an operation on elements of two lists, List.iteri, which is like `List.iter` except that the index of each element is passed as an argument to the function that is called for each element, and List.iteri2, which is a combination of the functionality of

`List.iter2` and `List.iteri`. The following code example illustrates these functions.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
List.iter (fun x -> printfn "List.iter: element is %d" x) list1
List.iteri(fun i x -> printfn "List.iteri: element %d is %d" i x) list1
List.iter2 (fun x y -> printfn "List.iter2: elements are %d %d" x y) list1 list2
List.iteri2 (fun i x y ->
                printfn "List.iteri2: element %d of list1 is %d element %d of list2 is %d"
                  i x i y)
            list1 list2
```

The output is as follows:

```
List.iter: element is 1
List.iter: element is 2
List.iter: element is 3
List.iteri: element 0 is 1
List.iteri: element 1 is 2
List.iteri: element 2 is 3
List.iter2: elements are 1 4
List.iter2: elements are 2 5
List.iter2: elements are 3 6
List.iteri2: element 0 of list1 is 1; element 0 of list2 is 4
List.iteri2: element 1 of list1 is 2; element 1 of list2 is 5
List.iteri2: element 2 of list1 is 3; element 2 of list2 is 6
```

Another frequently used function that transforms list elements is List.map, which enables you to apply a function to each element of a list and put all the results into a new list. List.map2 and List.map3 are variations that take multiple lists. You can also use List.mapi and List.mapi2, if, in addition to the element, the function needs to be passed the index of each element. The only difference between `List.mapi2` and `List.mapi` is that `List.mapi2` works with two lists. The following example illustrates List.map.

```
let list1 = [1; 2; 3]
let newList = List.map (fun x -> x + 1) list1
printfn "%A" newList
```

The output is as follows:

```
[2; 3; 4]
```

The following example shows the use of `List.map2`.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let sumList = List.map2 (fun x y -> x + y) list1 list2
printfn "%A" sumList
```

The output is as follows:

```
[5; 7; 9]
```

The following example shows the use of `List.map3`.

```
let newList2 = List.map3 (fun x y z -> x + y + z) list1 list2 [2; 3; 4]
printfn "%A" newList2
```

The output is as follows:

```
[7; 10; 13]
```

The following example shows the use of `List.mapi`.

```
let newListAddIndex = List.mapi (fun i x -> x + i) list1
printfn "%A" newListAddIndex
```

The output is as follows:

```
[1; 3; 5]
```

The following example shows the use of `List.mapi2`.

```
let listAddTimesIndex = List.mapi2 (fun i x y -> (x + y) * i) list1 list2
printfn "%A" listAddTimesIndex
```

The output is as follows:

```
[0; 7; 18]
```

List.collect is like `List.map`, except that each element produces a list and all these lists are concatenated into a final list. In the following code, each element of the list generates three numbers. These are all collected into one list.

```
let collectList = List.collect (fun x -> [for i in 1..3 -> x * i]) list1
printfn "%A" collectList
```

The output is as follows:

```
[1; 2; 3; 2; 4; 6; 3; 6; 9]
```

You can also use List.filter, which takes a Boolean condition and produces a new list that consists only of elements that satisfy the given condition.

```
let evenOnlyList = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6]
```

The resulting list is `[2; 4; 6]`.

A combination of map and filter, List.choose enables you to transform and select elements at the same time. `List.choose` applies a function that returns an option to each element of a list, and returns a new list of the results for elements when the function returns the option value `Some`.

The following code demonstrates the use of `List.choose` to select capitalized words out of a list of words.

```
let listWords = [ "and"; "Rome"; "Bob"; "apple"; "zebra" ]
let isCapitalized (string1:string) = System.Char.IsUpper string1[0]
let results = List.choose (fun elem ->
    match elem with
    | elem when isCapitalized elem -> Some(elem + "'s")
    | _ -> None) listWords
printfn "%A" results
```

The output is as follows:

```
["Rome's"; "Bob's"]
```

Operating on Multiple Lists

Lists can be joined together. To join two lists into one, use List.append. To join more than two lists, use List.concat.

```
let list1to10 = List.append [1; 2; 3] [4; 5; 6; 7; 8; 9; 10]
let listResult = List.concat [ [1; 2; 3]; [4; 5; 6]; [7; 8; 9] ]
List.iter (fun elem -> printf "%d " elem) list1to10
printfn ""
List.iter (fun elem -> printf "%d " elem) listResult
```

Fold and Scan Operations

Some list operations involve interdependencies between all of the list elements. The fold and scan operations are like `List.iter` and `List.map` in that you invoke a function on each element, but these operations provide an additional parameter called the *accumulator* that carries information through the computation.

Use `List.fold` to perform a calculation on a list.

The following code example demonstrates the use of List.fold to perform various operations.

The list is traversed; the accumulator `acc` is a value that is passed along as the calculation proceeds. The first argument takes the accumulator and the list element, and returns the interim result of the calculation for that list element. The second argument is the initial value of the accumulator.

```
let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 3 ] (sumList [ 1 .. 3 ])

// The following example computes the average of a list.
let averageList list = (List.fold (fun acc elem -> acc + float elem) 0.0 list / float list.Length)

// The following example computes the standard deviation of a list.
// The standard deviation is computed by taking the square root of the
// sum of the variances, which are the differences between each value
// and the average.
let stdDevList list =
    let avg = averageList list
    sqrt (List.fold (fun acc elem -> acc + (float elem - avg) ** 2.0 ) 0.0 list / float list.Length)

let testList listTest =
    printfn "List %A average: %f stddev: %f" listTest (averageList listTest) (stdDevList listTest)

testList [1; 1; 1]
testList [1; 2; 1]
testList [1; 2; 3]

// List.fold is the same as to List.iter when the accumulator is not used.
let printList list = List.fold (fun acc elem -> printfn "%A" elem) () list
printList [0.0; 1.0; 2.5; 5.1 ]

// The following example uses List.fold to reverse a list.
// The accumulator starts out as the empty list, and the function uses the cons operator
// to add each successive element to the head of the accumulator list, resulting in a
// reversed form of the list.
let reverseList list = List.fold (fun acc elem -> elem::acc) [] list
printfn "%A" (reverseList [1 .. 10])
```

The versions of these functions that have a digit in the function name operate on more than one list. For example, List.fold2 performs computations on two lists.

The following example demonstrates the use of `List.fold2`.

```
// Use List.fold2 to perform computations over two lists (of equal size) at the same time.
// Example: Sum the greater element at each list position.
let sumGreatest list1 list2 = List.fold2 (fun acc elem1 elem2 ->
                                              acc + max elem1 elem2) 0 list1 list2

let sum = sumGreatest [1; 2; 3] [3; 2; 1]
printfn "The sum of the greater of each pair of elements in the two lists is %d." sum
```

`List.fold` and List.scan differ in that `List.fold` returns the final value of the extra parameter, but `List.scan` returns the list of the intermediate values (along with the final value) of the extra parameter.

Each of these functions includes a reverse variation, for example, List.foldBack, which differs in the order in which the list is traversed and the order of the arguments. Also, `List.fold` and `List.foldBack` have variations, List.fold2 and List.foldBack2, that take two lists of equal length. The function that executes on each element can use corresponding elements of both lists to perform some action. The element types of the two lists can be different, as in the following example, in which one list contains transaction amounts for a bank account, and the other list contains the type of transaction: deposit or withdrawal.

```
// Discriminated union type that encodes the transaction type.
type Transaction =
    | Deposit
    | Withdrawal

let transactionTypes = [Deposit; Deposit; Withdrawal]
let transactionAmounts = [100.00; 1000.00; 95.00 ]
let initialBalance = 200.00

// Use fold2 to perform a calculation on the list to update the account balance.
let endingBalance = List.fold2 (fun acc elem1 elem2 ->
                                match elem1 with
                                | Deposit -> acc + elem2
                                | Withdrawal -> acc - elem2)
                                initialBalance
                                transactionTypes
                                transactionAmounts
printfn "%f" endingBalance
```

For a calculation like summation, `List.fold` and `List.foldBack` have the same effect because the result does not depend on the order of traversal. In the following example, `List.foldBack` is used to add the elements in a list.

```
let sumListBack list = List.foldBack (fun elem acc -> acc + elem) list 0
printfn "%d" (sumListBack [1; 2; 3])

// For a calculation in which the order of traversal is important, fold and foldBack have different
// results. For example, replacing fold with foldBack in the listReverse function
// produces a function that copies the list, rather than reversing it.
let copyList list = List.foldBack (fun elem acc -> elem::acc) list []
printfn "%A" (copyList [1 .. 10])
```

The following example returns to the bank account example. This time a new transaction type is added: an interest calculation. The ending balance now depends on the order of transactions.

```
type Transaction2 =
    | Deposit
    | Withdrawal
    | Interest

let transactionTypes2 = [Deposit; Deposit; Withdrawal; Interest]
let transactionAmounts2 = [100.00; 1000.00; 95.00; 0.05 / 12.0 ]
let initialBalance2 = 200.00

// Because fold2 processes the lists by starting at the head element,
// the interest is calculated last, on the balance of 1205.00.
let endingBalance2 = List.fold2 (fun acc elem1 elem2 ->
                                match elem1 with
                                | Deposit -> acc + elem2
                                | Withdrawal -> acc - elem2
                                | Interest -> acc * (1.0 + elem2))
                                initialBalance2
                                transactionTypes2
                                transactionAmounts2
printfn "%f" endingBalance2


// Because foldBack2 processes the lists by starting at end of the list,
// the interest is calculated first, on the balance of only 200.00.
let endingBalance3 = List.foldBack2 (fun elem1 elem2 acc ->
                                match elem1 with
                                | Deposit -> acc + elem2
                                | Withdrawal -> acc - elem2
                                | Interest -> acc * (1.0 + elem2))
                                transactionTypes2
                                transactionAmounts2
                                initialBalance2
printfn "%f" endingBalance3
```

The function List.reduce is somewhat like `List.fold` and `List.scan`, except that instead of passing around a separate accumulator, `List.reduce` takes a function that takes two arguments of the element type instead of just one, and one of those arguments acts as the accumulator, meaning that it stores the intermediate result of the computation. `List.reduce` starts by operating on the first two list elements, and then uses the result of the operation along with the next element. Because there is not a separate accumulator that has its own type, `List.reduce` can be used in place of `List.fold` only when the accumulator and the element type have the same type. The following code demonstrates the use of `List.reduce`. `List.reduce` throws an exception if the list provided has no elements.

In the following code, the first call to the lambda expression is given the arguments 2 and 4, and returns 6, and the next call is given the arguments 6 and 10, so the result is 16.

```
let sumAList list =
    try
        List.reduce (fun acc elem -> acc + elem) list
    with
       | :? System.ArgumentException as exc -> 0

let resultSum = sumAList [2; 4; 10]
printfn "%d " resultSum
```

Converting Between Lists and Other Collection Types

The `List` module provides functions for converting to and from both sequences and arrays. To convert to or from a sequence, use List.toSeq or List.ofSeq. To convert to or from an array, use List.toArray or List.ofArray.

Additional Operations

For information about additional operations on lists, see the library reference topic List Module.

See also

- F# Language Reference
- F# Types
- Sequences
- Arrays
- Options

Arrays (F#)

8/5/2022 • 17 minutes to read • Edit Online

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.

Create arrays

You can create arrays in several ways. You can create a small array by listing consecutive values between `[|` and `|]` and separated by semicolons, as shown in the following examples.

```
let array1 = [| 1; 2; 3 |]
```

You can also put each element on a separate line, in which case the semicolon separator is optional.

```
let array1 =
    [|
        1
        2
        3
     |]
```

The type of the array elements is inferred from the literals used and must be consistent. The following code causes an error because 1.0 is a float and 2 and 3 are integers.

```
// Causes an error.
// let array2 = [| 1.0; 2; 3 |]
```

You can also use sequence expressions to create arrays. Following is an example that creates an array of squares of integers from 1 to 10.

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

To create an array in which all the elements are initialized to zero, use `Array.zeroCreate`.

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

Access elements

You can access array elements by using brackets (`[` and `]`). The original dot syntax (`.[index]`) is still supported but no longer recommended as of F# 6.0.

```
array1[0]
```

Array indexes start at 0.

You can also access array elements by using slice notation, which enables you to specify a subrange of the array. Examples of slice notation follow.

```
// Accesses elements from 0 to 2.

array1[0..2]

// Accesses elements from the beginning of the array to 2.

array1[..2]

// Accesses elements from 2 to the end of the array.

array1[2..]
```

When slice notation is used, a new copy of the array is created.

Array types and modules

The type of all F# arrays is the .NET Framework type System.Array. Therefore, F# arrays support all the functionality available in System.Array.

The `Array` module supports operations on one-dimensional arrays. The modules `Array2D`, `Array3D`, and `Array4D` contain functions that support operations on arrays of two, three, and four dimensions, respectively. You can create arrays of rank greater than four by using System.Array.

Simple functions

`Array.get` gets an element. `Array.length` gives the length of an array. `Array.set` sets an element to a specified value. The following code example illustrates the use of these functions.

```
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
    Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
    printf "%s " (Array.get array1 i)
```

The output is as follows.

```
0 1 2 3 4 5 6 7 8 9
```

Functions that create arrays

Several functions create arrays without requiring an existing array. `Array.empty` creates a new array that does not contain any elements. `Array.create` creates an array of a specified size and sets all the elements to provided values. `Array.init` creates an array, given a dimension and a function to generate the elements. `Array.zeroCreate` creates an array in which all the elements are initialized to the zero value for the array's type. The following code demonstrates these functions.

```
let myEmptyArray = Array.empty
printfn "Length of empty array: %d" myEmptyArray.Length



printfn "Array of floats set to 5.0: %A" (Array.create 10 5.0)


printfn "Array of squares: %A" (Array.init 10 (fun index -> index * index))

let (myZeroArray : float array) = Array.zeroCreate 10
```

The output is as follows.

```
Length of empty array: 0
Area of floats set to 5.0: [|5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.copy` creates a new array that contains elements that are copied from an existing array. Note that the copy is a shallow copy, which means that if the element type is a reference type, only the reference is copied, not the underlying object. The following code example illustrates this.

```
open System.Text

let firstArray : StringBuilder array = Array.init 3 (fun index -> new StringBuilder(""))
let secondArray = Array.copy firstArray
// Reset an element of the first array to a new value.
firstArray[0] <- new StringBuilder("Test1")
// Change an element of the first array.
firstArray[1].Insert(0, "Test2") |> ignore
printfn "%A" firstArray
printfn "%A" secondArray
```

The output of the preceding code is as follows:

```
[|Test1; Test2; |]
[|; Test2; |]
```

The string `Test1` appears only in the first array because the operation of creating a new element overwrites the reference in `firstArray` but does not affect the original reference to an empty string that is still present in `secondArray`. The string `Test2` appears in both arrays because the `Insert` operation on the System.Text.StringBuilder type affects the underlying System.Text.StringBuilder object, which is referenced in both arrays.

`Array.sub` generates a new array from a subrange of an array. You specify the subrange by providing the starting index and the length. The following code demonstrates the use of `Array.sub`.

```
let a1 = [| 0 .. 99 |]
let a2 = Array.sub a1 5 10
printfn "%A" a2
```

The output shows that the subarray starts at element 5 and contains 10 elements.

```
[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14|]
```

`Array.append` creates a new array by combining two existing arrays.

The following code demonstrates **Array.append**.

```
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

The output of the preceding code is as follows.

```
[|1; 2; 3; 4; 5; 6|]
```

`Array.choose` selects elements of an array to include in a new array. The following code demonstrates `Array.choose`. Note that the element type of the array does not have to match the type of the value returned in the option type. In this example, the element type is `int` and the option is the result of a polynomial function, `elem*elem - 1`, as a floating point number.

```
printfn "%A" (Array.choose (fun elem -> if elem % 2 = 0 then
                                            Some(float (elem*elem - 1))
                                        else
                                            None) [| 1 .. 10 |])
```

The output of the preceding code is as follows.

```
[|3.0; 15.0; 35.0; 63.0; 99.0|]
```

`Array.collect` runs a specified function on each array element of an existing array and then collects the elements generated by the function and combines them into a new array. The following code demonstrates `Array.collect`.

```
printfn "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

The output of the preceding code is as follows.

```
[|0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

`Array.concat` takes a sequence of arrays and combines them into a single array. The following code demonstrates `Array.concat`.

```
Array.concat [ [|0..3|] ; [|4|] ]
//output [|0; 1; 2; 3; 4|]

Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]
```

The output of the preceding code is as follows.

```
[|(1, 1, 1); (1, 2, 2); (1, 3, 3); (2, 1, 2); (2, 2, 4); (2, 3, 6); (3, 1, 3);
(3, 2, 6); (3, 3, 9)|]
```

`Array.filter` takes a Boolean condition function and generates a new array that contains only those elements from the input array for which the condition is true. The following code demonstrates `Array.filter`.

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

The output of the preceding code is as follows.

```
[|2; 4; 6; 8; 10|]
```

`Array.rev` generates a new array by reversing the order of an existing array. The following code demonstrates `Array.rev`.

```
let stringReverse (s: string) =
    System.String(Array.rev (s.ToCharArray()))

printfn "%A" (stringReverse("!dlrow olleH"))
```

The output of the preceding code is as follows.

```
"Hello world!"
```

You can easily combine functions in the array module that transform arrays by using the pipeline operator (`|>`), as shown in the following example.

```
[| 1 .. 10 |]
|> Array.filter (fun elem -> elem % 2 = 0)
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)
|> Array.rev
|> printfn "%A"
```

The output is

```
[|100; 36; 16; 4|]
```

Multidimensional arrays

A multidimensional array can be created, but there is no syntax for writing a multidimensional array literal. Use the operator `array2D` to create an array from a sequence of sequences of array elements. The sequences can be array or list literals. For example, the following code creates a two-dimensional array.

```
let my2DArray = array2D [ [ 1; 0]; [0; 1] ]
```

You can also use the function `Array2D.init` to initialize arrays of two dimensions, and similar functions are available for arrays of three and four dimensions. These functions take a function that is used to create the elements. To create a two-dimensional array that contains elements set to an initial value instead of specifying a function, use the `Array2D.create` function, which is also available for arrays up to four dimensions. The following code example first shows how to create an array of arrays that contain the desired elements, and then uses `Array2D.init` to generate the desired two-dimensional array.

```
let arrayOfArrays = [| [| 1.0; 0.0 |]; [|0.0; 1.0 |] |]
let twoDimensionalArray = Array2D.init 2 2 (fun i j -> arrayOfArrays[i][j])
```

Array indexing and slicing syntax is supported for arrays up to rank 4. When you specify an index in multiple dimensions, you use commas to separate the indexes, as illustrated in the following code example.

```
twoDimensionalArray[0, 1] <- 1.0
```

The type of a two-dimensional array is written out as `<type>[,]` (for example, `int[,]`, `double[,]`), and the type of a three-dimensional array is written as `<type>[,,]`, and so on for arrays of higher dimensions.

Only a subset of the functions available for one-dimensional arrays is also available for multidimensional arrays.

Array slicing and multidimensional arrays

In a two-dimensional array (a matrix), you can extract a sub-matrix by specifying ranges and using a wildcard (

`*`) character to specify whole rows or columns.

```
// Get rows 1 to N from an NxM matrix (returns a matrix):
matrix[1.., *]

// Get rows 1 to 3 from a matrix (returns a matrix):
matrix[1..3, *]

// Get columns 1 to 3 from a matrix (returns a matrix):
matrix[*, 1..3]

// Get a 3x3 submatrix:
matrix[1..3, 1..3]
```

You can decompose a multidimensional array into subarrays of the same or lower dimension. For example, you can obtain a vector from a matrix by specifying a single row or column.

```
// Get row 3 from a matrix as a vector:
matrix[3, *]

// Get column 3 from a matrix as a vector:
matrix[*, 3]
```

You can use this slicing syntax for types that implement the element access operators and overloaded `GetSlice` methods. For example, the following code creates a Matrix type that wraps the F# 2D array, implements an Item property to provide support for array indexing, and implements three versions of `GetSlice`. If you can use this code as a template for your matrix types, you can use all the slicing operations that this section describes.

```
type Matrix<'T>(N: int, M: int) =
    let internalArray = Array2D.zeroCreate<'T> N M

    member this.Item
        with get(a: int, b: int) = internalArray[a, b]
        and set(a: int, b: int) (value:'T) = internalArray[a, b] <- value

    member this.GetSlice(rowStart: int option, rowFinish : int option, colStart: int option, colFinish : int
option) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray[rowStart..rowFinish, colStart..colFinish]

    member this.GetSlice(row: int, colStart: int option, colFinish: int option) =
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray[row, colStart..colFinish]

    member this.GetSlice(rowStart: int option, rowFinish: int option, col: int) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        internalArray[rowStart..rowFinish, col]

module test =
    let generateTestMatrix x y =
        let matrix = new Matrix<float>(3, 3)
        for i in 0..2 do
            for j in 0..2 do
                matrix[i, j] <- float(i) * x - float(j) * y
        matrix

    let test1 = generateTestMatrix 2.3 1.1
    let submatrix = test1[0..1, 0..1]
    printfn $"{submatrix}"

    let firstRow = test1[0,*]
    let secondRow = test1[1,*]
    let firstCol = test1[*,0]
    printfn $"{firstCol}"
```

Boolean functions on arrays

The functions `Array.exists` and `Array.exists2` test elements in either one or two arrays, respectively. These functions take a test function and return `true` if there is an element (or element pair for `Array.exists2`) that satisfies the condition.

The following code demonstrates the use of `Array.exists` and `Array.exists2`. In these examples, new functions are created by applying only one of the arguments, in these cases, the function argument.

```
let allNegative = Array.exists (fun elem -> abs (elem) = elem) >> not
printfn "%A" (allNegative [| -1; -2; -3 |])
printfn "%A" (allNegative [| -10; -1; 5 |])
printfn "%A" (allNegative [| 0 |])


let haveEqualElement = Array.exists2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (haveEqualElement [| 1; 2; 3 |] [| 3; 2; 1|])
```

The output of the preceding code is as follows.

```
true
false
false
true
```

Similarly, the function `Array.forall` tests an array to determine whether every element satisfies a Boolean condition. The variation `Array.forall2` does the same thing by using a Boolean function that involves elements of two arrays of equal length. The following code illustrates the use of these functions.

```
let allPositive = Array.forall (fun elem -> elem > 0)
printfn "%A" (allPositive [| 0; 1; 2; 3 |])
printfn "%A" (allPositive [| 1; 2; 3 |])


let allEqual = Array.forall2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (allEqual [| 1; 2 |] [| 1; 2 |])
printfn "%A" (allEqual [| 1; 2 |] [| 2; 1 |])
```

The output for these examples is as follows.

```
false
true
true
false
```

Search arrays

`Array.find` takes a Boolean function and returns the first element for which the function returns `true`, or raises a System.Collections.Generic.KeyNotFoundException if no element that satisfies the condition is found. `Array.findIndex` is like `Array.find`, except that it returns the index of the element instead of the element itself.

The following code uses `Array.find` and `Array.findIndex` to locate a number that is both a perfect square and perfect cube.

```
let arrayA = [| 2 .. 100 |]
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
printfn "The first element that is both a square and a cube is %d and its index is %d." element index
```

The output is as follows.

```
The first element that is both a square and a cube is 64 and its index is 62.
```

`Array.tryFind` is like `Array.find`, except that its result is an option type, and it returns `None` if no element is found. `Array.tryFind` should be used instead of `Array.find` when you do not know whether a matching element is in the array. Similarly, `Array.tryFindIndex` is like `Array.findIndex` except that the option type is the return value. If no element is found, the option is `None`.

The following code demonstrates the use of `Array.tryFind`. This code depends on the previous code.

```
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let lookForCubeAndSquare array1 =
    let result = Array.tryFind (fun elem -> isPerfectSquare elem && isPerfectCube elem) array1
    match result with
    | Some x -> printfn "Found an element: %d" x
    | None -> printfn "Failed to find a matching element."

lookForCubeAndSquare [| 1 .. 10 |]
lookForCubeAndSquare [| 100 .. 1000 |]
lookForCubeAndSquare [| 2 .. 50 |]
```

The output is as follows.

```
Found an element: 1
Found an element: 729
Failed to find a matching element.
```

Use `Array.tryPick` when you need to transform an element in addition to finding it. The result is the first element for which the function returns the transformed element as an option value, or `None` if no such element is found.

The following code shows the use of `Array.tryPick`. In this case, instead of a lambda expression, several local helper functions are defined to simplify the code.

```
let findPerfectSquareAndCube array1 =
    let delta = 1.0e-10
    let isPerfectSquare (x:int) =
        let y = sqrt (float x)
        abs(y - round y) < delta
    let isPerfectCube (x:int) =
        let y = System.Math.Pow(float x, 1.0/3.0)
        abs(y - round y) < delta
    // intFunction : (float -> float) -> int -> int
    // Allows the use of a floating point function with integers.
    let intFunction function1 number = int (round (function1 (float number)))
    let cubeRoot x = System.Math.Pow(x, 1.0/3.0)
    // testElement: int -> (int * int * int) option
    // Test an element to see whether it is a perfect square and a perfect
    // cube, and, if so, return the element, square root, and cube root
    // as an option value. Otherwise, return None.
    let testElement elem =
        if isPerfectSquare elem && isPerfectCube elem then
            Some(elem, intFunction sqrt elem, intFunction cubeRoot elem)
        else None
    match Array.tryPick testElement array1 with
    | Some (n, sqrt, cuberoot) -> printfn "Found an element %d with square root %d and cube root %d." n sqrt
cuberoot
    | None -> printfn "Did not find an element that is both a perfect square and a perfect cube."

findPerfectSquareAndCube [| 1 .. 10 |]
findPerfectSquareAndCube [| 2 .. 100 |]
findPerfectSquareAndCube [| 100 .. 1000 |]
findPerfectSquareAndCube [| 1000 .. 10000 |]
findPerfectSquareAndCube [| 2 .. 50 |]
```

The output is as follows.

```
Found an element 1 with square root 1 and cube root 1.
Found an element 64 with square root 8 and cube root 4.
Found an element 729 with square root 27 and cube root 9.
Found an element 4096 with square root 64 and cube root 16.
Did not find an element that is both a perfect square and a perfect cube.
```

Perform computations on arrays

The `Array.average` function returns the average of each element in an array. It is limited to element types that support exact division by an integer, which includes floating point types but not integral types. The `Array.averageBy` function returns the average of the results of calling a function on each element. For an array of integral type, you can use `Array.averageBy` and have the function convert each element to a floating point type for the computation.

Use `Array.max` or `Array.min` to get the maximum or minimum element, if the element type supports it. Similarly, `Array.maxBy` and `Array.minBy` allow a function to be executed first, perhaps to transform to a type that supports comparison.

`Array.sum` adds the elements of an array, and `Array.sumBy` calls a function on each element and adds the results together.

To execute a function on each element in an array without storing the return values, use `Array.iter`. For a function involving two arrays of equal length, use `Array.iter2`. If you also need to keep an array of the results of the function, use `Array.map` or `Array.map2`, which operates on two arrays at a time.

The variations `Array.iteri` and `Array.iteri2` allow the index of the element to be involved in the computation; the same is true for `Array.mapi` and `Array.mapi2`.

The functions `Array.fold`, `Array.foldBack`, `Array.reduce`, `Array.reduceBack`, `Array.scan`, and `Array.scanBack` execute algorithms that involve all the elements of an array. Similarly, the variations `Array.fold2` and `Array.foldBack2` perform computations on two arrays.

These functions for performing computations correspond to the functions of the same name in the List module. For usage examples, see Lists.

Modify arrays

`Array.set` sets an element to a specified value. `Array.fill` sets a range of elements in an array to a specified value. The following code provides an example of `Array.fill`.

```
let arrayFill1 = [| 1 .. 25 |]
Array.fill arrayFill1 2 20 0
printfn "%A" arrayFill1
```

The output is as follows.

```
[|1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 23; 24; 25|]
```

You can use `Array.blit` to copy a subsection of one array to another array.

Convert to and from other types

`Array.ofList` creates an array from a list. `Array.ofSeq` creates an array from a sequence. `Array.toList` and `Array.toSeq` convert to these other collection types from the array type.

Sort arrays

Use `Array.sort` to sort an array by using the generic comparison function. Use `Array.sortBy` to specify a function that generates a value, referred to as a *key*, to sort by using the generic comparison function on the key. Use `Array.sortWith` if you want to provide a custom comparison function. `Array.sort`, `Array.sortBy`, and `Array.sortWith` all return the sorted array as a new array. The variations `Array.sortInPlace`, `Array.sortInPlaceBy`, and `Array.sortInPlaceWith` modify the existing array instead of returning a new one.

Arrays and tuples

The functions `Array.zip` and `Array.unzip` convert arrays of tuple pairs to tuples of arrays and vice versa. `Array.zip3` and `Array.unzip3` are similar except that they work with tuples of three elements or tuples of three arrays.

Parallel computations on arrays

The module `Array.Parallel` contains functions for performing parallel computations on arrays. This module is not available in applications that target versions of the .NET Framework prior to version 4.

See also

- F# Language Reference
- F# Types

Slices

8/5/2022 • 5 minutes to read • Edit Online

This article explains how to take slices from existing F# types and how to define your own slices.

In F#, a slice is a subset of any data type. Slices are similar to indexers, but instead of yielding a single value from the underlying data structure, they yield multiple ones. Slices use the `..` operator syntax to select the range of specified indices in a data type. For more information, see the looping expression reference article.

F# currently has intrinsic support for slicing strings, lists, arrays, and multidimensional (2D, 3D, 4D) arrays. Slicing is most commonly used with F# arrays and lists. You can add slicing to your custom data types by using the `GetSlice` method in your type definition or in an in-scope type extension.

Slicing F# lists and arrays

The most common data types that are sliced are F# lists and arrays. The following example demonstrates how to slice lists:

```
// Generate a list of 100 integers
let fullList = [ 1 .. 100 ]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullList[1..5]
printfn $"Small slice: {smallSlice}"

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullList[..5]
printfn $"Unbounded beginning slice: {unboundedBeginning}"

// Create a slice from an index to the end of the list
let unboundedEnd = fullList[94..]
printfn $"Unbounded end slice: {unboundedEnd}"
```

Slicing arrays is just like slicing lists:

```
// Generate an array of 100 integers
let fullArray = [| 1 .. 100 |]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullArray[1..5]
printfn $"Small slice: {smallSlice}"

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullArray[..5]
printfn $"Unbounded beginning slice: {unboundedBeginning}"

// Create a slice from an index to the end of the list
let unboundedEnd = fullArray[94..]
printfn $"Unbounded end slice: {unboundedEnd}"
```

Prior to F# 6, slicing used the syntax `expr.[start..finish]` with the extra `.`. If you choose, you can still use this syntax. For more information, see RFC FS-1110.

Slicing multidimensional arrays

F# supports multidimensional arrays in the F# core library. As with one-dimensional arrays, slices of multidimensional arrays can also be useful. However, the introduction of additional dimensions mandates a slightly different syntax so that you can take slices of specific rows and columns.

The following examples demonstrate how to slice a 2D array:

```
// Generate a 3x3 2D matrix
let A = array2D [[1;2;3];[4;5;6];[7;8;9]]
printfn $"Full matrix:\n {A}"

// Take the first row
let row0 = A[0,*]
printfn $"{row0}"

// Take the first column
let col0 = A[*,0]
printfn $"{col0}"

// Take all rows but only two columns
let subA = A[*,0..1]
printfn $"{subA}"

// Take two rows and all columns
let subA' = A[0..1,*]
printfn $"{subA}"

// Slice a 2x2 matrix out of the full 3x3 matrix
let twoByTwo = A[0..1,0..1]
printfn $"{twoByTwo}"
```

Defining slices for other data structures

The F# core library defines slices for a limited set of types. If you wish to define slices for more data types, you can do so either in the type definition itself or in a type extension.

For example, here's how you might define slices for the ArraySegment<T> class to allow for convenient data manipulation:

```
open System

type ArraySegment<'TItem> with
    member segment.GetSlice(start, finish) =
        let start = defaultArg start 0
        let finish = defaultArg finish segment.Count
        ArraySegment(segment.Array, segment.Offset + start, finish - start)

let arr = ArraySegment [| 1 .. 10 |]
let slice = arr[2..5] //[ 3; 4; 5]
```

Another example using the Span<T> and ReadOnlySpan<T> types:

```
open System

type ReadOnlySpan<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

type Span<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn $"{arr}"

let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
printSpan sp[0..] // [|1; 2; 3; 4; 5|]
printSpan sp[..5] // [|1; 2; 3; 4; 5|]
printSpan sp[0..3] // [|1; 2; 3|]
printSpan sp[1..3] // |2; 3|]
```

Built-in F# slices are end-inclusive

All intrinsic slices in F# are end-inclusive; that is, the upper bound is included in the slice. For a given slice with starting index `x` and ending index `y`, the resulting slice will include the *yth* value.

```
// Define a new list
let xs = [1 .. 10]

printfn $"{xs[2..5]}" // Includes the 5th index
```

Built-in F# empty slices

F# lists, arrays, sequences, strings, multidimensional (2D, 3D, 4D) arrays will all produce an empty slice if the syntax could produce a slice that doesn't exist.

Consider the following example:

```
let l = [ 1..10 ]
let a = [| 1..10 |]
let s = "hello!"

let emptyList = l[-2..(-1)]
let emptyArray = a[-2..(-1)]
let emptyString = s[-2..(-1)]
```

IMPORTANT

C# developers may expect these to throw an exception rather than produce an empty slice. This is a design decision rooted in the fact that empty collections compose in F#. An empty F# list can be composed with another F# list, an empty string can be added to an existing string, and so on. It can be common to take slices based on values passed in as parameters, and being tolerant of out-of-bounds > by producing an empty collection fits with the compositional nature of F# code.

Fixed-index slices for 3D and 4D arrays

For F# 3D and 4D arrays, you can "fix" a particular index and slice other dimensions with that index fixed.

To illustrate this, consider the following 3D array:

z = 0

X\Y	0	1
0	0	1
1	2	3

z = 1

X\Y	0	1
0	4	5
1	6	7

If you want to extract the slice `[| 4; 5 |]` from the array, use a fixed-index slice.

```
let dim = 2
let m = Array3D.zeroCreate<int> dim dim dim

let mutable count = 0

for z in 0..dim-1 do
    for y in 0..dim-1 do
        for x in 0..dim-1 do
            m[x,y,z] <- count
            count <- count + 1

// Now let's get the [4;5] slice!
m[*, 0, 1]
```

The last line fixes the `y` and `z` indices of the 3D array and takes the rest of the `x` values that correspond to the matrix.

See also

- Indexed properties

Sequences

8/5/2022 • 19 minutes to read • Edit Online

A *sequence* is a logical series of elements all of one type. Sequences are particularly useful when you have a large, ordered collection of data but do not necessarily expect to use all of the elements. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used. Sequences are represented by the `seq<'T>` type, which is an alias for IEnumerable<T>. Therefore, any .NET type that implements IEnumerable<T> interface can be used as a sequence. The Seq module provides support for manipulations involving sequences.

Sequence Expressions

A *sequence expression* is an expression that evaluates to a sequence. Sequence expressions can take a number of forms. The simplest form specifies a range. For example, `seq { 1 .. 5 }` creates a sequence that contains five elements, including the endpoints 1 and 5. You can also specify an increment (or decrement) between two double periods. For example, the following code creates the sequence of multiples of 10.

```
// Sequence that has an increment.
seq { 0 .. 10 .. 100 }
```

Sequence expressions are made up of F# expressions that produce values of the sequence. You can also generate values programmatically:

```
seq { for i in 1 .. 10 -> i * i }
```

The previous sample uses the `->` operator, which allows you to specify an expression whose value will become a part of the sequence. You can only use `->` if every part of the code that follows it returns a value.

Alternatively, you can specify the `do` keyword, with an optional `yield` that follows:

```
seq { for i in 1 .. 10 do yield i * i }

// The 'yield' is implicit and doesn't need to be specified in most cases.
seq { for i in 1 .. 10 do i * i }
```

The following code generates a list of coordinate pairs along with an index into an array that represents the grid. Note that the first `for` expression requires a `do` to be specified.

```
let (height, width) = (10, 10)

seq {
    for row in 0 .. width - 1 do
        for col in 0 .. height - 1 ->
            (row, col, row*width + col)
    }
```

An `if` expression used in a sequence is a filter. For example, to generate a sequence of only prime numbers, assuming that you have a function `isprime` of type `int -> bool`, construct the sequence as follows.

```
seq { for n in 1 .. 100 do if isprime n then n }
```

As mentioned previously, `do` is required here because there is no `else` branch that goes with the `if`. If you try to use `->`, you'll get an error saying that not all branches return a value.

The `yield!` keyword

Sometimes, you may wish to include a sequence of elements into another sequence. To include a sequence within another sequence, you'll need to use the `yield!` keyword:

```
// Repeats '1 2 3 4 5' ten times
seq {
    for _ in 1..10 do
        yield! seq { 1; 2; 3; 4; 5}
}
```

Another way of thinking of `yield!` is that it flattens an inner sequence and then includes that in the containing sequence.

When `yield!` is used in an expression, all other single values must use the `yield` keyword:

```
// Combine repeated values with their values
seq {
    for x in 1..10 do
        yield x
        yield! seq { for i in 1..x -> i}
}
```

The previous example will produce the value of `x` in addition to all values from `1` to `x` for each `x`.

Examples

The first example uses a sequence expression that contains an iteration, a filter, and a yield to generate an array. This code prints a sequence of prime numbers between 1 and 100 to the console.

```
// Recursive isprime function.
let isprime n =
    let rec check i =
        i > n/2 || (n % i <> 0 && check (i + 1))
    check 2

let aSequence =
    seq {
        for n in 1..100 do
            if isprime n then
                n
    }

for x in aSequence do
    printfn "%d" x
```

The following example creates a multiplication table that consists of tuples of three elements, each consisting of two factors and the product:

```
let multiplicationTable =
    seq {
        for i in 1..9 do
            for j in 1..9 ->
                (i, j, i*j)
    }
```

The following example demonstrates the use of `yield!` to combine individual sequences into a single final sequence. In this case, the sequences for each subtree in a binary tree are concatenated in a recursive function to produce the final sequence.

```
// Yield the values of a binary tree in a sequence.
type Tree<'a> =
   | Tree of 'a * Tree<'a> * Tree<'a>
   | Leaf of 'a

// inorder : Tree<'a> -> seq<'a>
let rec inorder tree =
    seq {
      match tree with
          | Tree(x, left, right) ->
               yield! inorder left
               yield x
               yield! inorder right
          | Leaf x -> yield x
    }

let mytree = Tree(6, Tree(2, Leaf(1), Leaf(3)), Leaf(9))
let seq1 = inorder mytree
printfn "%A" seq1
```

Using Sequences

Sequences support many of the same functions as lists. Sequences also support operations such as grouping and counting by using key-generating functions. Sequences also support more diverse functions for extracting subsequences.

Many data types, such as lists, arrays, sets, and maps are implicitly sequences because they are enumerable collections. A function that takes a sequence as an argument works with any of the common F# data types, in addition to any .NET data type that implements `System.Collections.Generic.IEnumerable<'T>`. Contrast this to a function that takes a list as an argument, which can only take lists. The type `seq<'T>` is a type abbreviation for `IEnumerable<'T>`. This means that any type that implements the generic `System.Collections.Generic.IEnumerable<'T>`, which includes arrays, lists, sets, and maps in F#, and also most .NET collection types, is compatible with the `seq` type and can be used wherever a sequence is expected.

Module Functions

The Seq module in the FSharp.Collections namespace contains functions for working with sequences. These functions work with lists, arrays, maps, and sets as well, because all of those types are enumerable, and therefore can be treated as sequences.

Creating Sequences

You can create sequences by using sequence expressions, as described previously, or by using certain functions.

You can create an empty sequence by using Seq.empty, or you can create a sequence of just one specified element by using Seq.singleton.

```
let seqEmpty = Seq.empty
let seqOne = Seq.singleton 10
```

You can use Seq.init to create a sequence for which the elements are created by using a function that you provide. You also provide a size for the sequence. This function is just like List.init, except that the elements are not created until you iterate through the sequence. The following code illustrates the use of `Seq.init`.

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

The output is

```
0 10 20 30 40
```

By using Seq.ofArray and Seq.ofList<'T> Function, you can create sequences from arrays and lists. However, you can also convert arrays and lists to sequences by using a cast operator. Both techniques are shown in the following code.

```
// Convert an array to a sequence by using a cast.
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>

// Convert an array to a sequence by using Seq.ofArray.
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

By using Seq.cast, you can create a sequence from a weakly typed collection, such as those defined in `System.Collections`. Such weakly typed collections have the element type `System.Object` and are enumerated by using the non-generic `System.Collections.Generic.IEnumerable`1` type. The following code illustrates the use of `Seq.cast` to convert an `System.Collections.ArrayList` into a sequence.

```
open System

let arr = ResizeArray<int>(10)

for i in 1 .. 10 do
    arr.Add(10)

let seqCast = Seq.cast arr
```

You can define infinite sequences by using the Seq.initInfinite function. For such a sequence, you provide a function that generates each element from the index of the element. Infinite sequences are possible because of lazy evaluation; elements are created as needed by calling the function that you specify. The following code example produces an infinite sequence of floating point numbers, in this case the alternating series of reciprocals of squares of successive integers.

```
let seqInfinite =
    Seq.initInfinite (fun index ->
        let n = float (index + 1)
        1.0 / (n * n * (if ((index + 1) % 2 = 0) then 1.0 else -1.0)))

printfn "%A" seqInfinite
```

Seq.unfold generates a sequence from a computation function that takes a state and transforms it to produce each subsequent element in the sequence. The state is just a value that is used to compute each element, and can change as each element is computed. The second argument to `Seq.unfold` is the initial value that is used to start the sequence. `Seq.unfold` uses an option type for the state, which enables you to terminate the sequence by returning the `None` value. The following code shows two examples of sequences, `seq1` and `fib`, that are generated by an `unfold` operation. The first, `seq1`, is just a simple sequence with numbers up to 20. The second, `fib`, uses `unfold` to compute the Fibonacci sequence. Because each element in the Fibonacci sequence is the sum of the previous two Fibonacci numbers, the state value is a tuple that consists of the previous two numbers in the sequence. The initial value is `(1,1)`, the first two numbers in the sequence.

```
let seq1 =
    0 // Initial state
    |> Seq.unfold (fun state ->
        if (state > 20) then
            None
        else
            Some(state, state + 1))

printfn "The sequence seq1 contains numbers from 0 to 20."

for x in seq1 do
    printf "%d " x

let fib =
    (1, 1) // Initial state
    |> Seq.unfold (fun state ->
        if (snd state > 1000) then
            None
        else
            Some(fst state + snd state, (snd state, fst state + snd state)))

printfn "\nThe sequence fib contains Fibonacci numbers."
for x in fib do printf "%d " x
```

The output is as follows:

```
The sequence seq1 contains numbers from 0 to 20.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The sequence fib contains Fibonacci numbers.

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The following code is an example that uses many of the sequence module functions described here to generate and compute the values of infinite sequences. The code might take a few minutes to run.

```
// generateInfiniteSequence generates sequences of floating point
// numbers. The sequences generated are computed from the fDenominator
// function, which has the type (int -> float) and computes the
// denominator of each term in the sequence from the index of that
// term. The isAlternating parameter is true if the sequence has
// alternating signs.
let generateInfiniteSequence fDenominator isAlternating =
    if (isAlternating) then
        Seq.initInfinite (fun index ->
            1.0 /(fDenominator index) * (if (index % 2 = 0) then -1.0 else 1.0))
    else
        Seq.initInfinite (fun index -> 1.0 /(fDenominator index))

// The harmonic alternating series is like the harmonic series
// except that it has alternating signs.
let harmonicAlternatingSeries = generateInfiniteSequence (fun index -> float index) true

// This is the series of reciprocals of the odd numbers.
let oddNumberSeries = generateInfiniteSequence (fun index -> float (2 * index - 1)) true

// This is the series of recipocals of the squares.
let squaresSeries = generateInfiniteSequence (fun index -> float (index * index)) false

// This function sums a sequence, up to the specified number of terms.
let sumSeq length sequence =
    (0, 0.0)
    |>
    Seq.unfold (fun state ->
        let subtotal = snd state + Seq.item (fst state + 1) sequence
        if (fst state >= length) then
            None
        else
            Some(subtotal, (fst state + 1, subtotal)))

// This function sums an infinite sequence up to a given value
// for the difference (epsilon) between subsequent terms,
// up to a maximum number of terms, whichever is reached first.
let infiniteSum infiniteSeq epsilon maxIteration =
    infiniteSeq
    |> sumSeq maxIteration
    |> Seq.pairwise
    |> Seq.takeWhile (fun elem -> abs (snd elem - fst elem) > epsilon)
    |> List.ofSeq
    |> List.rev
    |> List.head
    |> snd

// Compute the sums for three sequences that converge, and compare
// the sums to the expected theoretical values.
let result1 = infiniteSum harmonicAlternatingSeries 0.00001 100000
printfn "Result: %f  ln2: %f" result1 (log 2.0)

let pi = Math.PI
let result2 = infiniteSum oddNumberSeries 0.00001 10000
printfn "Result: %f pi/4: %f" result2 (pi/4.0)

// Because this is not an alternating series, a much smaller epsilon
// value and more terms are needed to obtain an accurate result.
let result3 = infiniteSum squaresSeries 0.0000001 1000000
printfn "Result: %f pi*pi/6: %f" result3 (pi*pi/6.0)
```

Searching and Finding Elements

Sequences support functionality available with lists: Seq.exists, Seq.exists2, Seq.find, Seq.findIndex, Seq.pick, Seq.tryFind, and Seq.tryFindIndex. The versions of these functions that are available for sequences evaluate the sequence only up to the element that is being searched for. For examples, see Lists.

Obtaining Subsequences

Seq.filter and Seq.choose are like the corresponding functions that are available for lists, except that the filtering and choosing does not occur until the sequence elements are evaluated.

Seq.truncate creates a sequence from another sequence, but limits the sequence to a specified number of elements. Seq.take creates a new sequence that contains only a specified number of elements from the start of a sequence. If there are fewer elements in the sequence than you specify to take, `Seq.take` throws a `System.InvalidOperationException`. The difference between `Seq.take` and `Seq.truncate` is that `Seq.truncate` does not produce an error if the number of elements is fewer than the number you specify.

The following code shows the behavior of and differences between `Seq.truncate` and `Seq.take`.

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let truncatedSeq2 = Seq.truncate 20 mySeq
let takenSeq2 = Seq.take 20 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""

// Up to this point, the sequences are not evaluated.
// The following code causes the sequences to be evaluated.
truncatedSeq |> printSeq
truncatedSeq2 |> printSeq
takenSeq |> printSeq
// The following line produces a run-time error (in printSeq):
takenSeq2 |> printSeq
```

The output, before the error occurs, is as follows.

```
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
```

By using Seq.takeWhile, you can specify a predicate function (a Boolean function) and create a sequence from another sequence made up of those elements of the original sequence for which the predicate is `true`, but stop before the first element for which the predicate returns `false`. Seq.skip returns a sequence that skips a specified number of the first elements of another sequence and returns the remaining elements. Seq.skipWhile returns a sequence that skips the first elements of another sequence as long as the predicate returns `true`, and then returns the remaining elements, starting with the first element for which the predicate returns `false`.

The following code example illustrates the behavior of and differences between `Seq.takeWhile`, `Seq.skip`, and `Seq.skipWhile`.

```
// takeWhile
let mySeqLessThan10 = Seq.takeWhile (fun elem -> elem < 10) mySeq
mySeqLessThan10 |> printSeq

// skip
let mySeqSkipFirst5 = Seq.skip 5 mySeq
mySeqSkipFirst5 |> printSeq

// skipWhile
let mySeqSkipWhileLessThan10 = Seq.skipWhile (fun elem -> elem < 10) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

The output is as follows.

```
1 4 9
36 49 64 81 100
16 25 36 49 64 81 100
```

Transforming Sequences

Seq.pairwise creates a new sequence in which successive elements of the input sequence are grouped into tuples.

```
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqPairwise = Seq.pairwise (seq { for i in 1 .. 10 -> i*i })
printSeq seqPairwise

printfn ""
let seqDelta = Seq.map (fun elem -> snd elem - fst elem) seqPairwise
printSeq seqDelta
```

Seq.windowed is like `Seq.pairwise`, except that instead of producing a sequence of tuples, it produces a sequence of arrays that contain copies of adjacent elements (a *window*) from the sequence. You specify the number of adjacent elements you want in each array.

The following code example demonstrates the use of `Seq.windowed`. In this case the number of elements in the window is 3. The example uses `printSeq`, which is defined in the previous code example.

```
let seqNumbers = [ 1.0; 1.5; 2.0; 1.5; 1.0; 1.5 ] :> seq<float>
let seqWindows = Seq.windowed 3 seqNumbers
let seqMovingAverage = Seq.map Array.average seqWindows
printfn "Initial sequence: "
printSeq seqNumbers
printfn "\nWindows of length 3: "
printSeq seqWindows
printfn "\nMoving average: "
printSeq seqMovingAverage
```

The output is as follows.

Initial sequence:

```
1.0 1.5 2.0 1.5 1.0 1.5

Windows of length 3:
[|1.0; 1.5; 2.0|] [|1.5; 2.0; 1.5|] [|2.0; 1.5; 1.0|] [|1.5; 1.0; 1.5|]

Moving average:
1.5 1.666666667 1.5 1.333333333
```

Operations with Multiple Sequences

Seq.zip and Seq.zip3 take two or three sequences and produce a sequence of tuples. These functions are like the corresponding functions available for lists. There is no corresponding functionality to separate one sequence into two or more sequences. If you need this functionality for a sequence, convert the sequence to a list and use List.unzip.

Sorting, Comparing, and Grouping

The sorting functions supported for lists also work with sequences. This includes Seq.sort and Seq.sortBy. These functions iterate through the whole sequence.

You compare two sequences by using the Seq.compareWith function. The function compares successive elements in turn, and stops when it encounters the first unequal pair. Any additional elements do not contribute to the comparison.

The following code shows the use of `Seq.compareWith`.

```
let sequence1 = seq { 1 .. 10 }
let sequence2 = seq { 10 .. -1 .. 1 }

// Compare two sequences element by element.
let compareSequences =
    Seq.compareWith (fun elem1 elem2 ->
        if elem1 > elem2 then 1
        elif elem1 < elem2 then -1
        else 0)

let compareResult1 = compareSequences sequence1 sequence2
match compareResult1 with
| 1 -> printfn "Sequence1 is greater than sequence2."
| -1 -> printfn "Sequence1 is less than sequence2."
| 0 -> printfn "Sequence1 is equal to sequence2."
| _ -> failwith("Invalid comparison result.")
```

In the previous code, only the first element is computed and examined, and the result is -1.

Seq.countBy takes a function that generates a value called a *key* for each element. A key is generated for each element by calling this function on each element. `Seq.countBy` then returns a sequence that contains the key values, and a count of the number of elements that generated each value of the key.

```
let mySeq1 = seq { 1.. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let seqResult =
    mySeq1
    |> Seq.countBy (fun elem ->
        if elem % 3 = 0 then 0
        elif elem % 3 = 1 then 1
        else 2)

printSeq seqResult
```

The output is as follows.

```
(1, 34) (2, 33) (0, 33)
```

The previous output shows that there were 34 elements of the original sequence that produced the key 1, 33 values that produced the key 2, and 33 values that produced the key 0.

You can group elements of a sequence by calling Seq.groupBy. `Seq.groupBy` takes a sequence and a function that generates a key from an element. The function is executed on each element of the sequence. `Seq.groupBy` returns a sequence of tuples, where the first element of each tuple is the key and the second is a sequence of elements that produce that key.

The following code example shows the use of `Seq.groupBy` to partition the sequence of numbers from 1 to 100 into three groups that have the distinct key values 0, 1, and 2.

```
let sequence = seq { 1 .. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let sequences3 =
    sequences
    |> Seq.groupBy (fun index ->
        if (index % 3 = 0) then 0
        elif (index % 3 = 1) then 1
        else 2)

sequences3 |> printSeq
```

The output is as follows.

```
(1, seq [1; 4; 7; 10; ...]) (2, seq [2; 5; 8; 11; ...]) (0, seq [3; 6; 9; 12; ...])
```

You can create a sequence that eliminates duplicate elements by calling Seq.distinct. Or you can use Seq.distinctBy, which takes a key-generating function to be called on each element. The resulting sequence contains elements of the original sequence that have unique keys; later elements that produce a duplicate key to an earlier element are discarded.

The following code example illustrates the use of `Seq.distinct`. `Seq.distinct` is demonstrated by generating sequences that represent binary numbers, and then showing that the only distinct elements are 0 and 1.

```
let binary n =
    let rec generateBinary n =
        if (n / 2 = 0) then [n]
        else (n % 2) :: generateBinary (n / 2)

    generateBinary n
    |> List.rev
    |> Seq.ofList

printfn "%A" (binary 1024)

let resultSequence = Seq.distinct (binary 1024)
printfn "%A" resultSequence
```

The following code demonstrates `Seq.distinctBy` by starting with a sequence that contains negative and positive numbers and using the absolute value function as the key-generating function. The resulting sequence is missing all the positive numbers that correspond to the negative numbers in the sequence, because the negative numbers appear earlier in the sequence and therefore are selected instead of the positive numbers that have the same absolute value, or key.

```
let inputSequence = { -5 .. 10 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1

printfn "Original sequence: "
printSeq inputSequence

printfn "\nSequence with distinct absolute values: "
let seqDistinctAbsoluteValue = Seq.distinctBy (fun elem -> abs elem) inputSequence
printSeq seqDistinctAbsoluteValue
```

Readonly and Cached Sequences

Seq.readonly creates a read-only copy of a sequence. `Seq.readonly` is useful when you have a read-write collection, such as an array, and you do not want to modify the original collection. This function can be used to preserve data encapsulation. In the following code example, a type that contains an array is created. A property exposes the array, but instead of returning an array, it returns a sequence that is created from the array by using `Seq.readonly`.

```
type ArrayContainer(start, finish) =
    let internalArray = [| start .. finish |]
    member this.RangeSeq = Seq.readonly internalArray
    member this.RangeArray = internalArray

let newArray = new ArrayContainer(1, 10)
let rangeSeq = newArray.RangeSeq
let rangeArray = newArray.RangeArray

// These lines produce an error:
//let myArray = rangeSeq :> int array
//myArray[0] <- 0

// The following line does not produce an error.
// It does not preserve encapsulation.
rangeArray[0] <- 0
```

Seq.cache creates a stored version of a sequence. Use `Seq.cache` to avoid reevaluation of a sequence, or when you have multiple threads that use a sequence, but you must make sure that each element is acted upon only one time. When you have a sequence that is being used by multiple threads, you can have one thread that enumerates and computes the values for the original sequence, and remaining threads can use the cached sequence.

Performing Computations on Sequences

Simple arithmetic operations are like those of lists, such as Seq.average, Seq.sum, Seq.averageBy, Seq.sumBy, and so on.

Seq.fold, Seq.reduce, and Seq.scan are like the corresponding functions that are available for lists. Sequences support a subset of the full variations of these functions that lists support. For more information and examples, see Lists.

See also

- F# Language Reference
- F# Types

Reference Cells

8/5/2022 • 2 minutes to read • Edit Online

Reference cells are storage locations that enable you to create mutable values with reference semantics.

Syntax

```
ref expression
```

Remarks

You use the `ref` operator before a value to create a new reference cell that encapsulates the value. You can then change the underlying value because it is mutable.

A reference cell holds an actual value; it is not just an address. When you create a reference cell by using the `ref` operator, you create a copy of the underlying value as an encapsulated mutable value.

You can dereference a reference cell by using the `!` (bang) operator.

The following code example illustrates the declaration and use of reference cells.

```
// Declare a reference.
let refVar = ref 6

// Change the value referred to by the reference.
refVar := 50

// Dereference by using the ! operator.
printfn "%d" !refVar
```

The output is `50`.

Reference cells are instances of the `Ref` generic record type, which is declared as follows.

```
type Ref<'a> =
{ mutable contents: 'a }
```

The type `'a ref` is a synonym for `Ref<'a>`. The compiler and IntelliSense in the IDE display the former for this type, but the underlying definition is the latter.

The `ref` operator creates a new reference cell. The following code is the declaration of the `ref` operator.

```
let ref x = { contents = x }
```

The following table shows the features that are available on the reference cell.

OPERATOR, MEMBER, OR FIELD	DESCRIPTION	TYPE	DEFINITION

OPERATOR, MEMBER, OR FIELD	DESCRIPTION	TYPE	DEFINITION
`!` (dereference operator)	Returns the underlying value.	`'a ref -> 'a`	`let (!) r = r.contents`
`:=` (assignment operator)	Changes the underlying value.	`'a ref -> 'a -> unit`	`let (:=) r x = r.contents <- x`
`ref` (operator)	Encapsulates a value into a new reference cell.	`'a -> 'a ref`	`let ref x = { contents = x }`
`Value` (property)	Gets or sets the underlying value.	`unit -> 'a`	`member x.Value = x.contents`
`contents` (record field)	Gets or sets the underlying value.	`'a`	`let ref x = { contents = x }`

There are several ways to access the underlying value. The value returned by the dereference operator (`!`) is not an assignable value. Therefore, if you are modifying the underlying value, you must use the assignment operator (`:=`) instead.

Both the `Value` property and the `contents` field are assignable values. Therefore, you can use these to either access or change the underlying value, as shown in the following code.

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

The output is as follows.

```
10
10
11
12
```

The field `contents` is provided for compatibility with other versions of ML and will produce a warning during compilation. To disable the warning, use the `--mlcompatibility` compiler option. For more information, see Compiler Options.

C# programmers should know that `ref` in C# is not the same thing as `ref` in F#. The equivalent constructs in F# are byrefs, which are a different concept from reference cells.

Values marked as `mutable` may be automatically promoted to `'a ref` if captured by a closure; see Values.

See also

- F# Language Reference
- Parameters and Arguments
- Symbol and Operator Reference

- Values

Records (F#)

8/5/2022 • 7 minutes to read • Edit Online

Records represent simple aggregates of named values, optionally with members. They can either be structs or reference types. They are reference types by default.

Syntax

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
    [ member-list ]
```

Remarks

In the previous syntax, *typename* is the name of the record type, *label1* and *label2* are names of values, referred to as *labels*, and *type1* and *type2* are the types of these values. *member-list* is the optional list of members for the type. You can use the `[<Struct>]` attribute to create a struct record rather than a record which is a reference type.

Following are some examples.

```
// Labels are separated by semicolons when defined on the same line.
type Point = { X: float; Y: float; Z: float; }

// You can define labels on their own line with or without a semicolon.
type Customer =
    { First: string
      Last: string;
      SSN: uint32
      AccountNumber: uint32; }

// A struct record.
[<Struct>]
type StructPoint =
    { X: float
      Y: float
      Z: float }
```

When each label is on a separate line, the semicolon is optional.

You can set values in expressions known as *record expressions*. The compiler infers the type from the labels used (if the labels are sufficiently distinct from those of other record types). Braces ({ }) enclose the record expression. The following code shows a record expression that initializes a record with three float elements with labels `x`, `y` and `z`.

```
let mypoint = { X = 1.0; Y = 1.0; Z = -1.0; }
```

Do not use the shortened form if there could be another type that also has the same labels.

```
type Point = { X: float; Y: float; Z: float; }
type Point3D = { X: float; Y: float; Z: float }
// Ambiguity: Point or Point3D?
let mypoint3D = { X = 1.0; Y = 1.0; Z = 0.0; }
```

The labels of the most recently declared type take precedence over those of the previously declared type, so in the preceding example, `mypoint3D` is inferred to be `Point3D`. You can explicitly specify the record type, as in the following code.

```
let myPoint1 = { Point.X = 1.0; Y = 1.0; Z = 0.0; }
```

Methods can be defined for record types just as for class types.

Creating Records by Using Record Expressions

You can initialize records by using the labels that are defined in the record. An expression that does this is referred to as a *record expression*. Use braces to enclose the record expression and use the semicolon as a delimiter.

The following example shows how to create a record.

```
type MyRecord =
    { X: int
      Y: int
      Z: int }

let myRecord1 = { X = 1; Y = 2; Z = 3; }
```

The semicolons after the last field in the record expression and in the type definition are optional, regardless of whether the fields are all in one line.

When you create a record, you must supply values for each field. You cannot refer to the values of other fields in the initialization expression for any field.

In the following code, the type of `myRecord2` is inferred from the names of the fields. Optionally, you can specify the type name explicitly.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Another form of record construction can be useful when you have to copy an existing record, and possibly change some of the field values. The following line of code illustrates this.

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

This form of the record expression is called the *copy and update record expression*.

Records are immutable by default; however, you can easily create modified records by using a copy and update expression. You can also explicitly specify a mutable field.

```
type Car =
    { Make : string
      Model : string
      mutable Odometer : int }

let myCar = { Make = "Fabrikam"; Model = "Coupe"; Odometer = 108112 }
myCar.Odometer <- myCar.Odometer + 21
```

Don't use the DefaultValue attribute with record fields. A better approach is to define default instances of records with fields that are initialized to default values and then use a copy and update record expression to set any fields that differ from the default values.

```
// Rather than use [<DefaultValue>], define a default record.
type MyRecord =
    { Field1 : int
      Field2 : int }

let defaultRecord1 = { Field1 = 0; Field2 = 0 }
let defaultRecord2 = { Field1 = 1; Field2 = 25 }

// Use the with keyword to populate only a few chosen fields
// and leave the rest with default values.
let rr3 = { defaultRecord1 with Field2 = 42 }
```

Creating Mutually Recursive Records

Sometime when creating a record, you may want to have it depend on another type that you would like to define afterwards. This is a compile error unless you define the record types to be mutually recursive.

Defining mutually recursive records is done with the `and` keyword. This lets you link 2 or more record types together.

For example, the following code defines a `Person` and `Address` type as mutually recursive:

```
// Create a Person type and use the Address type that is not defined
type Person =
  { Name: string
    Age: int
    Address: Address }
// Define the Address type which is used in the Person record
and Address =
  { Line1: string
    Line2: string
    PostCode: string
    Occupant: Person }
```

To create instances of both, you do the following:

```
// Create a Person type and use the Address type that is not defined
let rec person =
  {
      Name = "Person name"
      Age = 12
      Address =
          {
              Line1 = "line 1"
              Line2 = "line 2"
              PostCode = "abc123"
              Occupant = person
          }
  }
```

If you were to define the previous example without the `and` keyword, then it would not compile. The `and` keyword is required for mutually recursive definitions.

Pattern Matching with Records

Records can be used with pattern matching. You can specify some fields explicitly and provide variables for other fields that will be assigned when a match occurs. The following code example illustrates this.

```
type Point3D = { X: float; Y: float; Z: float }
let evaluatePoint (point: Point3D) =
    match point with
    | { X = 0.0; Y = 0.0; Z = 0.0 } -> printfn "Point is at the origin."
    | { X = xVal; Y = 0.0; Z = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
    | { X = 0.0; Y = yVal; Z = 0.0 } -> printfn "Point is on the y-axis. Value is %f." yVal
    | { X = 0.0; Y = 0.0; Z = zVal } -> printfn "Point is on the z-axis. Value is %f." zVal
    | { X = xVal; Y = yVal; Z = zVal } -> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }
```

The output of this code is as follows.

```
Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).
```

Records and members

You can specify members on records much like you can with classes. There is no support for fields. A common approach is to define a `Default` static member for easy record construction:

```
type Person =
  { Name: string
    Age: int
    Address: string }

    static member Default =
        { Name = "Phillip"
          Age = 12
          Address = "123 happy fun street" }

let defaultPerson = Person.Default
```

If you use a self identifier, that identifier refers to the instance of the record whose member is called:

```
type Person =
  { Name: string
    Age: int
    Address: string }

    member this.WeirdToString() =
        this.Name + this.Address + string this.Age

let p = { Name = "a"; Age = 12; Address = "abc123" }
let weirdString = p.WeirdToString()
```

Differences Between Records and Classes

Record fields differ from class fields in that they are automatically exposed as properties, and they are used in the creation and copying of records. Record construction also differs from class construction. In a record type, you cannot define a constructor. Instead, the construction syntax described in this topic applies. Classes have no direct relationship between constructor parameters, fields, and properties.

Like union and structure types, records have structural equality semantics. Classes have reference equality semantics. The following code example demonstrates this.

```
type RecordTest = { X: int; Y: int }

let record1 = { X = 1; Y = 2 }
let record2 = { X = 1; Y = 2 }

if (record1 = record2) then
    printfn "The records are equal."
else
    printfn "The records are unequal."
```

The output of this code is as follows:

```
The records are equal.
```

If you write the same code with classes, the two class objects would be unequal because the two values would represent two objects on the heap and only the addresses would be compared (unless the class type overrides the `System.Object.Equals` method).

If you need reference equality for records, add the attribute `[<ReferenceEquality>]` above the record.

See also

- F# Types
- Classes
- F# Language Reference
- Reference-Equality
- Pattern Matching

Copy and Update Record Expressions

8/5/2022 • 2 minutes to read • Edit Online

A *copy and update record expression* is an expression that copies an existing record, updates specified fields, and returns the updated record.

Syntax

```
{ record-name with
    updated-labels }

{| anonymous-record-name with
    updated-labels |}
```

Remarks

Records and anonymous records are immutable by default, so it is not possible to update an existing record. To create an updated record all the fields of a record would have to be specified again. To simplify this task a *copy and update expression* can be used. This expression takes an existing record, creates a new one of the same type by using specified fields from the expression and the missing field specified by the expression.

This can be useful when you have to copy an existing record, and possibly change some of the field values.

Take for instance a newly created record.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

To update only two fields in that record you can use the *copy and update record expression*:

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

See also

- Records
- Anonymous Records
- F# Language Reference

Anonymous Records

8/5/2022 • 8 minutes to read • Edit Online

Anonymous records are simple aggregates of named values that don't need to be declared before use. You can declare them as either structs or reference types. They're reference types by default.

Syntax

The following examples demonstrate the anonymous record syntax. Items delimited as `[item]` are optional.

```
// Construct an anonymous record
let value-name = [struct] {| Label1: Type1; Label2: Type2; ...|}

// Use an anonymous record as a type parameter
let value-name = Type-Name<[struct] {| Label1: Type1; Label2: Type2; ...|}>

// Define a parameter with an anonymous record as input
let function-name (arg-name: [struct] {| Label1: Type1; Label2: Type2; ...|}) ...
```

Basic usage

Anonymous records are best thought of as F# record types that don't need to be declared before instantiation.

For example, here how you can interact with a function that produces an anonymous record:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

The following example expands on the previous one with a `printCircleStats` function that takes an anonymous record as input:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let printCircleStats r (stats: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Calling `printCircleStats` with any anonymous record type that doesn't have the same "shape" as the input type will fail to compile:

```
printCircleStats r {| Diameter = 2.0; Area = 4.0; MyCircumference = 12.566371 |}
// Two anonymous record types have mismatched sets of field names
// '["Area"; "Circumference"; "Diameter"]' and '["Area"; "Diameter"; "MyCircumference"]'
```

Struct anonymous records

Anonymous records can also be defined as struct with the optional `struct` keyword. The following example augments the previous one by producing and consuming a struct anonymous record:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    // Note that the keyword comes before the '{| |}' brace pair
    struct {| Area = a; Circumference = c; Diameter = d |}

// the 'struct' keyword also comes before the '{| |}' brace pair when declaring the parameter type
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Structness inference

Struct anonymous records also allow for "structness inference" where you do not need to specify the `struct` keyword at the call site. In this example, you elide the `struct` keyword when calling `printCircleStats`:

```
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

printCircleStats r {| Area = 4.0; Circumference = 12.6; Diameter = 12.6 |}
```

The reverse pattern - specifying `struct` when the input type is not a struct anonymous record - will fail to compile.

Embedding anonymous records within other types

It's useful to declare discriminated unions whose cases are records. But if the data in the records is the same type as the discriminated union, you must define all types as mutually recursive. Using anonymous records avoids this restriction. What follows is an example type and function that pattern matches over it:

```
type FullName = { FirstName: string; LastName: string }

// Note that using a named record for Manager and Executive would require mutually recursive definitions.
type Employee =
    | Engineer of FullName
    | Manager of {| Name: FullName; Reports: Employee list |}
    | Executive of {| Name: FullName; Reports: Employee list; Assistant: Employee |}

let getFirstName e =
    match e with
    | Engineer fullName -> fullName.FirstName
    | Manager m -> m.Name.FirstName
    | Executive ex -> ex.Name.FirstName
```

Copy and update expressions

Anonymous records support construction with copy and update expressions. For example, here's how you can construct a new instance of an anonymous record that copies an existing one's data:

```
let data = {| X = 1; Y = 2 |}
let data' = {| data with Y = 3 |}
```

However, unlike named records, anonymous records allow you to construct entirely different forms with copy and update expressions. The follow example takes the same anonymous record from the previous example and expands it into a new anonymous record:

```
let data = {| X = 1; Y = 2 |}
let expandedData = {| data with Z = 3 |} // Gives {| X=1; Y=2; Z=3 |}
```

It is also possible to construct anonymous records from instances of named records:

```
type R = { X: int }
let data = { X = 1 }
let data' = {| data with Y = 2 |} // Gives {| X=1; Y=2 |}
```

You can also copy data to and from reference and struct anonymous records:

```
// Copy data from a reference record into a struct anonymous record
type R1 = { X: int }
let r1 = { X = 1 }

let data1 = struct {| r1 with Y = 1 |}

// Copy data from a struct record into a reference anonymous record
[<Struct>]
type R2 = { X: int }
let r2 = { X = 1 }

let data2 = {| r1 with Y = 1 |}

// Copy the reference anonymous record data into a struct anonymous record
let data3 = struct {| data2 with Z = r2.X |}
```

Properties of anonymous records

Anonymous records have a number of characteristics that are essential to fully understanding how they can be used.

Anonymous records are nominal

Anonymous records are nominal types. They are best thought as named record types (which are also nominal) that do not require an up-front declaration.

Consider the following example with two anonymous record declarations:

```
let x = {| X = 1 |}
let y = {| Y = 1 |}
```

The `x` and `y` values have different types and are not compatible with one another. They are not equatable and they are not comparable. To illustrate this, consider a named record equivalent:

```
type X = { X: int }
type Y = { Y: int }

let x = { X = 1 }
let y = { Y = 1 }
```

There isn't anything inherently different about anonymous records when compared with their named record equivalents when concerning type equivalency or comparison.

Anonymous records use structural equality and comparison

Like record types, anonymous records are structurally equatable and comparable. This is only true if all constituent types support equality and comparison, like with record types. To support equality or comparison, two anonymous records must have the same "shape".

```
{| a = 1+1 |} = {| a = 2 |} // true
{| a = 1+1 |} > {| a = 1 |} // true

// error FS0001: Two anonymous record types have mismatched sets of field names '["a"]' and '["a"; "b"]'
{| a = 1 + 1 |} = {| a = 2;  b = 1|}
```

Anonymous records are serializable

You can serialize anonymous records just as you can with named records. Here is an example using Newtonsoft.Json:

```
open Newtonsoft.Json

let phillip' = {| name="Phillip"; age=28 |}
let philStr = JsonConvert.SerializeObject(phillip')

let phillip = JsonConvert.DeserializeObject<{|name: string; age: int|}>(philStr)
printfn $"Name: {phillip.name} Age: %d{phillip.age}"
```

Anonymous records are useful for sending lightweight data over a network without the need to define a domain for your serialized/deserialized types up front.

Anonymous records interoperate with C# anonymous types

It is possible to use a .NET API that requires the use of C# anonymous types. C# anonymous types are trivial to interoperate with by using anonymous records. The following example shows how to use anonymous records to call a LINQ overload that requires an anonymous type:

```
open System.Linq

let names = [ "Ana"; "Felipe"; "Emilia"]
let nameGrouping = names.Select(fun n -> {| Name = n; FirstLetter = n[0] |})
for ng in nameGrouping do
    printfn $"{ng.Name} has first letter {ng.FirstLetter}"
```

There are a multitude of other APIs used throughout .NET that require the use of passing in an anonymous type. Anonymous records are your tool for working with them.

Limitations

Anonymous records have some restrictions in their usage. Some are inherent to their design, but others are amenable to change.

Limitations with pattern matching

Anonymous records do not support pattern matching, unlike named records. There are three reasons:

1. A pattern would have to account for every field of an anonymous record, unlike named record types. This is because anonymous records do not support structural subtyping – they are nominal types.
2. Because of (1), there is no ability to have additional patterns in a pattern match expression, as each distinct pattern would imply a different anonymous record type.
3. Because of (2), any anonymous record pattern would be more verbose than the use of “dot” notation.

There is an open language suggestion to allow pattern matching in limited contexts.

Limitations with mutability

It is not currently possible to define an anonymous record with `mutable` data. There is an open language suggestion to allow mutable data.

Limitations with struct anonymous records

It is not possible to declare struct anonymous records as `IsByRefLike` or `IsReadOnly`. There is an open language suggestion to for `IsByRefLike` and `IsReadOnly` anonymous records.

Discriminated Unions

8/5/2022 • 9 minutes to read • Edit Online

Discriminated unions provide support for values that can be one of a number of named cases, possibly each with different values and types. Discriminated unions are useful for heterogeneous data; data that can have special cases, including valid and error cases; data that varies in type from one instance to another; and as an alternative for small object hierarchies. In addition, recursive discriminated unions are used to represent tree data structures.

Syntax

```
[ attributes ]
type [accessibility-modifier] type-name =
    | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]
    | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]

    [ member-list ]
```

Remarks

Discriminated unions are similar to union types in other languages, but there are differences. As with a union type in C++ or a variant type in Visual Basic, the data stored in the value is not fixed; it can be one of several distinct options. Unlike unions in these other languages, however, each of the possible options is given a *case identifier*. The case identifiers are names for the various possible types of values that objects of this type could be; the values are optional. If values are not present, the case is equivalent to an enumeration case. If values are present, each value can either be a single value of a specified type, or a tuple that aggregates multiple fields of the same or different types. You can give an individual field a name, but the name is optional, even if other fields in the same case are named.

Accessibility for discriminated unions defaults to `public`.

For example, consider the following declaration of a Shape type.

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

The preceding code declares a discriminated union Shape, which can have values of any of three cases: Rectangle, Circle, and Prism. Each case has a different set of fields. The Rectangle case has two named fields, both of type `float`, that have the names width and length. The Circle case has just one named field, radius. The Prism case has three fields, two of which (width and height) are named fields. Unnamed fields are referred to as anonymous fields.

You construct objects by providing values for the named and anonymous fields according to the following examples.

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

This code shows that you can either use the named fields in the initialization, or you can rely on the ordering of the fields in the declaration and just provide the values for each field in turn. The constructor call for `rect` in the previous code uses the named fields, but the constructor call for `circ` uses the ordering. You can mix the ordered fields and named fields, as in the construction of `prism`.

The `option` type is a simple discriminated union in the F# core library. The `option` type is declared as follows.

```
// The option type is a discriminated union.
type Option<'a> =
    | Some of 'a
    | None
```

The previous code specifies that the type `Option` is a discriminated union that has two cases, `Some` and `None`. The `Some` case has an associated value that consists of one anonymous field whose type is represented by the type parameter `'a`. The `None` case has no associated value. Thus the `option` type specifies a generic type that either has a value of some type or no value. The type `Option` also has a lowercase type alias, `option`, that is more commonly used.

The case identifiers can be used as constructors for the discriminated union type. For example, the following code is used to create values of the `option` type.

```
let myOption1 = Some(10.0)
let myOption2 = Some("string")
let myOption3 = None
```

The case identifiers are also used in pattern matching expressions. In a pattern matching expression, identifiers are provided for the values associated with the individual cases. For example, in the following code, `x` is the identifier given the value that is associated with the `Some` case of the `option` type.

```
let printValue opt =
    match opt with
    | Some x -> printfn "%A" x
    | None -> printfn "No value."
```

In pattern matching expressions, you can use named fields to specify discriminated union matches. For the Shape type that was declared previously, you can use the named fields as the following code shows to extract the values of the fields.

```
let getShapeWidth shape =
    match shape with
    | Rectangle(width = w) -> w
    | Circle(radius = r) -> 2. * r
    | Prism(width = w) -> w
```

Normally, the case identifiers can be used without qualifying them with the name of the union. If you want the name to always be qualified with the name of the union, you can apply the RequireQualifiedAccess attribute to the union type definition.

Unwrapping Discriminated Unions

In F# Discriminated Unions are often used in domain-modeling for wrapping a single type. It's easy to extract the underlying value via pattern matching as well. You don't need to use a match expression for a single case:

```
let ([UnionCaseIdentifier] [values]) = [UnionValue]
```

The following example demonstrates this:

```
type ShaderProgram = | ShaderProgram of id:int

let someFunctionUsingShaderProgram shaderProgram =
    let (ShaderProgram id) = shaderProgram
    // Use the unwrapped value
    ...
```

Pattern matching is also allowed directly in function parameters, so you can unwrap a single case there:

```
let someFunctionUsingShaderProgram (ShaderProgram id) =
    // Use the unwrapped value
    ...
```

Struct Discriminated Unions

You can also represent Discriminated Unions as structs. This is done with the `[<Struct>]` attribute.

```
[<Struct>]
type SingleCase = Case of string

[<Struct>]
type Multicase =
    | Case1 of Case1 : string
    | Case2 of Case2 : int
    | Case3 of Case3 : double
```

Because these are value types and not reference types, there are extra considerations compared with reference discriminated unions:

1. They are copied as value types and have value type semantics.
2. You cannot use a recursive type definition with a multicase struct Discriminated Union.
3. You must provide unique case names for a multicase struct Discriminated Union.

Using Discriminated Unions Instead of Object Hierarchies

You can often use a discriminated union as a simpler alternative to a small object hierarchy. For example, the following discriminated union could be used instead of a `Shape` base class that has derived types for circle, square, and so on.

```
type Shape =
  // The value here is the radius.
| Circle of float
  // The value here is the side length.
| EquilateralTriangle of double
  // The value here is the side length.
| Square of double
  // The values here are the height and width.
| Rectangle of double * double
```

Instead of a virtual method to compute an area or perimeter, as you would use in an object-oriented implementation, you can use pattern matching to branch to appropriate formulas to compute these quantities. In the following example, different formulas are used to compute the area, depending on the shape.

```
let pi = 3.141592654

let area myShape =
    match myShape with
    | Circle radius -> pi * radius * radius
    | EquilateralTriangle s -> (sqrt 3.0) / 4.0 * s * s
    | Square s -> s * s
    | Rectangle (h, w) -> h * w

let radius = 15.0
let myCircle = Circle(radius)
printfn "Area of circle that has radius %f: %f" radius (area myCircle)

let squareSide = 10.0
let mySquare = Square(squareSide)
printfn "Area of square that has side %f: %f" squareSide (area mySquare)

let height, width = 5.0, 10.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle that has height %f and width %f is %f" height width (area myRectangle)
```

The output is as follows:

```
Area of circle that has radius 15.000000: 706.858347
Area of square that has side 10.000000: 100.000000
Area of rectangle that has height 5.000000 and width 10.000000 is 50.000000
```

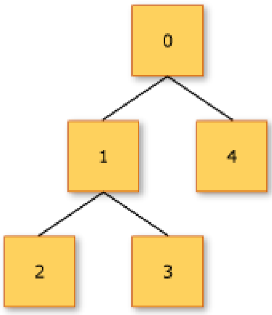
Using Discriminated Unions for Tree Data Structures

Discriminated unions can be recursive, meaning that the union itself can be included in the type of one or more cases. Recursive discriminated unions can be used to create tree structures, which are used to model expressions in programming languages. In the following code, a recursive discriminated union is used to create a binary tree data structure. The union consists of two cases, `Node`, which is a node with an integer value and left and right subtrees, and `Tip`, which terminates the tree.

```
type Tree =
    | Tip
    | Node of int * Tree * Tree

let rec sumTree tree =
    match tree with
    | Tip -> 0
    | Node(value, left, right) ->
        value + sumTree(left) + sumTree(right)
let myTree = Node(0, Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip)), Node(4, Tip, Tip))
let resultSumTree = sumTree myTree
```

In the previous code, `resultSumTree` has the value 10. The following illustration shows the tree structure for `myTree`.



Discriminated unions work well if the nodes in the tree are heterogeneous. In the following code, the type `Expression` represents the abstract syntax tree of an expression in a simple programming language that supports addition and multiplication of numbers and variables. Some of the union cases are not recursive and represent either numbers (`Number`) or variables (`Variable`). Other cases are recursive, and represent operations (`Add` and `Multiply`), where the operands are also expressions. The `Evaluate` function uses a match expression to recursively process the syntax tree.

```
type Expression =
    | Number of int
    | Add of Expression * Expression
    | Multiply of Expression * Expression
    | Variable of string

let rec Evaluate (env:Map<string,int>) exp =
    match exp with
    | Number n -> n
    | Add (x, y) -> Evaluate env x + Evaluate env y
    | Multiply (x, y) -> Evaluate env x * Evaluate env y
    | Variable id -> env[id]

let environment = Map [ "a", 1; "b", 2; "c", 3 ]

// Create an expression tree that represents
// the expression: a + 2 * b.
let expressionTree1 = Add(Variable "a", Multiply(Number 2, Variable "b"))

// Evaluate the expression a + 2 * b, given the
// table of values for the variables.
let result = Evaluate environment expressionTree1
```

When this code is executed, the value of `result` is 5.

Members

It is possible to define members on discriminated unions. The following example shows how to define a property and implement an interface:

```fsharp
open System

type IPrintable =
    abstract Print: unit -> unit

type Shape =
    | Circle of float
    | EquilateralTriangle of float
    | Square of float
    | Rectangle of float * float

    member this.Area =
        match this with
        | Circle r -> Math.PI * (r ** 2.0)
        | EquilateralTriangle s -> s * s * sqrt 3.0 / 4.0
        | Square s -> s * s
        | Rectangle(l, w) -> l * w

    interface IPrintable with
        member this.Print () =
            match this with
            | Circle r -> printfn $"Circle with radius %f{r}"
            | EquilateralTriangle s -> printfn $"Equilateral Triangle of side %f{s}"
            | Square s -> printfn $"Square with side %f{s}"
            | Rectangle(l, w) -> printfn $"Rectangle with length %f{l} and width %f{w}"
```

Common attributes

The following attributes are commonly seen in discriminated unions:

- `[<RequireQualifiedAccess>]`
- `[<NoEquality>]`
- `[<NoComparison>]`
- `[<Struct>]`

See also

- F# Language Reference

Classes (F#)

8/5/2022 • 8 minutes to read • Edit Online

Classes are types that represent objects that can have properties, methods, and events.

Syntax

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Remarks

Classes represent the fundamental description of .NET object types; the class is the primary type concept that supports object-oriented programming in F#.

In the preceding syntax, the `type-name` is any valid identifier. The `type-params` describes optional generic type parameters. It consists of type parameter names and constraints enclosed in angle brackets (`<` and `>`). For more information, see Generics and Constraints. The `parameter-list` describes constructor parameters. The first access modifier pertains to the type; the second pertains to the primary constructor. In both cases, the default is `public`.

You specify the base class for a class by using the `inherit` keyword. You must supply arguments, in parentheses, for the base class constructor.

You declare fields or function values that are local to the class by using `let` bindings, and you must follow the general rules for `let` bindings. The `do-bindings` section includes code to be executed upon object construction.

The `member-list` consists of additional constructors, instance and static method declarations, interface declarations, abstract bindings, and property and event declarations. These are described in Members.

The `identifier` that is used with the optional `as` keyword gives a name to the instance variable, or self identifier, which can be used in the type definition to refer to the instance of the type. For more information, see the section Self Identifiers later in this topic.

The keywords `class` and `end` that mark the start and end of the definition are optional.

Mutually recursive types, which are types that reference each other, are joined together with the `and` keyword just as mutually recursive functions are. For an example, see the section Mutually Recursive Types.

Constructors

The constructor is code that creates an instance of the class type. Constructors for classes work somewhat differently in F# than they do in other .NET languages. In an F# class, there is always a primary constructor whose arguments are described in the `parameter-list` that follows the type name, and whose body consists of the `let` (and `let rec`) bindings at the start of the class declaration and the `do` bindings that follow. The arguments of the primary constructor are in scope throughout the class declaration.

You can add additional constructors by using the `new` keyword to add a member, as follows:

`new`(`argument-list`) = `constructor-body`

The body of the new constructor must invoke the primary constructor that is specified at the top of the class declaration.

The following example illustrates this concept. In the following code, `MyClass` has two constructors, a primary constructor that takes two arguments and another constructor that takes no arguments.

```
type MyClass1(x: int, y: int) =
   do printfn "%d %d" x y
   new() = MyClass1(0, 0)
```

let and do Bindings

The `let` and `do` bindings in a class definition form the body of the primary class constructor, and therefore they run whenever a class instance is created. If a `let` binding is a function, then it is compiled into a member. If the `let` binding is a value that is not used in any function or member, then it is compiled into a variable that is local to the constructor. Otherwise, it is compiled into a field of the class. The `do` expressions that follow are compiled into the primary constructor and execute initialization code for every instance. Because any additional constructors always call the primary constructor, the `let` bindings and `do` bindings always execute regardless of which constructor is called.

Fields that are created by `let` bindings can be accessed throughout the methods and properties of the class; however, they cannot be accessed from static methods, even if the static methods take an instance variable as a parameter. They cannot be accessed by using the self identifier, if one exists.

Self Identifiers

A *self identifier* is a name that represents the current instance. Self identifiers resemble the `this` keyword in C# or C++ or `Me` in Visual Basic. You can define a self identifier in two different ways, depending on whether you want the self identifier to be in scope for the whole class definition or just for an individual method.

To define a self identifier for the whole class, use the `as` keyword after the closing parentheses of the constructor parameter list, and specify the identifier name.

To define a self identifier for just one method, provide the self identifier in the member declaration, just before the method name and a period (.) as a separator.

The following code example illustrates the two ways to create a self identifier. In the first line, the `as` keyword is used to define the self identifier. In the fifth line, the identifier `this` is used to define a self identifier whose scope is restricted to the method `PrintMessage`.

```
type MyClass2(dataIn) as self =
    let data = dataIn
    do
        self.PrintMessage()
    member this.PrintMessage() =
        printf "Creating MyClass2 with Data %d" data
```

Unlike in other .NET languages, you can name the self identifier however you want; you are not restricted to names such as `self`, `Me`, or `this`.

The self identifier that is declared with the `as` keyword is not initialized until after the base constructor. Therefore, when used before or inside the base constructor, `System.InvalidOperationException: The initialization of an object or value resulted in an object or value being accessed recursively before it was fully initialized.` will be raised during runtime. You can use the self identifier freely after the base constructor, such as in `let` bindings or `do` bindings.

Generic Type Parameters

Generic type parameters are specified in angle brackets (`<` and `>`), in the form of a single quotation mark followed by an identifier. Multiple generic type parameters are separated by commas. The generic type parameter is in scope throughout the declaration. The following code example shows how to specify generic type parameters.

```
type MyGenericClass<'a> (x: 'a) =
   do printfn "%A" x
```

Type arguments are inferred when the type is used. In the following code, the inferred type is a sequence of tuples.

```
let g1 = MyGenericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

Specifying Inheritance

The `inherit` clause identifies the direct base class, if there is one. In F#, only one direct base class is allowed. Interfaces that a class implements are not considered base classes. Interfaces are discussed in the Interfaces topic.

You can access the methods and properties of the base class from the derived class by using the language keyword `base` as an identifier, followed by a period (.) and the name of the member.

For more information, see Inheritance.

Members Section

You can define static or instance methods, properties, interface implementations, abstract members, event declarations, and additional constructors in this section. Let and do bindings cannot appear in this section. Because members can be added to a variety of F# types in addition to classes, they are discussed in a separate topic, Members.

Mutually Recursive Types

When you define types that reference each other in a circular way, you string together the type definitions by using the `and` keyword. The `and` keyword replaces the `type` keyword on all except the first definition, as follows.

```
open System.IO

type Folder(pathIn: string) =
  let path = pathIn
  let filenameArray : string array = Directory.GetFiles(path)
  member this.FileArray = Array.map (fun elem -> new File(elem, this)) filenameArray

and File(filename: string, containingFolder: Folder) =
   member this.Name = filename
   member this.ContainingFolder = containingFolder

let folder1 = new Folder(".")
for file in folder1.FileArray do
   printfn "%s" file.Name
```

The output is a list of all the files in the current directory.

When to Use Classes, Unions, Records, and Structures

Given the variety of types to choose from, you need to have a good understanding of what each type is designed for to select the appropriate type for a particular situation. Classes are designed for use in object-oriented programming contexts. Object-oriented programming is the dominant paradigm used in applications that are written for the .NET Framework. If your F# code has to work closely with the .NET Framework or another object-oriented library, and especially if you have to extend from an object-oriented type system such as a UI library, classes are probably appropriate.

If you are not interoperating closely with object-oriented code, or if you are writing code that is self-contained and therefore protected from frequent interaction with object-oriented code, you should consider using a mix of classes, records and discriminated unions. A single, well thought–out discriminated union, together with appropriate pattern matching code, can often be used as a simpler alternative to an object hierarchy. For more information about discriminated unions, see Discriminated Unions.

Records have the advantage of being simpler than classes, but records are not appropriate when the demands of a type exceed what can be accomplished with their simplicity. Records are basically simple aggregates of values, without separate constructors that can perform custom actions, without hidden fields, and without inheritance or interface implementations. Although members such as properties and methods can be added to records to make their behavior more complex, the fields stored in a record are still a simple aggregate of values. For more information about records, see Records.

Structures are also useful for small aggregates of data, but they differ from classes and records in that they are .NET value types. Classes and records are .NET reference types. The semantics of value types and reference types are different in that value types are passed by value. This means that they are copied bit for bit when they are passed as a parameter or returned from a function. They are also stored on the stack or, if they are used as a field, embedded inside the parent object instead of stored in their own separate location on the heap. Therefore, structures are appropriate for frequently accessed data when the overhead of accessing the heap is a problem. For more information about structures, see Structs.

See also

- F# Language Reference
- Members
- Inheritance
- Interfaces

Interfaces (F#)

8/5/2022 • 4 minutes to read • Edit Online

Interfaces specify sets of related members that other classes implement.

Syntax

```
// Interface declaration:
[ attributes ]
type [accessibility-modifier] interface-name =
    [ interface ]     [ inherit base-interface-name ...]
    abstract member1 : [ argument-types1 -> ] return-type1
    abstract member2 : [ argument-types2 -> ] return-type2
    ...
[ end ]

// Implementing, inside a class type definition:
interface interface-name with
    member self-identifier.member1argument-list = method-body1
    member self-identifier.member2argument-list = method-body2

// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
    { new interface-name with
        member self-identifier.member1argument-list = method-body1
        member self-identifier.member2argument-list = method-body2
        [ base-interface-definitions ]
    }
    member-list
```

Remarks

Interface declarations resemble class declarations except that no members are implemented. Instead, all the members are abstract, as indicated by the keyword `abstract`. You do not provide a method body for abstract methods. F# cannot define a default method implementation on an interface, but it is compatible with default implementations defined by C#. Default implementations using the `default` keyword are only supported when inheriting from a non-interface base class.

The default accessibility for interfaces is `public`.

You can optionally give each method parameter a name using normal F# syntax:

```
type ISprintable =
    abstract member Print : format:string -> unit
```

In the above `ISprintable` example, the `Print` method has a single parameter of the type `string` with the name `format`.

There are two ways to implement interfaces: by using object expressions, and by using class types. In either case, the class type or object expression provides method bodies for abstract methods of the interface. Implementations are specific to each type that implements the interface. Therefore, interface methods on different types might be different from each other.

The keywords `interface` and `end`, which mark the start and end of the definition, are optional when you use lightweight syntax. If you do not use these keywords, the compiler attempts to infer whether the type is a class or an interface by analyzing the constructs that you use. If you define a member or use other class syntax, the type is interpreted as a class.

The .NET coding style is to begin all interfaces with a capital `I`.

You can specify multiple parameters in two ways: F#-style and .NET-style. Both will compile the same way for .NET consumers, but F#-style will force F# callers to use F#-style parameter application and .NET-style will force F# callers to use tupled argument application.

```
type INumericFSharp =
    abstract Add: x: int -> y: int -> int

type INumericDotNet =
    abstract Add: x: int * y: int -> int
```

Implementing Interfaces by Using Class Types

You can implement one or more interfaces in a class type by using the `interface` keyword, the name of the interface, and the `with` keyword, followed by the interface member definitions, as shown in the following code.

```
type IPrintable =
   abstract member Print : unit -> unit

type SomeClass1(x: int, y: float) =
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y
```

Interface implementations are inherited, so any derived classes do not need to reimplement them.

Calling Interface Methods

Interface methods can be called only through the interface, not through any object of the type that implements the interface. Thus, you might have to upcast to the interface type by using the `:>` operator or the `upcast` operator in order to call these methods.

To call the interface method when you have an object of type `SomeClass`, you must upcast the object to the interface type, as shown in the following code.

```
let x1 = new SomeClass1(1, 2.0)
(x1 :> IPrintable).Print()
```

An alternative is to declare a method on the object that upcasts and calls the interface method, as in the following example.

```
type SomeClass2(x: int, y: float) =
   member this.Print() = (this :> IPrintable).Print()
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y

let x2 = new SomeClass2(1, 2.0)
x2.Print()
```

Implementing Interfaces by Using Object Expressions

Object expressions provide a short way to implement an interface. They are useful when you do not have to create a named type, and you just want an object that supports the interface methods, without any additional methods. An object expression is illustrated in the following code.

```
let makePrintable(x: int, y: float) =
    { new IPrintable with
              member this.Print() = printfn "%d %f" x y }
let x3 = makePrintable(1, 2.0)
x3.Print()
```

Interface Inheritance

Interfaces can inherit from one or more base interfaces.

```
type Interface1 =
    abstract member Method1 : int -> int

type Interface2 =
    abstract member Method2 : int -> int

type Interface3 =
    inherit Interface1
    inherit Interface2
    abstract member Method3 : int -> int

type MyClass() =
    interface Interface3 with
        member this.Method1(n) = 2 * n
        member this.Method2(n) = n + 100
        member this.Method3(n) = n / 10
```

Implementing interfaces with default implementations

C# supports defining interfaces with default implementations, like so:

```
using System;

namespace CSharp
{
    public interface MyDim
    {
        public int Z => 0;
    }
}
```

These are directly consumable from F#:

```
open CSharp

// You can implement the interface via a class
type MyType() =
    member _.M() = ()

    interface MyDim

let md = MyType() :> MyDim
printfn $"DIM from C#: %d{md.Z}"

// You can also implement it via an object expression
let md' = { new MyDim }
printfn $"DIM from C# but via Object Expression: %d{md'.Z}"
```

You can override a default implementation with `override`, like overriding any virtual member.

Any members in an interface that do not have a default implementation must still be explicitly implemented.

Implementing the same interface at different generic instantiations

F# supports implementing the same interface at different generic instantiations like so:

```
type IA<'T> =
    abstract member Get : unit -> 'T

type MyClass() =
    interface IA<int> with
        member x.Get() = 1
    interface IA<string> with
        member x.Get() = "hello"

let mc = MyClass()
let iaInt = mc :> IA<int>
let iaString = mc :> IA<string>

iaInt.Get() // 1
iaString.Get() // "hello"
```

See also

- F# Language Reference
- Object Expressions
- Classes

Members

8/5/2022 • 2 minutes to read • Edit Online

This section describes members of F# object types.

Remarks

Members are features that are part of a type definition and are declared with the `member` keyword. F# object types such as records, classes, discriminated unions, interfaces, and structures support members. For more information, see Records, Classes, Discriminated Unions, Interfaces, and Structs.

Members typically make up the public interface for a type, which is why they are public unless otherwise specified. Members can also be declared private or internal. For more information, see Access Control. Signatures files can also be used to expose or not expose certain members of a type. For more information, see Signatures.

Private fields and `do` bindings, which are used only with classes, are not true members, because they are never part of the public interface of a type and are not declared with the `member` keyword, but they are described in this section also.

Related Topics

TOPIC	DESCRIPTION
`let` Bindings in Classes	Describes the definition of private fields and functions in classes.
`do` Bindings in Classes	Describes the specification of object initialization code.
Properties	Describes property members in classes and other types.
Indexed Properties	Describes array-like properties in classes and other types.
Methods	Describes functions that are members of a type.
Constructors	Describes special functions that initialize objects of a type.
Operator Overloading	Describes the definition of customized operators for types.
Events	Describes the definition of events and event handling support in F#.
Structs	Describes the definition of structs in F#.
Explicit Fields	Describes the definition of uninitialized fields in a type.

Constructors

8/5/2022 • 7 minutes to read • Edit Online

This article describes how to define and use constructors to create and initialize class and structure objects.

Construction of class objects

Objects of class types have constructors. There are two kinds of constructors. One is the primary constructor, whose parameters appear in parentheses just after the type name. You specify other, optional additional constructors by using the `new` keyword. Any such additional constructors must call the primary constructor.

The primary constructor contains `let` and `do` bindings that appear at the start of the class definition. A `let` binding declares private fields and methods of the class; a `do` binding executes code. For more information about `let` bindings in class constructors, see `let` Bindings in Classes. For more information about `do` bindings in constructors, see `do` Bindings in Classes.

Regardless of whether the constructor you want to call is a primary constructor or an additional constructor, you can create objects by using a `new` expression, with or without the optional `new` keyword. You initialize your objects together with constructor arguments, either by listing the arguments in order and separated by commas and enclosed in parentheses, or by using named arguments and values in parentheses. You can also set properties on an object during the construction of the object by using the property names and assigning values just as you use named constructor arguments.

The following code illustrates a class that has a constructor and various ways of creating objects:

```
// This class has a primary constructor that takes three arguments
// and an additional constructor that calls the primary constructor.
type MyClass(x0, y0, z0) =
    let mutable x = x0
    let mutable y = y0
    let mutable z = z0
    do
        printfn "Initialized object that has coordinates (%d, %d, %d)" x y z
    member this.X with get() = x and set(value) = x <- value
    member this.Y with get() = y and set(value) = y <- value
    member this.Z with get() = z and set(value) = z <- value
    new() = MyClass(0, 0, 0)

// Create by using the new keyword.
let myObject1 = new MyClass(1, 2, 3)
// Create without using the new keyword.
let myObject2 = MyClass(4, 5, 6)
// Create by using named arguments.
let myObject3 = MyClass(x0 = 7, y0 = 8, z0 = 9)
// Create by using the additional constructor.
let myObject4 = MyClass()
```

The output is as follows:

```
Initialized object that has coordinates (1, 2, 3)
Initialized object that has coordinates (4, 5, 6)
Initialized object that has coordinates (7, 8, 9)
Initialized object that has coordinates (0, 0, 0)
```

Construction of structures

Structures follow all the rules of classes. Therefore, you can have a primary constructor, and you can provide additional constructors by using `new`. However, there is one important difference between structures and classes: structures can have a parameterless constructor (that is, one with no arguments) even if no primary constructor is defined. The parameterless constructor initializes all the fields to the default value for that type, usually zero or its equivalent. Any constructors that you define for structures must have at least one argument so that they do not conflict with the parameterless constructor.

Also, structures often have fields that are created by using the `val` keyword; classes can also have these fields. Structures and classes that have fields defined by using the `val` keyword can also be initialized in additional constructors by using record expressions, as shown in the following code.

```
type MyStruct =
    struct
       val X : int
       val Y : int
       val Z : int
       new(x, y, z) = { X = x; Y = y; Z = z }
    end

let myStructure1 = new MyStruct(1, 2, 3)
```

For more information, see Explicit Fields: The `val` Keyword.

Executing side effects in constructors

A primary constructor in a class can execute code in a `do` binding. However, what if you have to execute code in an additional constructor, without a `do` binding? To do this, you use the `then` keyword.

```
 // Executing side effects in the primary constructor and
// additional constructors.
type Person(nameIn : string, idIn : int) =
    let mutable name = nameIn
    let mutable id = idIn
    do printfn "Created a person object."
    member this.Name with get() = name and set(v) = name <- v
    member this.ID with get() = id and set(v) = id <- v
    new() =
        Person("Invalid Name", -1)
        then
            printfn "Created an invalid person object."

let person1 = new Person("Humberto Acevedo", 123458734)
let person2 = new Person()
```

The side effects of the primary constructor still execute. Therefore, the output is as follows:

```
Created a person object.
Created a person object.
Created an invalid person object.
```

The reason why `then` is required instead of another `do` is that the `do` keyword has its standard meaning of delimiting a `unit`-returning expression when present in the body of an additional constructor. It only has special meaning in the context of primary constructors.

Self identifiers in constructors

In other members, you provide a name for the current object in the definition of each member. You can also put the self identifier on the first line of the class definition by using the `as` keyword immediately following the constructor parameters. The following example illustrates this syntax.

```
type MyClass1(x) as this =
    // This use of the self identifier produces a warning - avoid.
    let x1 = this.X
    // This use of the self identifier is acceptable.
    do printfn "Initializing object with X =%d" this.X
    member this.X = x
```

In additional constructors, you can also define a self identifier by putting the `as` clause right after the constructor parameters. The following example illustrates this syntax:

```
type MyClass2(x : int) =
    member this.X = x
    new() as this = MyClass2(0) then printfn "Initializing with X = %d" this.X
```

Problems can occur when you try to use an object before it is fully defined. Therefore, uses of the self identifier can cause the compiler to emit a warning and insert additional checks to ensure the members of an object are not accessed before the object is initialized. You should only use the self identifier in the `do` bindings of the primary constructor, or after the `then` keyword in additional constructors.

The name of the self identifier does not have to be `this`. It can be any valid identifier.

Assigning values to properties at initialization

You can assign values to the properties of a class object in the initialization code by appending a list of assignments of the form `property = value` to the argument list for a constructor. This is shown in the following code example:

```
 type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
       with get() = number
       and set(value) = number <- value
    member this.FirstName
       with get() = firstName
       and set(value) = firstName <- value
    member this.LastName
       with get() = lastName
       and set(value) = lastName <- value
    member this.Balance
       with get() = balance
       and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

The following version of the previous code illustrates the combination of ordinary arguments, optional arguments, and property settings in one constructor call:

```
type Account(accountNumber : int, ?first: string, ?last: string, ?bal : float) =
   let mutable balance = defaultArg bal 0.0
   let mutable number = accountNumber
   let mutable firstName = defaultArg first ""
   let mutable lastName = defaultArg last ""
   member this.AccountNumber
      with get() = number
      and set(value) = number <- value
   member this.FirstName
      with get() = firstName
      and set(value) = firstName <- value
   member this.LastName
      with get() = lastName
      and set(value) = lastName <- value
   member this.Balance
      with get() = balance
      and set(value) = balance <- value
   member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
   member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(8782108, bal = 543.33,
                          FirstName="Raman", LastName="Iyer")
```

Constructors in inherited class

When inheriting from a base class that has a constructor, you must specify its arguments in the inherit clause. For more information, see Constructors and inheritance.

Static constructors or type constructors

In addition to specifying code for creating objects, static `let` and `do` bindings can be authored in class types that execute before the type is first used to perform initialization at the type level. For more information, see `let` Bindings in Classes and `do` Bindings in Classes.

See also

- Members

let Bindings in Classes

8/5/2022 • 2 minutes to read • Edit Online

You can define private fields and private functions for F# classes by using `let` bindings in the class definition.

Syntax

```
// Field.
[static] let [ mutable ] binding1 [ and ... binding-n ]

// Function.
[static] let [ rec ] binding1 [ and ... binding-n ]
```

Remarks

The previous syntax appears after the class heading and inheritance declarations but before any member definitions. The syntax is like that of `let` bindings outside of classes, but the names defined in a class have a scope that is limited to the class. A `let` binding creates a private field or function; to expose data or functions publicly, declare a property or a member method.

A `let` binding that is not static is called an instance `let` binding. Instance `let` bindings execute when objects are created. Static `let` bindings are part of the static initializer for the class, which is guaranteed to execute before the type is first used.

The code within instance `let` bindings can use the primary constructor's parameters.

Attributes and accessibility modifiers are not permitted on `let` bindings in classes.

The following code examples illustrate several types of `let` bindings in classes.

```
type PointWithCounter(a: int, b: int) =
    // A variable i.
    let mutable i = 0

    // A let binding that uses a pattern.
    let (x, y) = (a, b)

    // A private function binding.
    let privateFunction x y = x * x + 2*y

    // A static let binding.
    static let mutable count = 0

    // A do binding.
    do
       count <- count + 1

    member this.Prop1 = x
    member this.Prop2 = y
    member this.CreatedCount = count
    member this.FunctionValue = privateFunction x y

let point1 = PointWithCounter(10, 52)

printfn "%d %d %d %d" (point1.Prop1) (point1.Prop2) (point1.CreatedCount) (point1.FunctionValue)
```

The output is as follows.

```
10 52 1 204
```

Alternative Ways to Create Fields

You can also use the `val` keyword to create a private field. When using the `val` keyword, the field is not given a value when the object is created, but instead is initialized with a default value. For more information, see Explicit Fields: The val Keyword.

You can also define private fields in a class by using a member definition and adding the keyword `private` to the definition. This can be useful if you expect to change the accessibility of a member without rewriting your code. For more information, see Access Control.

See also

- Members
- `do` Bindings in Classes
- `let` Bindings

do Bindings in Classes

8/5/2022 • 2 minutes to read • Edit Online

A `do` binding in a class definition performs actions when the object is constructed or, for a static `do` binding, when the type is first used.

Syntax

```
[static] do expression
```

Remarks

A `do` binding appears together with or after `let` bindings but before member definitions in a class definition. Although the `do` keyword is optional for `do` bindings at the module level, it is not optional for `do` bindings in a class definition.

For the construction of every object of any given type, non-static `do` bindings and non-static `let` bindings are executed in the order in which they appear in the class definition. Multiple `do` bindings can occur in one type. The non-static `let` bindings and the non-static `do` bindings become the body of the primary constructor. The code in the non-static `do` bindings section can reference the primary constructor parameters and any values or functions that are defined in the `let` bindings section.

Non-static `do` bindings can access members of the class as long as the class has a self identifier that is defined by an `as` keyword in the class heading, and as long as all uses of those members are qualified with the self identifier for the class.

Because `let` bindings initialize the private fields of a class, which is often necessary to guarantee that members behave as expected, `do` bindings are usually put after `let` bindings so that code in the `do` binding can execute with a fully initialized object. If your code attempts to use a member before the initialization is complete, an InvalidOperationException is raised.

Static `do` bindings can reference static members or fields of the enclosing class but not instance members or fields. Static `do` bindings become part of the static initializer for the class, which is guaranteed to execute before the class is first used.

Attributes are ignored for `do` bindings in types. If an attribute is required for code that executes in a `do` binding, it must be applied to the primary constructor.

In the following code, a class has a static `do` binding and a non-static `do` binding. The object has a constructor that has two parameters, `a` and `b`, and two private fields are defined in the `let` bindings for the class. Two properties are also defined. All of these are in scope in the non-static `do` bindings section, as is illustrated by the line that prints all those values.

```
open System

type MyType(a:int, b:int) as this =
    inherit Object()
    let x = 2*a
    let y = 2*b
    do printfn "Initializing object %d %d %d %d %d %d"
               a b x y (this.Prop1) (this.Prop2)
    static do printfn "Initializing MyType."
    member this.Prop1 = 4*x
    member this.Prop2 = 4*y
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)

let obj1 = new MyType(1, 2)
```

The output is as follows.

```
Initializing MyType.
Initializing object 1 2 2 4 8 16
```

See also

- Members
- Classes
- Constructors
- `let` Bindings in Classes
- `do` Bindings

Properties (F#)

8/5/2022 • 6 minutes to read • Edit Online

Properties are members that represent values associated with an object.

Syntax

```
// Property that has both get and set defined.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with [accessibility-modifier] get() =
    get-function-body
and [accessibility-modifier] set parameter =
    set-function-body

// Alternative syntax for a property that has get and set.
[ attributes-for-get ]
[ static ] member [accessibility-modifier-for-get] [self-identifier.]PropertyName =
    get-function-body
[ attributes-for-set ]
[ static ] member [accessibility-modifier-for-set] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName =
    get-function-body

// Alternative syntax for property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with get() =
    get-function-body

// Property that has set only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Automatically implemented properties.
[ attributes ]
[ static ] member val [accessibility-modifier] PropertyName = initialization-expression [ with get, set ]
```

Remarks

Properties represent the "has a" relationship in object-oriented programming, representing data that is associated with object instances or, for static properties, with the type.

You can declare properties in two ways, depending on whether you want to explicitly specify the underlying value (also called the backing store) for the property, or if you want to allow the compiler to automatically generate the backing store for you. Generally, you should use the more explicit way if the property has a non-trivial implementation and the automatic way when the property is just a simple wrapper for a value or variable. To declare a property explicitly, use the `member` keyword. This declarative syntax is followed by the syntax that specifies the `get` and `set` methods, also named *accessors*. The various forms of the explicit syntax shown in the syntax section are used for read/write, read-only, and write-only properties. For read-only properties, you define only a `get` method; for write-only properties, define only a `set` method. Note that when a property has both `get` and `set` accessors, the alternative syntax enables you to specify attributes and accessibility modifiers that are different for each accessor, as is shown in the following code.

```
// A read-only property.
member this.MyReadOnlyProperty = myInternalValue
// A write-only property.
member this.MyWriteOnlyProperty with set (value) = myInternalValue <- value
// A read-write property.
member this.MyReadWriteProperty
    with get () = myInternalValue
    and set (value) = myInternalValue <- value
```

For read/write properties, which have both a `get` and `set` method, the order of `get` and `set` can be reversed. Alternatively, you can provide the syntax shown for `get` only and the syntax shown for `set` only instead of using the combined syntax. Doing this makes it easier to comment out the individual `get` or `set` method, if that is something you might need to do. This alternative to using the combined syntax is shown in the following code.

```
member this.MyReadWriteProperty with get () = myInternalValue
member this.MyReadWriteProperty with set (value) = myInternalValue <- value
```

Private values that hold the data for properties are called *backing stores*. To have the compiler create the backing store automatically, use the keywords `member val`, omit the self-identifier, then provide an expression to initialize the property. If the property is to be mutable, include `with get, set`. For example, the following class type includes two automatically implemented properties. `Property1` is read-only and is initialized to the argument provided to the primary constructor, and `Property2` is a settable property initialized to an empty string:

```
type MyClass(property1 : int) =
member val Property1 = property1
member val Property2 = "" with get, set
```

Automatically implemented properties are part of the initialization of a type, so they must be included before any other member definitions, just like `let` bindings and `do` bindings in a type definition. Note that the expression that initializes an automatically implemented property is only evaluated upon initialization, and not every time the property is accessed. This behavior is in contrast to the behavior of an explicitly implemented property. What this effectively means is that the code to initialize these properties is added to the constructor of a class. Consider the following code that shows this difference:

```
type MyClass() =
    let random  = new System.Random()
    member val AutoProperty = random.Next() with get, set
    member this.ExplicitProperty = random.Next()

let class1 = new MyClass()

printfn $"class1.AutoProperty = %d{class1.AutoProperty}"
printfn $"class1.ExplicitProperty = %d{class1.ExplicitProperty}"
```

Output

```
class1.AutoProperty = 1853799794
class1.AutoProperty = 1853799794
class1.ExplicitProperty = 978922705
class1.ExplicitProperty = 1131210765
```

The output of the preceding code shows that the value of AutoProperty is unchanged when called repeatedly, whereas the ExplicitProperty changes each time it is called. This demonstrates that the expression for an automatically implemented property is not evaluated each time, as is the getter method for the explicit property.

> **WARNING**
> There are some libraries, such as the Entity Framework (`System.Data.Entity`) that perform custom operations in base class constructors that don't work well with the initialization of automatically implemented properties. In those cases, try using explicit properties.

Properties can be members of classes, structures, discriminated unions, records, interfaces, and type extensions and can also be defined in object expressions.

Attributes can be applied to properties. To apply an attribute to a property, write the attribute on a separate line before the property. For more information, see Attributes.

By default, properties are public. Accessibility modifiers can also be applied to properties. To apply an accessibility modifier, add it immediately before the name of the property if it is meant to apply to both the `get` and `set` methods; add it before the `get` and `set` keywords if different accessibility is required for each accessor. The *accessibility-modifier* can be one of the following: `public`, `private`, `internal`. For more information, see Access Control.

Property implementations are executed each time a property is accessed.

Static and Instance Properties

Properties can be static or instance properties. Static properties can be invoked without an instance and are used for values associated with the type, not with individual objects. For static properties, omit the self-identifier. The self-identifier is required for instance properties.

The following static property definition is based on a scenario in which you have a static field `myStaticValue` that is the backing store for the property.

```
static member MyStaticProperty
    with get() = myStaticValue
    and set(value) = myStaticValue <- value
```

Properties can also be array-like, in which case they are called *indexed properties*. For more information, see Indexed Properties.

Type Annotation for Properties

In many cases, the compiler has enough information to infer the type of a property from the type of the backing store, but you can set the type explicitly by adding a type annotation.

```
// To apply a type annotation to a property that does not have an explicit
// get or set, apply the type annotation directly to the property.
member this.MyProperty1 : int = myInternalValue
// If there is a get or set, apply the type annotation to the get or set method.
member this.MyProperty2 with get() : int = myInternalValue
```

Using Property set Accessors

You can set properties that provide `set` accessors by using the `<-` operator.

```
// Assume that the constructor argument sets the initial value of the
// internal backing store.
let mutable myObject = new MyType(10)
myObject.MyProperty <- 20
printfn "%d" (myObject.MyProperty)
```

The output is `20`.

Abstract Properties

Properties can be abstract. As with methods, `abstract` just means that there is a virtual dispatch associated with the property. Abstract properties can be truly abstract, that is, without a definition in the same class. The class that contains such a property is therefore an abstract class. Alternatively, abstract can just mean that a property is virtual, and in that case, a definition must be present in the same class. Note that abstract properties must not be private, and if one accessor is abstract, the other must also be abstract. For more information about abstract classes, see Abstract Classes.

```
// Abstract property in abstract class.
// The property is an int type that has a get and
// set method
[<AbstractClass>]
type AbstractBase() =
   abstract Property1 : int with get, set

// Implementation of the abstract property
type Derived1() =
   inherit AbstractBase()
   let mutable value = 10
   override this.Property1 with get() = value and set(v : int) = value <- v

// A type with a "virtual" property.
 type Base1() =
   let mutable value = 10
   abstract Property1 : int with get, set
   default this.Property1 with get() = value and set(v : int) = value <- v

// A derived type that overrides the virtual property
type Derived2() =
   inherit Base1()
   let mutable value2 = 11
   override this.Property1 with get() = value2 and set(v) = value2 <- v
```

See also

- Members
- Methods

Methods

8/5/2022 • 7 minutes to read • Edit Online

A *method* is a function that is associated with a type. In object-oriented programming, methods are used to expose and implement the functionality and behavior of objects and types.

Syntax

```
// Instance method definition.
[ attributes ]
member [inline] self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Static method definition.
[ attributes ]
static member [inline] method-name parameter-list [ : return-type ] =
    method-body

// Abstract method declaration or virtual dispatch slot.
[ attributes ]
abstract member method-name : type-signature

// Virtual method declaration and default implementation.
[ attributes ]
abstract member method-name : type-signature
[ attributes ]
default self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Override of inherited virtual method.
[ attributes ]
override self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Optional and DefaultParameterValue attributes on input parameters
[ attributes ]
[ modifier ] member [inline] self-identifier.method-name ([<Optional; DefaultParameterValue( default-value
)>] input) [ : return-type ]
```

Remarks

In the previous syntax, you can see the various forms of method declarations and definitions. In longer method bodies, a line break follows the equal sign (=), and the whole method body is indented.

Attributes can be applied to any method declaration. They precede the syntax for a method definition and are usually listed on a separate line. For more information, see Attributes.

Methods can be marked `inline`. For information about `inline`, see Inline Functions.

Non-inline methods can be used recursively within the type; there is no need to explicitly use the `rec` keyword.

Instance Methods

Instance methods are declared with the `member` keyword and a *self-identifier*, followed by a period (.) and the method name and parameters. As is the case for `let` bindings, the *parameter-list* can be a pattern. Typically, you enclose method parameters in parentheses in a tuple form, which is the way methods appear in F# when they are created in other .NET Framework languages. However, the curried form (parameters separated by spaces) is also common, and other patterns are supported also.

The following example illustrates the definition and use of a non-abstract instance method.

```
type SomeType(factor0: int) =
   let factor = factor0
   member this.SomeMethod(a, b, c) =
      (a + b + c) * factor

   member this.SomeOtherMethod(a, b, c) =
      this.SomeMethod(a, b, c) * factor
```

Within instance methods, do not use the self identifier to access fields defined by using let bindings. Use the self identifier when accessing other members and properties.

Static Methods

The keyword `static` is used to specify that a method can be called without an instance and is not associated with an object instance. Otherwise, methods are instance methods.

The example in the next section shows fields declared with the `let` keyword, property members declared with the `member` keyword, and a static method declared with the `static` keyword.

The following example illustrates the definition and use of static methods. Assume that these method definitions are in the `SomeType` class in the previous section.

```
static member SomeStaticMethod(a, b, c) =
   (a + b + c)

static member SomeOtherStaticMethod(a, b, c) =
   SomeType.SomeStaticMethod(a, b, c) * 100
```

Abstract and Virtual Methods

The keyword `abstract` indicates that a method has a virtual dispatch slot and might not have a definition in the class. A *virtual dispatch slot* is an entry in an internally maintained table of functions that is used at run time to look up virtual function calls in an object-oriented type. The virtual dispatch mechanism is the mechanism that implements *polymorphism*, an important feature of object-oriented programming. A class that has at least one abstract method without a definition is an *abstract class*, which means that no instances can be created of that class. For more information about abstract classes, see Abstract Classes.

Abstract method declarations do not include a method body. Instead, the name of the method is followed by a colon (:) and a type signature for the method. The type signature of a method is the same as that shown by IntelliSense when you pause the mouse pointer over a method name in the Visual Studio Code Editor, except without parameter names. Type signatures are also displayed by the interpreter, fsi.exe, when you are working interactively. The type signature of a method is formed by listing out the types of the parameters, followed by the return type, with appropriate separator symbols. Curried parameters are separated by `->` and tuple parameters are separated by `*`. The return value is always separated from the arguments by a `->` symbol. Parentheses can be used to group complex parameters, such as when a function type is a parameter, or to indicate when a tuple is treated as a single parameter rather than as two parameters.

You can also give abstract methods default definitions by adding the definition to the class and using the `default` keyword, as shown in the syntax block in this topic. An abstract method that has a definition in the same class is equivalent to a virtual method in other .NET Framework languages. Whether or not a definition exists, the `abstract` keyword creates a new dispatch slot in the virtual function table for the class.

Regardless of whether a base class implements its abstract methods, derived classes can provide implementations of abstract methods. To implement an abstract method in a derived class, define a method that has the same name and signature in the derived class, except use the `override` or `default` keyword, and provide the method body. The keywords `override` and `default` mean exactly the same thing. Use `override` if the new method overrides a base class implementation; use `default` when you create an implementation in the same class as the original abstract declaration. Do not use the `abstract` keyword on the method that implements the method that was declared abstract in the base class.

The following example illustrates an abstract method `Rotate` that has a default implementation, the equivalent of a .NET Framework virtual method.

```
type Ellipse(a0 : float, b0 : float, theta0 : float) =
    let mutable axis1 = a0
    let mutable axis2 = b0
    let mutable rotAngle = theta0
    abstract member Rotate: float -> unit
    default this.Rotate(delta : float) = rotAngle <- rotAngle + delta
```

The following example illustrates a derived class that overrides a base class method. In this case, the override changes the behavior so that the method does nothing.

```
type Circle(radius : float) =
    inherit Ellipse(radius, radius, 0.0)
     // Circles are invariant to rotation, so do nothing.
    override this.Rotate(_) = ()
```

Overloaded Methods

Overloaded methods are methods that have identical names in a given type but that have different arguments. In F#, optional arguments are usually used instead of overloaded methods. However, overloaded methods are permitted in the language, provided that the arguments are in tuple form, not curried form.

Optional Arguments

Starting with F# 4.1, you can also have optional arguments with a default parameter value in methods. This is to help facilitate interoperation with C# code. The following example demonstrates the syntax:

```
// A class with a method M, which takes in an optional integer argument.
type C() =
    _.M([<Optional; DefaultParameterValue(12)>] i) = i + 1
```

Note that the value passed in for `DefaultParameterValue` must match the input type. In the above sample, it is an `int`. Attempting to pass a non-integer value into `DefaultParameterValue` would result in a compile error.

Example: Properties and Methods

The following example contains a type that has examples of fields, private functions, properties, and a static method.

```
type RectangleXY(x1 : float, y1: float, x2: float, y2: float) =
    // Field definitions.
    let height = y2 - y1
    let width = x2 - x1
    let area = height * width
    // Private functions.
    static let maxFloat (x: float) (y: float) =
      if x >= y then x else y
    static let minFloat (x: float) (y: float) =
      if x <= y then x else y
    // Properties.
    // Here, "this" is used as the self identifier,
    // but it can be any identifier.
    member this.X1 = x1
    member this.Y1 = y1
    member this.X2 = x2
    member this.Y2 = y2
    // A static method.
    static member intersection(rect1 : RectangleXY, rect2 : RectangleXY) =
       let x1 = maxFloat rect1.X1 rect2.X1
       let y1 = maxFloat rect1.Y1 rect2.Y1
       let x2 = minFloat rect1.X2 rect2.X2
       let y2 = minFloat rect1.Y2 rect2.Y2
       let result : RectangleXY option =
         if ( x2 > x1 && y2 > y1) then
           Some (RectangleXY(x1, y1, x2, y2))
         else
           None
       result

// Test code.
let testIntersection =
    let r1 = RectangleXY(10.0, 10.0, 20.0, 20.0)
    let r2 = RectangleXY(15.0, 15.0, 25.0, 25.0)
    let r3 : RectangleXY option = RectangleXY.intersection(r1, r2)
    match r3 with
    | Some(r3) -> printfn "Intersection rectangle: %f %f %f %f" r3.X1 r3.Y1 r3.X2 r3.Y2
    | None -> printfn "No intersection found."

testIntersection
```

See also

- Members

Parameters and Arguments

8/5/2022 • 10 minutes to read • Edit Online

This topic describes language support for defining parameters and passing arguments to functions, methods, and properties. It includes information about how to pass by reference, and how to define and use methods that can take a variable number of arguments.

Parameters and Arguments

The term *parameter* is used to describe the names for values that are expected to be supplied. The term *argument* is used for the values provided for each parameter.

Parameters can be specified in tuple or curried form, or in some combination of the two. You can pass arguments by using an explicit parameter name. Parameters of methods can be specified as optional and given a default value.

Parameter Patterns

Parameters supplied to functions and methods are, in general, patterns separated by spaces. This means that, in principle, any of the patterns described in Match Expressions can be used in a parameter list for a function or member.

Methods usually use the tuple form of passing arguments. This achieves a clearer result from the perspective of other .NET languages because the tuple form matches the way arguments are passed in .NET methods.

The curried form is most often used with functions created by using `let` bindings.

The following pseudocode shows examples of tuple and curried arguments.

```
// Tuple form.
member this.SomeMethod(param1, param2) = ...
// Curried form.
let function1 param1 param2 = ...
```

Combined forms are possible when some arguments are in tuples and some are not.

```
let function2 param1 (param2a, param2b) param3 = ...
```

Other patterns can also be used in parameter lists, but if the parameter pattern does not match all possible inputs, there might be an incomplete match at run time. The exception `MatchFailureException` is generated when the value of an argument does not match the patterns specified in the parameter list. The compiler issues a warning when a parameter pattern allows for incomplete matches. At least one other pattern is commonly useful for parameter lists, and that is the wildcard pattern. You use the wildcard pattern in a parameter list when you simply want to ignore any arguments that are supplied. The following code illustrates the use of the wildcard pattern in an argument list.

```
let makeList _ = [ for i in 1 .. 100 -> i * i ]
// The arguments 100 and 200 are ignored.
let list1 = makeList 100
let list2 = makeList 200
```

The wildcard pattern can be useful whenever you do not need the arguments passed in, such as in the main entry point to a program, when you are not interested in the command-line arguments that are normally supplied as a string array, as in the following code.

```
[<EntryPoint>]
let main _ =
    printfn "Entry point!"
    0
```

Other patterns that are sometimes used in arguments are the `as` pattern, and identifier patterns associated with discriminated unions and active patterns. You can use the single-case discriminated union pattern as follows.

```
type Slice = Slice of int * int * string

let GetSubstring1 (Slice(p0, p1, text)) =
    printfn "Data begins at %d and ends at %d in string %s" p0 p1 text
    text[p0..p1]

let substring = GetSubstring1 (Slice(0, 4, "Et tu, Brute?"))
printfn "Substring: %s" substring
```

The output is as follows.

```
Data begins at 0 and ends at 4 in string Et tu, Brute?
Et tu
```

Active patterns can be useful as parameters, for example, when transforming an argument into a desired format, as in the following example:

```
type Point = { x : float; y : float }

let (| Polar |) { x = x; y = y} =
    ( sqrt (x*x + y*y), System.Math.Atan (y/ x) )

let radius (Polar(r, _)) = r
let angle (Polar(_, theta)) = theta
```

You can use the `as` pattern to store a matched value as a local value, as is shown in the following line of code.

```
let GetSubstring2 (Slice(p0, p1, text) as s) = s
```

Another pattern that is used occasionally is a function that leaves the last argument unnamed by providing, as the body of the function, a lambda expression that immediately performs a pattern match on the implicit argument. An example of this is the following line of code.

```
let isNil = function [] -> true | _::_ -> false
```

This code defines a function that takes a generic list and returns `true` if the list is empty, and `false` otherwise. The use of such techniques can make code more difficult to read.

Occasionally, patterns that involve incomplete matches are useful, for example, if you know that the lists in your program have only three elements, you might use a pattern like the following in a parameter list.

```
let sum [a; b; c;] = a + b + c
```

The use of patterns that have incomplete matches is best reserved for quick prototyping and other temporary uses. The compiler will issue a warning for such code. Such patterns cannot cover the general case of all possible inputs and therefore are not suitable for component APIs.

Named Arguments

Arguments for methods can be specified by position in a comma-separated argument list, or they can be passed to a method explicitly by providing the name, followed by an equal sign and the value to be passed in. If specified by providing the name, they can appear in a different order from that used in the declaration.

Named arguments can make code more readable and more adaptable to certain types of changes in the API, such as a reordering of method parameters.

Named arguments are allowed only for methods, not for `let`-bound functions, function values, or lambda expressions.

The following code example demonstrates the use of named arguments.

```
type SpeedingTicket() =
    member this.GetMPHOver(speed: int, limit: int) = speed - limit

let CalculateFine (ticket : SpeedingTicket) =
    let delta = ticket.GetMPHOver(limit = 55, speed = 70)
    if delta < 20 then 50.0 else 100.0

let ticket1 : SpeedingTicket = SpeedingTicket()
printfn "%f" (CalculateFine ticket1)
```

In a call to a class constructor, you can set the values of properties of the class by using a syntax similar to that of named arguments. The following example shows this syntax.

```
 type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
       with get() = number
       and set(value) = number <- value
    member this.FirstName
       with get() = firstName
       and set(value) = firstName <- value
    member this.LastName
       with get() = lastName
       and set(value) = lastName <- value
    member this.Balance
       with get() = balance
       and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

For more information, see Constructors (F#).

Optional Parameters

You can specify an optional parameter for a method by using a question mark in front of the parameter name. Optional parameters are interpreted as the F# option type, so you can query them in the regular way that option types are queried, by using a `match` expression with `Some` and `None`. Optional parameters are permitted only on members, not on functions created by using `let` bindings.

You can pass existing optional values to method by parameter name, such as `?arg=None` or `?arg=Some(3)` or `?arg=arg`. This can be useful when building a method that passes optional arguments to another method.

You can also use a function `defaultArg`, which sets a default value of an optional argument. The `defaultArg` function takes the optional parameter as the first argument and the default value as the second.

The following example illustrates the use of optional parameters.

```
type DuplexType =
    | Full
    | Half

type Connection(?rate0 : int, ?duplex0 : DuplexType, ?parity0 : bool) =
    let duplex = defaultArg duplex0 Full
    let parity = defaultArg parity0 false
    let mutable rate = match rate0 with
                        | Some rate1 -> rate1
                        | None -> match duplex with
                                  | Full -> 9600
                                  | Half -> 4800
    do printfn "Baud Rate: %d Duplex: %A Parity: %b" rate duplex parity

let conn1 = Connection(duplex0 = Full)
let conn2 = Connection(duplex0 = Half)
let conn3 = Connection(300, Half, true)
let conn4 = Connection(?duplex0 = None)
let conn5 = Connection(?duplex0 = Some(Full))

let optionalDuplexValue : option<DuplexType> = Some(Half)
let conn6 = Connection(?duplex0 = optionalDuplexValue)
```

The output is as follows.

```
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
Baud Rate: 300 Duplex: Half Parity: true
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
```

For the purposes of C# and Visual Basic interop you can use the attributes `[<Optional; DefaultParameterValue<(...)>]` in F#, so that callers will see an argument as optional. This is equivalent to defining the argument as optional in C# as in `MyMethod(int i = 3)`.

```
open System
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue("Hello world")>] message) =
        printfn $"{message}"
```

You can also specify a new object as a default parameter value. For example, the `Foo` member could have an optional `CancellationToken` as input instead:

```
open System.Threading
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue(CancellationToken())>] ct: CancellationToken) =
        printfn $"{ct}"
```

The value given as argument to `DefaultParameterValue` must match the type of the parameter. For example, the following is not allowed:

```
type C =
    static member Wrong([<Optional; DefaultParameterValue("string")>] i:int) = ()
```

In this case, the compiler generates a warning and will ignore both attributes altogether. Note that the default value `null` needs to be type-annotated, as otherwise the compiler infers the wrong type, i.e. `[<Optional; DefaultParameterValue(null:obj)>] o:obj`.

Passing by Reference

Passing an F# value by reference involves byrefs, which are managed pointer types. Guidance for which type to use is as follows:

- Use `inref<'T>` if you only need to read the pointer.
- Use `outref<'T>` if you only need to write to the pointer.
- Use `byref<'T>` if you need to both read from and write to the pointer.

```
let example1 (x: inref<int>) = printfn $"It's %d{x}"

let example2 (x: outref<int>) = x <- x + 1

let example3 (x: byref<int>) =
    printfn $"It's %d{x}"
    x <- x + 1

let test () =
    // No need to make it mutable, since it's read-only
    let x = 1
    example1 &x

    // Needs to be mutable, since we write to it
    let mutable y = 2
    example2 &y
    example3 &y // Now 'y' is 3
```

Because the parameter is a pointer and the value is mutable, any changes to the value are retained after the execution of the function.

You can use a tuple as a return value to store any `out` parameters in .NET library methods. Alternatively, you can treat the `out` parameter as a `byref` parameter. The following code example illustrates both ways.

```
// TryParse has a second parameter that is an out parameter
// of type System.DateTime.
let (b, dt) = System.DateTime.TryParse("12-20-04 12:21:00")

printfn "%b %A" b dt

// The same call, using an address of operator.
let mutable dt2 = System.DateTime.Now
let b2 = System.DateTime.TryParse("12-20-04 12:21:00", &dt2)

printfn "%b %A" b2 dt2
```

Parameter Arrays

Occasionally it is necessary to define a function that takes an arbitrary number of parameters of heterogeneous type. It would not be practical to create all the possible overloaded methods to account for all the types that could be used. The .NET implementations provide support for such methods through the parameter array feature. A method that takes a parameter array in its signature can be provided with an arbitrary number of parameters. The parameters are put into an array. The type of the array elements determines the parameter types that can be passed to the function. If you define the parameter array with `System.Object` as the element type, then client code can pass values of any type.

In F#, parameter arrays can only be defined in methods. They cannot be used in standalone functions or functions that are defined in modules.

You define a parameter array by using the `ParamArray` attribute. The `ParamArray` attribute can only be applied to the last parameter.

The following code illustrates both calling a .NET method that takes a parameter array and the definition of a type in F# that has a method that takes a parameter array.

```
open System

type X() =
    member this.F([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

[<EntryPoint>]
let main _ =
    // call a .NET method that takes a parameter array, passing values of various types
    Console.WriteLine("a {0} {1} {2} {3} {4}", 1, 10.0, "Hello world", 1u, true)

    let xobj = new X()
    // call an F# method that takes a parameter array, passing values of various types
    xobj.F("a", 1, 10.0, "Hello world", 1u, true)
    0
```

When run in a project, the output of the previous code is as follows:

```
a 1 10 Hello world 1 True
"a"
1
10.0
"Hello world"
1u
true
```

See also

- Members

Indexed Properties

8/5/2022 • 3 minutes to read • Edit Online

When defining a class that abstracts over ordered data, it can sometimes be helpful to provide indexed access to that data without exposing the underlying implementation. This is done with the `Item` member.

Syntax

Syntax for expressions:

```
// Looking up an indexed property
expr[idx]

/// Assign to an indexed property
expr[idx] <- elementExpr
```

Syntax for member declarations:

```
// Indexed property that can be read and written to
member self-identifier.Item
    with get(index-values) =
        get-member-body
    and set index-values values-to-set =
        set-member-body

// Indexed property can only be read
member self-identifier.Item
    with get(index-values) =
        get-member-body

// Indexed property that can only be set
member self-identifier.Item
    with set index-values values-to-set =
        set-member-body
```

Remarks

The forms of the previous syntax show how to define indexed properties that have both a `get` and a `set` method, have a `get` method only, or have a `set` method only. You can also combine both the syntax shown for get only and the syntax shown for set only, and produce a property that has both get and set. This latter form allows you to put different accessibility modifiers and attributes on the get and set methods.

By using the name `Item`, the compiler treats the property as a default indexed property. A *default indexed property* is a property that you can access by using array-like syntax on the object instance. For example, if `o` is an object of the type that defines this property, the syntax `o[index]` is used to access the property.

The syntax for accessing a non-default indexed property is to provide the name of the property and the index in parentheses, just like a regular member. For example, if the property on `o` is called `Ordinal`, you write `o.Ordinal(index)` to access it.

Regardless of which form you use, you should always use the curried form for the set method on an indexed property. For information about curried functions, see Functions.

Prior to F# 6, the syntax `expr.[idx]` was used for indexing. You can activate an optional informational warning (`/warnon:3566` or property `<WarnOn>3566</WarnOn>`) to report uses of the `expr.[idx]` notation.

Example

The following code example illustrates the definition and use of default and non-default indexed properties that have get and set methods.

```
type NumberStrings() =
   let mutable ordinals = [| "one"; "two"; "three"; "four"; "five";
                             "six"; "seven"; "eight"; "nine"; "ten" |]
   let mutable cardinals = [| "first"; "second"; "third"; "fourth";
                              "fifth"; "sixth"; "seventh"; "eighth";
                              "ninth"; "tenth" |]
   member this.Item
      with get(index) = ordinals[index]
      and set index value = ordinals[index] <- value
   member this.Ordinal
      with get(index) = ordinals[index]
      and set index value = ordinals[index] <- value
   member this.Cardinal
      with get(index) = cardinals[index]
      and set index value = cardinals[index] <- value

let nstrs = new NumberStrings()
nstrs[0] <- "ONE"
for i in 0 .. 9 do
  printf "%s " nstrs[i]
printfn ""

nstrs.Cardinal(5) <- "6th"

for i in 0 .. 9 do
  printf "%s " (nstrs.Ordinal(i))
  printf "%s " (nstrs.Cardinal(i))
printfn ""
```

Output

```
ONE two three four five six seven eight nine ten
ONE first two second three third four fourth five fifth six 6th
seven seventh eight eighth nine ninth ten tenth
```

Indexed Properties with multiple index values

Indexed properties can have more than one index value. In that case, the values are separated by commas when the property is used. The set method in such a property must have two curried arguments, the first of which is a tuple containing the keys, and the second of which is the value to set.

The following code demonstrates the use of an indexed property with multiple index values.

```
open System.Collections.Generic

/// Basic implementation of a sparse matrix based on a dictionary
type SparseMatrix() =
    let table = new Dictionary<(int * int), float>()
    member _.Item
        // Because the key is comprised of two values, 'get' has two index values
        with get(key1, key2) = table[(key1, key2)]

        // 'set' has two index values and a new value to place in the key's position
        and set (key1, key2) value = table[(key1, key2)] <- value

let sm = new SparseMatrix()
for i in 1..1000 do
    sm[i, i] <- float i * float i
```

See also

- Members

Operator Overloading

8/5/2022 • 7 minutes to read • Edit Online

This topic describes how to overload arithmetic operators in a class or record type, and at the global level.

Syntax

```
// Overloading an operator as a class or record member.
static member (operator-symbols) (parameter-list) =
    method-body
// Overloading an operator at the global level
let [inline] (operator-symbols) parameter-list = function-body
```

Remarks

In the previous syntax, the *operator-symbol* is one of `+`, `-`, `*`, `/`, `=`, and so on. The *parameter-list* specifies the operands in the order they appear in the usual syntax for that operator. The *method-body* constructs the resulting value.

Operator overloads for operators must be static. Operator overloads for unary operators, such as `+` and `-`, must use a tilde (`~`) in the *operator-symbol* to indicate that the operator is a unary operator and not a binary operator, as shown in the following declaration.

```
static member (~-) (v : Vector)
```

The following code illustrates a vector class that has just two operators, one for unary minus and one for multiplication by a scalar. In the example, two overloads for scalar multiplication are needed because the operator must work regardless of the order in which the vector and scalar appear.

```
type Vector(x: float, y : float) =
   member this.x = x
   member this.y = y
   static member (~-) (v : Vector) =
     Vector(-1.0 * v.x, -1.0 * v.y)
   static member (*) (v : Vector, a) =
     Vector(a * v.x, a * v.y)
   static member (*) (a, v: Vector) =
     Vector(a * v.x, a * v.y)
   override this.ToString() =
     this.x.ToString() + " " + this.y.ToString()

let v1 = Vector(1.0, 2.0)

let v2 = v1 * 2.0
let v3 = 2.0 * v1

let v4 = - v2

printfn "%s" (v1.ToString())
printfn "%s" (v2.ToString())
printfn "%s" (v3.ToString())
printfn "%s" (v4.ToString())
```

Creating New Operators

You can overload all the standard operators, but you can also create new operators out of sequences of certain characters. Allowed operator characters are `!`, `$`, `%`, `&`, `*`, `+`, `-`, `.`, `/`, `<`, `=`, `>`, `?`, `@`, `^`, `|`, and `~`. The `~` character has the special meaning of making an operator unary, and is not part of the operator character sequence. Not all operators can be made unary.

Depending on the exact character sequence you use, your operator will have a certain precedence and associativity. Associativity can be either left to right or right to left and is used whenever operators of the same level of precedence appear in sequence without parentheses.

The operator character `.` does not affect precedence, so that, for example, if you want to define your own version of multiplication that has the same precedence and associativity as ordinary multiplication, you could create operators such as `.*`.

The `$` operator must stand alone and without additional symbols.

A table that shows the precedence of all operators in F# can be found in Symbol and Operator Reference.

Overloaded Operator Names

When the F# compiler compiles an operator expression, it generates a method that has a compiler-generated name for that operator. This is the name that appears in the Microsoft intermediate language (MSIL) for the method, and also in reflection and IntelliSense. You do not normally need to use these names in F# code.

The following table shows the standard operators and their corresponding generated names.

OPERATOR	GENERATED NAME
`[]`	`op_Nil`
`::`	`op_Cons`
`+`	`op_Addition`
`-`	`op_Subtraction`
`*`	`op_Multiply`
`/`	`op_Division`
`@`	`op_Append`
`^`	`op_Concatenate`
`%`	`op_Modulus`
`&&&`	`op_BitwiseAnd`
`|||`	`op_BitwiseOr`
`^^^`	`op_ExclusiveOr`

OPERATOR	GENERATED NAME
`<<<`	`op_LeftShift`
`~~~`	`op_LogicalNot`
`>>>`	`op_RightShift`
`~+`	`op_UnaryPlus`
`~-`	`op_UnaryNegation`
`=`	`op_Equality`
`<=`	`op_LessThanOrEqual`
`>=`	`op_GreaterThanOrEqual`
`<`	`op_LessThan`
`>`	`op_GreaterThan`
`?`	`op_Dynamic`
`?<-`	`op_DynamicAssignment`
`|>`	`op_PipeRight`
`<|`	`op_PipeLeft`
`!`	`op_Dereference`
`>>`	`op_ComposeRight`
`<<`	`op_ComposeLeft`
`<@ @>`	`op_Quotation`
`<@@ @@>`	`op_QuotationUntyped`
`+=`	`op_AdditionAssignment`
`-=`	`op_SubtractionAssignment`
`*=`	`op_MultiplyAssignment`
`/=`	`op_DivisionAssignment`
`..`	`op_Range`

OPERATOR	GENERATED NAME
`.. ..`	`op_RangeStep`

Note that the `not` operator in F# does not emit `op_Inequality` because it is not a symbolic operator. It is a function that emits IL that negates a boolean expression.

Other combinations of operator characters that are not listed here can be used as operators and have names that are made up by concatenating names for the individual characters from the following table. For example, +! becomes `op_PlusBang`.

OPERATOR CHARACTER	NAME
`>`	`Greater`
`<`	`Less`
`+`	`Plus`
`-`	`Minus`
`*`	`Multiply`
`/`	`Divide`
`=`	`Equals`
`~`	`Twiddle`
`$`	`Dollar`
`%`	`Percent`
`.`	`Dot`
`&`	`Amp`
`|`	`Bar`
`@`	`At`
`^`	`Hat`
`!`	`Bang`
`?`	`Qmark`
`(`	`LParen`
`,`	`Comma`

OPERATOR CHARACTER	NAME
`)`	`RParen`
`[`	`LBrack`
`]`	`RBrack`

Prefix and Infix Operators

Prefix operators are expected to be placed in front of an operand or operands, much like a function. *Infix* operators are expected to be placed between the two operands.

Only certain operators can be used as prefix operators. Some operators are always prefix operators, others can be infix or prefix, and the rest are always infix operators. Operators that begin with `!`, except `!=`, and the operator `~`, or repeated sequences of `~`, are always prefix operators. The operators `+`, `-`, `+.`, `-.`, `&`, `&&`, `%`, and `%%` can be prefix operators or infix operators. You distinguish the prefix version of these operators from the infix version by adding a `~` at the beginning of a prefix operator when it is defined. The `~` is not used when you use the operator, only when it is defined.

Example

The following code illustrates the use of operator overloading to implement a fraction type. A fraction is represented by a numerator and a denominator. The function `hcf` is used to determine the highest common factor, which is used to reduce fractions.

```
// Determine the highest common factor between
// two positive integers, a helper for reducing
// fractions.
let rec hcf a b =
  if a = 0u then b
  elif a<b then hcf a (b - a)
  else hcf (a - b) b

// type Fraction: represents a positive fraction
// (positive rational number).
type Fraction =
   {
      // n: Numerator of fraction.
      n : uint32
      // d: Denominator of fraction.
      d : uint32
   }

   // Produce a string representation. If the
   // denominator is "1", do not display it.
   override this.ToString() =
      if (this.d = 1u)
        then this.n.ToString()
        else this.n.ToString() + "/" + this.d.ToString()

   // Add two fractions.
   static member (+) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.d + f2.n * f1.d
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Adds a fraction and a positive integer.
   static member (+) (f1: Fraction, i : uint32) =
```

```
      let nTemp = f1.n + i * f1.d
      let dTemp = f1.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Adds a positive integer and a fraction.
   static member (+) (i : uint32, f2: Fraction) =
      let nTemp = f2.n + i * f2.d
      let dTemp = f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Subtract one fraction from another.
   static member (-) (f1 : Fraction, f2 : Fraction) =
      if (f2.n * f1.d > f1.n * f2.d)
        then failwith "This operation results in a negative number, which is not supported."
      let nTemp = f1.n * f2.d - f2.n * f1.d
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Multiply two fractions.
   static member (*) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.n
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Divide two fractions.
   static member (/) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.d
      let dTemp = f2.n * f1.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // A full set of operators can be quite lengthy. For example,
   // consider operators that support other integral data types,
   // with fractions, on the left side and the right side for each.
   // Also consider implementing unary operators.

let fraction1 = { n = 3u; d = 4u }
let fraction2 = { n = 1u; d = 2u }
let result1 = fraction1 + fraction2
let result2 = fraction1 - fraction2
let result3 = fraction1 * fraction2
let result4 = fraction1 / fraction2
let result5 = fraction1 + 1u
printfn "%s + %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result1.ToString())
printfn "%s - %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result2.ToString())
printfn "%s * %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result3.ToString())
printfn "%s / %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result4.ToString())
printfn "%s + 1 = %s" (fraction1.ToString()) (result5.ToString())
```

Output:

```
3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2
3/4 + 1 = 7/4
```

Operators at the Global Level

You can also define operators at the global level. The following code defines an operator `+?`.

```
let inline (+?) (x: int) (y: int) = x + 2*y
printf "%d" (10 +? 1)
```

The output of the above code is `12`.

You can redefine the regular arithmetic operators in this manner because the scoping rules for F# dictate that newly defined operators take precedence over the built-in operators.

The keyword `inline` is often used with global operators, which are often small functions that are best integrated into the calling code. Making operator functions inline also enables them to work with statically resolved type parameters to produce statically resolved generic code. For more information, see Inline Functions and Statically Resolved Type Parameters.

See also

- Members

Explicit Fields: The val Keyword

8/5/2022 • 4 minutes to read • Edit Online

The `val` keyword is used to declare a location to store a value in a class or structure type, without initializing it. Storage locations declared in this manner are called *explicit fields*. Another use of the `val` keyword is in conjunction with the `member` keyword to declare an auto-implemented property. For more information on auto-implemented properties, see Properties.

Syntax

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

Remarks

The usual way to define fields in a class or structure type is to use a `let` binding. However, `let` bindings must be initialized as part of the class constructor, which is not always possible, necessary, or desirable. You can use the `val` keyword when you want a field that is uninitialized.

Explicit fields can be static or non-static. The *access-modifier* can be `public`, `private`, or `internal`. By default, explicit fields are public. This differs from `let` bindings in classes, which are always private.

The DefaultValue attribute is required on explicit fields in class types that have a primary constructor. This attribute specifies that the field is initialized to zero. The type of the field must support zero-initialization. A type supports zero-initialization if it is one of the following:

- A primitive type that has a zero value.
- A type that supports a null value, either as a normal value, as an abnormal value, or as a representation of a value. This includes classes, tuples, records, functions, interfaces, .NET reference types, the `unit` type, and discriminated union types.
- A .NET value type.
- A structure whose fields all support a default zero value.

For example, an immutable field called `someField` has a backing field in the .NET compiled representation with the name `someField@`, and you access the stored value using a property named `someField`.

For a mutable field, the .NET compiled representation is a .NET field.

> **WARNING**
> The .NET Framework namespace `System.ComponentModel` contains an attribute that has the same name. For information about this attribute, see DefaultValueAttribute.

The following code shows the use of explicit fields and, for comparison, a `let` binding in a class that has a primary constructor. Note that the `let`-bound field `myInt1` is private. When the `let`-bound field `myInt1` is referenced from a member method, the self identifier `this` is not required. But when you are referencing the explicit fields `myInt2` and `myString`, the self identifier is required.

```
type MyType() =
    let mutable myInt1 = 10
    [<DefaultValue>] val mutable myInt2 : int
    [<DefaultValue>] val mutable myString : string
    member this.SetValsAndPrint( i: int, str: string) =
       myInt1 <- i
       this.myInt2 <- i + 1
       this.myString <- str
       printfn "%d %d %s" myInt1 (this.myInt2) (this.myString)

let myObject = new MyType()
myObject.SetValsAndPrint(11, "abc")
// The following line is not allowed because let bindings are private.
// myObject.myInt1 <- 20
myObject.myInt2 <- 30
myObject.myString <- "def"

printfn "%d %s" (myObject.myInt2) (myObject.myString)
```

The output is as follows:

```
11 12 abc
30 def
```

The following code shows the use of explicit fields in a class that does not have a primary constructor. In this case, the `DefaultValue` attribute is not required, but all the fields must be initialized in the constructors that are defined for the type.

```
type MyClass =
    val a : int
    val b : int
    // The following version of the constructor is an error
    // because b is not initialized.
    // new (a0, b0) = { a = a0; }
    // The following version is acceptable because all fields are initialized.
    new(a0, b0) = { a = a0; b = b0; }

let myClassObj = new MyClass(35, 22)
printfn "%d %d" (myClassObj.a) (myClassObj.b)
```

The output is `35 22`.

The following code shows the use of explicit fields in a structure. Because a structure is a value type, it automatically has a parameterless constructor that sets the values of its fields to zero. Therefore, the `DefaultValue` attribute is not required.

```
type MyStruct =
    struct
        val mutable myInt : int
        val mutable myString : string
    end

let mutable myStructObj = new MyStruct()
myStructObj.myInt <- 11
myStructObj.myString <- "xyz"

printfn "%d %s" (myStructObj.myInt) (myStructObj.myString)
```

The output is `11 xyz`.

Beware, if you are going to initialize your structure with `mutable` fields without `mutable` keyword, your assignments will work on a copy of the structure which will be discarded right after assignment. Therefore your structure won't change.

```
[<Struct>]
type Foo =
    val mutable bar: string
    member self.ChangeBar bar = self.bar <- bar
    new (bar) = {bar = bar}

let foo = Foo "1"
foo.ChangeBar "2" //make implicit copy of Foo, changes the copy, discards the copy, foo remains unchanged
printfn "%s" foo.bar //prints 1

let mutable foo' = Foo "1"
foo'.ChangeBar "2" //changes foo'
printfn "%s" foo'.bar //prints 2
```

Explicit fields are not intended for routine use. In general, when possible you should use a `let` binding in a class instead of an explicit field. Explicit fields are useful in certain interoperability scenarios, such as when you need to define a structure that will be used in a platform invoke call to a native API, or in COM interop scenarios. For more information, see External Functions. Another situation in which an explicit field might be necessary is when you are working with an F# code generator which emits classes without a primary constructor. Explicit fields are also useful for thread-static variables or similar constructs. For more information, see `System.ThreadStaticAttribute`.

When the keywords `member val` appear together in a type definition, it is a definition of an automatically implemented property. For more information, see Properties.

See also

- Properties
- Members
- `let` Bindings in Classes

Object Expressions

8/5/2022 • 2 minutes to read • Edit Online

An *object expression* is an expression that creates a new instance of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.

Syntax

```
// When typename is a class:
{ new typename [type-params]arguments with
    member-definitions
    [ additional-interface-definitions ]
}
// When typename is not a class:
{ new typename [generic-type-args] with
    member-definitions
    [ additional-interface-definitions ]
}
```

Remarks

In the previous syntax, the *typename* represents an existing class type or interface type. *type-params* describes the optional generic type parameters. The *arguments* are used only for class types, which require constructor parameters. The *member-definitions* are overrides of base class methods, or implementations of abstract methods from either a base class or an interface.

The following example illustrates several different types of object expressions.

```fsharp
// This object expression specifies a System.Object but overrides the
// ToString method.
let obj1 = { new System.Object() with member x.ToString() = "F#" }
printfn $"{obj1}"

// This object expression implements the IFormattable interface.
let delimiter(delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member x.ToString(format: string, provider: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value }

let obj2 = delimiter("{","}", "Bananas!");

printfn "%A" (System.String.Format("{0:D}", obj2))

// Define two interfaces
type IFirst =
  abstract F : unit -> unit
  abstract G : unit -> unit

type ISecond =
  inherit IFirst
  abstract H : unit -> unit
  abstract J : unit -> unit

// This object expression implements both interfaces.
let implementer() =
    { new ISecond with
        member this.H() = ()
        member this.J() = ()
      interface IFirst with
        member this.F() = ()
        member this.G() = () }
```

Using Object Expressions

You use object expressions when you want to avoid the extra code and overhead that is required to create a new, named type. If you use object expressions to minimize the number of types created in a program, you can reduce the number of lines of code and prevent the unnecessary proliferation of types. Instead of creating many types just to handle specific situations, you can use an object expression that customizes an existing type or provides an appropriate implementation of an interface for the specific case at hand.

See also

- F# Language Reference

Type extensions

8/5/2022 • 5 minutes to read • Edit Online

Type extensions (also called *augmentations*) are a family of features that let you add new members to a previously defined object type. The three features are:

- Intrinsic type extensions
- Optional type extensions
- Extension methods

Each can be used in different scenarios and has different tradeoffs.

Syntax

```
// Intrinsic and optional extensions
type typename with
    member self-identifier.member-name =
        body
    ...

// Extension methods
open System.Runtime.CompilerServices

[<Extension>]
type Extensions() =
    [<Extension>]
    static member extension-name (ty: typename, [args]) =
        body
    ...
```

Intrinsic type extensions

An intrinsic type extension is a type extension that extends a user-defined type.

Intrinsic type extensions must be defined in the same file **and** in the same namespace or module as the type they're extending. Any other definition will result in them being optional type extensions.

Intrinsic type extensions are sometimes a cleaner way to separate functionality from the type declaration. The following example shows how to define an intrinsic type extension:

```
namespace Example

type Variant =
    | Num of int
    | Str of string

module Variant =
    let print v =
        match v with
        | Num n -> printf "Num %d" n
        | Str s -> printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
    member x.Print() = Variant.print x
```

Using a type extension allows you to separate each of the following:

- The declaration of a `Variant` type
- Functionality to print the `Variant` class depending on its "shape"
- A way to access the printing functionality with object-style `.` -notation

This is an alternative to defining everything as a member on `Variant`. Although it is not an inherently better approach, it can be a cleaner representation of functionality in some situations.

Intrinsic type extensions are compiled as members of the type they augment, and appear on the type when the type is examined by reflection.

Optional type extensions

An optional type extension is an extension that appears outside the original module, namespace, or assembly of the type being extended.

Optional type extensions are useful for extending a type that you have not defined yourself. For example:

```
module Extensions

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for _ in 1 .. n -> x
        }
```

You can now access `RepeatElements` as if it's a member of IEnumerable<T> as long as the `Extensions` module is opened in the scope that you are working in.

Optional extensions do not appear on the extended type when examined by reflection. Optional extensions must be in modules, and they're only in scope when the module that contains the extension is open or is otherwise in scope.

Optional extension members are compiled to static members for which the object instance is passed implicitly as the first parameter. However, they act as if they're instance members or static members according to how they're declared.

Optional extension members are also not visible to C# or Visual Basic consumers. They can only be consumed in other F# code.

Generic limitation of intrinsic and optional type extensions

It's possible to declare a type extension on a generic type where the type variable is constrained. The requirement is that the constraint of the extension declaration matches the constraint of the declared type.

However, even when constraints are matched between a declared type and a type extension, it's possible for a constraint to be inferred by the body of an extended member that imposes a different requirement on the type parameter than the declared type. For example:

```
open System.Collections.Generic

// NOT POSSIBLE AND FAILS TO COMPILE!
//
// The member 'Sum' has a different requirement on 'T than the type IEnumerable<'T>
type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
```

There is no way to get this code to work with an optional type extension:

- As is, the `Sum` member has a different constraint on `'T` (`static member get_Zero` and `static member (+)`) than what the type extension defines.
- Modifying the type extension to have the same constraint as `Sum` will no longer match the defined constraint on `IEnumerable<'T>`.
- Changing `member this.Sum` to `member inline this.Sum` will give an error that type constraints are mismatched.

What is desired are static methods that "float in space" and can be presented as if they're extending a type. This is where extension methods become necessary.

Extension methods

Finally, extension methods (sometimes called "C# style extension members") can be declared in F# as a static member method on a class.

Extension methods are useful for when you wish to define extensions on a generic type that will constrain the type variable. For example:

```
namespace Extensions

open System.Collections.Generic
open System.Runtime.CompilerServices

[<Extension>]
type IEnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs
```

When used, this code will make it appear as if `Sum` is defined on IEnumerable<T>, so long as `Extensions` has been opened or is in scope.

For the extension to be available to VB.NET code, an extra `ExtensionAttribute` is required at the assembly level:

```
module AssemblyInfo
open System.Runtime.CompilerServices
[<assembly:Extension>]
do ()
```

Other remarks

Type extensions also have the following attributes:

- Any type that can be accessed can be extended.
- Intrinsic and optional type extensions can define *any* member type, not just methods. So extension properties are also possible, for example.
- The `self-identifier` token in the syntax represents the instance of the type being invoked, just like ordinary

members.

- Extended members can be static or instance members.
- Type variables on a type extension must match the constraints of the declared type.

The following limitations also exist for type extensions:

- Type extensions do not support virtual or abstract methods.
- Type extensions do not support override methods as augmentations.
- Type extensions do not support Statically Resolved Type Parameters.
- Optional Type extensions do not support constructors as augmentations.
- Type extensions cannot be defined on type abbreviations.
- Type extensions are not valid for `byref<'T>` (though they can be declared).
- Type extensions are not valid for attributes (though they can be declared).
- You can define extensions that overload other methods of the same name, but the F# compiler gives preference to non-extension methods if there is an ambiguous call.

Finally, if multiple intrinsic type extensions exist for one type, all members must be unique. For optional type extensions, members in different type extensions to the same type can have the same names. Ambiguity errors occur only if client code opens two different scopes that define the same member names.

See also

- F# Language Reference
- Members

Inheritance

8/5/2022 • 3 minutes to read • Edit Online

Inheritance is used to model the "is-a" relationship, or subtyping, in object-oriented programming.

Specifying Inheritance Relationships

You specify inheritance relationships by using the `inherit` keyword in a class declaration. The basic syntactical form is shown in the following example.

```
type MyDerived(...) =
    inherit MyBase(...)
```

A class can have at most one direct base class. If you do not specify a base class by using the `inherit` keyword, the class implicitly inherits from `System.Object`.

Inherited Members

If a class inherits from another class, the methods and members of the base class are available to users of the derived class as if they were direct members of the derived class.

Any let bindings and constructor parameters are private to a class and, therefore, cannot be accessed from derived classes.

The keyword `base` is available in derived classes and refers to the base class instance. It is used like the self-identifier.

Virtual Methods and Overrides

Virtual methods (and properties) work somewhat differently in F# as compared to other .NET languages. To declare a new virtual member, you use the `abstract` keyword. You do this regardless of whether you provide a default implementation for that method. Thus a complete definition of a virtual method in a base class follows this pattern:

```
abstract member [method-name] : [type]

default [self-identifier].[method-name] [argument-list] = [method-body]
```

And in a derived class, an override of this virtual method follows this pattern:

```
override [self-identifier].[method-name] [argument-list] = [method-body]
```

If you omit the default implementation in the base class, the base class becomes an abstract class.

The following code example illustrates the declaration of a new virtual method `function1` in a base class and how to override it in a derived class.

```
type MyClassBase1() =
   let mutable z = 0
   abstract member function1 : int -> int
   default u.function1(a : int) = z <- z + a; z

type MyClassDerived1() =
   inherit MyClassBase1()
   override u.function1(a: int) = a + 1
```

Constructors and Inheritance

The constructor for the base class must be called in the derived class. The arguments for the base class constructor appear in the argument list in the `inherit` clause. The values that are used must be determined from the arguments supplied to the derived class constructor.

The following code shows a base class and a derived class, where the derived class calls the base class constructor in the inherit clause:

```
type MyClassBase2(x: int) =
   let mutable z = x * x
   do for i in 1..z do printf "%d " i


type MyClassDerived2(y: int) =
   inherit MyClassBase2(y * 2)
   do for i in 1..y do printf "%d " i
```

In the case of multiple constructors, the following code can be used. The first line of the derived class constructors is the `inherit` clause, and the fields appear as explicit fields that are declared with the `val` keyword. For more information, see Explicit Fields: The `val` Keyword.

```
type BaseClass =
    val string1 : string
    new (str) = { string1 = str }
    new () = { string1 = "" }

type DerivedClass =
    inherit BaseClass

    val string2 : string
    new (str1, str2) = { inherit BaseClass(str1); string2 = str2 }
    new (str2) = { inherit BaseClass(); string2 = str2 }

let obj1 = DerivedClass("A", "B")
let obj2 = DerivedClass("A")
```

Alternatives to Inheritance

In cases where a minor modification of a type is required, consider using an object expression as an alternative to inheritance. The following example illustrates the use of an object expression as an alternative to creating a new derived type:

```
open System

let object1 = { new Object() with
      override this.ToString() = "This overrides object.ToString()"
      }

printfn "%s" (object1.ToString())
```

For more information about object expressions, see Object Expressions.

When you are creating object hierarchies, consider using a discriminated union instead of inheritance. Discriminated unions can also model varied behavior of different objects that share a common overall type. A single discriminated union can often eliminate the need for a number of derived classes that are minor variations of each other. For information about discriminated unions, see Discriminated Unions.

See also

- Object Expressions
- F# Language Reference

Abstract Classes

8/5/2022 • 4 minutes to read • Edit Online

Abstract classes are classes that leave some or all members unimplemented, so that implementations can be provided by derived classes.

Syntax

```
// Abstract class syntax.
[<AbstractClass>]
type [ accessibility-modifier ] abstract-class-name =
[ inherit base-class-or-interface-name ]
[ abstract-member-declarations-and-member-definitions ]

// Abstract member syntax.
abstract member member-name : type-signature
```

Remarks

In object-oriented programming, an abstract class is used as a base class of a hierarchy, and represents common functionality of a diverse set of object types. As the name "abstract" implies, abstract classes often do not correspond directly onto concrete entities in the problem domain. However, they do represent what many different concrete entities have in common.

Abstract classes must have the `AbstractClass` attribute. They can have implemented and unimplemented members. The use of the term *abstract* when applied to a class is the same as in other .NET languages; however, the use of the term *abstract* when applied to methods (and properties) is a little different in F# from its use in other .NET languages. In F#, when a method is marked with the `abstract` keyword, this indicates that a member has an entry, known as a *virtual dispatch slot*, in the internal table of virtual functions for that type. In other words, the method is virtual, although the `virtual` keyword is not used in F#. The keyword `abstract` is used on virtual methods regardless of whether the method is implemented. The declaration of a virtual dispatch slot is separate from the definition of a method for that dispatch slot. Therefore, the F# equivalent of a virtual method declaration and definition in another .NET language is a combination of both an abstract method declaration and a separate definition, with either the `default` keyword or the `override` keyword. For more information and examples, see Methods.

A class is considered abstract only if there are abstract methods that are declared but not defined. Therefore, classes that have abstract methods are not necessarily abstract classes. Unless a class has undefined abstract methods, do not use the **AbstractClass** attribute.

In the previous syntax, *accessibility-modifier* can be `public`, `private` or `internal`. For more information, see Access Control.

As with other types, abstract classes can have a base class and one or more base interfaces. Each base class or interface appears on a separate line together with the `inherit` keyword.

The type definition of an abstract class can contain fully defined members, but it can also contain abstract members. The syntax for abstract members is shown separately in the previous syntax. In this syntax, the *type signature* of a member is a list that contains the parameter types in order and the return types, separated by `->` tokens and/or `*` tokens as appropriate for curried and tupled parameters. The syntax for abstract member type signatures is the same as that used in signature files and that shown by IntelliSense in the Visual Studio Code

Editor.

The following code illustrates an abstract class Shape, which has two non-abstract derived classes, Square and Circle. The example shows how to use abstract classes, methods, and properties. In the example, the abstract class Shape represents the common elements of the concrete entities circle and square. The common features of all shapes (in a two-dimensional coordinate system) are abstracted out into the Shape class: the position on the grid, an angle of rotation, and the area and perimeter properties. These can be overridden, except for position, the behavior of which individual shapes cannot change.

The rotation method can be overridden, as in the Circle class, which is rotation invariant because of its symmetry. So in the Circle class, the rotation method is replaced by a method that does nothing.

```
// An abstract class that has some methods and properties defined
// and some left abstract.
[<AbstractClass>]
type Shape2D(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
    let mutable rotAngle = 0.0

    // These properties are not declared abstract. They
    // cannot be overriden.
    member this.CenterX with get() = x and set xval = x <- xval
    member this.CenterY with get() = y and set yval = y <- yval

    // These properties are abstract, and no default implementation
    // is provided. Non-abstract derived classes must implement these.
    abstract Area : float with get
    abstract Perimeter : float  with get
    abstract Name : string with get

    // This method is not declared abstract. It cannot be
    // overridden.
    member this.Move dx dy =
       x <- x + dx
       y <- y + dy

    // An abstract method that is given a default implementation
    // is equivalent to a virtual method in other .NET languages.
    // Rotate changes the internal angle of rotation of the square.
    // Angle is assumed to be in degrees.
    abstract member Rotate: float -> unit
    default this.Rotate(angle) = rotAngle <- rotAngle + angle

type Square(x, y, sideLengthIn) =
    inherit Shape2D(x, y)
    member this.SideLength = sideLengthIn
    override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"

type Circle(x, y, radius) =
    inherit Shape2D(x, y)
    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
    // Rotating a circle does nothing, so use the wildcard
    // character to discard the unused argument and
    // evaluate to unit.
    override this.Rotate(_) = ()
    override this.Name = "Circle"

let square1 = new Square(0.0, 0.0, 10.0)
let circle1 = new Circle(0.0, 0.0, 5.0)
circle1.CenterX <- 1.0
circle1.CenterY <- -2.0
```

```
square1.Move -1.0 2.0
square1.Rotate 45.0
circle1.Rotate 45.0
printfn "Perimeter of square with side length %f is %f, %f"
        (square1.SideLength) (square1.Area) (square1.Perimeter)
printfn "Circumference of circle with radius %f is %f, %f"
        (circle1.Radius) (circle1.Area) (circle1.Perimeter)

let shapeList : list<Shape2D> = [ (square1 :> Shape2D);
                                  (circle1 :> Shape2D) ]
List.iter (fun (elem : Shape2D) ->
              printfn "Area of %s: %f" (elem.Name) (elem.Area))
          shapeList
```

Output:

```
Perimeter of square with side length 10.000000 is 40.000000
Circumference of circle with radius 5.000000 is 31.415927
Area of Square: 100.000000
Area of Circle: 78.539816
```

See also

- Classes
- Members
- Methods
- Properties

Structures

8/5/2022 • 4 minutes to read • Edit Online

A *structure* is a compact object type that can be more efficient than a class for types that have a small amount of data and simple behavior.

Syntax

```
[ attributes ]
type [accessibility-modifier] type-name =
    struct
        type-definition-elements-and-members
    end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
    type-definition-elements-and-members
```

Remarks

Structures are *value types*, which means that they are stored directly on the stack or, when they are used as fields or array elements, inline in the parent type. Unlike classes and records, structures have pass-by-value semantics. This means that they are useful primarily for small aggregates of data that are accessed and copied frequently.

In the previous syntax, two forms are shown. The first is not the lightweight syntax, but it is nevertheless frequently used because, when you use the `struct` and `end` keywords, you can omit the `StructAttribute` attribute, which appears in the second form. You can abbreviate `StructAttribute` to just `Struct`.

The *type-definition-elements-and-members* in the previous syntax represents member declarations and definitions. Structures can have constructors and mutable and immutable fields, and they can declare members and interface implementations. For more information, see Members.

Structures cannot participate in inheritance, cannot contain `let` or `do` bindings, and cannot recursively contain fields of their own type (although they can contain reference cells that reference their own type).

Because structures do not allow `let` bindings, you must declare fields in structures by using the `val` keyword. The `val` keyword defines a field and its type but does not allow initialization. Instead, `val` declarations are initialized to zero or null. For this reason, structures that have an implicit constructor (that is, parameters that are given immediately after the structure name in the declaration) require that `val` declarations be annotated with the `DefaultValue` attribute. Structures that have a defined constructor still support zero-initialization. Therefore, the `DefaultValue` attribute is a declaration that such a zero value is valid for the field. Implicit constructors for structures do not perform any actions because `let` and `do` bindings aren't allowed on the type, but the implicit constructor parameter values passed in are available as private fields.

Explicit constructors might involve initialization of field values. When you have a structure that has an explicit constructor, it still supports zero-initialization; however, you do not use the `DefaultValue` attribute on the `val` declarations because it conflicts with the explicit constructor. For more information about `val` declarations, see Explicit Fields: The `val` Keyword.

Attributes and accessibility modifiers are allowed on structures, and follow the same rules as those for other types. For more information, see Attributes and Access Control.

The following code examples illustrate structure definitions.

```
// In Point3D, three immutable values are defined.
// x, y, and z will be initialized to 0.0.
type Point3D =
    struct
        val x: float
        val y: float
        val z: float
    end

// In Point2D, two immutable values are defined.
// It also has a member which computes a distance between itself and another Point2D.
// Point2D has an explicit constructor.
// You can create zero-initialized instances of Point2D, or you can
// pass in arguments to initialize the values.
type Point2D =
    struct
        val X: float
        val Y: float
        new(x: float, y: float) = { X = x; Y = y }

        member this.GetDistanceFrom(p: Point2D) =
            let dX = (p.X - this.X) ** 2.0
            let dY = (p.Y - this.Y) ** 2.0

            dX + dY
            |> sqrt
    end
```

ByRefLike structs

You can define your own structs that can adhere to `byref` -like semantics: see Byrefs for more information. This is done with the IsByRefLikeAttribute attribute:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` does not imply `Struct`. Both must be present on the type.

A " `byref` -like" struct in F# is a stack-bound value type. It is never allocated on the managed heap. A `byref` -like struct is useful for high-performance programming, as it is enforced with set of strong checks about lifetime and non-capture. The rules are:

- They can be used as function parameters, method parameters, local variables, method returns.
- They cannot be static or instance members of a class or normal struct.
- They cannot be captured by any closure construct (`async` methods or lambda expressions).
- They cannot be used as a generic parameter.

Although these rules very strongly restrict usage, they do so to fulfill the promise of high-performance computing in a safe manner.

ReadOnly structs

You can annotate structs with the IsReadOnlyAttribute attribute. For example:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsReadOnly` does not imply `Struct`. You must add both to have an `IsReadOnly` struct.

Use of this attribute emits metadata letting F# and C# know to treat it as `inref<'T>` and `in ref`, respectively.

Defining a mutable value inside of a readonly struct produces an error.

Struct Records and Discriminated Unions

You can represent Records and Discriminated Unions as structs with the `[<Struct>]` attribute. See each article to learn more.

See also

- F# Language Reference
- Classes
- Records
- Members

Computation Expressions

8/5/2022 • 15 minutes to read • Edit Online

Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. Depending on the kind of computation expression, they can be thought of as a way to express monads, monoids, monad transformers, and applicative functors. However, unlike other languages (such as *do-notation* in Haskell), they are not tied to a single abstraction, and do not rely on macros or other forms of metaprogramming to accomplish a convenient and context-sensitive syntax.

Overview

Computations can take many forms. The most common form of computation is single-threaded execution, which is easy to understand and modify. However, not all forms of computation are as straightforward as single-threaded execution. Some examples include:

- Non-deterministic computations
- Asynchronous computations
- Effectful computations
- Generative computations

More generally, there are *context-sensitive* computations that you must perform in certain parts of an application. Writing context-sensitive code can be challenging, as it is easy to "leak" computations outside of a given context without abstractions to prevent you from doing so. These abstractions are often challenging to write by yourself, which is why F# has a generalized way to do so called **computation expressions**.

Computation expressions offer a uniform syntax and abstraction model for encoding context-sensitive computations.

Every computation expression is backed by a *builder* type. The builder type defines the operations that are available for the computation expression. See Creating a New Type of Computation Expression, which shows how to create a custom computation expression.

Syntax overview

All computation expressions have the following form:

```
builder-expr { cexper }
```

In this form, `builder-expr` is the name of a builder type that defines the computation expression, and `cexper` is the expression body of the computation expression. For example, `async` computation expression code can look like this:

```
let fetchAndDownload url =
    async {
        let! data = downloadData url

        let processedData = processData data

        return processedData
    }
```

There is a special, additional syntax available within a computation expression, as shown in the previous example. The following expression forms are possible with computation expressions:

```
expr { let! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }
expr { match! ... }
```

Each of these keywords, and other standard F# keywords are only available in a computation expression if they have been defined in the backing builder type. The only exception to this is `match!`, which is itself syntactic sugar for the use of `let!` followed by a pattern match on the result.

The builder type is an object that defines special methods that govern the way the fragments of the computation expression are combined; that is, its methods control how the computation expression behaves. Another way to describe a builder class is to say that it enables you to customize the operation of many F# constructs, such as loops and bindings.

`let!`

The `let!` keyword binds the result of a call to another computation expression to a name:

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

If you bind the call to a computation expression with `let`, you will not get the result of the computation expression. Instead, you will have bound the value of the *unrealized* call to that computation expression. Use `let!` to bind to the result.

`let!` is defined by the `Bind(x, f)` member on the builder type.

`do!`

The `do!` keyword is for calling a computation expression that returns a `unit`-like type (defined by the `Zero` member on the builder):

```
let doThingsAsync data url =
    async {
        do! submitData data url
        ...
    }
```

For the async workflow, this type is `Async<unit>`. For other computation expressions, the type is likely to be `CExpType<unit>`.

`do!` is defined by the `Bind(x, f)` member on the builder type, where `f` produces a `unit`.

`yield`

The `yield` keyword is for returning a value from the computation expression so that it can be consumed as an IEnumerable<T>:

```
let squares =
    seq {
        for i in 1..10 do
            yield i * i
    }

for sq in squares do
    printfn $"%d{sq}"
```

In most cases, it can be omitted by callers. The most common way to omit `yield` is with the `->` operator:

```
let squares =
    seq {
        for i in 1..10 -> i * i
    }

for sq in squares do
    printfn $"%d{sq}"
```

For more complex expressions that might yield many different values, and perhaps conditionally, simply omitting the keyword can do:

```
let weekdays includeWeekend =
    seq {
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    }
```

As with the yield keyword in C#, each element in the computation expression is yielded back as it is iterated.

`yield` is defined by the `Yield(x)` member on the builder type, where `x` is the item to yield back.

`yield!`

The `yield!` keyword is for flattening a collection of values from a computation expression:

```
let squares =
    seq {
        for i in 1..3 -> i * i
    }

let cubes =
    seq {
        for i in 1..3 -> i * i * i
    }

let squaresAndCubes =
    seq {
        yield! squares
        yield! cubes
    }

printfn $"{squaresAndCubes}"  // Prints - 1; 4; 9; 1; 8; 27
```

When evaluated, the computation expression called by `yield!` will have its items yielded back one-by-one, flattening the result.

`yield!` is defined by the `YieldFrom(x)` member on the builder type, where `x` is a collection of values.

Unlike `yield`, `yield!` must be explicitly specified. Its behavior isn't implicit in computation expressions.

`return`

The `return` keyword wraps a value in the type corresponding to the computation expression. Aside from computation expressions using `yield`, it is used to "complete" a computation expression:

```
let req = // 'req' is of type 'Async<data>'
    async {
        let! data = fetch url
        return data
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return` is defined by the `Return(x)` member on the builder type, where `x` is the item to wrap.

`return!`

The `return!` keyword realizes the value of a computation expression and wraps that result in the type corresponding to the computation expression:

```
let req = // 'req' is of type 'Async<data>'
    async {
        return! fetch url
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return!` is defined by the `ReturnFrom(x)` member on the builder type, where `x` is another computation expression.

`match!`

The `match!` keyword allows you to inline a call to another computation expression and pattern match on its result:

```
let doThingsAsync url =
    async {
        match! callService url with
        | Some data -> ...
        | None -> ...
    }
```

When calling a computation expression with `match!`, it will realize the result of the call like `let!`. This is often used when calling a computation expression where the result is an optional.

Built-in computation expressions

The F# core library defines four built-in computation expressions: Sequence Expressions, Async expressions, Task expressions, and Query Expressions.

Creating a New Type of Computation Expression

You can define the characteristics of your own computation expressions by creating a builder class and defining certain special methods on the class. The builder class can optionally define the methods as listed in the following table.

The following table describes methods that can be used in a workflow builder class.

METHOD	TYPICAL SIGNATURE(S)	DESCRIPTION
`Bind`	`M<'T> * ('T -> M<'U>) -> M<'U>`	Called for `let!` and `do!` in computation expressions.
`Delay`	`(unit -> M<'T>) -> Delayed<'T>`	Wraps a computation expression as a function. `Delayed<'T>` can be any type, commonly `M<'T>` or `unit -> M<'T>` are used. The default implementation returns a `M<'T>`.
`Return`	`'T -> M<'T>`	Called for `return` in computation expressions.
`ReturnFrom`	`M<'T> -> M<'T>`	Called for `return!` in computation expressions.

METHOD	TYPICAL SIGNATURE(S)	DESCRIPTION
`Run`	`Delayed<'T> -> M<'T>` or `M<'T> -> 'T`	Executes a computation expression.
`Combine`	`M<'T> * Delayed<'T> -> M<'T>` or `M<unit> * M<'T> -> M<'T>`	Called for sequencing in computation expressions.

`For`	`seq<'T> * ('T -> M<'U>) -> M<'U>`
	or
	
	`seq<'T> * ('T -> M<'U>) -> seq<M<'U>>`
		Called for `for...do` expressions in computation expressions.

`TryFinally`	`Delayed<'T> * (unit -> unit) -> M<'T>`	Called for `try...finally` expressions in computation expressions.
`TryWith`	`Delayed<'T> * (exn -> M<'T>) -> M<'T>`	Called for `try...with` expressions in computation expressions.
`Using`	`'T * ('T -> M<'U>) -> M<'U> when 'T :> IDisposable`	Called for `use` bindings in computation expressions.

`While`	`(unit -> bool) * Delayed<'T> -> M<'T>`
	or
	
	`(unit -> bool) * Delayed<unit> -> M<unit>`
		Called for `while...do` expressions in computation expressions.

`Yield`	`'T -> M<'T>`	Called for `yield` expressions in computation expressions.
`YieldFrom`	`M<'T> -> M<'T>`	Called for `yield!` expressions in computation expressions.
`Zero`	`unit -> M<'T>`	Called for empty `else` branches of `if...then` expressions in computation expressions.
`Quote`	`Quotations.Expr<'T> -> Quotations.Expr<'T>`	Indicates that the computation expression is passed to the `Run` member as a quotation. It translates all instances of a computation into a quotation.

Many of the methods in a builder class use and return an `M<'T>` construct, which is typically a separately defined type that characterizes the kind of computations being combined, for example, `Async<'T>` for async expressions and `Seq<'T>` for sequence workflows. The signatures of these methods enable them to be combined and nested with each other, so that the workflow object returned from one construct can be passed to the next.

Many functions use the result of `Delay` as an argument: `Run`, `While`, `TryWith`, `TryFinally`, and `Combine`. The `Delayed<'T>` type is the return type of `Delay` and consequently the parameter to these functions. `Delayed<'T>` can be an arbitrary type that does not need to be related to `M<'T>`; commonly `M<'T>` or `(unit -> M<'T>)` are used. The default implementation is `M<'T>`. See here for a more in-depth look.

The compiler, when it parses a computation expression, converts the expression into a series of nested function calls by using the methods in the preceding table and the code in the computation expression. The nested expression is of the following form:

```
builder.Run(builder.Delay(fun () -> {| cexpr |}))
```

In the above code, the calls to `Run` and `Delay` are omitted if they are not defined in the computation expression builder class. The body of the computation expression, here denoted as `{| cexpr |}`, is translated into calls involving the methods of the builder class by the translations described in the following table. The computation expression `{| cexpr |}` is defined recursively according to these translations where `expr` is an F# expression and `cexpr` is a computation expression.

<table>
<tr><th>EXPRESSION</th><th>TRANSLATION</th></tr>
<tr><td><code>{ let binding in cexpr }</code></td><td><code>let binding in {| cexpr |}</code></td></tr>
<tr><td><code>{ let! pattern = expr in cexpr }</code></td><td><code>builder.Bind(expr, (fun pattern -> {| cexpr |}))</code></td></tr>
<tr><td><code>{ do! expr in cexpr }</code></td><td><code>builder.Bind(expr, (fun () -> {| cexpr |}))</code></td></tr>
<tr><td><code>{ yield expr }</code></td><td><code>builder.Yield(expr)</code></td></tr>
<tr><td><code>{ yield! expr }</code></td><td><code>builder.YieldFrom(expr)</code></td></tr>
<tr><td><code>{ return expr }</code></td><td><code>builder.Return(expr)</code></td></tr>
<tr><td><code>{ return! expr }</code></td><td><code>builder.ReturnFrom(expr)</code></td></tr>
<tr><td><code>{ use pattern = expr in cexpr }</code></td><td><code>builder.Using(expr, (fun pattern -> {| cexpr |}))</code></td></tr>
<tr><td><code>{ use! value = expr in cexpr }</code></td><td><code>builder.Bind(expr, (fun value ->
builder.Using(value, (fun value -> { cexpr }))))</code></td></tr>
<tr><td><code>{ if expr then cexpr0 |}</code></td><td><code>if expr then { cexpr0 } else builder.Zero()</code></td></tr>
<tr><td><code>{ if expr then cexpr0 else cexpr1 |}</code></td><td><code>if expr then { cexpr0 } else { cexpr1 }</code></td></tr>
<tr><td><code>{ match expr with | pattern_i -> cexpr_i }</code></td><td><code>match expr with | pattern_i -> { cexpr_i }</code></td></tr>
<tr><td><code>{ for pattern in expr do cexpr }</code></td><td><code>builder.For(enumeration, (fun pattern -> { cexpr
}))</code></td></tr>
<tr><td><code>{ for identifier = expr1 to expr2 do cexpr }</code></td><td><code>builder.For(enumeration, (fun identifier -> { cexpr
}))</code></td></tr>
<tr><td><code>{ while expr do cexpr }</code></td><td><code>builder.While(fun () -> expr, builder.Delay({ cexpr
}))</code></td></tr>
<tr><td><code>{ try cexpr with | pattern_i -> expr_i }</code></td><td><code>builder.TryWith(builder.Delay({ cexpr }), (fun value -> match value wi
pattern_i -> expr_i | exn ->
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Capture(exn).Th</code></td></tr>
<tr><td><code>{ try cexpr finally expr }</code></td><td><code>builder.TryFinally(builder.Delay({ cexpr }), (fun
() -> expr))</code></td></tr>
<tr><td><code>{ cexpr1; cexpr2 }</code></td><td><code>builder.Combine({ cexpr1 }, { cexpr2 })</code></td></tr>
<tr><td><code>{ other-expr; cexpr }</code></td><td><code>expr; { cexpr }</code></td></tr>
<tr><td><code>{ other-expr }</code></td><td><code>expr; builder.Zero()</code></td></tr>
</table>

In the previous table, `other-expr` describes an expression that is not otherwise listed in the table. A builder class does not need to implement all of the methods and support all of the translations listed in the previous table. Those constructs that are not implemented are not available in computation expressions of that type. For example, if you do not want to support the `use` keyword in your computation expressions, you can omit the definition of `Use` in your builder class.

The following code example shows a computation expression that encapsulates a computation as a series of steps that can be evaluated one step at a time. A discriminated union type, `OkOrException`, encodes the error state of the expression as evaluated so far. This code demonstrates several typical patterns that you can use in your computation expressions, such as boilerplate implementations of some of the builder methods.

```
/// Represents computations that can be run step by step
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =

    /// Bind a computation using 'func'.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    /// Return the final value
    let result value = Done value

    /// The catch for the computations. Stitch try/with throughout
    /// the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Error exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Error exn -> result (Error exn))

    /// The delay operator.
    let delay func = NotYetDone (fun () -> func())

    /// The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    /// The tryFinally operator.
    /// This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Error exn -> raise exn)

    /// The tryWith operator.
    /// This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
        |> bind (function Ok value -> result value | Error exn -> handler exn)

    /// The whileLoop operator.
    /// This is boilerplate in terms of "result" and "bind".
    let rec whileLoop pred body =
        if pred() then body |> bind (fun _ -> whileLoop pred body)
        else result ()

    /// The sequential composition operator.
    /// This is boilerplate in terms of "result" and "bind".
    let combine expr1 expr2 =
        expr1 |> bind (fun () -> expr2)

    /// The using operator.
    /// This is boilerplate in terms of "tryFinally" and "Dispose".
    let using (resource: #System.IDisposable) func =
        tryFinally (func resource) (fun () -> resource.Dispose())
```

```
    /// The forLoop operator.
    /// This is boilerplate in terms of "catch", "result", and "bind".
    let forLoop (collection:seq<_>) func =
        let ie = collection.GetEnumerator()
        tryFinally
            (whileLoop
                (fun () -> ie.MoveNext())
                (delay (fun () -> let value = ie.Current in func value)))
            (fun () -> ie.Dispose())

/// The builder class.
type EventuallyBuilder() =
    member x.Bind(comp, func) = Eventually.bind func comp
    member x.Return(value) = Eventually.result value
    member x.ReturnFrom(value) = value
    member x.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member x.Delay(func) = Eventually.delay func
    member x.Zero() = Eventually.result ()
    member x.TryWith(expr, handler) = Eventually.tryWith expr handler
    member x.TryFinally(expr, compensation) = Eventually.tryFinally expr compensation
    member x.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member x.Using(resource, expr) = Eventually.using resource expr

let eventually = new EventuallyBuilder()

let comp =
    eventually {
        for x in 1..2 do
            printfn $" x = %d{x}"
        return 3 + 4
    }

/// Try the remaining lines in F# interactive to see how this
/// computation expression works in practice.
let step x = Eventually.step x

// returns "NotYetDone <closure>"
comp |> step

// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step

// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step
```

A computation expression has an underlying type, which the expression returns. The underlying type may represent a computed result or a delayed computation that can be performed, or it may provide a way to iterate through some type of collection. In the previous example, the underlying type was `Eventually<_>`. For a sequence expression, the underlying type is System.Collections.Generic.IEnumerable<T>. For a query expression, the underlying type is System.Linq.IQueryable. For an async expression, the underlying type is `Async`. The `Async` object represents the work to be performed to compute the result. For example, you call `Async.RunSynchronously` to execute a computation and return the result.

Custom Operations

You can define a custom operation on a computation expression and use a custom operation as an operator in a computation expression. For example, you can include a query operator in a query expression. When you define a custom operation, you must define the Yield and For methods in the computation expression. To define a custom operation, put it in a builder class for the computation expression, and then apply the `CustomOperationAttribute`. This attribute takes a string as an argument, which is the name to be used in a custom operation. This name comes into scope at the start of the opening curly brace of the computation expression. Therefore, you shouldn't use identifiers that have the same name as a custom operation in this block. For example, avoid the use of identifiers such as `all` or `last` in query expressions.

Extending existing Builders with new Custom Operations

If you already have a builder class, its custom operations can be extended from outside of this builder class. Extensions must be declared in modules. Namespaces cannot contain extension members except in the same file and the same namespace declaration group where the type is defined.

The following example shows the extension of the existing `FSharp.Linq.QueryBuilder` class.

```
open System
open FSharp.Linq

type QueryBuilder with

    [<CustomOperation("existsNot")>]
    member _.ExistsNot (source: QuerySource<'T, 'Q>, predicate) =
        System.Linq.Enumerable.Any (source.Source, Func<_,_>(predicate)) |> not
```

Custom operations can be overloaded. For more information, see F# RFC FS-1056 - Allow overloads of custom keywords in computation expressions.

Compiling computation expressions efficiently

F# computation expressions that suspend execution can be compiled to highly efficient state machines through careful use of a low-level feature called *resumable code*. Resumable code is documented in F# RFC FS-1087 and used for Task Expressions.

F# computation expressions that are synchronous (that is, they don't suspend execution) can alternatively be compiled to efficient state machines by using inline functions including the `InlineIfLambda` attribute. Examples are given in F# RFC FS-1098.

List expressions, array expressions, and sequence expressions are given special treatment by the F# compiler to ensure generation of high-performance code.

See also

- F# Language Reference
- Sequences
- Async expressions
- Task expressions
- Query Expressions
- Series on Computation Expressions from F# for Fun and Profit

Async expressions

8/5/2022 • 5 minutes to read • Edit Online

This article describes support in F# for async expressions. Async expressions provide one way of performing computations asynchronously, that is, without blocking execution of other work. For example, asynchronous computations can be used to write apps that have UIs that remain responsive to users as the application performs other work.

Asynchronous code can also be authored using task expressions, which create .NET tasks directly. Using task expressions is preferred when interoperating extensively with .NET libraries that create or consume .NET tasks. When writing most asynchronous code in F#, F# async expressions are preferred because they are more succinct, more compositional, and avoid certain caveats associated with .NET tasks.

Syntax

```
async { expression }
```

Remarks

In the previous syntax, the computation represented by `expression` is set up to run asynchronously, that is, without blocking the current computation thread when asynchronous sleep operations, I/O, and other asynchronous operations are performed. Asynchronous computations are often started on a background thread while execution continues on the current thread. The type of the expression is `Async<'T>`, where `'T` is the type returned by the expression when the `return` keyword is used.

The `Async` class provides methods that support several scenarios. The general approach is to create `Async` objects that represent the computation or computations that you want to run asynchronously, and then start these computations by using one of the triggering functions. The triggering you use depends on whether you want to use the current thread, a background thread, or a .NET task object. For example, to start an async computation on the current thread, you can use `Async.StartImmediate`. When you start an async computation from the UI thread, you do not block the main event loop that processes user actions such as keystrokes and mouse activity, so your application remains responsive.

Asynchronous Binding by Using let!

In an async expression, some expressions and operations are synchronous, and some are asynchronous. When you call a method asynchronously, instead of an ordinary `let` binding, you use `let!`. The effect of `let!` is to enable execution to continue on other computations or threads as the computation is being performed. After the right side of the `let!` binding returns, the rest of the async expression resumes execution.

The following code shows the difference between `let` and `let!`. The line of code that uses `let` just creates an asynchronous computation as an object that you can run later by using, for example, `Async.StartImmediate` or `Async.RunSynchronously`. The line of code that uses `let!` starts the computation and performs an asynchronous wait: the thread is suspended until the result is available, at which point execution continues.

```
// let just stores the result as an asynchronous operation.
let (result1 : Async<byte[]>) = stream.AsyncRead(bufferSize)
// let! completes the asynchronous operation and returns the data.
let! (result2 : byte[])  = stream.AsyncRead(bufferSize)
```

`let!` can only be used to await F# async computations `Async<T>` directly. You can await other kinds of asynchronous operations indirectly:

- .NET tasks, Task<TResult> and the non-generic Task, by combining with `Async.AwaitTask`
- .NET value tasks, ValueTask<TResult> and the non-generic ValueTask, by combining with `.AsTask()` and `Async.AwaitTask`
- Any object following the "GetAwaiter" pattern specified in F# RFC FS-1097, by combining with `task { return! expr } |> Async.AwaitTask`.

Control Flow

Async expressions can include control-flow constructs, such as `for .. in .. do`, `while .. do`, `try .. with ..`, `try .. finally ..`, `if .. then .. else`, and `if .. then ..`. These may, in turn, include further async constructs, with the exception of the `with` and `finally` handlers, which execute synchronously.

F# async expressions don't support asynchronous `try .. finally ..`. You can use a task expression for this case.

`use` and `use!` bindings

Within async expressions, `use` bindings can bind to values of type IDisposable. For the latter, the disposal cleanup operation is executed asynchronously.

In addition to `let!`, you can use `use!` to perform asynchronous bindings. The difference between `let!` and `use!` is the same as the difference between `let` and `use`. For `use!`, the object is disposed of at the close of the current scope. Note that in the current release of F#, `use!` does not allow a value to be initialized to null, even though `use` does.

Asynchronous Primitives

A method that performs a single asynchronous task and returns the result is called an *asynchronous primitive*, and these are designed specifically for use with `let!`. Several asynchronous primitives are defined in the F# core library. Two such methods for Web applications are defined in the module `FSharp.Control.WebExtensions`: `WebRequest.AsyncGetResponse` and `WebClient.AsyncDownloadString`. Both primitives download data from a Web page, given a URL. `AsyncGetResponse` produces a `System.Net.WebResponse` object, and `AsyncDownloadString` produces a string that represents the HTML for a Web page.

Several primitives for asynchronous I/O operations are included in the `FSharp.Control.CommonExtensions` module. These extension methods of the `System.IO.Stream` class are `Stream.AsyncRead` and `Stream.AsyncWrite`.

You can also write your own asynchronous primitives by defining a function or method whose body is an async expression.

To use asynchronous methods in the .NET Framework that are designed for other asynchronous models with the F# asynchronous programming model, you create a function that returns an F# `Async` object. The F# library has functions that make this easy to do.

One example of using async expressions is included here; there are many others in the documentation for the methods of the Async class.

This example shows how to use async expressions to execute code in parallel.

In the following code example, a function `fetchAsync` gets the HTML text returned from a Web request. The `fetchAsync` function contains an asynchronous block of code. When a binding is made to the result of an asynchronous primitive, in this case `AsyncDownloadString`, `let!` is used instead of `let`.

You use the function `Async.RunSynchronously` to execute an asynchronous operation and wait for its result. As an example, you can execute multiple asynchronous operations in parallel by using the `Async.Parallel` function together with the `Async.RunSynchronously` function. The `Async.Parallel` function takes a list of the `Async` objects, sets up the code for each `Async` task object to run in parallel, and returns an `Async` object that represents the parallel computation. Just as for a single operation, you call `Async.RunSynchronously` to start the execution.

The `runAll` function launches three async expressions in parallel and waits until they have all completed.

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

See also

- F# Language Reference
- Computation Expressions
- Task Expressions
- Control.Async Class

Tasks expressions

8/5/2022 • 6 minutes to read • Edit Online

This article describes support in F# for task expressions, which are similar to async expressions but allow you to author .NET tasks directly. Like async expressions, task expressions execute code asynchronously, that is, without blocking execution of other work.

Asynchronous code is normally authored using async expressions. Using task expressions is preferred when interoperating extensively with .NET libraries that create or consume .NET tasks. Task expressions can also improve performance and the debugging experience. However, task expressions come with some limitations, which are described later in the article.

Syntax

```
task { expression }
```

In the previous syntax, the computation represented by `expression` is set up to run as a .NET task. The task is started immediately after this code is executed and runs on the current thread until its first asynchronous operation is performed (for example, an asynchronous sleep, asynchronous I/O, or other primitive asynchronous operation). The type of the expression is `Task<'T>`, where `'T` is the type returned by the expression when the `return` keyword is used.

Binding by using let!

In a task expression, some expressions and operations are synchronous, and some are asynchronous. When you await the result of an asynchronous operation, instead of an ordinary `let` binding, you use `let!`. The effect of `let!` is to enable execution to continue on other computations or threads as the computation is being performed. After the right side of the `let!` binding returns, the rest of the task resumes execution.

The following code shows the difference between `let` and `let!`. The line of code that uses `let` just creates a task as an object that you can await later by using, for example, `task.Wait()` or `task.Result`. The line of code that uses `let!` starts the task and awaits its result.

```
// let just stores the result as a task.
let (result1 : Task<int>) = stream.ReadAsync(buffer, offset, count, cancellationToken)
// let! completes the asynchronous operation and returns the data.
let! (result2 : int)  = stream.ReadAsync(buffer, offset, count, cancellationToken)
```

F# `task { }` expressions can await the following kinds of asynchronous operations:

- .NET tasks, Task<TResult> and the non-generic Task.
- .NET value tasks, ValueTask<TResult> and the non-generic ValueTask.
- F# async computations `Async<T>`.
- Any object following the "GetAwaiter" pattern specified in F# RFC FS-1097.

`return` expressions

Within task expressions, `return expr` is used to return the result of a task.

`return!` expressions

Within task expressions, `return! expr` is used to return the result of another task. It is equivalent to using `let!` and then immediately returning the result.

Control flow

Task expressions can include the control-flow constructs `for .. in .. do`, `while .. do`, `try .. with ..`, `try .. finally ..`, `if .. then .. else`, and `if .. then ..`. These may in turn include further task constructs, except for the `with` and `finally` handlers, which execute synchronously. If you need an asynchronous `try .. finally ..`, use a `use` binding in combination with an object of type `IAsyncDisposable`.

`use` and `use!` bindings

Within task expressions, `use` bindings can bind to values of type IDisposable or IAsyncDisposable. For the latter, the disposal cleanup operation is executed asynchronously.

In addition to `let!`, you can use `use!` to perform asynchronous bindings. The difference between `let!` and `use!` is the same as the difference between `let` and `use`. For `use!`, the object is disposed of at the close of the current scope. Note that in F# 6, `use!` does not allow a value to be initialized to null, even though `use` does.

Value Tasks

Value tasks are structs used to avoid allocations in task-based programming. A value task is an ephemeral value that's turned into a real task by using `.AsTask()`.

To create a value task from a task expression, use `|> ValueTask<ReturnType>` or `|> ValueTask`. For example:

```
let makeTask() =
    task { return 1 }

makeTask() |> ValueTask<int>
```

Adding cancellation tokens and cancellation checks

Unlike F# async expressions, task expressions do not implicitly pass a cancellation token and don't implicitly perform cancellation checks. If your code requires a cancellation token, you should specify the cancellation token as a parameter. For example:

```
open System.Threading

let someTaskCode (cancellationToken: CancellationToken) =
    task {
        cancellationToken.ThrowIfCancellationRequested()
        printfn $"continuing..."
    }
```

If you intend to correctly make your code cancelable, carefully check that you pass the cancellation token through to all .NET library operations that support cancellation. For example, `Stream.ReadAsync` has multiple overloads, one of which accepts a cancellation token. If you do not use this overload, that specific asynchronous read operation will not be cancelable.

Background tasks

By default, .NET tasks are scheduled using SynchronizationContext.Current if present. This allows tasks to serve as cooperative, interleaved agents executing on a user interface thread without blocking the UI. If not present, task continuations are scheduled to the .NET thread pool.

In practice, it's often desirable that library code that generates tasks ignores the synchronization context and instead always switches to the .NET thread pool, if necessary. You can achieve this using `backgroundTask { }`:

```
backgroundTask { expression }
```

A background task ignores any `SynchronizationContext.Current` in the following sense: if started on a thread with non-null `SynchronizationContext.Current`, it switches to a background thread in the thread pool using `Task.Run`. If started on a thread with null `SynchronizationContext.Current`, it executes on that same thread.

> **NOTE**
>
> In practice, this means that calls to `ConfigureAwait(false)` are not typically needed in F# task code. Instead, tasks that are intended to run in the background should be authored using `backgroundTask { ... }`. Any outer task binding to a background task will resynchronize to the `SynchronizationContext.Current` on completion of the background task.

Limitations of tasks regarding tailcalls

Unlike F# async expressions, task expressions do not support tailcalls. That is, when `return!` is executed, the current task is registered as awaiting the task whose result is being returned. This means that recursive functions and methods implemented using task expressions may create unbounded chains of tasks, and these may use unbounded stack or heap. For example, consider the following code:

```
let rec taskLoopBad (count: int) : Task<string> =
    task {
        if count = 0 then
            return "done!"
        else
            printfn $"looping..., count = {count}"
            return! taskLoopBad (count-1)
    }

let t = taskLoopBad 10000000
t.Wait()
```

This coding style should not be used with task expressions—it will create a chain of 10000000 tasks and cause a `StackOverflowException`. If an asynchronous operation is added on each loop invocation, the code will use an essentially unbounded heap. Consider switching this code to use an explicit loop, for example:

```
let taskLoopGood (count: int) : Task<string> =
    task {
        for i in count .. 1 do
            printfn $"looping... count = {count}"
        return "done!"
    }

let t = taskLoopGood 10000000
t.Wait()
```

If asynchronous tailcalls are required, use an F# async expression, which does support tailcalls. For example:

```
let rec asyncLoopGood (count: int) =
    async {
        if count = 0 then
            return "done!"
        else
            printfn $"looping..., count = {count}"
            return! asyncLoopGood (count-1)
    }

let t = asyncLoopGood 1000000 |> Async.StartAsTask
t.Wait()
```

Task implementation

Tasks are implemented using Resumable Code, a new feature in F# 6. Tasks are compiled into "Resumable State Machines" by the F# compiler. These are described in detail in the Resumable code RFC, and in an F# compiler community session.

See also

- F# Language Reference
- Computation Expressions
- Async Expressions
- Resumable State Machines - F# Compiler Community Session
- Resumable Code - RFC FS-1087
- Task
- Task<TResult>
- ValueTask
- ValueTask<TResult>

Lazy Expressions

8/5/2022 • 2 minutes to read • Edit Online

Lazy expressions are expressions that are not evaluated immediately, but are instead evaluated when the result is needed. This can help to improve the performance of your code.

Syntax

```
let identifier = lazy ( expression )
```

Remarks

In the previous syntax, *expression* is code that is evaluated only when a result is required, and *identifier* is a value that stores the result. The value is of type `Lazy<'T>`, where the actual type that is used for `'T` is determined from the result of the expression.

Lazy expressions enable you to improve performance by restricting the execution of an expression to only those situations in which a result is needed.

To force the expressions to be performed, you call the method `Force`. `Force` causes the execution to be performed only one time. Subsequent calls to `Force` return the same result, but do not execute any code.

The following code illustrates the use of lazy expressions and the use of `Force`. In this code, the type of `result` is `Lazy<int>`, and the `Force` method returns an `int`.

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

Lazy evaluation, but not the `Lazy` type, is also used for sequences. For more information, see Sequences.

See also

- F# Language Reference
- LazyExtensions module

Namespaces (F#)

8/5/2022 • 4 minutes to read • Edit Online

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of F# program elements. Namespaces are typically top-level elements in F# files.

Syntax

```
namespace [rec] [parent-namespaces.]identifier
```

Remarks

If you want to put code in a namespace, the first declaration in the file must declare the namespace. The contents of the entire file then become part of the namespace, provided no other namespaces declaration exists further in the file. If that is the case, then all code up until the next namespace declaration is considered to be within the first namespace.

Namespaces cannot directly contain values and functions. Instead, values and functions must be included in modules, and modules are included in namespaces. Namespaces can contain types, modules.

XML doc comments can be declared above a namespace, but they're ignored. Compiler directives can also be declared above a namespace.

Namespaces can be declared explicitly with the namespace keyword, or implicitly when declaring a module. To declare a namespace explicitly, use the namespace keyword followed by the namespace name. The following example shows a code file that declares a namespace `Widgets` with a type and a module included in that namespace.

```
namespace Widgets

type MyWidget1 =
    member this.WidgetName = "Widget1"

module WidgetsModule =
    let widgetName = "Widget2"
```

If the entire contents of the file are in one module, you can also declare namespaces implicitly by using the `module` keyword and providing the new namespace name in the fully qualified module name. The following example shows a code file that declares a namespace `Widgets` and a module `WidgetsModule`, which contains a function.

```
module Widgets.WidgetModule

let widgetFunction x y =
   printfn "%A %A" x y
```

The following code is equivalent to the preceding code, but the module is a local module declaration. In that case, the namespace must appear on its own line.

```
namespace Widgets

module WidgetModule =

    let widgetFunction x y =
        printfn "%A %A" x y
```

If more than one module is required in the same file in one or more namespaces, you must use local module declarations. When you use local module declarations, you cannot use the qualified namespace in the module declarations. The following code shows a file that has a namespace declaration and two local module declarations. In this case, the modules are contained directly in the namespace; there is no implicitly created module that has the same name as the file. Any other code in the file, such as a `do` binding, is in the namespace but not in the inner modules, so you need to qualify the module member `widgetFunction` by using the module name.

```
namespace Widgets

module WidgetModule1 =
   let widgetFunction x y =
      printfn "Module1 %A %A" x y
module WidgetModule2 =
   let widgetFunction x y =
      printfn "Module2 %A %A" x y

module useWidgets =

  do
     WidgetModule1.widgetFunction 10 20
     WidgetModule2.widgetFunction 5 6
```

The output of this example is as follows.

```
Module1 10 20
Module2 5 6
```

For more information, see Modules.

Nested Namespaces

When you create a nested namespace, you must fully qualify it. Otherwise, you create a new top-level namespace. Indentation is ignored in namespace declarations.

The following example shows how to declare a nested namespace.

```
namespace Outer

    // Full name: Outer.MyClass
    type MyClass() =
       member this.X(x) = x + 1

// Fully qualify any nested namespaces.
namespace Outer.Inner

    // Full name: Outer.Inner.MyClass
    type MyClass() =
       member this.Prop1 = "X"
```

Namespaces in Files and Assemblies

Namespaces can span multiple files in a single project or compilation. The term *namespace fragment* describes the part of a namespace that is included in one file. Namespaces can also span multiple assemblies. For example, the `System` namespace includes the whole .NET Framework, which spans many assemblies and contains many nested namespaces.

Global Namespace

You use the predefined namespace `global` to put names in the .NET top-level namespace.

```
namespace global

type SomeType() =
    member this.SomeMember = 0
```

You can also use global to reference the top-level .NET namespace, for example, to resolve name conflicts with other namespaces.

```
global.System.Console.WriteLine("Hello World!")
```

Recursive namespaces

Namespaces can also be declared as recursive to allow for all contained code to be mutually recursive. This is done via `namespace rec`. Use of `namespace rec` can alleviate some pains in not being able to write mutually referential code between types and modules. The following is an example of this:

```
namespace rec MutualReferences

type Orientation = Up | Down
type PeelState = Peeled | Unpeeled

// This exception depends on the type below.
exception DontSqueezeTheBananaException of Banana

type Banana(orientation : Orientation) =
    member val IsPeeled = false with get, set
    member val Orientation = orientation with get, set
    member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

    member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
    member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on the
exception above.

module BananaHelpers =
    let peel (b: Banana) =
        let flip (banana: Banana) =
            match banana.Orientation with
            | Up ->
                banana.Orientation <- Down
                banana
            | Down -> banana

        let peelSides (banana: Banana) =
            banana.Sides
            |> List.map (function
                         | Unpeeled -> Peeled
                         | Peeled -> Peeled)

        match b.Orientation with
        | Up ->   b |> flip |> peelSides
        | Down -> b |> peelSides
```

Note that the exception `DontSqueezeTheBananaException` and the class `Banana` both refer to each other. Additionally, the module `BananaHelpers` and the class `Banana` also refer to each other. This wouldn't be possible to express in F# if you removed the `rec` keyword from the `MutualReferences` namespace.

This feature is also available for top-level Modules.

See also

- F# Language Reference
- Modules
- F# RFC FS-1009 - Allow mutually referential types and modules over larger scopes within files

Modules

8/5/2022 • 6 minutes to read • Edit Online

In the context of F#, a *module* is a grouping of F# code, such as values, types, and function values, in an F# program. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

Syntax

```
// Top-level module declaration.
module [accessibility-modifier] [qualified-namespace.]module-name
declarations
// Local module declaration.
module [accessibility-modifier] module-name =
    declarations
```

Remarks

An F# module is a grouping of F# code constructs such as types, values, function values, and code in `do` bindings. It is implemented as a common language runtime (CLR) class that has only static members. There are two types of module declarations, depending on whether the whole file is included in the module: a top-level module declaration and a local module declaration. A top-level module declaration includes the whole file in the module. A top-level module declaration can appear only as the first declaration in a file.

In the syntax for the top-level module declaration, the optional *qualified-namespace* is the sequence of nested namespace names that contains the module. The qualified namespace does not have to be previously declared.

You do not have to indent declarations in a top-level module. You do have to indent all declarations in local modules. In a local module declaration, only the declarations that are indented under that module declaration are part of the module.

If a code file does not begin with a top-level module declaration or a namespace declaration, the whole contents of the file, including any local modules, becomes part of an implicitly created top-level module that has the same name as the file, without the extension, with the first letter converted to uppercase. For example, consider the following file.

```
// In the file program.fs.
let x = 40
```

This file would be compiled as if it were written in this manner:

```
module Program
let x = 40
```

If you have multiple modules in a file, you must use a local module declaration for each module. If an enclosing namespace is declared, these modules are part of the enclosing namespace. If an enclosing namespace is not declared, the modules become part of the implicitly created top-level module. The following code example shows a code file that contains multiple modules. The compiler implicitly creates a top-level module named `Multiplemodules`, and `MyModule1` and `MyModule2` are nested in that top-level module.

```
// In the file multiplemodules.fs.
// MyModule1
module MyModule1 =
    // Indent all program elements within modules that are declared with an equal sign.
    let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

    // Use a qualified name to access the function.
    // from MyModule1.
    let module2Function x =
        x * (MyModule1.module1Function module2Value)
```

If you have multiple files in a project or in a single compilation, or if you are building a library, you must include a namespace declaration or module declaration at the top of the file. The F# compiler only determines a module name implicitly when there is only one file in a project or compilation command line, and you are creating an application.

The *accessibility-modifier* can be one of the following: `public`, `private`, `internal`. For more information, see Access Control. The default is public.

Referencing Code in Modules

When you reference functions, types, and values from another module, you must either use a qualified name or open the module. If you use a qualified name, you must specify the namespaces, the module, and the identifier for the program element you want. You separate each part of the qualified path with a dot (.), as follows.

`Namespace1.Namespace2.ModuleName.Identifier`

You can open the module or one or more of the namespaces to simplify the code. For more information about opening namespaces and modules, see Import Declarations: The `open` Keyword.

The following code example shows a top-level module that contains all the code up to the end of the file.

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

To use this code from another file in the same project, you either use qualified names or you open the module before you use the functions, as shown in the following examples.

```
// Fully qualify the function name.
let result1 = Arithmetic.add 5 9
// Open the module.
open Arithmetic
let result2 = add 5 9
```

Nested Modules

Modules can be nested. Inner modules must be indented as far as outer module declarations to indicate that they are inner modules, not new modules. For example, compare the following two examples. Module `Z` is an inner module in the following code.

```
module Y =
    let x = 1

    module Z =
        let z = 5
```

But module `Z` is a sibling to module `Y` in the following code.

```
module Y =
    let x = 1

module Z =
    let z = 5
```

Module `Z` is also a sibling module in the following code, because it is not indented as far as other declarations in module `Y`.

```
module Y =
        let x = 1

    module Z =
        let z = 5
```

Finally, if the outer module has no declarations and is followed immediately by another module declaration, the new module declaration is assumed to be an inner module, but the compiler will warn you if the second module definition is not indented farther than the first.

```
// This code produces a warning, but treats Z as a inner module.
module Y =
module Z =
    let z = 5
```

To eliminate the warning, indent the inner module.

```
module Y =
    module Z =
        let z = 5
```

If you want all the code in a file to be in a single outer module and you want inner modules, the outer module does not require the equal sign, and the declarations, including any inner module declarations, that will go in the outer module do not have to be indented. Declarations inside the inner module declarations do have to be indented. The following code shows this case.

```
// The top-level module declaration can be omitted if the file is named
// TopLevel.fs or topLevel.fs, and the file is the only file in an
// application.
module TopLevel

let topLevelX = 5

module Inner1 =
    let inner1X = 1
module Inner2 =
    let inner2X = 5
```

Recursive modules

F# 4.1 introduces the notion of modules which allow for all contained code to be mutually recursive. This is done via `module rec`. Use of `module rec` can alleviate some pains in not being able to write mutually referential code between types and modules. The following is an example of this:

```
module rec RecursiveModule =
    type Orientation = Up | Down
    type PeelState = Peeled | Unpeeled

    // This exception depends on the type below.
    exception DontSqueezeTheBananaException of Banana

    type Banana(orientation : Orientation) =
        member val IsPeeled = false with get, set
        member val Orientation = orientation with get, set
        member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

        member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
        member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on
the exception above.

    module BananaHelpers =
        let peel (b: Banana) =
            let flip (banana: Banana) =
                match banana.Orientation with
                | Up ->
                    banana.Orientation <- Down
                    banana
                | Down -> banana

            let peelSides (banana: Banana) =
                banana.Sides
                |> List.map (function
                             | Unpeeled -> Peeled
                             | Peeled -> Peeled)

            match b.Orientation with
            | Up ->   b |> flip |> peelSides
            | Down -> b |> peelSides
```

Note that the exception `DontSqueezeTheBananaException` and the class `Banana` both refer to each other. Additionally, the module `BananaHelpers` and the class `Banana` also refer to each other. This would not be possible to express in F# if you removed the `rec` keyword from the `RecursiveModule` module.

This capability is also possible in Namespaces with F# 4.1.

See also

- F# Language Reference
- Namespaces
- F# RFC FS-1009 - Allow mutually referential types and modules over larger scopes within files

Import declarations: The `open` keyword

8/5/2022 • 3 minutes to read • Edit Online

An *import declaration* specifies a module or namespace whose elements you can reference without using a fully qualified name.

Syntax

```
open module-or-namespace-name
open type type-name
```

Remarks

Referencing code by using the fully qualified namespace or module path every time can create code that is hard to write, read, and maintain. Instead, you can use the `open` keyword for frequently used modules and namespaces so that when you reference a member of that module or namespace, you can use the short form of the name instead of the fully qualified name. This keyword is similar to the `using` keyword in C#, `using namespace` in Visual C++, and `Imports` in Visual Basic.

The module or namespace provided must be in the same project or in a referenced project or assembly. If it is not, you can add a reference to the project, or use the `-reference` command-line option (or its abbreviation, `-r`). For more information, see Compiler Options.

The import declaration makes the names available in the code that follows the declaration, up to the end of the enclosing namespace, module, or file.

When you use multiple import declarations, they should appear on separate lines.

The following code shows the use of the `open` keyword to simplify code.

```
// Without the import declaration, you must include the full
// path to .NET Framework namespaces such as System.IO.
let writeToFile1 filename (text: string) =
  let stream1 = new System.IO.FileStream(filename, System.IO.FileMode.Create)
  let writer = new System.IO.StreamWriter(stream1)
  writer.WriteLine(text)

// Open a .NET Framework namespace.
open System.IO

// Now you do not have to include the full paths.
let writeToFile2 filename (text: string) =
  let stream1 = new FileStream(filename, FileMode.Create)
  let writer = new StreamWriter(stream1)
  writer.WriteLine(text)

writeToFile2 "file1.txt" "Testing..."
```

The F# compiler does not emit an error or warning when ambiguities occur when the same name occurs in more than one open module or namespace. When ambiguities occur, F# gives preference to the more recently opened module or namespace. For example, in the following code, `empty` means `Seq.empty`, even though `empty` is located in both the `List` and `Seq` modules.

```
open List
open Seq
printfn %"{empty}"
```

Therefore, be careful when you open modules or namespaces such as `List` or `Seq` that contain members that have identical names; instead, consider using the qualified names. You should avoid any situation in which the code is dependent upon the order of the import declarations.

Open type declarations

F# supports `open` on a type like so:

```
open type System.Math
PI
```

This will expose all accessible static fields and members on the type.

You can also `open` F#-defined record and discriminated union types to expose static members. In the case of discriminated unions, you can also expose the union cases. This can be helpful for accessing union cases in a type declared inside of a module that you may not want to open, like so:

```
module M =
    type DU = A | B | C

    let someOtherFunction x = x + 1

// Open only the type inside the module
open type M.DU

printfn "%A" A
```

Namespaces That Are Open by Default

Some namespaces are so frequently used in F# code that they are opened implicitly without the need of an explicit import declaration. The following table shows the namespaces that are open by default.

NAMESPACE	DESCRIPTION
`FSharp.Core`	Contains basic F# type definitions for built-in types such as `int` and `float`.
`FSharp.Core.Operators`	Contains basic arithmetic operations such as `+` and `*`.
`FSharp.Collections`	Contains immutable collection classes such as `List` and `Array`.
`FSharp.Control`	Contains types for control constructs such as lazy evaluation and async expressions.
`FSharp.Text`	Contains functions for formatted IO, such as the `printf` function.

AutoOpen Attribute

You can apply the `AutoOpen` attribute to an assembly if you want to automatically open a namespace or module when the assembly is referenced. You can also apply the `AutoOpen` attribute to a module to automatically open that module when the parent module or namespace is opened. For more information, see AutoOpenAttribute.

RequireQualifiedAccess Attribute

Some modules, records, or union types may specify the `RequireQualifiedAccess` attribute. When you reference elements of those modules, records, or unions, you must use a qualified name regardless of whether you include an import declaration. If you use this attribute strategically on types that define commonly used names, you help avoid name collisions and thereby make code more resilient to changes in libraries. For more information, see RequireQualifiedAccessAttribute.

See also

- F# Language Reference
- Namespaces
- Modules

Signatures

8/5/2022 • 4 minutes to read • Edit Online

A signature file contains information about the public signatures of a set of F# program elements, such as types, namespaces, and modules. It can be used to specify the accessibility of these program elements.

Remarks

For each F# code file, you can have a *signature file*, which is a file that has the same name as the code file but with the extension .fsi instead of .fs. Signature files can also be added to the compilation command-line if you are using the command line directly. To distinguish between code files and signature files, code files are sometimes referred to as *implementation files*. In a project, the signature file should precede the associated code file.

A signature file describes the namespaces, modules, types, and members in the corresponding implementation file. You use the information in a signature file to specify what parts of the code in the corresponding implementation file can be accessed from code outside the implementation file, and what parts are internal to the implementation file. The namespaces, modules, and types that are included in the signature file must be a subset of the namespaces, modules, and types that are included in the implementation file. With some exceptions noted later in this topic, those language elements that are not listed in the signature file are considered private to the implementation file. If no signature file is found in the project or command line, the default accessibility is used.

For more information about the default accessibility, see Access Control.

In a signature file, you do not repeat the definition of the types and the implementations of each method or function. Instead, you use the signature for each method and function, which acts as a complete specification of the functionality that is implemented by a module or namespace fragment. The syntax for a type signature is the same as that used in abstract method declarations in interfaces and abstract classes, and is also shown by IntelliSense and by the F# interpreter fsi.exe when it displays correctly compiled input.

If there is not enough information in the type signature to indicate whether a type is sealed, or whether it is an interface type, you must add an attribute that indicates the nature of the type to the compiler. Attributes that you use for this purpose are described in the following table.

ATTRIBUTE	DESCRIPTION
`[<Sealed>]`	For a type that has no abstract members, or that should not be extended.
`[<Interface>]`	For a type that is an interface.

The compiler produces an error if the attributes are not consistent between the signature and the declaration in the implementation file.

Use the keyword `val` to create a signature for a value or function value. The keyword `type` introduces a type signature.

You can generate a signature file by using the `--sig` compiler option. Generally, you do not write .fsi files manually. Instead, you generate .fsi files by using the compiler, add them to your project, if you have one, and edit them by removing methods and functions that you do not want to be accessible.

There are several rules for type signatures:

- Type abbreviations in an implementation file must not match a type without an abbreviation in a signature file.

- Records and discriminated unions must expose either all or none of their fields and constructors, and the order in the signature must match the order in the implementation file. Classes can reveal some, all, or none of their fields and methods in the signature.

- Classes and structures that have constructors must expose the declarations of their base classes (the `inherits` declaration). Also, classes and structures that have constructors must expose all of their abstract methods and interface declarations.

- Interface types must reveal all their methods and interfaces.

The rules for value signatures are as follows:

- Modifiers for accessibility (`public`, `internal`, and so on) and the `inline` and `mutable` modifiers in the signature must match those in the implementation.

- The number of generic type parameters (either implicitly inferred or explicitly declared) must match, and the types and type constraints in generic type parameters must match.

- If the `Literal` attribute is used, it must appear in both the signature and the implementation, and the same literal value must be used for both.

- The pattern of parameters (also known as the *arity*) of signatures and implementations must be consistent.

- If parameter names in a signature file differ from the corresponding implementation file, the name in the signature file will be used instead, which may cause issues when debugging or profiling. If you wish to be notified of such mismatches, enable warning 3218 in your project file or when invoking the compiler (see `--warnon` under Compiler Options).

The following code example shows an example of a signature file that has namespace, module, function value, and type signatures together with the appropriate attributes. It also shows the corresponding implementation file.

```
// Module1.fsi

namespace Library1
  module Module1 =
    val function1 : int -> int
    type Type1 =
        new : unit -> Type1
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Sealed>]
    type Type2 =
        new : unit -> Type2
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

The following code shows the implementation file.

```
namespace Library1

module Module1 =

    let function1 x = x + 1


    type Type1() =
        member type1.method1() =
            printfn "type1.method1"
        member type1.method2() =
            printfn "type1.method2"


    [<Sealed>]
    type Type2() =
        member type2.method1() =
            printfn "type2.method1"
        member type2.method2() =
            printfn "type2.method2"

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

See also

- F# Language Reference
- Access Control
- Compiler Options

Access Control

8/5/2022 • 3 minutes to read • Edit Online

Access control refers to declaring which clients can use certain program elements, such as types, methods, and functions.

Basics of Access Control

In F#, the access control specifiers `public`, `internal`, and `private` can be applied to modules, types, methods, value definitions, functions, properties, and explicit fields.

- `public` indicates that the entity can be accessed by all callers.
- `internal` indicates that the entity can be accessed only from the same assembly.
- `private` indicates that the entity can be accessed only from the enclosing type or module.

> **NOTE**
>
> The access specifier `protected` is not used in F#, although it is acceptable if you are using types authored in languages that do support `protected` access. Therefore, if you override a protected method, your method remains accessible only within the class and its descendents.

In general, the specifier is put in front of the name of the entity, except when a `mutable` or `inline` specifier is used, which appear after the access control specifier.

If no access specifier is used, the default is `public`, except for `let` bindings in a type, which are always `private` to the type.

Signatures in F# provide another mechanism for controlling access to F# program elements. Signatures are not required for access control. For more information, see Signatures.

Rules for Access Control

Access control is subject to the following rules:

- Inheritance declarations (that is, the use of `inherit` to specify a base class for a class), interface declarations (that is, specifying that a class implements an interface), and abstract members always have the same accessibility as the enclosing type. Therefore, an access control specifier cannot be used on these constructs.
- Accessibility for individual cases in a discriminated union is determined by the accessibility of the discriminated union itself. That is, a particular union case is no less accessible than the union itself.
- Accessibility for individual fields of a record type is determined by the accessibility of the record itself. That is, a particular record label is no less accessible than the record itself.

Example

The following code illustrates the use of access control specifiers. There are two files in the project, `Module1.fs` and `Module2.fs`. Each file is implicitly a module. Therefore, there are two modules, `Module1` and `Module2`. A private type and an internal type are defined in `Module1`. The private type cannot be accessed from `Module2`, but the internal type can.

```
// Module1.fs

module Module1

// This type is not usable outside of this file
type private MyPrivateType() =
   // x is private since this is an internal let binding
   let x = 5
   // X is private and does not appear in the QuickInfo window
   // when viewing this type in the Visual Studio editor
   member private this.X() = 10
   member this.Z() = x * 100

type internal MyInternalType() =
   let x = 5
   member private this.X() = 10
   member this.Z() = x * 100

// Top-level let bindings are public by default,
// so "private" and "internal" are needed here since a
// value cannot be more accessible than its type.
let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

// let bindings at the top level are public by default,
// so result1 and result2 are public.
let result1 = myPrivateObj.Z
let result2 = myInternalObj.Z
```

The following code tests the accessibility of the types created in `Module1.fs`.

```
// Module2.fs
module Module2

open Module1

// The following line is an error because private means
// that it cannot be accessed from another file or module
// let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

let result = myInternalObj.Z
```

See also

- F# Language Reference
- Signatures

Document your code with XML comments

8/5/2022 • 6 minutes to read • Edit Online

You can produce documentation from triple-slash (///) code comments in F#. XML comments can precede declarations in code files (.fs) or signature (.fsi) files.

XML documentation comments are a special kind of comment, added above the definition of any user-defined type or member. They are special because they can be processed by the compiler to generate an XML documentation file at compile time. The compiler-generated XML file can be distributed alongside your .NET assembly so that IDEs can use tooltips to show quick information about types or members. Additionally, the XML file can be run through tools like fsdocs to generate API reference websites.

XML documentation comments, like all other comments, are ignored by the compiler, unless the options described below are enabled to check the validity and completeness of comments at compile time.

You can generate the XML file at compile time by doing one of the following:

- You can add a `GenerateDocumentationFile` element to the `<PropertyGroup>` section of your `.fsproj` project file, which generates an XML file in the project directory with the same root filename as the assembly. For example:

  ```
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  ```

  For more information, see GenerateDocumentationFile property.

- If you are developing an application using Visual Studio, right-click on the project and select **Properties**. In the properties dialog, select the **Build** tab, and check **XML documentation file**. You can also change the location to which the compiler writes the file.

There are two ways to write XML documentation comments: with and without XML tags. Both use triple-slash comments.

Comments without XML tags

If a `///` comment does not start with a `<`, then the entire comment text is taken as the summary documentation for the code construct that immediately follows. Use this method when you want to write only a brief summary for each construct.

The comment is encoded to XML during documentation preparation, so characters such as `<`, `>`, and `&` need not be escaped. If you don't specify a summary tag explicitly, you should not specify other tags, such as **param** or **returns** tags.

The following example shows the alternative method, without XML tags. In this example, the entire text in the comment is considered a summary.

```
/// Creates a new string whose characters are the result of applying
/// the function mapping to each of the characters of the input string
/// and concatenating the resulting strings.
val collect : (char -> string) -> string -> string
```

Comments with XML tags

If a comment body begins with `<` (normally `<summary>`), then it is treated as an XML formatted comment body using XML tags. This second enables you to specify separate notes for a short summary, additional remarks, documentation for each parameter and type parameter and exceptions thrown, and a description of the return value.

The following is a typical XML documentation comment in a signature file:

```
/// <summary>Builds a new string whose characters are the results of applying the function <c>mapping</c>
/// to each of the characters of the input string and concatenating the resulting
/// strings.</summary>
/// <param name="mapping">The function to produce a string from each character of the input string.</param>
///<param name="str">The input string.</param>
///<returns>The concatenated string.</returns>
///<exception cref="System.ArgumentNullException">Thrown when the input string is null.</exception>
val collect : (char -> string) -> string -> string
```

Recommended Tags

If you are using XML tags, the following table describes the outer tags recognized in F# XML code comments.

TAG SYNTAX	DESCRIPTION
`<summary>` ***text*** `</summary>`	Specifies that *text* is a brief description of the program element. The description is usually one or two sentences.
`<remarks>` ***text*** `</remarks>`	Specifies that *text* contains supplementary information about the program element.
`<param name="` ***name*** `">` ***description*** `</param>`	Specifies the name and description for a function or method parameter.
`<typeparam name="` ***name*** `">` ***description*** `</typeparam>`	Specifies the name and description for a type parameter.
`<returns>` ***text*** `</returns>`	Specifies that *text* describes the return value of a function or method.
`<exception cref="` ***type*** `">` ***description*** `</exception>`	Specifies the type of exception that can be generated and the circumstances under which it is thrown.
`<seealso cref="` ***reference*** `"/>`	Specifies a See Also link to the documentation for another type. The *reference* is the name as it appears in the XML documentation file. See Also links usually appear at the bottom of a documentation page.

The following table describes the tags for use inside description sections:

TAG SYNTAX	DESCRIPTION
`<para>` ***text*** `</para>`	Specifies a paragraph of text. This is used to separate text inside the **remarks** tag.
`<code>` ***text*** `</code>`	Specifies that *text* is multiple lines of code. This tag can be used by documentation generators to display text in a font that is appropriate for code.

TAG SYNTAX	DESCRIPTION
`<paramref name="` *name* `"/>`	Specifies a reference to a parameter in the same documentation comment.
`<typeparamref name="` *name* `"/>`	Specifies a reference to a type parameter in the same documentation comment.
`<c>` *text* `</c>`	Specifies that *text* is inline code. This tag can be used by documentation generators to display text in a font that is appropriate for code.
`<see cref="` *reference* `">` *text* `</see>`	Specifies an inline link to another program element. The *reference* is the name as it appears in the XML documentation file. The *text* is the text shown in the link.

User-defined tags

The previous tags represent those that are recognized by the F# compiler and typical F# editor tooling. However, a user is free to define their own tags. Tools like fsdocs bring support for extra tags like <namespacedoc>. Custom or in-house documentation generation tools can also be used with the standard tags and multiple output formats from HTML to PDF can be supported.

Compile-time checking

When `--warnon:3390` is enabled, the compiler verifies the syntax of the XML and the parameters referred to in `<param>` and `<paramref>` tags.

Documenting F# Constructs

F# constructs such as modules, members, union cases, and record fields are documented by a `///` comment immediately prior to their declaration. If needed, implicit constructors of classes are documented by giving a `///` comment prior to the argument list. For example:

```
/// This is the type
type SomeType
      /// This is the implicit constructor
      (a: int, b: int) =

    /// This is the member
    member _.Sum() = a + b
```

Limitations

Some features of XML documentation in C# and other .NET languages are not supported in F#.

- In F#, cross-references must use the full XML signature of the corresponding symbol, for example `cref="T:System.Console"`. Simple C#-style cross-references such as `cref="Console"` are not elaborated to full XML signatures and these elements are not checked by the F# compiler. Some documentation tooling may allow the use of these cross-references by subsequent processing, but the full signatures should be used.
- The tags `<include>`, `<inheritdoc>` are not supported by the F# compiler. No error is given if they are used, but they are simply copied to the generated documentation file without otherwise affecting the documentation generated.

- Cross-references are not checked by the F# compiler, even when `-warnon:3390` is used.
- The names used in the tags `<typeparam>` and `<typeparamref>` are not checked by the F# compiler, even when `--warnon:3390` is used.
- No warnings are given if documentation is missing, even when `--warnon:3390` is used.

Recommendations

Documenting code is recommended for many reasons. What follows are some best practices, general use case scenarios, and things that you should know when using XML documentation tags in your F# code.

- Enable the option `--warnon:3390` in your code to help ensure your XML documentation is valid XML.
- Consider adding signature files to separate long XML documentation comments from your implementation.
- For the sake of consistency, all publicly visible types and their members should be documented. If you must do it, do it all.
- At a bare minimum, modules, types, and their members should have a plain `///` comment or `<summary>` tag. This will show in an autocompletion tooltip window in F# editing tools.
- Documentation text should be written using complete sentences ending with full stops.

See also

- C# XML Documentation Comments (C# Programming Guide).
- F# Language Reference
- Compiler Options

Console Applications

8/5/2022 • 2 minutes to read • Edit Online

In this article, you learn how to structure an F# console application.

Implicit Entry Point

By default, F# applications use an implicit entry point. For example, for the following program, the entry point is implicit and, when the program is run, the code executes from the first line to the last:

```
open System

let printSomeText() =
    let text = "Hello" + "World"
    printfn $"text = {text}"

let showCommandLineArgs() =
    for arg in Environment.GetCommandLineArgs() do
        printfn $"arg = {arg}"

printSomeText()
showCommandLineArgs()
exit 100
```

Explicit Entry Point

If you want, you can use an explicit entry point. This is usually done for one or all of the following reasons:

- You prefer to access the command-line arguments via an argument passed to a function, rather than using `System.Environment.GetCommandLineArgs()`.
- You want to return an error code via a return result, rather than using `exit`.
- You want to unit test the code in the last file of your console application.

The following example illustrates a simple `main` function with an explicit entry point.

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    // Return 0. This indicates success.
    0
```

When this code is executed with the command line `EntryPoint.exe 1 2 3`, the output is as follows.

```
Arguments passed to function : [|"1"; "2"; "3"|]
```

Syntax

```
[<EntryPoint>]
let-function-binding
```

Remarks

In the previous syntax, *let-function-binding* is the definition of a function in a `let` binding.

The entry point to a program that is compiled as an executable file is where execution formally starts. You specify the entry point to an F# application by applying the `EntryPoint` attribute to the program's `main` function. This function (created by using a `let` binding) must be the last function in the last compiled file. The last compiled file is the last file in the project or the last file that is passed to the command line.

The entry point function has type `string array -> int`. The arguments provided on the command line are passed to the `main` function in the array of strings. The first element of the array is the first argument; the name of the executable file is not included in the array, as it is in some other languages. The return value is used as the exit code for the process. Zero usually indicates success; nonzero values indicate an error. There is no convention for the specific meaning of nonzero return codes; the meanings of the return codes are application-specific.

See also

- Tour of F#
- Functions
- let Bindings

Query expressions

8/5/2022 • 41 minutes to read • Edit Online

Query expressions enable you to query a data source and put the data in a desired form. Query expressions provide support for LINQ in F#.

Syntax

```
query { expression }
```

Remarks

Query expressions are a type of computation expression similar to sequence expressions. Just as you specify a sequence by providing code in a sequence expression, you specify a set of data by providing code in a query expression. In a sequence expression, the `yield` keyword identifies data to be returned as part of the resulting sequence. In query expressions, the `select` keyword performs the same function. In addition to the `select` keyword, F# also supports a number of query operators that are much like the parts of a SQL SELECT statement. Here is an example of a simple query expression, along with code that connects to the Northwind OData source.

```
// Use the OData type provider to create types that can be used to access the Northwind database.
// Add References to FSharp.Data.TypeProviders and System.Data.Services.Client
open Microsoft.FSharp.Data.TypeProviders

type Northwind = ODataService<"http://services.odata.org/Northwind/Northwind.svc">
let db = Northwind.GetDataContext()

// A query expression.
let query1 =
    query {
        for customer in db.Customers do
            select customer
    }

// Print results
query1
|> Seq.iter (fun customer -> printfn "Company: %s Contact: %s" customer.CompanyName customer.ContactName)
```

In the previous code example, the query expression is in curly braces. The meaning of the code in the expression is, return every customer in the Customers table in the database in the query results. Query expressions return a type that implements IQueryable<T> and IEnumerable<T>, and so they can be iterated using the Seq module as the example shows.

Every computation expression type is built from a builder class. The builder class for the query computation expression is `QueryBuilder`. For more information, see Computation Expressions and QueryBuilder Class.

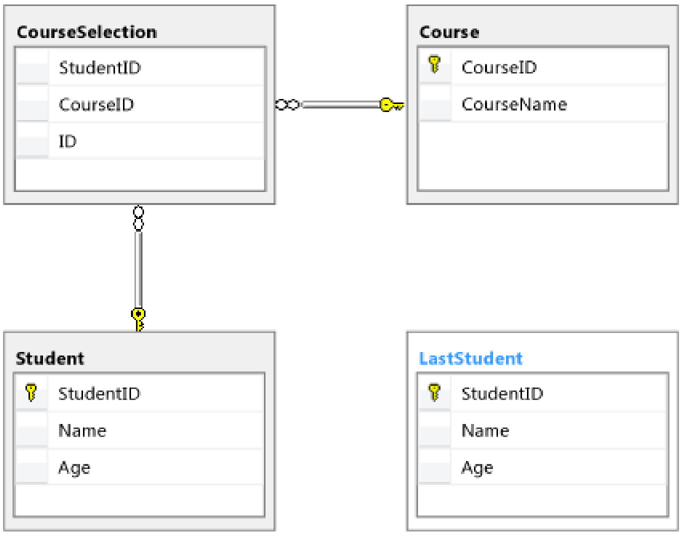
Query Operators

Query operators enable you to specify the details of the query, such as to put criteria on records to be returned, or specify the sorting order of results. The query source must support the query operator. If you attempt to use an unsupported query operator, `System.NotSupportedException` will be thrown.

Only expressions that can be translated to SQL are allowed in query expressions. For example, no function calls are allowed in the expressions when you use the `where` query operator.

Table 1 shows available query operators. In addition, see Table2, which compares SQL queries and the equivalent F# query expressions later in this topic. Some query operators aren't supported by some type providers. In particular, the OData type provider is limited in the query operators that it supports due to limitations in OData.

This table assumes a database in the following form:



The code in the tables that follow also assumes the following database connection code. Projects should add references to System.Data, System.Data.Linq, and FSharp.Data.TypeProviders assemblies. The code that creates this database is included at the end of this topic.

```
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq
open Microsoft.FSharp.Linq

type schema = SqlDataConnection< @"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;" >

let db = schema.GetDataContext()

// Needed for some query operator examples:
let data = [ 1; 5; 7; 11; 18; 21]
```

Table 1. Query Operators

OPERATOR	DESCRIPTION

`contains`	Determines whether the selected elements include a specified element.
	`query {`
	`    for student in db.Student do`
	`    select student.Age.Value`
	`    contains 11`
	`}`

`count`	Returns the number of selected elements.
	
	`query {`
	`    for student in db.Student do`
	`    select student`
	`    count`
	`}`

`last`	Selects the last element of those selected so far.
	
	`query {`
	`    for number in data do`
	`    last`
	`}`

`lastOrDefault`	Selects the last element of those selected so far, or a default value if no element is found.
	
	`query {`
	`    for number in data do`
	`    where (number < 0)`
	`    lastOrDefault`
	`}`

`exactlyOne`	Selects the single, specific element selected so far. If multiple elements are present, an exception is thrown.
	
	`query {`
	`    for student in db.Student do`
	`    where (student.StudentID = 1)`
	`    select student`
	`    exactlyOne`
	`}`

`exactlyOneOrDefault`	Selects the single, specific element of those selected so far, or a default value if that element is not found.
	
	`query {`
	`    for student in db.Student do`
	`    where (student.StudentID = 1)`
	`    select student`
	`    exactlyOneOrDefault`
	`}`

`headOrDefault`	Selects the first element of those selected so far, or a default value if the sequence contains no elements.
	
	`query {`
	`    for student in db.Student do`
	`    select student`
	`    headOrDefault`
	`}`

`select`	Projects each of the elements selected so far.
	
	`query {`
	`    for student in db.Student do`
	`    select student`
	`}`

`where`	Selects elements based on a specified predicate.
	
	`query {`
	`    for student in db.Student do`
	`    where (student.StudentID > 4)`
	`    select student`
	`}`

`minBy`	Selects a value for each element selected so far and returns the minimum resulting value.
	
	`query {`
	`    for student in db.Student do`
	`    minBy student.StudentID`
	`}`

`maxBy`	Selects a value for each element selected so far and returns the maximum resulting value.
	
	`query {`
	`    for student in db.Student do`
	`    maxBy student.StudentID`
	`}`

`groupBy`	Groups the elements selected so far according to a specified key selector.
	
	`query {`
	`    for student in db.Student do`
	`    groupBy student.Age into g`
	`    select (g.Key, g.Count())`
	`}`

<table>
<tr><td><code>sortBy</code></td><td>Sorts the elements selected so far in ascending order by the given sorting key.
<pre><code>query {
    for student in db.Student do
    sortBy student.Name
    select student
}</code></pre></td></tr>
<tr><td><code>sortByDescending</code></td><td>Sorts the elements selected so far in descending order by the given sorting key.
<pre><code>query {
    for student in db.Student do
    sortByDescending student.Name
    select student
}</code></pre></td></tr>
<tr><td><code>thenBy</code></td><td>Performs a subsequent ordering of the elements selected so far in ascending order by the given sorting key. This operator may only be used after a <code>sortBy</code>, <code>sortByDescending</code>, <code>thenBy</code>, or <code>thenByDescending</code>.
<pre><code>query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenBy student.Name
    select student
}</code></pre></td></tr>
<tr><td><code>thenByDescending</code></td><td>Performs a subsequent ordering of the elements selected so far in descending order by the given sorting key. This operator may only be used after a <code>sortBy</code>, <code>sortByDescending</code>, <code>thenBy</code>, or <code>thenByDescending</code>.
<pre><code>query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenByDescending student.Name
    select student
}</code></pre></td></tr>
</table>

`groupValBy`	Selects a value for each element selected so far and groups the elements by the given key.
	
	`query {`
	`    for student in db.Student do`
	`    groupValBy student.Name student.Age into g`
	`    select (g, g.Key, g.Count())`
	`}`

`join`	Correlates two sets of selected values based on matching keys. Note that the order of the keys around the = sign in a join expression is significant. In all joins, if the line is split after the `->` symbol, the indentation must be indented at least as far as the keyword `for`.
	
	`query {`
	`    for student in db.Student do`
	`    join selection in db.CourseSelection`
	`        on (student.StudentID =`
	`selection.StudentID)`
	`    select (student, selection)`
	`}`

`groupJoin`	Correlates two sets of selected values based on matching keys and groups the results. Note that the order of the keys around the = sign in a join expression is significant.
	
	`query {`
	`    for student in db.Student do`
	`    groupJoin courseSelection in`
	`db.CourseSelection`
	`        on (student.StudentID =`
	`courseSelection.StudentID) into g`
	`    for courseSelection in g do`
	`    join course in db.Course`
	`        on (courseSelection.CourseID =`
	`course.CourseID)`
	`    select (student.Name, course.CourseName)`
	`}`

`leftOuterJoin`	Correlates two sets of selected values based on matching keys and groups the results. If any group is empty, a group with a single default value is used instead. Note that the order of the keys around the = sign in a join expression is significant.
	
	`query {`
	`    for student in db.Student do`
	`    leftOuterJoin selection in db.CourseSelection`
	`        on (student.StudentID = selection.StudentID) into result`
	`    for selection in result.DefaultIfEmpty() do`
	`    select (student, selection)`
	`}`

`sumByNullable`	Selects a nullable value for each element selected so far and returns the sum of these values. If any nullable does not have a value, it is ignored.
	
	`query {`
	`    for student in db.Student do`
	`    sumByNullable student.Age`
	`}`

`minByNullable`	Selects a nullable value for each element selected so far and returns the minimum of these values. If any nullable does not have a value, it is ignored.
	
	`query {`
	`    for student in db.Student do`
	`    minByNullable student.Age`
	`}`

`maxByNullable`	Selects a nullable value for each element selected so far and returns the maximum of these values. If any nullable does not have a value, it is ignored.
	
	`query {`
	`    for student in db.Student do`
	`    maxByNullable student.Age`
	`}`

`averageByNullable`	Selects a nullable value for each element selected so far and returns the average of these values. If any nullable does not have a value, it is ignored.
	
	`query {`
	`    for student in db.Student do`
	`    averageByNullable (Nullable.float`
	`student.Age)`
	`}`

`averageBy`	Selects a value for each element selected so far and returns the average of these values.
	
	`query {`
	`    for student in db.Student do`
	`    averageBy (float student.StudentID)`
	`}`

`distinct`	Selects distinct elements from the elements selected so far.
	
	`query {`
	`    for student in db.Student do`
	`    join selection in db.CourseSelection`
	`        on (student.StudentID =`
	`selection.StudentID)`
	`    distinct`
	`}`

`exists`	Determines whether any element selected so far satisfies a condition.
	
	`query {`
	`    for student in db.Student do`
	`    where`
	`        (query {`
	`            for courseSelection in`
	`db.CourseSelection do`
	`            exists (courseSelection.StudentID =`
	`student.StudentID) })`
	`    select student`
	`}`

`find`	Selects the first element selected so far that satisfies a specified condition.
	
	`query {`
	`    for student in db.Student do`
	`    find (student.Name = "Abercrombie, Kim")`
	`}`

`all`	Determines whether all elements selected so far satisfy a condition.
	<pre>query {
	    for student in db.Student do
	    all (SqlMethods.Like(student.Name, "%,%"))
	}</pre>

`head`	Selects the first element from those selected so far.
	<pre>query {
	    for student in db.Student do
	    head
	}</pre>

`nth`	Selects the element at a specified index amongst those selected so far.
	<pre>query {
	    for numbers in data do
	    nth 3
	}</pre>

`skip`	Bypasses a specified number of the elements selected so far and then selects the remaining elements.
	<pre>query {
	    for student in db.Student do
	    skip 1
	}</pre>

`skipWhile`	Bypasses elements in a sequence as long as a specified condition is true and then selects the remaining elements.
	<pre>query {
	    for number in data do
	    skipWhile (number < 3)
	    select student
	}</pre>

`sumBy`	Selects a value for each element selected so far and returns the sum of these values.
	<pre>query {
	    for student in db.Student do
	    sumBy student.StudentID
	}</pre>

`take`	Selects a specified number of contiguous elements from those selected so far.
	
	`query {`
	`    for student in db.Student do`
	`    select student`
	`    take 2`
	`}`

`takeWhile`	Selects elements from a sequence as long as a specified condition is true, and then skips the remaining elements.
	
	`query {`
	`    for number in data do`
	`    takeWhile (number < 10)`
	`}`

`sortByNullable`	Sorts the elements selected so far in ascending order by the given nullable sorting key.
	
	`query {`
	`    for student in db.Student do`
	`    sortByNullable student.Age`
	`    select student`
	`}`

`sortByNullableDescending`	Sorts the elements selected so far in descending order by the given nullable sorting key.
	
	`query {`
	`    for student in db.Student do`
	`    sortByNullableDescending student.Age`
	`    select student`
	`}`

`thenByNullable`	Performs a subsequent ordering of the elements selected so far in ascending order by the given nullable sorting key. This operator may only be used immediately after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`, or their nullable variants.
	
	`query {`
	`    for student in db.Student do`
	`    sortBy student.Name`
	`    thenByNullable student.Age`
	`    select student`
	`}`

`thenByNullableDescending`	Performs a subsequent ordering of the elements selected so far in descending order by the given nullable sorting key. This operator may only be used immediately after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`, or their nullable variants.
	
	`query {`
	`    for student in db.Student do`
	`    sortBy student.Name`
	`    thenByNullableDescending student.Age`
	`    select student`
	`}`

Comparison of Transact-SQL and F# Query Expressions

The following table shows some common Transact-SQL queries and their equivalents in F#. The code in this table also assumes the same database as the previous table and the same initial code to set up the type provider.

Table 2. Transact-SQL and F# Query Expressions

TRANSACT-SQL (NOT CASE SENSITIVE)	F# QUERY EXPRESSION (CASE SENSITIVE)

Select all fields from table.
 	
`SELECT * FROM Student`
	`// All students.`
	`query {`
	`    for student in db.Student do`
	`    select student`
	`}`

Count records in a table.
 	
`SELECT COUNT( * ) FROM Student`
	`// Count of students.`
	`query {`
	`    for student in db.Student do`
	`    count`
	`}`

`EXISTS`	

```
SELECT * FROM Student
WHERE EXISTS
  (SELECT * FROM CourseSelection
   WHERE CourseSelection.StudentID =
Student.StudentID)
```

```
// Find students who have signed up at least
one course.
query {
    for student in db.Student do
    where
        (query {
            for courseSelection in
db.CourseSelection do
            exists (courseSelection.StudentID =
student.StudentID) })
    select student
}
```

Grouping

```
SELECT Student.Age, COUNT( * ) FROM Student
GROUP BY Student.Age
```

```
// Group by age and count.
query {
    for n in db.Student do
    groupBy n.Age into g
    select (g.Key, g.Count())
}
// OR
query {
    for n in db.Student do
    groupValBy n.Age n.Age into g
    select (g.Key, g.Count())
}
```

Grouping with condition.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING student.Age > 10
```

```
// Group students by age where age > 10.
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Key.HasValue && g.Key.Value > 10)
    select (g.Key, g.Count())
}
```

Grouping with count condition.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
```

```
// Group students by age and count number of
students
// at each age with more than 1 student.
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
```

Grouping, counting, and summing.

```
SELECT Student.Age, COUNT( * ),
SUM(Student.Age) as total
FROM Student
GROUP BY Student.Age
```

```
// Group students by age and sum ages.
query {
    for student in db.Student do
    groupBy student.Age into g
    let total =
        query {
            for student in g do
            sumByNullable student.Age
        }
    select (g.Key, g.Count(), total)
}
```

Grouping, counting, and ordering by count.

```
SELECT Student.Age, COUNT( * ) as myCount
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
ORDER BY COUNT( * ) DESC
```

```
// Group students by age, count number of
students
// at each age, and display all with count > 1
// in descending order of count.
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
```

`IN` a set of specified values

```
SELECT *
FROM Student
WHERE Student.StudentID IN (1, 2, 5, 10)
```

```
// Select students where studentID is one of a
given list.
let idQuery =
    query {
        for id in [1; 2; 5; 10] do
        select id
    }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
```

`LIKE` and `TOP`.

```
-- '_e%' matches strings where the second
character is 'e'
SELECT TOP 2 * FROM Student
WHERE Student.Name LIKE '_e%'
```

```
// Look for students with Name match _e%
pattern and take first two.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name,
"_e%") )
    select student
    take 2
}
```

`LIKE` with pattern match set.

```
-- '[abc]%' matches strings where the first
character is
-- 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[abc]%'
```

```
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[abc]%") )
    select student
}
```

`LIKE` with set exclusion pattern.

```
-- '[^abc]%' matches strings where the first
character is
-- not 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
// Look for students with name matching
[^abc]%% pattern.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[^abc]%") )
    select student
}
```

`LIKE` on one field, but select a different field.

```
SELECT StudentID AS ID FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%")
)
    select n.StudentID
}
```

`LIKE`, with substring search.

```
SELECT * FROM Student
WHERE Student.Name like '%A%'
```

```
// Using Contains as a query filter.
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
```

Simple `JOIN` with two tables.

```
SELECT * FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join Student and CourseSelection tables.
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID =
selection.StudentID)
    select (student, selection)
}
```

`LEFT JOIN` with two tables.

```
SELECT * FROM Student
LEFT JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
//Left Join Student and CourseSelection tables.
query {
    for student in db.Student do
    leftOuterJoin selection in
db.CourseSelection
        on (student.StudentID =
selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
```

`JOIN` with `COUNT`

```
SELECT COUNT( * ) FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join with count.
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
```

`DISTINCT`
`SELECT DISTINCT StudentID FROM CourseSelection`
	`// Join with distinct.`
	`query {`
	`    for student in db.Student do`
	`    join selection in db.CourseSelection`
	`        on (student.StudentID =`
	`selection.StudentID)`
	`    distinct`
	`}`

Distinct count.
`SELECT DISTINCT COUNT(StudentID) FROM`
`CourseSelection`
	`// Join with distinct and count.`
	`query {`
	`    for n in db.Student do`
	`    join e in db.CourseSelection`
	`        on (n.StudentID = e.StudentID)`
	`    distinct`
	`    count`
	`}`

`BETWEEN`
`SELECT * FROM Student`
`WHERE Student.Age BETWEEN 10 AND 15`
	`// Selecting students with ages between 10 and`
	`15.`
	`query {`
	`    for student in db.Student do`
	`    where (student.Age ?>= 10 && student.Age ?<`
	`15)`
	`    select student`
	`}`

`OR`
`SELECT * FROM Student`
`WHERE Student.Age = 11 OR Student.Age = 12`
	`// Selecting students with age that's either 11`
	`or 12.`
	`query {`
	`    for student in db.Student do`
	`    where (student.Age.Value = 11 ||`
	`student.Age.Value = 12)`
	`    select student`
	`}`

`OR` with ordering
`SELECT * FROM Student`
`WHERE Student.Age = 12 OR Student.Age = 13`
`ORDER BY Student.Age DESC`
	`// Selecting students in a certain age range`
	`and sorting.`
	`query {`
	`    for n in db.Student do`
	`    where (n.Age.Value = 12 || n.Age.Value =`
	`13)`
	`    sortByNullableDescending n.Age`
	`    select n`
	`}`

`TOP`, `OR`, and ordering.
<pre>SELECT TOP 2 student.Name FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
ORDER BY Student.Name DESC</pre>
	<pre>// Selecting students with certain ages,
	// taking account of the possibility of nulls.
	query {
	    for student in db.Student do
	    where
	        ((student.Age.HasValue &&
	student.Age.Value = 11) ||
	         (student.Age.HasValue &&
	student.Age.Value = 12))
	    sortByDescending student.Name
	    select student.Name
	    take 2
	}</pre>

`UNION` of two queries.
<pre>SELECT * FROM Student
UNION
SELECT * FROM lastStudent</pre>
	<pre>let query1 =
	    query {
	        for n in db.Student do
	        select (n.Name, n.Age)
	    }
	
	let query2 =
	    query {
	        for n in db.LastStudent do
	        select (n.Name, n.Age)
	    }
	
	query2.Union (query1)</pre>

Intersection of two queries.
<pre>SELECT * FROM Student
INTERSECT
SELECT * FROM LastStudent</pre>
	<pre>let query1 =
	    query {
	        for n in db.Student do
	        select (n.Name, n.Age)
	    }
	
	let query2 =
	    query {
	        for n in db.LastStudent do
	        select (n.Name, n.Age)
	    }
	
	query1.Intersect(query2)</pre>

`CASE` condition.	

```
SELECT student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special value.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Multiple cases.

```
SELECT Student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  WHEN 0 THEN 1000
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special values.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue &&
student.Age.Value = 0 then
             (student.StudentID,
System.Nullable<int>(1000), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Multiple tables.

```
SELECT * FROM Student, Course
```

```
// Multiple table select.
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
```

Multiple joins.

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Multiple joins.
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID =
course.CourseID)
    select (student.Name, course.CourseName)
}
```

Multiple left outer joins.

```
SELECT Student.Name, Course.CourseName
FROM Student
LEFT OUTER JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
LEFT OUTER JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Using leftOuterJoin with multiple joins.
query {
    for student in db.Student do
    leftOuterJoin courseSelection in
db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty()
do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID =
course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
```

The following code can be used to create the sample database for these examples.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

USE [master];
GO

IF EXISTS (SELECT * FROM sys.databases WHERE name = 'MyDatabase')
DROP DATABASE MyDatabase;
GO

-- Create the MyDatabase database.
CREATE DATABASE MyDatabase COLLATE SQL_Latin1_General_CP1_CI_AS;
GO

-- Specify a simple recovery model
-- to keep the log growth to a minimum.
ALTER DATABASE MyDatabase
SET RECOVERY SIMPLE;
GO

USE MyDatabase;
GO

CREATE TABLE [dbo].[Course] (
[CourseID]   INT           NOT NULL,
[CourseName] NVARCHAR (50) NOT NULL,
PRIMARY KEY CLUSTERED ([CourseID] ASC)
);

CREATE TABLE [dbo].[Student] (
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

CREATE TABLE [dbo].[CourseSelection] (
[ID]        INT NOT NULL,
[StudentID] INT NOT NULL,
[CourseID]  INT NOT NULL,
PRIMARY KEY CLUSTERED ([ID] ASC),
CONSTRAINT [FK_CourseSelection_ToTable] FOREIGN KEY ([StudentID]) REFERENCES [dbo].[Student] ([StudentID])
ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT [FK_CourseSelection_Course_1] FOREIGN KEY ([CourseID]) REFERENCES [dbo].[Course] ([CourseID]) ON
DELETE NO ACTION ON UPDATE NO ACTION
```

```
);

CREATE TABLE [dbo].[LastStudent] (
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

-- Insert data into the tables.
USE MyDatabase
INSERT INTO Course (CourseID, CourseName)
VALUES(1, 'Algebra I');
INSERT INTO Course (CourseID, CourseName)
VALUES(2, 'Trigonometry');
INSERT INTO Course (CourseID, CourseName)
VALUES(3, 'Algebra II');
INSERT INTO Course (CourseID, CourseName)
VALUES(4, 'History');
INSERT INTO Course (CourseID, CourseName)
VALUES(5, 'English');
INSERT INTO Course (CourseID, CourseName)
VALUES(6, 'French');
INSERT INTO Course (CourseID, CourseName)
VALUES(7, 'Chinese');

INSERT INTO Student (StudentID, Name, Age)
VALUES(1, 'Abercrombie, Kim', 10);
INSERT INTO Student (StudentID, Name, Age)
VALUES(2, 'Abolrous, Hazen', 14);
INSERT INTO Student (StudentID, Name, Age)
VALUES(3, 'Hance, Jim', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(4, 'Adams, Terry', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(5, 'Hansen, Claus', 11);
INSERT INTO Student (StudentID, Name, Age)
VALUES(6, 'Penor, Lori', 13);
INSERT INTO Student (StudentID, Name, Age)
VALUES(7, 'Perham, Tom', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(8, 'Peng, Yun-Feng', NULL);

INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(1, 1, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(2, 1, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(3, 1, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(4, 2, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(5, 2, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(6, 2, 6);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(7, 2, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(8, 3, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(9, 3, 1);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(10, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(11, 4, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(12, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(13, 5, 3);
```

```
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(14, 5, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(15, 7, 3);
```

The following code contains the sample code that appears in this topic.

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema = SqlDataConnection<"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = schema.GetDataContext()

let data = [1; 5; 7; 11; 18; 21]

type Nullable<'T when 'T : ( new : unit -> 'T) and 'T : struct and 'T :> ValueType > with
    member this.Print() =
        if this.HasValue then this.Value.ToString()
        else "NULL"

printfn "\ncontains query operator"
query {
    for student in db.Student do
    select student.Age.Value
    contains 11
}
|> printfn "Is at least one student age 11? %b"

printfn "\ncount query operator"
query {
    for student in db.Student do
    select student
    count
}
|> printfn "Number of students: %d"

printfn "\nlast query operator."
let num =
    query {
        for number in data do
        sortBy number
        last
    }
printfn "Last number: %d" num

open Microsoft.FSharp.Linq

printfn "\nlastOrDefault query operator."
query {
    for number in data do
    sortBy number
    lastOrDefault
}
|> printfn "lastOrDefault: %d"

printfn "\nexactlyOne query operator."
let student2 =
    query {
```

```
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOne
    }
printfn "Student with StudentID = 1 is %s" student2.Name

printfn "\nexactlyOneOrDefault query operator."
let student3 =
    query {
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOneOrDefault
    }
printfn "Student with StudentID = 1 is %s" student3.Name

printfn "\nheadOrDefault query operator."
let student4 =
    query {
        for student in db.Student do
        select student
        headOrDefault
    }
printfn "head student is %s" student4.Name

printfn "\nselect query operator."
query {
    for student in db.Student do
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nwhere query operator."
query {
    for student in db.Student do
    where (student.StudentID > 4)
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nminBy query operator."
let student5 =
    query {
        for student in db.Student do
        minBy student.StudentID
    }

printfn "\nmaxBy query operator."
let student6 =
    query {
        for student in db.Student do
        maxBy student.StudentID
    }

printfn "\ngroupBy query operator."
query {
    for student in db.Student do
    groupBy student.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "Age: %s Count at that age: %d" (age.Print()) count)

printfn "\nsortBy query operator."
query {
    for student in db.Student do
    sortBy student.Name
    select student
}
```

```
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nsortByDescending query operator."
query {
    for student in db.Student do
    sortByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nthenBy query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenBy student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\nthenByDescending query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\ngroupValBy query operator."
query {
    for student in db.Student do
    groupValBy student.Name student.Age into g
    select (g, g.Key, g.Count())
}
|> Seq.iter (fun (group, age, count) ->
    printfn "Age: %s Count at that age: %d" (age.Print()) count
    group |> Seq.iter (fun name -> printfn "Name: %s" name))

printfn "\n sumByNullable query operator"
query {
    for student in db.Student do
    sumByNullable student.Age
}
|> (fun sum -> printfn "Sum of ages: %s" (sum.Print()))

printfn "\n minByNullable"
query {
    for student in db.Student do
    minByNullable student.Age
}
|> (fun age -> printfn "Minimum age: %s" (age.Print()))

printfn "\n maxByNullable"
query {
    for student in db.Student do
    maxByNullable student.Age
}
|> (fun age -> printfn "Maximum age: %s" (age.Print()))

printfn "\n averageBy"
query {
    for student in db.Student do
    averageBy (float student.StudentID)
}
|> printfn "Average student ID: %f"

printfn "\n averageByNullable"
```

```
printfn "\n averageByNullable"
query {
    for student in db.Student do
    averageByNullable (Nullable.float student.Age)
}
|> (fun avg -> printfn "Average age: %s" (avg.Print()))

printfn "\n find query operator"
query {
    for student in db.Student do
    find (student.Name = "Abercrombie, Kim")
}
|> (fun student -> printfn "Found a match with StudentID = %d" student.StudentID)

printfn "\n all query operator"
query {
    for student in db.Student do
    all (SqlMethods.Like(student.Name, "%,%"))
}
|> printfn "Do all students have a comma in the name? %b"

printfn "\n head query operator"
query {
    for student in db.Student do
    head
}
|> (fun student -> printfn "Found the head student with StudentID = %d" student.StudentID)

printfn "\n nth query operator"
query {
    for numbers in data do
    nth 3
}
|> printfn "Third number is %d"

printfn "\n skip query operator"
query {
    for student in db.Student do
    skip 1
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n skipWhile query operator"
query {
    for number in data do
    skipWhile (number < 3)
    select number
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sumBy query operator"
query {
    for student in db.Student do
    sumBy student.StudentID
}
|> printfn "Sum of student IDs: %d"

printfn "\n take query operator"
query {
    for student in db.Student do
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n takeWhile query operator"
query {
    for number in data do
    takeWhile (number < 10)
}
```

```
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sortByNullable query operator"
query {
    for student in db.Student do
    sortByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n sortByNullableDescending query operator"
query {
    for student in db.Student do
    sortByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullable query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullableDescending query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "All students: "
query {
    for student in db.Student do
    select student
}
|> Seq.iter (fun student -> printfn "%s %d %s" student.Name student.StudentID (student.Age.Print()))

printfn "\nCount of students: "
query {
    for student in db.Student do
    count
}
|> (fun count -> printfn "Student count: %d" count)

printfn "\nExists."
query {
    for student in db.Student do
    where
        (query {
            for courseSelection in db.CourseSelection do
            exists (courseSelection.StudentID = student.StudentID) })
    select student
}
|> Seq.iter (fun student -> printfn "%A" student.Name)

printfn "\n Group by age and count"
query {
    for n in db.Student do
```

```
    groupBy n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\n Group value by age."
query {
    for n in db.Student do
    groupValBy n.Age n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\nGroup students by age where age > 10."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Key.HasValue && g.Key.Value > 10)
    select (g, g.Key)
}
|> Seq.iter (fun (students, age) ->
    printfn "Age: %s" (age.Value.ToString())
    students
    |> Seq.iter (fun student -> printfn "%s" student.Name))

printfn "\nGroup students by age and print counts of number of students at each age with more than 1
student."
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
|> Seq.iter (fun (age, ageCount) ->
    printfn "Age: %s Count: %d" (age.Print()) ageCount)

printfn "\nGroup students by age and sum ages."
query {
    for student in db.Student do
    groupBy student.Age into g
    let total = query { for student in g do sumByNullable student.Age }
    select (g.Key, g.Count(), total)
}
|> Seq.iter (fun (age, count, total) ->
    printfn "Age: %d" (age.GetValueOrDefault())
    printfn "Count: %d" count
    printfn "Total years: %s" (total.ToString()))

printfn "\nGroup students by age and count number of students at each age, and display all with count > 1 in
descending order of count."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, myCount) ->
    printfn "Age: %s" (age.Print())
    printfn "Count: %d" myCount)

printfn "\n Select students from a set of IDs"
let idList = [1; 2; 5; 10]
let idQuery =
    query { for id in idList do select id }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
```

```
}
|> Seq.iter (fun student ->
    printfn "Name: %s" student.Name)

printfn "\nLook for students with Name match _e%% pattern and take first two."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%") )
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with Name matching [abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[^abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern and select ID."
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%") )
    select n.StudentID
}
|> Seq.iter (fun id -> printfn "%d" id)

printfn "\n Using Contains as a query filter."
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nSearching for names from a list."
let names = [|"a";"b";"c"|]
query {
    for student in db.Student do
    if names.Contains (student.Name) then select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nJoin Student and CourseSelection tables."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    select (student, selection)
}
|> Seq.iter (fun (student, selection) -> printfn "%d %s %d" student.StudentID student.Name
selection.CourseID)

printfn "\nLeft Join Student and CourseSelection tables."
query {
    for student in db.Student do
    leftOuterJoin selection in db.CourseSelection
        on (student.StudentID = selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
```

```
    select (student, selection)
}
|> Seq.iter (fun (student, selection) ->
    let selectionID, studentID, courseID =
        match selection with
        | null -> "NULL", "NULL", "NULL"
        | sel -> (sel.ID.ToString(), sel.StudentID.ToString(), sel.CourseID.ToString())
    printfn "%d %s %d %s %s %s" student.StudentID student.Name (student.Age.GetValueOrDefault()) selectionID
studentID courseID)

printfn "\nJoin with count"
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
|> printfn "%d"

printfn "\n Join with distinct."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    distinct
}
|> Seq.iter (fun (student, selection) -> printfn "%s %d" student.Name selection.CourseID)

printfn "\n Join with distinct and count."
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
|> printfn "%d"

printfn "\n Selecting students with age between 10 and 15."
query {
    for student in db.Student do
    where (student.Age.Value >= 10 && student.Age.Value < 15)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students with age either 11 or 12."
query {
    for student in db.Student do
    where (student.Age.Value = 11 || student.Age.Value = 12)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students in a certain age range and sorting."
query {
    for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
|> Seq.iter (fun student -> printfn "%s %s" student.Name (student.Age.Print()))

printfn "\n Selecting students with certain ages, taking account of possibility of nulls."
query {
    for student in db.Student do
    where
        ((student.Age.HasValue && student.Age.Value = 11) ||
         (student.Age.HasValue && student.Age.Value = 12))
```

```
    sortByDescending student.Name
    select student.Name
    take 2
}
|> Seq.iter (fun name -> printfn "%s" name)

printfn "\n Union of two queries."
module Queries =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query2.Union (query1)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Intersect of two queries."
module Queries2 =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query1.Intersect(query2)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Using if statement to alter results for special value."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Using if statement to alter results special values."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue && student.Age.Value = 0 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Multiple table select."
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
|> Seq.iteri (fun index (student, course) ->
    if index = 0 then
        printfn "StudentID Name Age CourseID CourseName"
    printfn "%d %s %s %d %s" student.StudentID student.Name (student.Age.Print()) course.CourseID
course.CourseName)
```

```
printfn "\nMultiple Joins"
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID = course.CourseID)
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

printfn "\nMultiple Left Outer Joins"
query {
    for student in db.Student do
    leftOuterJoin courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty() do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID = course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)
```

And here is the full output when this code is run in F# Interactive.

```
--> Referenced 'C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\3.0\Runtime\v4.0\Type
Providers\FSharp.Data.TypeProviders.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.Linq.dll'

contains query operator
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp5E3C.dll'...
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp611A.dll'...
Is at least one student age 11? true

count query operator
Number of students: 8

last query operator.
Last number: 21

lastOrDefault query operator.
lastOrDefault: 21

exactlyOne query operator.
Student with StudentID = 1 is Abercrombie, Kim

exactlyOneOrDefault query operator.
Student with StudentID = 1 is Abercrombie, Kim

headOrDefault query operator.
head student is Abercrombie, Kim

select query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng
```

```
where query operator.
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng

minBy query operator.

maxBy query operator.

groupBy query operator.
Age: NULL Count at that age: 1
Age: 10 Count at that age: 1
Age: 11 Count at that age: 1
Age: 12 Count at that age: 3
Age: 13 Count at that age: 1
Age: 14 Count at that age: 1

sortBy query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 4 Adams, Terry
StudentID, Name: 3 Hance, Jim
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom

sortByDescending query operator.
StudentID, Name: 7 Perham, Tom
StudentID, Name: 6 Penor, Lori
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 1 Abercrombie, Kim

thenBy query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Adams, Terry
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Perham, Tom
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

thenByDescending query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Perham, Tom
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Adams, Terry
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

groupValBy query operator.
Age: NULL Count at that age: 1
Name: Peng, Yun-Feng
Age: 10 Count at that age: 1
Name: Abercrombie, Kim
Age: 11 Count at that age: 1
Name: Hansen, Claus
Age: 12 Count at that age: 3
Name: Hance, Jim
Name: Adams, Terry
Name: Perham, Tom
Age: 13 Count at that age: 1
Name: Penor, Lori
```

```
Name: Penor, Lori
Age: 14 Count at that age: 1
Name: Abolrous, Hazen

sumByNullable query operator
Sum of ages: 84

minByNullable
Minimum age: 10

maxByNullable
Maximum age: 14

averageBy
Average student ID: 4.500000

averageByNullable
Average age: 12

find query operator
Found a match with StudentID = 1

all query operator
Do all students have a comma in the name? true

head query operator
Found the head student with StudentID = 1

nth query operator
Third number is 11

skip query operator
StudentID = 2
StudentID = 3
StudentID = 4
StudentID = 5
StudentID = 6
StudentID = 7
StudentID = 8

skipWhile query operator
Number = 5
Number = 7
Number = 11
Number = 18
Number = 21

sumBy query operator
Sum of student IDs: 36

take query operator
StudentID = 1
StudentID = 2

takeWhile query operator
Number = 1
Number = 5
Number = 7

sortByNullable query operator
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 2 Abolrous, Hazen 14

sortByNullableDescending query operator
```

```
sortByNullableDescending query operator
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 8 Peng, Yun-Feng NULL

thenByNullable query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12

thenByNullableDescending query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
All students:
Abercrombie, Kim 1 10
Abolrous, Hazen 2 14
Hance, Jim 3 12
Adams, Terry 4 12
Hansen, Claus 5 11
Penor, Lori 6 13
Perham, Tom 7 12
Peng, Yun-Feng 8 NULL

Count of students:
Student count: 8

Exists.
"Abercrombie, Kim"
"Abolrous, Hazen"
"Hance, Jim"
"Adams, Terry"
"Hansen, Claus"
"Perham, Tom"

Group by age and count
NULL 1
10 1
11 1
12 3
13 1
14 1

Group value by age.
NULL 1
10 1
11 1
12 3
13 1
14 1

Group students by age where age > 10.
Age: 11
Hansen, Claus
```

```
Age: 12
Hance, Jim
Adams, Terry
Perham, Tom
Age: 13
Penor, Lori
Age: 14
Abolrous, Hazen

Group students by age and print counts of number of students at each age with more than 1 student.
Age: 12 Count: 3

Group students by age and sum ages.
Age: 0
Count: 1
Total years:
Age: 10
Count: 1
Total years: 10
Age: 11
Count: 1
Total years: 11
Age: 12
Count: 3
Total years: 36
Age: 13
Count: 1
Total years: 13
Age: 14
Count: 1
Total years: 14

Group students by age and count number of students at each age, and display all with count > 1 in descending
order of count.
Age: 12
Count: 3

Select students from a set of IDs
Name: Abercrombie, Kim
Name: Abolrous, Hazen
Name: Hansen, Claus

Look for students with Name match _e% pattern and take first two.
Penor, Lori
Perham, Tom

Look for students with Name matching [abc]% pattern.
Abercrombie, Kim
Abolrous, Hazen
Adams, Terry

Look for students with name matching [^abc]% pattern.
Hance, Jim
Hansen, Claus
Penor, Lori
Perham, Tom
Peng, Yun-Feng

Look for students with name matching [^abc]% pattern and select ID.
3
5
6
7
8

Using Contains as a query filter.
Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
```

```
Adams, Terry
Hansen, Claus
Perham, Tom

Searching for names from a list.

Join Student and CourseSelection tables.
2 Abolrous, Hazen 2
3 Hance, Jim 3
5 Hansen, Claus 5
2 Abolrous, Hazen 2
5 Hansen, Claus 5
6 Penor, Lori 6
3 Hance, Jim 3
2 Abolrous, Hazen 2
1 Abercrombie, Kim 1
2 Abolrous, Hazen 2
5 Hansen, Claus 5
2 Abolrous, Hazen 2
3 Hance, Jim 3
2 Abolrous, Hazen 2
3 Hance, Jim 3

Left Join Student and CourseSelection tables.
1 Abercrombie, Kim 10 9 3 1
2 Abolrous, Hazen 14 1 1 2
2 Abolrous, Hazen 14 4 2 2
2 Abolrous, Hazen 14 8 3 2
2 Abolrous, Hazen 14 10 4 2
2 Abolrous, Hazen 14 12 4 2
2 Abolrous, Hazen 14 14 5 2
3 Hance, Jim 12 2 1 3
3 Hance, Jim 12 7 2 3
3 Hance, Jim 12 13 5 3
3 Hance, Jim 12 15 7 3
4 Adams, Terry 12 NULL NULL NULL
5 Hansen, Claus 11 3 1 5
5 Hansen, Claus 11 5 2 5
5 Hansen, Claus 11 11 4 5
6 Penor, Lori 13 6 2 6
7 Perham, Tom 12 NULL NULL NULL
8 Peng, Yun-Feng 0 NULL NULL NULL

Join with count
15

Join with distinct.
Abercrombie, Kim 2
Abercrombie, Kim 3
Abercrombie, Kim 5
Abolrous, Hazen 2
Abolrous, Hazen 5
Abolrous, Hazen 6
Abolrous, Hazen 3
Hance, Jim 2
Hance, Jim 1
Adams, Terry 2
Adams, Terry 5
Adams, Terry 2
Hansen, Claus 3
Hansen, Claus 2
Perham, Tom 3

Join with distinct and count.
15

Selecting students with age between 10 and 15.
Abercrombie, Kim
Abolrous, Hazen
```

```
Hance, Jim
Adams, Terry
Hansen, Claus
Penor, Lori
Perham, Tom

Selecting students with age either 11 or 12.
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Selecting students in a certain age range and sorting.
Penor, Lori 13
Perham, Tom 12
Hance, Jim 12
Adams, Terry 12

Selecting students with certain ages, taking account of possibility of nulls.
Hance, Jim
Adams, Terry

Union of two queries.
Abercrombie, Kim 10
Abolrous, Hazen 14
Hance, Jim 12
Adams, Terry 12
Hansen, Claus 11
Penor, Lori 13
Perham, Tom 12
Peng, Yun-Feng NULL

Intersect of two queries.

Using if statement to alter results for special value.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Using if statement to alter results special values.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Multiple table select.
StudentID Name Age CourseID CourseName
1 Abercrombie, Kim 10 1 Algebra I
2 Abolrous, Hazen 14 1 Algebra I
3 Hance, Jim 12 1 Algebra I
4 Adams, Terry 12 1 Algebra I
5 Hansen, Claus 11 1 Algebra I
6 Penor, Lori 13 1 Algebra I
7 Perham, Tom 12 1 Algebra I
8 Peng, Yun-Feng NULL 1 Algebra I
1 Abercrombie, Kim 10 2 Trigonometry
2 Abolrous, Hazen 14 2 Trigonometry
3 Hance, Jim 12 2 Trigonometry
4 Adams, Terry 12 2 Trigonometry
5 Hansen, Claus 11 2 Trigonometry
```

```
6 Penor, Lori 13 2 Trigonometry
7 Perham, Tom 12 2 Trigonometry
8 Peng, Yun-Feng NULL 2 Trigonometry
1 Abercrombie, Kim 10 3 Algebra II
2 Abolrous, Hazen 14 3 Algebra II
3 Hance, Jim 12 3 Algebra II
4 Adams, Terry 12 3 Algebra II
5 Hansen, Claus 11 3 Algebra II
6 Penor, Lori 13 3 Algebra II
7 Perham, Tom 12 3 Algebra II
8 Peng, Yun-Feng NULL 3 Algebra II
1 Abercrombie, Kim 10 4 History
2 Abolrous, Hazen 14 4 History
3 Hance, Jim 12 4 History
4 Adams, Terry 12 4 History
5 Hansen, Claus 11 4 History
6 Penor, Lori 13 4 History
7 Perham, Tom 12 4 History
8 Peng, Yun-Feng NULL 4 History
1 Abercrombie, Kim 10 5 English
2 Abolrous, Hazen 14 5 English
3 Hance, Jim 12 5 English
4 Adams, Terry 12 5 English
5 Hansen, Claus 11 5 English
6 Penor, Lori 13 5 English
7 Perham, Tom 12 5 English
8 Peng, Yun-Feng NULL 5 English
1 Abercrombie, Kim 10 6 French
2 Abolrous, Hazen 14 6 French
3 Hance, Jim 12 6 French
4 Adams, Terry 12 6 French
5 Hansen, Claus 11 6 French
6 Penor, Lori 13 6 French
7 Perham, Tom 12 6 French
8 Peng, Yun-Feng NULL 6 French
1 Abercrombie, Kim 10 7 Chinese
2 Abolrous, Hazen 14 7 Chinese
3 Hance, Jim 12 7 Chinese
4 Adams, Terry 12 7 Chinese
5 Hansen, Claus 11 7 Chinese
6 Penor, Lori 13 7 Chinese
7 Perham, Tom 12 7 Chinese
8 Peng, Yun-Feng NULL 7 Chinese

Multiple Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Perham, Tom Algebra II

Multiple Left Outer Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
```

```
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Penor, Lori
Perham, Tom Algebra II
Peng, Yun-Feng

type schema
val db : schema.ServiceTypes.SimpleDataContextTypes.MyDatabase1
val student : System.Data.Linq.Table<schema.ServiceTypes.Student>
val data : int list = [1; 5; 7; 11; 18; 21]
type Nullable<'T
                when 'T : (new : unit ->  'T) and 'T : struct and
                     'T :> System.ValueType> with
  member Print : unit -> string
val num : int = 21
val student2 : schema.ServiceTypes.Student
val student3 : schema.ServiceTypes.Student
val student4 : schema.ServiceTypes.Student
val student5 : int = 1
val student6 : int = 8
val idList : int list = [1; 2; 5; 10]
val idQuery : seq<int>
val names : string [] = [|"a"; "b"; "c"|]
module Queries = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
module Queries2 = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
```

See also

- F# Language Reference
- QueryBuilder Class
- Computation Expressions

Null Values

8/5/2022 • 3 minutes to read • Edit Online

This topic describes how the null value is used in F#.

Null Value

The null value is not normally used in F# for values or variables. However, null appears as an abnormal value in certain situations. If a type is defined in F#, null is not permitted as a regular value unless the AllowNullLiteral attribute is applied to the type. If a type is defined in some other .NET language, null is a possible value, and when you are interoperating with such types, your F# code might encounter null values.

For a type defined in F# and used strictly from F#, the only way to create a null value using the F# library directly is to use Unchecked.defaultof or Array.zeroCreate. However, for an F# type that is used from other .NET languages, or if you are using that type with an API that is not written in F#, such as the .NET Framework, null values can occur.

You can use the `option` type in F# when you might use a reference variable with a possible null value in another .NET language. Instead of null, with an F# `option` type, you use the option value `None` if there is no object. You use the option value `Some(obj)` with an object `obj` when there is an object. For more information, see Options. Note that you can still pack a `null` value into an Option if, for `Some x`, `x` happens to be `null`. Because of this, it is important you use `None` when a value is `null`.

The `null` keyword is a valid keyword in F#, and you have to use it when you are working with .NET Framework APIs or other APIs that are written in another .NET language. The two situations in which you might need a null value are when you call a .NET API and pass a null value as an argument, and when you interpret the return value or an output parameter from a .NET method call.

To pass a null value to a .NET method, just use the `null` keyword in the calling code. The following code example illustrates this.

```
open System

// Pass a null value to a .NET method.
let ParseDateTime (str: string) =
    let (success, res) = DateTime.TryParse(str, null, System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None
```

To interpret a null value that is obtained from a .NET method, use pattern matching if you can. The following code example shows how to use pattern matching to interpret the null value that is returned from `ReadLine` when it tries to read past the end of an input stream.

```
// Open a file and create a stream reader.
let fileStream1 =
    try
        System.IO.File.OpenRead("TextFile1.txt")
    with
        | :? System.IO.FileNotFoundException -> printfn "Error: TextFile1.txt not found."; exit(1)

let streamReader = new System.IO.StreamReader(fileStream1)

// ProcessNextLine returns false when there is no more input;
// it returns true when there is more input.
let ProcessNextLine nextLine =
    match nextLine with
    | null -> false
    | inputString ->
        match ParseDateTime inputString with
        | Some(date) -> printfn "%s" (date.ToLocalTime().ToString())
        | None -> printfn "Failed to parse the input."
        true

// A null value returned from .NET method ReadLine when there is
// no more input.
while ProcessNextLine (streamReader.ReadLine()) do ()
```

Null values for F# types can also be generated in other ways, such as when you use `Array.zeroCreate`, which calls `Unchecked.defaultof`. You must be careful with such code to keep the null values encapsulated. In a library intended only for F#, you do not have to check for null values in every function. If you are writing a library for interoperation with other .NET languages, you might have to add checks for null input parameters and throw an `ArgumentNullException`, just as you do in C# or Visual Basic code.

You can use the following code to check if an arbitrary value is null.

```
match box value with
| null -> printf "The value is null."
| _ -> printf "The value is not null."
```

See also

- Values
- Match Expressions

Nullable value types

8/5/2022 • 2 minutes to read • Edit Online

A *nullable value type* `Nullable<'T>` represents any struct type that could also be `null`. This is helpful when interacting with libraries and components that may choose to represent these kinds of types, like integers, with a `null` value for efficiency reasons. The underlying type that backs this construct is System.Nullable<T>.

Syntax

```
Nullable<'T>
Nullable value
```

Declare and assign with values

Declaring a nullable value type is just like declaring any wrapper-like type in F#:

```
open System

let x = 12
let nullableX = Nullable<int> x
```

You can also elide the generic type parameter and allow type inference to resolve it:

```
open System

let x = 12
let nullableX = Nullable x
```

To assign to a nullable value type, you'll need to also be explicit. There is no implicit conversion for F#-defined nullable value types:

```
open System

let mutable x = Nullable 12
x <- Nullable 13
```

Assign null

You cannot directly assign `null` to a nullable value type. Use `Nullable()` instead:

```
let mutable a = Nullable 42
a <- Nullable()
```

This is because `Nullable<'T>` does not have `null` as a proper value.

Pass and assign to members

A key difference between working with members and F# values is that nullable value types can be implicitly inferred when you're working with members. Consider the following method that takes a nullable value type as input:

```
type C() =
    member _.M(x: Nullable<int>) = x.HasValue
    member val NVT = Nullable 12 with get, set

let c = C()
c.M(12)
c.NVT <- 12
```

In the previous example, you can pass `12` to the method `M`. You can also assign `12` to the auto property `NVT`. If the input can be constructed as a nullable value type and it matches the target type, the F# compiler will implicitly convert such calls or assignments.

Examine a nullable value type instance

Unlike Options, which are a generalized construct for representing a possible value, nullable value types are not used with pattern matching. Instead, you need to use an `if` expression and check the `HasValue` property.

To get the underlying value, use the `Value` property after a `HasValue` check, like so:

```
open System

let a = Nullable 42

if a.HasValue then
    printfn $"{a} is {a.Value}"
else
    printfn $"{a} has no value."
```

Nullable operators

Operations on nullable value types, such as arithmetic or comparison, can require the use of nullable operators.

You can convert from one nullable value type to another using conversion operators from the `FSharp.Linq` namespace:

```
open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt
```

You can also use an appropriate non-nullable operator to convert to a primitive type, risking an exception if it has no value:

```
open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt

printfn $"value is %f{float nullableFloat}"
```

You can also use nullable operators as a short-hand for checking `HasValue` and `Value`:

```
open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt

let isBigger = nullableFloat ?> 1.0
let isBiggerLongForm = nullableFloat.HasValue && nullableFloat.Value > 1.0
```

The `?>` comparison checks if the left-hand side is nullable and only succeeds if it has a value. It is equivalent to the line that follows it.

See also

- Structs
- F# Options

Delegates (F#)

8/5/2022 • 3 minutes to read • Edit Online

A delegate represents a function call as an object. In F#, you ordinarily should use function values to represent functions as first-class values; however, delegates are used in the .NET Framework and so are needed when you interoperate with APIs that expect them. They may also be used when authoring libraries designed for use from other .NET Framework languages.

Syntax

```
type delegate-typename = delegate of type1 -> type2
```

Remarks

In the previous syntax, `type1` represents the argument type or types and `type2` represents the return type. The argument types that are represented by `type1` are automatically curried. This suggests that for this type you use a tuple form if the arguments of the target function are curried, and a parenthesized tuple for arguments that are already in the tuple form. The automatic currying removes a set of parentheses, leaving a tuple argument that matches the target method. Refer to the code example for the syntax you should use in each case.

Delegates can be attached to F# function values, and static or instance methods. F# function values can be passed directly as arguments to delegate constructors. For a static method, you construct the delegate by using the name of the class and the method. For an instance method, you provide the object instance and method in one argument. In both cases, the member access operator (`.`) is used.

The `Invoke` method on the delegate type calls the encapsulated function. Also, delegates can be passed as function values by referencing the Invoke method name without the parentheses.

The following code shows the syntax for creating delegates that represent various methods in a class. Depending on whether the method is a static method or an instance method, and whether it has arguments in the tuple form or the curried form, the syntax for declaring and assigning the delegate is a little different.

```
type Test1() =
  static member add(a : int, b : int) =
     a + b
  static member add2 (a : int) (b : int) =
     a + b

  member x.Add(a : int, b : int) =
     a + b
  member x.Add2 (a : int) (b : int) =
     a + b


// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg: Delegate1) (a: int) (b: int) =
   dlg.Invoke(a, b)
let InvokeDelegate2 (dlg: Delegate2) (a: int) (b: int) =
   dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 = Delegate1(Test1.add)
let del2 = Delegate2(Test1.add2)

let testObject = Test1()

// For instance methods, use the instance value name, the dot operator, and the instance method name.
let del3 = Delegate1(testObject.Add)
let del4 = Delegate2(testObject.Add2)

for (a, b) in [ (100, 200); (10, 20) ] do
  printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del2 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del4 a b)
```

The following code shows some of the different ways you can work with delegates.

```fsharp
type Delegate1 = delegate of int * char -> string

let replicate n c = String.replicate n (string c)

// An F# function value constructed from an unapplied let-bound function
let function1 = replicate

// A delegate object constructed from an F# function value
let delObject = Delegate1(function1)

// An F# function value constructed from an unapplied .NET member
let functionValue = delObject.Invoke

List.map (fun c -> functionValue(5,c)) ['a'; 'b'; 'c']
|> List.iter (printfn "%s")

// Or if you want to get back the same curried signature
let replicate' n c =  delObject.Invoke(n,c)

// You can pass a lambda expression as an argument to a function expecting a compatible delegate type
// System.Array.ConvertAll takes an array and a converter delegate that transforms an element from
// one type to another according to a specified function.
let stringArray = System.Array.ConvertAll([|'a';'b'|], fun c -> replicate' 3 c)
printfn "%A" stringArray
```

The output of the previous code example is as follows.

```
aaaaa
bbbbb
ccccc
[|"aaa"; "bbb"|]
```

See also

- F# Language Reference
- Parameters and Arguments
- Events

Enumerations

8/5/2022 • 2 minutes to read • Edit Online

Enumerations, also known as *enums*, are integral types where labels are assigned to a subset of the values. You can use them in place of literals to make code more readable and maintainable.

Syntax

```
type enum-name =
| value1 = integer-literal1
| value2 = integer-literal2
...
```

Remarks

An enumeration looks much like a discriminated union that has simple values, except that the values can be specified. The values are typically integers that start at 0 or 1, or integers that represent bit positions. If an enumeration is intended to represent bit positions, you should also use the Flags attribute.

The underlying type of the enumeration is determined from the literal that is used, so that, for example, you can use literals with a suffix, such as `1u`, `2u`, and so on, for an unsigned integer (`uint32`) type.

When you refer to the named values, you must use the name of the enumeration type itself as a qualifier, that is, `enum-name.value1`, not just `value1`. This behavior differs from that of discriminated unions. This is because enumerations always have the RequireQualifiedAccess attribute.

The following code shows the declaration and use of an enumeration.

```
// Declaration of an enumeration.
type Color =
   | Red = 0
   | Green = 1
   | Blue = 2
// Use of an enumeration.
let col1 : Color = Color.Red
```

You can easily convert enumerations to the underlying type by using the appropriate operator, as shown in the following code.

```
// Conversion to an integral type.
let n = int col1
```

Enumerated types can have one of the following underlying types: `sbyte`, `byte`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, and `char`. Enumeration types are represented in the .NET Framework as types that are inherited from `System.Enum`, which in turn is inherited from `System.ValueType`. Thus, they are value types that are located on the stack or inline in the containing object, and any value of the underlying type is a valid value of the enumeration. This is significant when pattern matching on enumeration values, because you have to provide a pattern that catches the unnamed values.

The `enum` function in the F# library can be used to generate an enumeration value, even a value other than one of the predefined, named values. You use the `enum` function as follows.

```
let col2 = enum<Color>(3)
```

The default `enum` function works with type `int32`. Therefore, it cannot be used with enumeration types that have other underlying types. Instead, use the following.

```
type uColor =
   | Red = 0u
   | Green = 1u
   | Blue = 2u
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

Additionally, cases for enums are always emitted as `public`. This is so that they align with C# and the rest of the .NET platform.

See also

- F# Language Reference
- Casting and Conversions

Events

8/5/2022 • 6 minutes to read • Edit Online

Events enable you to associate function calls with user actions and are important in GUI programming. Events can also be triggered by your applications or by the operating system.

Handling Events

When you use a GUI library like Windows Forms or Windows Presentation Foundation (WPF), much of the code in your application runs in response to events that are predefined by the library. These predefined events are members of GUI classes such as forms and controls. You can add custom behavior to a preexisting event, such as a button click, by referencing the specific named event of interest (for example, the `Click` event of the `Form` class) and invoking the `Add` method, as shown in the following code. If you run this from F# Interactive, omit the call to `System.Windows.Forms.Application.Run(System.Windows.Forms.Form)`.

```
open System.Windows.Forms

let form = new Form(Text="F# Windows Form",
                    Visible = true,
                    TopMost = true)

form.Click.Add(fun evArgs -> System.Console.Beep())
Application.Run(form)
```

The type of the `Add` method is `('a -> unit) -> unit`. Therefore, the event handler method takes one parameter, typically the event arguments, and returns `unit`. The previous example shows the event handler as a lambda expression. The event handler can also be a function value, as in the following code example. The following code example also shows the use of the event handler parameters, which provide information specific to the type of event. For a `MouseMove` event, the system passes a `System.Windows.Forms.MouseEventArgs` object, which contains the `X` and `Y` position of the pointer.

```
open System.Windows.Forms

let Beep evArgs =
    System.Console.Beep( )


let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)

let MouseMoveEventHandler (evArgs : System.Windows.Forms.MouseEventArgs) =
    form.Text <- System.String.Format("{0},{1}", evArgs.X, evArgs.Y)

form.Click.Add(Beep)
form.MouseMove.Add(MouseMoveEventHandler)
Application.Run(form)
```

Creating Custom Events

F# events are represented by the F# Event type, which implements the IEvent interface. `IEvent` is itself an interface that combines the functionality of two other interfaces, `System.IObservable<'T>` and IDelegateEvent.

Therefore, `Event`s have the equivalent functionality of delegates in other languages, plus the additional functionality from `IObservable`, which means that F# events support event filtering and using F# first-class functions and lambda expressions as event handlers. This functionality is provided in the Event module.

To create an event on a class that acts just like any other .NET Framework event, add to the class a `let` binding that defines an `Event` as a field in a class. You can specify the desired event argument type as the type argument, or leave it blank and have the compiler infer the appropriate type. You also must define an event member that exposes the event as a CLI event. This member should have the CLIEvent attribute. It is declared like a property and its implementation is just a call to the Publish property of the event. Users of your class can use the `Add` method of the published event to add a handler. The argument for the `Add` method can be a lambda expression. You can use the `Trigger` property of the event to raise the event, passing the arguments to the handler function. The following code example illustrates this. In this example, the inferred type argument for the event is a tuple, which represents the arguments for the lambda expression.

```
open System.Collections.Generic

type MyClassWithCLIEvent() =

    let event1 = new Event<_>()

    [<CLIEvent>]
    member this.Event1 = event1.Publish

    member this.TestEvent(arg) =
        event1.Trigger(this, arg)

let classWithEvent = new MyClassWithCLIEvent()
classWithEvent.Event1.Add(fun (sender, arg) ->
        printfn "Event1 occurred! Object data: %s" arg)

classWithEvent.TestEvent("Hello World!")

System.Console.ReadLine() |> ignore
```

The output is as follows.

```
Event1 occurred! Object data: Hello World!
```

The additional functionality provided by the `Event` module is illustrated here. The following code example illustrates the basic use of `Event.create` to create an event and a trigger method, add two event handlers in the form of lambda expressions, and then trigger the event to execute both lambda expressions.

```
type MyType() =
    let myEvent = new Event<_>()

    member this.AddHandlers() =
       Event.add (fun string1 -> printfn "%s" string1) myEvent.Publish
       Event.add (fun string1 -> printfn "Given a value: %s" string1) myEvent.Publish

    member this.Trigger(message) =
       myEvent.Trigger(message)

let myMyType = MyType()
myMyType.AddHandlers()
myMyType.Trigger("Event occurred.")
```

The output of the previous code is as follows.

```
Event occurred.
Given a value: Event occurred.
```

Processing Event Streams

Instead of just adding an event handler for an event by using the Event.add function, you can use the functions in the `Event` module to process streams of events in highly customized ways. To do this, you use the forward pipe (`|>`) together with the event as the first value in a series of function calls, and the `Event` module functions as subsequent function calls.

The following code example shows how to set up an event for which the handler is only called under certain conditions.

```
let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)
form.MouseMove
    |> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)
    |> Event.add ( fun evArgs ->
        form.BackColor <- System.Drawing.Color.FromArgb(
            evArgs.X, evArgs.Y, evArgs.X ^^^ evArgs.Y) )
```

The Observable module contains similar functions that operate on observable objects. Observable objects are similar to events but only actively subscribe to events if they themselves are being subscribed to.

Implementing an Interface Event

As you develop UI components, you often start by creating a new form or a new control that inherits from an existing form or control. Events are frequently defined on an interface, and, in that case, you must implement the interface to implement the event. The `System.ComponentModel.INotifyPropertyChanged` interface defines a single `System.ComponentModel.INotifyPropertyChanged.PropertyChanged` event. The following code illustrates how to implement the event that this inherited interface defined:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

type AppForm() as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property-changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property-changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    // Expose the PropertyChanged event as a first class .NET event.
    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = propertyChanged.Publish.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = propertyChanged.Publish.RemoveHandler(handler)

    // This is the event-handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property {args.PropertyName} changed its value to {newValue}"

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
let inpc = appForm :> INotifyPropertyChanged
inpc.PropertyChanged.Add(appForm.OnPropertyChanged)
Application.Run(appForm)
```

If you want to hook up the event in the constructor, the code is a bit more complicated because the event hookup must be in a `then` block in an additional constructor, as in the following example:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

// Create a private constructor with a dummy argument so that the public
// constructor can have no arguments.
type AppForm private (dummy) as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = this.PropertyChanged.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = this.PropertyChanged.RemoveHandler(handler)

    // This is the event handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property {args.PropertyName} changed its value to {newValue}"

    new() as this =
        new AppForm(0)
        then
            let inpc = this :> INotifyPropertyChanged
            inpc.PropertyChanged.Add(this.OnPropertyChanged)

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
Application.Run(appForm)
```

See also

- Members
- Handling and Raising Events
- Lambda Expressions: The `fun` Keyword

External Functions

8/5/2022 • 2 minutes to read • Edit Online

This topic describes F# language support for calling functions in native code.

Syntax

```
[<DllImport( arguments )>]
extern declaration
```

Remarks

In the previous syntax, *arguments* represents arguments that are supplied to the `System.Runtime.InteropServices.DllImportAttribute` attribute. The first argument is a string that represents the name of the DLL that contains this function, without the .dll extension. Additional arguments can be supplied for any of the public properties of the `System.Runtime.InteropServices.DllImportAttribute` class, such as the calling convention.

Assume you have a native C++ DLL that contains the following exported function.

```
#include <stdio.h>
extern "C" void __declspec(dllexport) HelloWorld()
{
    printf("Hello world, invoked by F#!\n");
}
```

You can call this function from F# by using the following code.

```
open System.Runtime.InteropServices

module InteropWithNative =
    [<DllImport(@"C:\bin\nativedll", CallingConvention = CallingConvention.Cdecl)>]
    extern void HelloWorld()

InteropWithNative.HelloWorld()
```

Interoperability with native code is referred to as *platform invoke* and is a feature of the CLR. For more information, see Interoperating with Unmanaged Code. The information in that section is applicable to F#.

See also

- Functions

Attributes (F#)

8/5/2022 • 3 minutes to read • Edit Online

Attributes enable metadata to be applied to a programming construct.

Syntax

```
[<target:attribute-name(arguments)>]
```

Remarks

In the previous syntax, the *target* is optional and, if present, specifies the kind of program entity that the attribute applies to. Valid values for *target* are shown in the table that appears later in this document.

The *attribute-name* refers to the name (possibly qualified with namespaces) of a valid attribute type, with or without the suffix `Attribute` that is usually used in attribute type names. For example, the type `ObsoleteAttribute` can be shortened to just `Obsolete` in this context.

The *arguments* are the arguments to the constructor for the attribute type. If an attribute has a parameterless constructor, the argument list and parentheses can be omitted. Attributes support both positional arguments and named arguments. *Positional arguments* are arguments that are used in the order in which they appear. Named arguments can be used if the attribute has public properties. You can set these by using the following syntax in the argument list.

```
property-name = property-value
```

Such property initializations can be in any order, but they must follow any positional arguments. The following is an example of an attribute that uses positional arguments and property initializations:

```
open System.Runtime.InteropServices

[<DllImport("kernel32", SetLastError=true)>]
extern bool CloseHandle(nativeint handle)
```

In this example, the attribute is `DllImportAttribute`, here used in shortened form. The first argument is a positional parameter and the second is a property.

Attributes are a .NET programming construct that enables an object known as an *attribute* to be associated with a type or other program element. The program element to which an attribute is applied is known as the *attribute target*. The attribute usually contains metadata about its target. In this context, metadata could be any data about the type other than its fields and members.

Attributes in F# can be applied to the following programming constructs: functions, methods, assemblies, modules, types (classes, records, structures, interfaces, delegates, enumerations, unions, and so on), constructors, properties, fields, parameters, type parameters, and return values. Attributes are not allowed on `let` bindings inside classes, expressions, or workflow expressions.

Typically, the attribute declaration appears directly before the declaration of the attribute target. Multiple attribute declarations can be used together, as follows:

```
[<Owner("Jason Carlson")>]
[<Company("Microsoft")>]
type SomeType1 =
```

You can query attributes at run time by using .NET reflection.

You can declare multiple attributes individually, as in the previous code example, or you can declare them in one set of brackets if you use a semicolon to separate the individual attributes and constructors, as follows:

```
[<Owner("Darren Parker"); Company("Microsoft")>]
type SomeType2 =
```

Typically encountered attributes include the `Obsolete` attribute, attributes for security considerations, attributes for COM support, attributes that relate to ownership of code, and attributes indicating whether a type can be serialized. The following example demonstrates the use of the `Obsolete` attribute.

```
open System

[<Obsolete("Do not use. Use newFunction instead.")>]
let obsoleteFunction x y =
  x + y

let newFunction x y =
  x + 2 * y

// The use of the obsolete function produces a warning.
let result1 = obsoleteFunction 10 100
let result2 = newFunction 10 100
```

For the attribute targets `assembly` and `module`, you apply the attributes to a top-level `do` binding in your assembly. You can include the word `assembly` or ``` ``module`` ``` in the attribute declaration, as follows:

```
open System.Reflection
[<assembly:AssemblyVersionAttribute("1.0.0.0")>]
[<``module``:MyCustomModuleAttribute>]
do
   printfn "Executing..."
```

If you omit the attribute target for an attribute applied to a `do` binding, the F# compiler attempts to determine the attribute target that makes sense for that attribute. Many attribute classes have an attribute of type `System.AttributeUsageAttribute` that includes information about the possible targets supported for that attribute. If the `System.AttributeUsageAttribute` indicates that the attribute supports functions as targets, the attribute is taken to apply to the main entry point of the program. If the `System.AttributeUsageAttribute` indicates that the attribute supports assemblies as targets, the compiler takes the attribute to apply to the assembly. Most attributes do not apply to both functions and assemblies, but in cases where they do, the attribute is taken to apply to the program's main function. If the attribute target is specified explicitly, the attribute is applied to the specified target.

Although you do not usually need to specify the attribute target explicitly, valid values for *target* in an attribute along with examples of usage are shown in the following table:

ATTRIBUTE TARGET	EXAMPLE

assembly	`[<assembly: AssemblyVersion("1.0.0.0")>]`
module	``` [<``module``: MyCustomAttributeThatWorksOnModules>] ```
return	`let function1 x : [<return: MyCustomAttributeThatWorksOnReturns>] int = x + 1`
field	`[<DefaultValue>] val mutable x: int`
property	`[<Obsolete>] this.MyProperty = x`
param	`member this.MyMethod([<Out>] x : ref<int>) = x := 10`

type	`[<type: StructLayout(LayoutKind.Sequential)>]`
	`type MyStruct =`
	`  struct`
	`    val x : byte`
	`    val y : int`
	`  end`

See also

- F# Language Reference

Code quotations

8/5/2022 • 7 minutes to read • Edit Online

This article describes *code quotations*, a language feature that enables you to generate and work with F# code expressions programmatically. This feature lets you generate an abstract syntax tree that represents F# code. The abstract syntax tree can then be traversed and processed according to the needs of your application. For example, you can use the tree to generate F# code or generate code in some other language.

Quoted expressions

A *quoted expression* is an F# expression in your code that is delimited in such a way that it is not compiled as part of your program, but instead is compiled into an object that represents an F# expression. You can mark a quoted expression in one of two ways: either with type information or without type information. If you want to include type information, you use the symbols `<@` and `@>` to delimit the quoted expression. If you do not need type information, you use the symbols `<@@` and `@@>`. The following code shows typed and untyped quotations.

```
open Microsoft.FSharp.Quotations
// A typed code quotation.
let expr : Expr<int> = <@ 1 + 1 @>
// An untyped code quotation.
let expr2 : Expr = <@@ 1 + 1 @@>
```

Traversing a large expression tree is faster if you do not include type information. The resulting type of an expression quoted with the typed symbols is `Expr<'T>`, where the type parameter has the type of the expression as determined by the F# compiler's type inference algorithm. When you use code quotations without type information, the type of the quoted expression is the non-generic type Expr. You can call the Raw property on the typed `Expr` class to obtain the untyped `Expr` object.

There are various static methods that allow you to generate F# expression objects programmatically in the `Expr` class without using quoted expressions.

A code quotation must include a complete expression. For a `let` binding, for example, you need both the definition of the bound name and another expression that uses the binding. In verbose syntax, this is an expression that follows the `in` keyword. At the top level in a module, this is just the next expression in the module, but in a quotation, it is explicitly required.

Therefore, the following expression is not valid.

```
// Not valid:
// <@ let f x = x + 1 @>
```

But the following expressions are valid.

```
// Valid:
<@ let f x = x + 10 in f 20 @>
// Valid:
<@
    let f x = x + 10
    f 20
@>
```

To evaluate F# quotations, you must use the F# Quotation Evaluator. It provides support for evaluating and executing F# expression objects.

F# quotations also retain type constraint information. Consider the following example:

```
open FSharp.Linq.RuntimeHelpers

let eval q = LeafExpressionConverter.EvaluateQuotation q

let inline negate x = -x
// val inline negate: x: ^a ->  ^a when  ^a : (static member ( ~- ) :  ^a ->  ^a)

<@ negate 1.0 @>  |> eval
```

The constraint generated by the `inline` function is retained in the code quotation. The `negate` function's quoted form can now be evaluated.

Expr type

An instance of the `Expr` type represents an F# expression. Both the generic and the non-generic `Expr` types are documented in the F# library documentation. For more information, see FSharp.Quotations Namespace and Quotations.Expr Class.

Splicing operators

Splicing enables you to combine literal code quotations with expressions that you have created programmatically or from another code quotation. The `%` and `%%` operators enable you to add an F# expression object into a code quotation. You use the `%` operator to insert a typed expression object into a typed quotation; you use the `%%` operator to insert an untyped expression object into an untyped quotation. Both operators are unary prefix operators. Thus if `expr` is an untyped expression of type `Expr`, the following code is valid.

```
<@@ 1 + %%expr @@>
```

And if `expr` is a typed quotation of type `Expr<int>`, the following code is valid.

```
<@ 1 + %expr @>
```

Example 1

Description

The following example illustrates the use of code quotations to put F# code into an expression object and then print the F# code that represents the expression. A function `println` is defined that contains a recursive function `print` that displays an F# expression object (of type `Expr`) in a friendly format. There are several active patterns in the FSharp.Quotations.Patterns and FSharp.Quotations.DerivedPatterns modules that can be used to analyze expression objects. This example does not include all the possible patterns that might appear in an F# expression. Any unrecognized pattern triggers a match to the wildcard pattern (`_`) and is rendered by using the `ToString` method, which, on the `Expr` type, lets you know the active pattern to add to your match expression.

Code

```
module Print
```

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let println expr =
    let rec print expr =
        match expr with
        | Application(expr1, expr2) ->
            // Function application.
            print expr1
            printf " "
            print expr2
        | SpecificCall <@@ (+) @@> (_, _, exprList) ->
            // Matches a call to (+). Must appear before Call pattern.
            print exprList.Head
            printf " + "
            print exprList.Tail.Head
        | Call(exprOpt, methodInfo, exprList) ->
            // Method or module function call.
            match exprOpt with
            | Some expr -> print expr
            | None -> printf "%s" methodInfo.DeclaringType.Name
            printf ".%s(" methodInfo.Name
            if (exprList.IsEmpty) then printf ")" else
            print exprList.Head
            for expr in exprList.Tail do
                printf ","
                print expr
            printf ")"
        | Int32(n) ->
            printf "%d" n
        | Lambda(param, body) ->
            // Lambda expression.
            printf "fun (%s:%s) -> " param.Name (param.Type.ToString())
            print body
        | Let(var, expr1, expr2) ->
            // Let binding.
            if (var.IsMutable) then
                printf "let mutable %s = " var.Name
            else
                printf "let %s = " var.Name
            print expr1
            printf " in "
            print expr2
        | PropertyGet(_, propOrValInfo, _) ->
            printf "%s" propOrValInfo.Name
        | String(str) ->
            printf "%s" str
        | Value(value, typ) ->
            printf "%s" (value.ToString())
        | Var(var) ->
            printf "%s" var.Name
        | _ -> printf "%s" (expr.ToString())
    print expr
    printfn ""


let a = 2

// exprLambda has type "(int -> int)".
let exprLambda = <@ fun x -> x + 1 @>
// exprCall has type unit.
let exprCall = <@ a + 1 @>

println exprLambda
println exprCall
println <@@ let f x = x + 10 in f 10 @@>
```

Output

```
fun (x:System.Int32) -> x + 1
a + 1
let f = fun (x:System.Int32) -> x + 10 in f 10
```

Example 2

Description

You can also use the three active patterns in the ExprShape module to traverse expression trees with fewer active patterns. These active patterns can be useful when you want to traverse a tree but you do not need all the information in most of the nodes. When you use these patterns, any F# expression matches one of the following three patterns: `ShapeVar` if the expression is a variable, `ShapeLambda` if the expression is a lambda expression, or `ShapeCombination` if the expression is anything else. If you traverse an expression tree by using the active patterns as in the previous code example, you have to use many more patterns to handle all possible F# expression types, and your code will be more complex. For more information, see ExprShape.ShapeVar|ShapeLambda|ShapeCombination Active Pattern.

The following code example can be used as a basis for more complex traversals. In this code, an expression tree is created for an expression that involves a function call, `add`. The SpecificCall active pattern is used to detect any call to `add` in the expression tree. This active pattern assigns the arguments of the call to the `exprList` value. In this case, there are only two, so these are pulled out and the function is called recursively on the arguments. The results are inserted into a code quotation that represents a call to `mul` by using the splice operator (`%%`). The `println` function from the previous example is used to display the results.

The code in the other active pattern branches just regenerates the same expression tree, so the only change in the resulting expression is the change from `add` to `mul`.

Code

```
module Module1
open Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape

let add x y = x + y
let mul x y = x * y

let rec substituteExpr expression =
    match expression with
    | SpecificCall <@@ add @@> (_, _, exprList) ->
        let lhs = substituteExpr exprList.Head
        let rhs = substituteExpr exprList.Tail.Head
        <@@ mul %%lhs %%rhs @@>
    | ShapeVar var -> Expr.Var var
    | ShapeLambda (var, expr) -> Expr.Lambda (var, substituteExpr expr)
    | ShapeCombination(shapeComboObject, exprList) ->
        RebuildShapeCombination(shapeComboObject, List.map substituteExpr exprList)

let expr1 = <@@ 1 + (add 2 (add 3 4)) @@>
println expr1
let expr2 = substituteExpr expr1
println expr2
```

Output

```
1 + Module1.add(2,Module1.add(3,4))
1 + Module1.mul(2,Module1.mul(3,4))
```

See also

- F# Language Reference

Nameof

8/5/2022 • 2 minutes to read • Edit Online

The `nameof` expression produces a string constant that matches the name in source for nearly any F# construct in source.

Syntax

```
nameof symbol
nameof<'TGeneric>
```

Remarks

`nameof` works by resolving the symbol passed to it and produces the name of that symbol as it is declared in your source code. This is useful in various scenarios, such as logging, and protects your logging against changes in source code.

```
let months =
    [
        "January"; "February"; "March"; "April";
        "May"; "June"; "July"; "August"; "September";
        "October"; "November"; "December"
    ]

let lookupMonth month =
    if (month > 12 || month < 1) then
        invalidArg (nameof month) ($"Value passed in was %d{month}.")

    months[month-1]

printfn "%s" (lookupMonth 12)
printfn "%s" (lookupMonth 1)
printfn "%s" (lookupMonth 13)
```

The last line will throw an exception and `"month"` will be shown in the error message.

You can take a name of nearly every F# construct:

```
module M =
    let f x = nameof x

printfn $"{(M.f 12)]}"
printfn $"{(nameof M)}"
printfn $"{(nameof M.f)}"
```

`nameof` is not a first-class function and cannot be used as such. That means it cannot be partially applied and values cannot be piped into it via F# pipeline operators.

Nameof on operators

Operators in F# can be used in two ways, as an operator text itself, or a symbol representing the compiled form. `nameof` on an operator will produce the name of the operator as it is declared in source. To get the compiled name, use the compiled name in source:

```
nameof(+) // "+"
nameof op_Addition // "op_Addition"
```

Nameof on generics

You can also take a name of a generic type parameter, but the syntax is different:

```
let f<'a> () = nameof<'a>
f() // "a"
```

`nameof<'TGeneric>` will take the name of the symbol as defined in source, not the name of the type substituted at a call site.

The reason why the syntax is different is to align with other F# intrinsic operators like `typeof<>` and `typedefof<>`. This makes F# consistent with respect to operators that act on generic types and anything else in source.

Nameof in pattern matching

The `nameof` pattern lets you use `nameof` in a pattern match expression like so:

```
let f (str: string) =
    match str with
    | nameof str -> "It's 'str'!"
    | _ -> "It is not 'str'!"

f "str" // matches
f "asdf" // does not match
```

Nameof with instance members

F# requires an instance in order to extract the name of an instance member with `nameof`. If an instance is not easily available, then one can be obtained using `Unchecked.defaultof`.

```
type MyRecord = { MyField: int }
type MyClass() =
    member _.MyProperty = ()
    member _.MyMethod () = ()

nameof Unchecked.defaultof<MyRecord>.MyField   // MyField
nameof Unchecked.defaultof<MyClass>.MyProperty // MyProperty
nameof Unchecked.defaultof<MyClass>.MyMethod   // MyMethod
```

Caller information

8/5/2022 • 2 minutes to read • Edit Online

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePath	Full path of the source file that contains the caller. This is the file path at compile time.	`String`
CallerLineNumber	Line number in the source file at which the method is called.	`Integer`
CallerMemberName	Method or property name of the caller. See the Member Names section later in this topic.	`String`

Example

The following example shows how you might use these attributes to trace a caller.

```
open System.Diagnostics
open System.Runtime.CompilerServices
open System.Runtime.InteropServices

type Tracer() =
    member _.DoTrace(message: string,
                     [<CallerMemberName; Optional; DefaultParameterValue("")>] memberName: string,
                     [<CallerFilePath; Optional; DefaultParameterValue("")>] path: string,
                     [<CallerLineNumber; Optional; DefaultParameterValue(0)>] line: int) =
        Trace.WriteLine(sprintf $"Message: {message}")
        Trace.WriteLine(sprintf $"Member name: {memberName}")
        Trace.WriteLine(sprintf $"Source file path: {path}")
        Trace.WriteLine(sprintf $"Source line number: {line}")
```

Remarks

Caller Info attributes can only be applied to optional parameters. The Caller Info attributes cause the compiler to write the proper value for each optional parameter decorated with a Caller Info attribute.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that Rename Refactoring doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the CallerMemberName attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the member to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- Attributes
- Named arguments
- Optional parameters

Source Line, File, and Path Identifiers

8/5/2022 • 2 minutes to read • Edit Online

The identifiers `__LINE__`, `__SOURCE_DIRECTORY__` and `__SOURCE_FILE__` are built-in values that enable you to access the source line number, directory and file name in your code.

Syntax

```
__LINE__
__SOURCE_DIRECTORY__
__SOURCE_FILE__
```

Remarks

Each of these values has type `string`.

The following table summarizes the source line, file, and path identifiers that are available in F#. These identifiers are not preprocessor macros; they are built-in values that are recognized by the compiler.

PREDEFINED IDENTIFIER	DESCRIPTION
`__LINE__`	Evaluates to the current line number, considering `#line` directives.
`__SOURCE_DIRECTORY__`	Evaluates to the current full path of the source directory, considering `#line` directives.
`__SOURCE_FILE__`	Evaluates to the current source file name, without its path, considering `#line` directives.

For more information about the `#line` directive, see Compiler Directives.

Example

The following code example demonstrates the use of these values.

```
let printSourceLocation() =
    printfn "Line: %s" __LINE__
    printfn "Source Directory: %s" __SOURCE_DIRECTORY__
    printfn "Source File: %s" __SOURCE_FILE__
printSourceLocation()
```

Output:

```
Line: 4
Source Directory: C:\Users\username\Documents\Visual Studio 2017\Projects\SourceInfo\SourceInfo
Source File: Program.fs
```

See also

- Compiler Directives
- F# Language Reference

Plain text formatting

8/5/2022 • 11 minutes to read • Edit Online

F# supports type-checked formatting of plain text using `printf`, `printfn`, `sprintf`, and related functions. For example,

```
dotnet fsi

> printfn "Hello %s, %d + %d is %d" "world" 2 2 (2+2);;
```

gives the output

```
Hello world, 2 + 2 is 4
```

F# also allows structured values to be formatted as plain text. For example, consider the following example that formats the output as a matrix-like display of tuples.

```
dotnet fsi

> printfn "%A" [ for i in 1 .. 5 -> [ for j in 1 .. 5 -> (i, j) ] ];;

[[(1, 1); (1, 2); (1, 3); (1, 4); (1, 5)];
 [(2, 1); (2, 2); (2, 3); (2, 4); (2, 5)];
 [(3, 1); (3, 2); (3, 3); (3, 4); (3, 5)];
 [(4, 1); (4, 2); (4, 3); (4, 4); (4, 5)];
 [(5, 1); (5, 2); (5, 3); (5, 4); (5, 5)]]
```

Structured plain text formatting is activated when you use the `%A` format in `printf` formatting strings. It's also activated when formatting the output of values in F# interactive, where the output includes extra information and is additionally customizable. Plain text formatting is also observable through any calls to `x.ToString()` on F# union and record values, including those that occur implicitly in debugging, logging, and other tooling.

Checking of `printf` -format strings

A compile-time error will be reported if a `printf` formatting function is used with an argument that doesn't match the printf format specifiers in the format string. For example,

```
sprintf "Hello %s" (2+2)
```

gives the output

```
  sprintf "Hello %s" (2+2)
  ----------------------^

stdin(3,25): error FS0001: The type 'string' does not match the type 'int'
```

Technically speaking, when using `printf` and other related functions, a special rule in the F# compiler checks the string literal passed as the format string, ensuring the subsequent arguments applied are of the correct type to match the format specifiers used.

Format specifiers for `printf`

Format specifications for `printf` formats are strings with `%` markers that indicate format. Format placeholders consist of `%[flags][width][.precision][type]` where the type is interpreted as follows:

FORMAT SPECIFIER	TYPE(S)	REMARKS
`%b`	`bool` (`System.Boolean`)	Formatted as `true` or `false`
`%s`	`string` (`System.String`)	Formatted as its unescaped contents
`%c`	`char` (`System.Char`)	Formatted as the character literal
`%d`, `%i`	a basic integer type	Formatted as a decimal integer, signed if the basic integer type is signed
`%u`	a basic integer type	Formatted as an unsigned decimal integer
`%x`, `%X`	a basic integer type	Formatted as an unsigned hexadecimal number (a-f or A-F for hex digits respectively)
`%o`	a basic integer type	Formatted as an unsigned octal number
`%B`	a basic integer type	Formatted as an unsigned binary number
`%e`, `%E`	a basic floating point type	Formatted as a signed value having the form `[-]d.dddde[sign]ddd` where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is `+` or `-`
`%f`, `%F`	a basic floating point type	Formatted as a signed value having the form `[-]dddd.dddd`, where `dddd` is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
`%g`, `%G`	a basic floating point type	Formatted using as a signed value printed in `%f` or `%e` format, whichever is more compact for the given value and precision.
`%M`	a `decimal` (`System.Decimal`) value	Formatted using the `"G"` format specifier for `System.Decimal.ToString(format)`

FORMAT SPECIFIER	TYPE(S)	REMARKS
`%O`	any value	Formatted by boxing the object and calling its `System.Object.ToString()` method
`%A`	any value	Formatted using structured plain text formatting with the default layout settings
`%a`	any value	Requires two arguments: a formatting function accepting a context parameter and the value, and the particular value to print
`%t`	any value	Requires one argument: a formatting function accepting a context parameter that either outputs or returns the appropriate text
`%%`	(none)	Requires no arguments and prints a plain percent sign: `%`

Basic integer types are `byte` (`System.Byte`), `sbyte` (`System.SByte`), `int16` (`System.Int16`), `uint16` (`System.UInt16`), `int32` (`System.Int32`), `uint32` (`System.UInt32`), `int64` (`System.Int64`), `uint64` (`System.UInt64`), `nativeint` (`System.IntPtr`), and `unativeint` (`System.UIntPtr`). Basic floating point types are `float` (`System.Double`), `float32` (`System.Single`), and `decimal` (`System.Decimal`).

The optional width is an integer indicating the minimal width of the result. For instance, `%6d` prints an integer, prefixing it with spaces to fill at least six characters. If width is `*`, then an extra integer argument is taken to specify the corresponding width.

Valid flags are:

FLAG	EFFECT	REMARKS
`0`	Add zeros instead of spaces to make up the required width	
`-`	Left justify the result within the specified width	
`+`	Add a `+` character if the number is positive (to match a `-` sign for negatives)	
space character	Add an extra space if the number is positive (to match a '-' sign for negatives)	

The printf `#` flag is invalid and a compile-time error will be reported if it is used.

Values are formatted using invariant culture. Culture settings are irrelevant to `printf` formatting except when they affect the results of `%O` and `%A` formatting. For more information, see structured plain text formatting.

`%A` formatting

The `%A` format specifier is used to format values in a human-readable way, and can also be useful for reporting diagnostic information.

Primitive values

When formatting plain text using the `%A` specifier, F# numeric values are formatted with their suffix and invariant culture. Floating point values are formatted using 10 places of floating point precision. For example,

```
printfn "%A" (1L, 3n, 5u, 7, 4.03f, 5.000000001, 5.0000000001)
```

produces

```
(1L, 3n, 5u, 7, 4.03000021f, 5.000000001, 5.0)
```

When using the `%A` specifier, strings are formatted using quotes. Escape codes are not added and instead the raw characters are printed. For example,

```
printfn "%A" ("abc", "a\tb\nc\"d")
```

produces

```
("abc", "a      b
c"d")
```

.NET values

When formatting plain text using the `%A` specifier, non-F# .NET objects are formatted by using `x.ToString()` using the default settings of .NET given by `System.Globalization.CultureInfo.CurrentCulture` and `System.Globalization.CultureInfo.CurrentUICulture`. For example,

```
open System.Globalization

let date = System.DateTime(1999, 12, 31)

CultureInfo.CurrentCulture <- CultureInfo.GetCultureInfo("de-DE")
printfn "Culture 1: %A" date

CultureInfo.CurrentCulture <- CultureInfo.GetCultureInfo("en-US")
printfn "Culture 2: %A" date
```

produces

```
Culture 1: 31.12.1999 00:00:00
Culture 2: 12/31/1999 12:00:00 AM
```

Structured values

When formatting plain text using the `%A` specifier, block indentation is used for F# lists and tuples. This is shown in the previous example. The structure of arrays is also used, including multi-dimensional arrays. Single-dimensional arrays are shown with `[| ... |]` syntax. For example,

```
printfn "%A" [| for i in 1 .. 20 -> (i, i*i) |]
```

produces

```
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25); (6, 36); (7, 49); (8, 64); (9, 81);
  (10, 100); (11, 121); (12, 144); (13, 169); (14, 196); (15, 225); (16, 256);
  (17, 289); (18, 324); (19, 361); (20, 400)|]
```

The default print width is 80. This width can be customized by using a print width in the format specifier. For example,

```
printfn "%10A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%20A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%50A" [| for i in 1 .. 5 -> (i, i*i) |]
```

produces

```
[|(1, 1);
  (2, 4);
  (3, 9);
  (4, 16);
  (5, 25)|]
[|(1, 1); (2, 4);
  (3, 9); (4, 16);
  (5, 25)|]
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
```

Specifying a print width of 0 results in no print width being used. A single line of text will result, except where embedded strings in the output contain line breaks. For example

```
printfn "%0A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%0A" [| for i in 1 .. 5 -> "abc\ndef" |]
```

produces

```
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
[|"abc
def"; "abc
def"; "abc
def"; "abc
def"; "abc
def"|]
```

A depth limit of 4 is used for sequence (`IEnumerable`) values, which are shown as `seq { ...}`. A depth limit of 100 is used for list and array values. For example,

```
printfn "%A" (seq { for i in 1 .. 10 -> (i, i*i) })
```

produces

```
seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
```

Block indentation is also used for the structure of public record and union values. For example,

```
type R = { X : int list; Y : string list }

printfn "%A" { X =  [ 1;2;3 ]; Y = ["one"; "two"; "three"] }
```

produces

```
{ X = [1; 2; 3]
  Y = ["one"; "two"; "three"] }
```

If `%+A` is used, then the private structure of records and unions is also revealed by using reflection. For example

```
type internal R =
    { X : int list; Y : string list }
    override _.ToString() = "R"

let internal data = { X = [ 1;2;3 ]; Y = ["one"; "two"; "three"] }

printfn "external view:\n%A" data

printfn "internal view:\n%+A" data
```

produces

```
external view:
R

internal view:
{ X = [1; 2; 3]
  Y = ["one"; "two"; "three"] }
```

Large, cyclic, or deeply nested values

Large structured values are formatted to a maximum overall object node count of 10000. Deeply nested values are formatted to a depth of 100. In both cases `...` is used to elide some of the output. For example,

```
type Tree =
    | Tip
    | Node of Tree * Tree

let rec make n =
    if n = 0 then
        Tip
    else
        Node(Tip, make (n-1))

printfn "%A" (make 1000)
```

produces a large output with some parts elided:

```
Node(Tip, Node(Tip, ....Node (..., ...)...))
```

Cycles are detected in the object graphs and `...` is used at places where cycles are detected. For example

```
type R = { mutable Links: R list }
let r = { Links = [] }
r.Links <- [r]
printfn "%A" r
```

produces

```
{ Links = [...] }
```

Lazy, null, and function values

Lazy values are printed as `Value is not created` or equivalent text when the value has not yet been evaluated.

Null values are printed as `null` unless the static type of the value is determined to be a union type where `null` is a permitted representation.

F# function values are printed as their internally generated closure name, for example, `<fun:it@43-7>`.

Customize plain text formatting with `StructuredFormatDisplay`

When using the `%A` specifier, the presence of the `StructuredFormatDisplay` attribute on type declarations is respected. This can be used to specify surrogate text and property to display a value. For example:

```
[<StructuredFormatDisplay("Counts({Clicks})")>]
type Counts = { Clicks:int list}

printfn "%20A" {Clicks=[0..20]}
```

produces

```
Counts([0; 1; 2; 3;
        4; 5; 6; 7;
        8; 9; 10; 11;
        12; 13; 14;
        15; 16; 17;
        18; 19; 20])
```

Customize plain text formatting by overriding `ToString`

The default implementation of `ToString` is observable in F# programming. Often, the default results aren't suitable for use in either programmer-facing information display or user output, and as a result it is common to override the default implementation.

By default, F# record and union types override the implementation of `ToString` with an implementation that uses `sprintf "%+A"`. For example,

```
type Counts = { Clicks:int list }

printfn "%s" ({Clicks=[0..10]}.ToString())
```

produces

```
{ Clicks = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10] }
```

For class types, no default implementation of `ToString` is provided and the .NET default is used, which reports the name of the type. For example,

```
type MyClassType(clicks: int list) =
   member _.Clicks = clicks

let data = [ MyClassType([1..5]); MyClassType([1..5]) ]
printfn "Default structured print gives this:\n%A" data
printfn "Default ToString gives:\n%s" (data.ToString())
```

produces

```
Default structured print gives this:
[MyClassType; MyClassType]
Default ToString gives:
[MyClassType; MyClassType]
```

Adding an override for `ToString` can give better formatting.

```
type MyClassType(clicks: int list) =
   member _.Clicks = clicks
   override _.ToString() = sprintf "MyClassType(%0A)" clicks

let data = [ MyClassType([1..5]); MyClassType([1..5]) ]
printfn "Now structured print gives this:\n%A" data
printfn "Now ToString gives:\n%s" (data.ToString())
```

produces

```
Now structured print gives this:
[MyClassType([1; 2; 3; 4; 5]); MyClassType([1; 2; 3; 4; 5])]
Now ToString gives:
[MyClassType([1; 2; 3; 4; 5]); MyClassType([1; 2; 3; 4; 5])]
```

Customize plain text formatting with `StructuredFormatDisplay` and `ToString`

To achieve consistent formatting for `%A` and `%O` format specifiers, combine the use of `StructuredFormatDisplay` with an override of `ToString`. For example,

```
[<StructuredFormatDisplay("{DisplayText}")>]
type MyRecord =
    {
        a: int
    }
    member this.DisplayText = this.ToString()

    override _.ToString() = "Custom ToString"
```

Evaluating the following definitions

```
let myRec = { a = 10 }
let myTuple = (myRec, myRec)
let s1 = sprintf $"{myRec}"
let s2 = sprintf $"{myTuple}"
let s3 = sprintf $"%A{myTuple}"
let s4 = sprintf $"{[myRec; myRec]}"
let s5 = sprintf $"%A{[myRec; myRec]}"
```

gives the text

```
val myRec: MyRecord = Custom ToString
val myTuple: MyRecord * MyRecord = (Custom ToString, Custom ToString)
val s1: string = "Custom ToString"
val s2: string = "(Custom ToString, Custom ToString)"
val s3: string = "(Custom ToString, Custom ToString)"
val s4: string = "[Custom ToString; Custom ToString]"
val s5: string = "[Custom ToString; Custom ToString]"
```

The use of `StructuredFormatDisplay` with the supporting `DisplayText` property means the fact that the `myRec` is a structural record type is ignored during structured printing, and the override of `ToString()` is preferred in all circumstances.

An implementation of the `System.IFormattable` interface can be added for further customization in the presence of .NET format specifications.

F# Interactive structured printing

F# Interactive (`dotnet fsi`) uses an extended version of structured plain text formatting to report values and allows additional customizability. For more information, see F# Interactive.

Customize debug displays

Debuggers for .NET respect the use of attributes such as `DebuggerDisplay` and `DebuggerTypeProxy`, and these affect the structured display of objects in debugger inspection windows. The F# compiler automatically generated these attributes for discriminated union and record types, but not class, interface, or struct types.

These attributes are ignored in F# plain text formatting, but it can be useful to implement these methods to improve displays when debugging F# types.

See also

- Strings
- Records
- Discriminated Unions
- F# Interactive

Type Providers

8/5/2022 • 2 minutes to read • Edit Online

An F# type provider is a component that provides types, properties, and methods for use in your program. Type Providers generate what are known as **Provided Types**, which are generated by the F# compiler and are based on an external data source.

For example, an F# Type Provider for SQL can generate types representing tables and columns in a relational database. In fact, this is what the SQLProvider Type Provider does.

Provided Types depend on input parameters to a Type Provider. Such input can be a sample data source (such as a JSON schema file), a URL pointing directly to an external service, or a connection string to a data source. A Type Provider can also ensure that groups of types are only expanded on demand; that is, they are expanded if the types are actually referenced by your program. This allows for the direct, on-demand integration of large-scale information spaces such as online data markets in a strongly typed way.

Generative and Erased Type Providers

Type Providers come in two forms: Generative and Erased.

Generative Type Providers produce types that can be written as .NET types into the assembly in which they are produced. This allows them to be consumed from code in other assemblies. This means that the typed representation of the data source must generally be one that is feasible to represent with .NET types.

Erasing Type Providers produce types that can only be consumed in the assembly or project they are generated from. The types are ephemeral; that is, they are not written into an assembly and cannot be consumed by code in other assemblies. They can contain *delayed* members, allowing you to use provided types from a potentially infinite information space. They are useful for using a small subset of a large and interconnected data source.

Commonly used Type Providers

The following widely-used libraries contain Type Providers for different uses:

- FSharp.Data includes Type Providers for JSON, XML, CSV, and HTML document formats and resources.
- SQLProvider provides strongly typed access to relation databases through object mapping and F# LINQ queries against these data sources.
- FSharp.Data.SqlClient has a set of type providers for compile-time checked embedding of T-SQL in F#.
- Azure Storage Type provider provides types for Azure Blobs, Tables, and Queues, allowing you to access these resources without needing to specify resource names as strings throughout your program.
- FSharp.Data.GraphQL contains the **GraphQLProvider**, which provides types based on a GraphQL server specified by URL.

Where necessary, you can create your own custom type providers, or reference type providers that have been created by others. For example, assume your organization has a data service providing a large and growing number of named data sets, each with its own stable data schema. You may choose to create a type provider that reads the schemas and presents the latest available data sets to the programmer in a strongly typed way.

See also

- Tutorial: Create a Type Provider
- F# Language Reference

Tutorial: Create a Type Provider

8/5/2022 • 35 minutes to read • Edit Online

The type provider mechanism in F# is a significant part of its support for information rich programming. This tutorial explains how to create your own type providers by walking you through the development of several simple type providers to illustrate the basic concepts. For more information about the type provider mechanism in F#, see Type Providers.

The F# ecosystem contains a range of type providers for commonly used Internet and enterprise data services. For example:

- FSharp.Data includes type providers for JSON, XML, CSV and HTML document formats.
- SwaggerProvider includes two generative type providers that generate object model and HTTP clients for APIs described by OpenApi 3.0 and Swagger 2.0 schemas.
- FSharp.Data.SqlClient has a set of type providers for compile-time checked embedding of T-SQL in F#.

You can create custom type providers, or you can reference type providers that others have created. For example, your organization could have a data service that provides a large and growing number of named data sets, each with its own stable data schema. You can create a type provider that reads the schemas and presents the current data sets to the programmer in a strongly typed way.

Before You Start

The type provider mechanism is primarily designed for injecting stable data and service information spaces into the F# programming experience.

This mechanism isn’t designed for injecting information spaces whose schema changes during program execution in ways that are relevant to program logic. Also, the mechanism isn't designed for intra-language meta-programming, even though that domain contains some valid uses. You should use this mechanism only where necessary and where the development of a type provider yields very high value.

You should avoid writing a type provider where a schema isn't available. Likewise, you should avoid writing a type provider where an ordinary (or even an existing) .NET library would suffice.

Before you start, you might ask the following questions:

- Do you have a schema for your information source? If so, what’s the mapping into the F# and .NET type system?
- Can you use an existing (dynamically typed) API as a starting point for your implementation?
- Will you and your organization have enough uses of the type provider to make writing it worthwhile? Would a normal .NET library meet your needs?
- How much will your schema change?
- Will it change during coding?
- Will it change between coding sessions?
- Will it change during program execution?

Type providers are best suited to situations where the schema is stable at run time and during the lifetime of compiled code.

A Simple Type Provider

This sample is Samples.HelloWorldTypeProvider, similar to the samples in the `examples` directory of the F# Type Provider SDK. The provider makes available a "type space" that contains 100 erased types, as the following code shows by using F# signature syntax and omitting the details for all except `Type1`. For more information about erased types, see Details About Erased Provided Types later in this topic.

```
namespace Samples.HelloWorldTypeProvider

type Type1 =
    /// This is a static property.
    static member StaticProperty : string

    /// This constructor takes no arguments.
    new : unit -> Type1

    /// This constructor takes one argument.
    new : data:string -> Type1

    /// This is an instance property.
    member InstanceProperty : int

    /// This is an instance method.
    member InstanceMethod : x:int -> char

    nested type NestedType =
        /// This is StaticProperty1 on NestedType.
        static member StaticProperty1 : string
        …
        /// This is StaticProperty100 on NestedType.
        static member StaticProperty100 : string

type Type2 =
…
…

type Type100 =
…
```

Note that the set of types and members provided is statically known. This example doesn't leverage the ability of providers to provide types that depend on a schema. The implementation of the type provider is outlined in the following code, and the details are covered in later sections of this topic.

> **WARNING**
>
> There may be differences between this code and the online samples.

```
namespace Samples.FSharp.HelloWorldTypeProvider

open System
open System.Reflection
open ProviderImplementation.ProvidedTypes
open FSharp.Core.CompilerServices
open FSharp.Quotations

// This type defines the type provider. When compiled to a DLL, it can be added
// as a reference to an F# command-line compilation, script, or project.
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =

  // Inheriting from this type provides implementations of ITypeProvider
  // in terms of the provided types below.
  inherit TypeProviderForNamespaces(config)

  let namespaceName = "Samples.HelloWorldTypeProvider"
  let thisAssembly = Assembly.GetExecutingAssembly()

  // Make one provided type, called TypeN.
  let makeOneProvidedType (n:int) =
  …
  // Now generate 100 types
  let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

  // And add them to the namespace
  do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()
```

To use this provider, open a separate instance of Visual Studio, create an F# script, and then add a reference to the provider from your script by using #r as the following code shows:

```
#r @".\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]
[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35
```

Then look for the types under the `Samples.HelloWorldTypeProvider` namespace that the type provider generated.

Before you recompile the provider, make sure that you have closed all instances of Visual Studio and F# Interactive that are using the provider DLL. Otherwise, a build error will occur because the output DLL will be locked.

To debug this provider by using print statements, make a script that exposes a problem with the provider, and then use the following code:

```
fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

To debug this provider by using Visual Studio, open the Developer Command Prompt for Visual Studio with administrative credentials, and run the following command:

```
devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

As an alternative, open Visual Studio, open the Debug menu, choose `Debug/Attach to process…`, and attach to another `devenv` process where you're editing your script. By using this method, you can more easily target particular logic in the type provider by interactively typing expressions into the second instance (with full IntelliSense and other features).

You can disable Just My Code debugging to better identify errors in generated code. For information about how to enable or disable this feature, see Navigating through Code with the Debugger. Also, you can also set first-chance exception catching by opening the `Debug` menu and then choosing `Exceptions` or by choosing the Ctrl+Alt+E keys to open the `Exceptions` dialog box. In that dialog box, under `Common Language Runtime Exceptions`, select the `Thrown` check box.

Implementation of the Type Provider

This section walks you through the principal sections of the type provider implementation. First, you define the type for the custom type provider itself:

```
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =
```

This type must be public, and you must mark it with the TypeProvider attribute so that the compiler will recognize the type provider when a separate F# project references the assembly that contains the type. The *config* parameter is optional, and, if present, contains contextual configuration information for the type provider instance that the F# compiler creates.

Next, you implement the ITypeProvider interface. In this case, you use the `TypeProviderForNamespaces` type from the `ProvidedTypes` API as a base type. This helper type can provide a finite collection of eagerly provided namespaces, each of which directly contains a finite number of fixed, eagerly provided types. In this context, the provider *eagerly* generates types even if they aren't needed or used.

```
inherit TypeProviderForNamespaces(config)
```

Next, define local private values that specify the namespace for the provided types, and find the type provider assembly itself. This assembly is used later as the logical parent type of the erased types that are provided.

```
let namespaceName = "Samples.HelloWorldTypeProvider"
let thisAssembly = Assembly.GetExecutingAssembly()
```

Next, create a function to provide each of the types Type1…Type100. This function is explained in more detail later in this topic.

```
let makeOneProvidedType (n:int) = …
```

Next, generate the 100 provided types:

```
let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]
```

Next, add the types as a provided namespace:

```
do this.AddNamespace(namespaceName, types)
```

Finally, add an assembly attribute that indicates that you are creating a type provider DLL:

```
[<assembly:TypeProviderAssembly>]
do()
```

Providing One Type And Its Members

The `makeOneProvidedType` function does the real work of providing one of the types.

```
let makeOneProvidedType (n:int) =
…
```

This step explains the implementation of this function. First, create the provided type (for example, Type1, when n = 1, or Type57, when n = 57).

```
// This is the provided type. It is an erased provided type and, in compiled code,
// will appear as type 'obj'.
let t = ProvidedTypeDefinition(thisAssembly, namespaceName,
                               "Type" + string n,
                               baseType = Some typeof<obj>)
```

You should note the following points:

- This provided type is erased. Because you indicate that the base type is `obj`, instances will appear as values of type obj in compiled code.
- When you specify a non-nested type, you must specify the assembly and namespace. For erased types, the assembly should be the type provider assembly itself.

Next, add XML documentation to the type. This documentation is delayed, that is, computed on-demand if the host compiler needs it.

```
t.AddXmlDocDelayed (fun () -> $"""This provided type {"Type" + string n}""")
```

Next you add a provided static property to the type:

```
let staticProp = ProvidedProperty(propertyName = "StaticProperty",
                                  propertyType = typeof<string>,
                                  isStatic = true,
                                  getterCode = (fun args -> <@@ "Hello!" @@>))
```

Getting this property will always evaluate to the string "Hello!". The `GetterCode` for the property uses an F# quotation, which represents the code that the host compiler generates for getting the property. For more information about quotations, see Code Quotations (F#).

Add XML documentation to the property.

```
staticProp.AddXmlDocDelayed(fun () -> "This is a static property")
```

Now attach the provided property to the provided type. You must attach a provided member to one and only one type. Otherwise, the member will never be accessible.

```
t.AddMember staticProp
```

Now create a provided constructor that takes no parameters.

```
let ctor = ProvidedConstructor(parameters = [ ],
                               invokeCode = (fun args -> <@@ "The object data" :> obj @@>))
```

The `InvokeCode` for the constructor returns an F# quotation, which represents the code that the host compiler generates when the constructor is called. For example, you can use the following constructor:

```
new Type10()
```

An instance of the provided type will be created with underlying data "The object data". The quoted code includes a conversion to obj because that type is the erasure of this provided type (as you specified when you declared the provided type).

Add XML documentation to the constructor, and add the provided constructor to the provided type:

```
ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

t.AddMember ctor
```

Create a second provided constructor that takes one parameter:

```
let ctor2 =
ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],
                    invokeCode = (fun args -> <@@ (%%(args[0]) : string) :> obj @@>))
```

The `InvokeCode` for the constructor again returns an F# quotation, which represents the code that the host compiler generated for a call to the method. For example, you can use the following constructor:

```
new Type10("ten")
```

An instance of the provided type is created with underlying data "ten". You may have already noticed that the `InvokeCode` function returns a quotation. The input to this function is a list of expressions, one per constructor parameter. In this case, an expression that represents the single parameter value is available in `args[0]`. The code for a call to the constructor coerces the return value to the erased type `obj`. After you add the second provided constructor to the type, you create a provided instance property:

```
let instanceProp =
    ProvidedProperty(propertyName = "InstanceProperty",
                     propertyType = typeof<int>,
                     getterCode= (fun args ->
                        <@@ ((%%(args[0]) : obj) :?> string).Length @@>))
instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")
t.AddMember instanceProp
```

Getting this property will return the length of the string, which is the representation object. The `GetterCode` property returns an F# quotation that specifies the code that the host compiler generates to get the property. Like `InvokeCode`, the `GetterCode` function returns a quotation. The host compiler calls this function with a list of arguments. In this case, the arguments include just the single expression that represents the instance upon which the getter is being called, which you can access by using `args[0]`. The implementation of `GetterCode` then splices into the result quotation at the erased type `obj`, and a cast is used to satisfy the compiler's mechanism for checking types that the object is a string. The next part of `makeOneProvidedType` provides an instance method with one parameter.

```
let instanceMeth =
    ProvidedMethod(methodName = "InstanceMethod",
                   parameters = [ProvidedParameter("x",typeof<int>)],
                   returnType = typeof<char>,
                   invokeCode = (fun args ->
                       <@@ ((%%(args[0]) : obj) :?> string).Chars(%%(args[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")
// Add the instance method to the type.
t.AddMember instanceMeth
```

Finally, create a nested type that contains 100 nested properties. The creation of this nested type and its properties is delayed, that is, computed on-demand.

```
t.AddMembersDelayed(fun () ->
  let nestedType = ProvidedTypeDefinition("NestedType", Some typeof<obj>)

  nestedType.AddMembersDelayed (fun () ->
    let staticPropsInNestedType =
      [
          for i in 1 .. 100 ->
              let valueOfTheProperty = "I am string "  + string i

              let p =
                ProvidedProperty(propertyName = "StaticProperty" + string i,
                  propertyType = typeof<string>,
                  isStatic = true,
                  getterCode= (fun args -> <@@ valueOfTheProperty @@>))

              p.AddXmlDocDelayed(fun () ->
                  $"This is StaticProperty{i} on NestedType")

              p
      ]

    staticPropsInNestedType)

  [nestedType])
```

Details about Erased Provided Types

The example in this section provides only *erased provided types*, which are particularly useful in the following situations:

- When you are writing a provider for an information space that contains only data and methods.
- When you are writing a provider where accurate runtime-type semantics aren't critical for practical use of the information space.
- When you are writing a provider for an information space that is so large and interconnected that it isn't technically feasible to generate real .NET types for the information space.

In this example, each provided type is erased to type `obj`, and all uses of the type will appear as type `obj` in compiled code. In fact, the underlying objects in these examples are strings, but the type will appear as `System.Object` in .NET compiled code. As with all uses of type erasure, you can use explicit boxing, unboxing, and casting to subvert erased types. In this case, a cast exception that isn't valid may result when the object is used. A provider runtime can define its own private representation type to help protect against false representations. You can't define erased types in F# itself. Only provided types may be erased. You must understand the ramifications, both practical and semantic, of using either erased types for your type provider or a provider that provides erased types. An erased type has no real .NET type. Therefore, you cannot do accurate reflection over the type, and you might subvert erased types if you use runtime casts and other techniques that rely on exact runtime type semantics. Subversion of erased types frequently results in type cast exceptions at run time.

Choosing Representations for Erased Provided Types

For some uses of erased provided types, no representation is required. For example, the erased provided type might contain only static properties and members and no constructors, and no methods or properties would return an instance of the type. If you can reach instances of an erased provided type, you must consider the following questions:

What is the erasure of a provided type?

- The erasure of a provided type is how the type appears in compiled .NET code.
- The erasure of a provided erased class type is always the first non-erased base type in the inheritance chain of the type.
- The erasure of a provided erased interface type is always `System.Object`.

What are the representations of a provided type?

- The set of possible objects for an erased provided type are called its representations. In the example in this document, the representations of all the erased provided types `Type1..Type100` are always string objects.

All representations of a provided type must be compatible with the erasure of the provided type. (Otherwise, either the F# compiler will give an error for a use of the type provider, or unverifiable .NET code that isn't valid will be generated. A type provider isn't valid if it returns code that gives a representation that isn't valid.)

You can choose a representation for provided objects by using either of the following approaches, both of which are very common:

- If you're simply providing a strongly typed wrapper over an existing .NET type, it often makes sense for your type to erase to that type, use instances of that type as representations, or both. This approach is appropriate when most of the existing methods on that type still make sense when using the strongly typed version.
- If you want to create an API that differs significantly from any existing .NET API, it makes sense to create runtime types that will be the type erasure and representations for the provided types.

The example in this document uses strings as representations of provided objects. Frequently, it may be appropriate to use other objects for representations. For example, you may use a dictionary as a property bag:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))
```

As an alternative, you may define a type in your type provider that will be used at run time to form the representation, along with one or more runtime operations:

```
type DataObject() =
    let data = Dictionary<string,obj>()
    member x.RuntimeOperation() = data.Count
```

Provided members can then construct instances of this object type:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))
```

In this case, you may (optionally) use this type as the type erasure by specifying this type as the `baseType` when constructing the `ProvidedTypeDefinition`:

```
ProvidedTypeDefinition(…, baseType = Some typeof<DataObject> )
…
ProvidedConstructor(…, InvokeCode = (fun args -> <@@ new DataObject() @@>), …)
```

Key Lessons

The previous section explained how to create a simple erasing type provider that provides a range of types, properties, and methods. This section also explained the concept of type erasure, including some of the advantages and disadvantages of providing erased types from a type provider, and discussed representations of erased types.

A Type Provider That Uses Static Parameters

The ability to parameterize type providers by static data enables many interesting scenarios, even in cases when the provider doesn't need to access any local or remote data. In this section, you'll learn some basic techniques for putting together such a provider.

Type Checked Regex Provider

Imagine that you want to implement a type provider for regular expressions that wraps the .NET Regex libraries in an interface that provides the following compile-time guarantees:

- Verifying whether a regular expression is valid.
- Providing named properties on matches that are based on any group names in the regular expression.

This section shows you how to use type providers to create a `RegexTyped` type that the regular expression pattern parameterizes to provide these benefits. The compiler will report an error if the supplied pattern isn't valid, and the type provider can extract the groups from the pattern so that you can access them by using named properties on matches. When you design a type provider, you should consider how its exposed API should look to end users and how this design will translate to .NET code. The following example shows how to use such an API to get the components of the area code:

```
type T = RegexTyped< @"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)">
let reg = T()
let result = T.IsMatch("425-555-2345")
let r = reg.Match("425-555-2345").Group_AreaCode.Value //r equals "425"
```

The following example shows how the type provider translates these calls:

```
let reg = new Regex(@"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)")
let result = reg.IsMatch("425-123-2345")
let r = reg.Match("425-123-2345").Groups["AreaCode"].Value //r equals "425"
```

Note the following points:

- The standard Regex type represents the parameterized `RegexTyped` type.
- The `RegexTyped` constructor results in a call to the Regex constructor, passing in the static type argument for the pattern.

- The results of the `Match` method are represented by the standard Match type.
- Each named group results in a provided property, and accessing the property results in a use of an indexer on a match's `Groups` collection.

The following code is the core of the logic to implement such a provider, and this example omits the addition of all members to the provided type. For information about each added member, see the appropriate section later in this topic.

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

          match parameterValues with
          | [| :? string as pattern|] ->

            // Create an instance of the regular expression.
            //
            // This will fail with System.ArgumentException if the regular expression is not valid.
            // The exception will escape the type provider and be reported in client code.
            let r = System.Text.RegularExpressions.Regex(pattern)

            // Declare the typed regex provided type.
            // The type erasure of this type is 'obj', even though the representation will always be a Regex
            // This, combined with hiding the object methods, makes the IntelliSense experience simpler.
            let ty =
              ProvidedTypeDefinition(
                thisAssembly,
                rootNamespace,
                typeName,
                baseType = Some baseTy)

            ...

            ty
          | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Note the following points:

- The type provider takes two static parameters: the `pattern`, which is mandatory, and the `options`, which are optional (because a default value is provided).

- After the static arguments are supplied, you create an instance of the regular expression. This instance will throw an exception if the Regex is malformed, and this error will be reported to users.
- Within the `DefineStaticParameters` callback, you define the type that will be returned after the arguments are supplied.
- This code sets `HideObjectMethods` to true so that the IntelliSense experience will remain streamlined. This attribute causes the `Equals`, `GetHashCode`, `Finalize`, and `GetType` members to be suppressed from IntelliSense lists for a provided object.
- You use `obj` as the base type of the method, but you'll use a `Regex` object as the runtime representation of this type, as the next example shows.
- The call to the `Regex` constructor throws an ArgumentException when a regular expression isn't valid. The compiler catches this exception and reports an error message to the user at compile time or in the Visual Studio editor. This exception enables regular expressions to be validated without running an application.

The type defined above isn't useful yet because it doesn't contain any meaningful methods or properties. First, add a static `IsMatch` method:

```
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%%args[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified input string."
ty.AddMember isMatch
```

The previous code defines a method `IsMatch`, which takes a string as input and returns a `bool`. The only tricky part is the use of the `args` argument within the `InvokeCode` definition. In this example, `args` is a list of quotations that represents the arguments to this method. If the method is an instance method, the first argument represents the `this` argument. However, for a static method, the arguments are all just the explicit arguments to the method. Note that the type of the quoted value should match the specified return type (in this case, `bool`). Also note that this code uses the `AddXmlDoc` method to make sure that the provided method also has useful documentation, which you can supply through IntelliSense.

Next, add an instance Match method. However, this method should return a value of a provided `Match` type so that the groups can be accessed in a strongly typed fashion. Thus, you first declare the `Match` type. Because this type depends on the pattern that was supplied as a static argument, this type must be nested within the parameterized type definition:

```
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

ty.AddMember matchTy
```

You then add one property to the Match type for each group. At run time, a match is represented as a Match value, so the quotation that defines the property must use the Groups indexed property to get the relevant group.

```
for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
    let prop =
      ProvidedProperty(
        propertyName = group,
        propertyType = typeof<Group>,
        getterCode = fun args -> <@@ ((%%args[0]:obj) :?> Match).Groups[group] @@>)
        prop.AddXmlDoc($"""Gets the ""{group}"" group from this match""")
    matchTy.AddMember prop
```

Again, note that you're adding XML documentation to the provided property. Also note that a property can be read if a `GetterCode` function is provided, and the property can be written if a `SetterCode` function is provided, so the resulting property is read only.

Now you can create an instance method that returns a value of this `Match` type:

```
let matchMethod =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%%args[0]:obj) :?> Regex).Match(%%args[1]) :> obj @@>)

matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this regular
expression"

ty.AddMember matchMeth
```

Because you are creating an instance method, `args[0]` represents the `RegexTyped` instance on which the method is being called, and `args[1]` is the input argument.

Finally, provide a constructor so that instances of the provided type can be created.

```
let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern, options) :> obj @@>)

ctor.AddXmlDoc("Initializes a regular expression instance.")

ty.AddMember ctor
```

The constructor merely erases to the creation of a standard .NET Regex instance, which is again boxed to an object because `obj` is the erasure of the provided type. With that change, the sample API usage that specified earlier in the topic works as expected. The following code is complete and final:

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types.
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
```

```
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.

                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.

                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,
                        typeName,
                        baseType = Some baseTy)

                ty.AddXmlDoc "A strongly typed interface to the regular expression '%s'"

                // Provide strongly typed version of Regex.IsMatch static method.
                let isMatch =
                    ProvidedMethod(
                        methodName = "IsMatch",
                        parameters = [ProvidedParameter("input", typeof<string>)],
                        returnType = typeof<bool>,
                        isStatic = true,
                        invokeCode = fun args -> <@@ Regex.IsMatch(%%args[0], pattern) @@>)

                isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified
input string"

                ty.AddMember isMatch

                // Provided type for matches
                // Again, erase to obj even though the representation will always be a Match
                let matchTy =
                    ProvidedTypeDefinition(
                        "MatchType",
                        baseType = Some baseTy,
                        hideObjectMethods = true)

                // Nest the match type within parameterized Regex type.
                ty.AddMember matchTy

                // Add group properties to match type
                for group in r.GetGroupNames() do
                    // Ignore the group named 0, which represents all input.
                    if group <> "0" then
                        let prop =
                          ProvidedProperty(
                            propertyName = group,
                            propertyType = typeof<Group>,
                            getterCode = fun args -> <@@ ((%%args[0]:obj) :?> Match).Groups[group] @@>)
                        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
                        matchTy.AddMember(prop)

                // Provide strongly typed version of Regex.Match instance method.
                let matchMeth =
                  ProvidedMethod(
                    methodName = "Match",
```

```
                     parameters = [ProvidedParameter("input", typeof<string>)],
                     returnType = matchTy,
                     invokeCode = fun args -> <@@ ((%%args[0]:obj) :?> Regex).Match(%%args[1]) :> obj @@>)
                 matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this
regular expression"

                 ty.AddMember matchMeth

                 // Declare a constructor.
                 let ctor =
                   ProvidedConstructor(
                     parameters = [],
                     invokeCode = fun args -> <@@ Regex(pattern) :> obj @@>)

                 // Add documentation to the constructor.
                 ctor.AddXmlDoc "Initializes a regular expression instance"

                 ty.AddMember ctor

                 ty
             | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Key Lessons

This section explained how to create a type provider that operates on its static parameters. The provider checks the static parameter and provides operations based on its value.

A Type Provider That Is Backed By Local Data

Frequently you might want type providers to present APIs based on not only static parameters but also information from local or remote systems. This section discusses type providers that are based on local data, such as local data files.

Simple CSV File Provider

As a simple example, consider a type provider for accessing scientific data in Comma Separated Value (CSV) format. This section assumes that the CSV files contain a header row followed by floating point data, as the following table illustrates:

DISTANCE (METER)	TIME (SECOND)
50.0	3.7
100.0	5.2
150.0	6.4

This section shows how to provide a type that you can use to get rows with a `Distance` property of type `float<meter>` and a `Time` property of type `float<second>`. For simplicity, the following assumptions are made:

- Header names are either unit-less or have the form "Name (unit)" and don't contain commas.
- Units are all System International (SI) units as the FSharp.Data.UnitSystems.SI.UnitNames Module (F#) module defines.
- Units are all simple (for example, meter) rather than compound (for example, meter/second).

- All columns contain floating point data.

A more complete provider would loosen these restrictions.

Again the first step is to consider how the API should look. Given an `info.csv` file with the contents from the previous table (in comma-separated format), users of the provider should be able to write code that resembles the following example:

```
let info = new MiniCsv<"info.csv">()
for row in info.Data do
let time = row.Time
printfn $"{float time}"
```

In this case, the compiler should convert these calls into something like the following example:

```
let info = new CsvFile("info.csv")
for row in info.Data do
let (time:float) = row[1]
printfn $"%f{float time}"
```

The optimal translation will require the type provider to define a real `CsvFile` type in the type provider's assembly. Type providers often rely on a few helper types and methods to wrap important logic. Because measures are erased at run time, you can use a `float[]` as the erased type for a row. The compiler will treat different columns as having different measure types. For example, the first column in our example has type `float<meter>`, and the second has `float<second>`. However, the erased representation can remain quite simple.

The following code shows the core of the implementation.

```
// Simple type wrapping CSV data
type CsvFile(filename) =
    // Cache the sequence of all data lines (all lines but the first)
    let data =
        seq {
            for line in File.ReadAllLines(filename) |> Seq.skip 1 ->
                line.Split(',') |> Array.map float
        }
        |> Seq.cache
    member _.Data = data

[<TypeProvider>]
type public MiniCsvProvider(cfg:TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces(cfg)

    // Get the assembly and namespace used to house the provided types.
    let asm = System.Reflection.Assembly.GetExecutingAssembly()
    let ns = "Samples.FSharp.MiniCsvProvider"

    // Create the main provided type.
    let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

    // Parameterize the type by the file to use as a template.
    let filename = ProvidedStaticParameter("filename", typeof<string>)
    do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

        // Resolve the filename relative to the resolution folder.
        let resolvedFilename = Path.Combine(cfg.ResolutionFolder, filename)

        // Get the first line from the file.
        let headerLine = File.ReadLines(resolvedFilename) |> Seq.head

        // Define a provided type for each row, erasing to a float[].
        let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))
```

```
        // Extract header names from the file, splitting on commas.
        // use Regex matching to get the position in the row at which the field occurs
        let headers = Regex.Matches(headerLine, "[^,]+")

        // Add one property per CSV field.
        for i in 0 .. headers.Count - 1 do
            let headerText = headers[i].Value

            // Try to decompose this header into a name and unit.
            let fieldName, fieldTy =
                let m = Regex.Match(headerText, @"(?<field>.+) \((?<unit>.+)\)")
                if m.Success then

                    let unitName = m.Groups["unit"].Value
                    let units = ProvidedMeasureBuilder.Default.SI unitName
                    m.Groups["field"].Value, ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>,
[units])

                else
                    // no units, just treat it as a normal float
                    headerText, typeof<float>

            let prop =
                ProvidedProperty(fieldName, fieldTy,
                    getterCode = fun [row] -> <@@ (%%row:float[])[i] @@>)

            // Add metadata that defines the property's location in the referenced file.
            prop.AddDefinitionLocation(1, headers[i].Index + 1, filename)
            rowTy.AddMember(prop)

        // Define the provided type, erasing to CsvFile.
        let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

        // Add a parameterless constructor that loads the file that was used to define the schema.
        let ctor0 =
            ProvidedConstructor([],
                invokeCode = fun [] -> <@@ CsvFile(resolvedFilename) @@>)
        ty.AddMember ctor0

        // Add a constructor that takes the file name to load.
        let ctor1 = ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],
            invokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>)
        ty.AddMember ctor1

        // Add a more strongly typed Data property, which uses the existing property at run time.
        let prop =
            ProvidedProperty("Data", typedefof<seq<_>>.MakeGenericType(rowTy),
                getterCode = fun [csvFile] -> <@@ (%%csvFile:CsvFile).Data @@>)
        ty.AddMember prop

        // Add the row type as a nested type.
        ty.AddMember rowTy
        ty)

    // Add the type to the namespace.
    do this.AddNamespace(ns, [csvTy])
```

Note the following points about the implementation:

- Overloaded constructors allow either the original file or one that has an identical schema to be read. This pattern is common when you write a type provider for local or remote data sources, and this pattern allows a local file to be used as the template for remote data.
- You can use the TypeProviderConfig value that's passed in to the type provider constructor to resolve relative file names.

- You can use the `AddDefinitionLocation` method to define the location of the provided properties. Therefore, if you use `Go To Definition` on a provided property, the CSV file will open in Visual Studio.
- You can use the `ProvidedMeasureBuilder` type to look up the SI units and to generate the relevant `float<_>` types.

Key Lessons

This section explained how to create a type provider for a local data source with a simple schema that's contained in the data source itself.

Going Further

The following sections include suggestions for further study.

A Look at the Compiled Code for Erased Types

To give you some idea of how the use of the type provider corresponds to the code that's emitted, look at the following function by using the `HelloWorldTypeProvider` that's used earlier in this topic.

```
let function1 () =
    let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")
    obj1.InstanceProperty
```

Here's an image of the resulting code decompiled by using ildasm.exe:

```
.class public abstract auto ansi sealed Module1
extends [mscorlib]System.Object
{
.custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAtt
ribute::.ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags)
= ( 01 00 07 00 00 00 00 00 )
.method public static int32  function1() cil managed
{
// Code size       24 (0x18)
.maxstack  3
.locals init ([0] object obj1)
IL_0000:  nop
IL_0001:  ldstr      "some data"
IL_0006:  unbox.any  [mscorlib]System.Object
IL_000b:  stloc.0
IL_000c:  ldloc.0
IL_000d:  call       !!0 [FSharp.Core_2]Microsoft.FSharp.Core.LanguagePrimit
ives/IntrinsicFunctions::UnboxGeneric<string>(object)
IL_0012:  callvirt   instance int32 [mscorlib_3]System.String::get_Length()
IL_0017:  ret
} // end of method Module1::function1

} // end of class Module1
```

As the example shows, all mentions of the type `Type1` and the `InstanceProperty` property have been erased, leaving only operations on the runtime types involved.

Design and Naming Conventions for Type Providers

Observe the following conventions when authoring type providers.

Providers for Connectivity Protocols In general, names of most provider DLLs for data and service connectivity protocols, such as OData or SQL connections, should end in `TypeProvider` or `TypeProviders`. For example, use a DLL name that resembles the following string:

`Fabrikam.Management.BasicTypeProviders.dll`

Ensure that your provided types are members of the corresponding namespace, and indicate the connectivity protocol that you implemented:

```
Fabrikam.Management.BasicTypeProviders.WmiConnection<…>
Fabrikam.Management.BasicTypeProviders.DataProtocolConnection<…>
```

Utility Providers for General Coding. For a utility type provider such as that for regular expressions, the type provider may be part of a base library, as the following example shows:

```
#r "Fabrikam.Core.Text.Utilities.dll"
```

In this case, the provided type would appear at an appropriate point according to normal .NET design conventions:

```
open Fabrikam.Core.Text.RegexTyped

let regex = new RegexTyped<"a+b+a+b+">()
```

Singleton Data Sources. Some type providers connect to a single dedicated data source and provide only data. In this case, you should drop the `TypeProvider` suffix and use normal conventions for .NET naming:

```
#r "Fabrikam.Data.Freebase.dll"

let data = Fabrikam.Data.Freebase.Astronomy.Asteroids
```

For more information, see the `GetConnection` design convention that's described later in this topic.

Design Patterns for Type Providers

The following sections describe design patterns you can use when authoring type providers.

The GetConnection Design Pattern

Most type providers should be written to use the `GetConnection` pattern that's used by the type providers in FSharp.Data.TypeProviders.dll, as the following example shows:

```
#r "Fabrikam.Data.WebDataStore.dll"

type Service = Fabrikam.Data.WebDataStore<…static connection parameters…>

let connection = Service.GetConnection(…dynamic connection parameters…)

let data = connection.Astronomy.Asteroids
```

Type Providers Backed By Remote Data and Services

Before you create a type provider that's backed by remote data and services, you must consider a range of issues that are inherent in connected programming. These issues include the following considerations:

- schema mapping
- liveness and invalidation in the presence of schema change
- schema caching
- asynchronous implementations of data access operations
- supporting queries, including LINQ queries

- credentials and authentication

This topic doesn't explore these issues further.

Additional Authoring Techniques

When you write your own type providers, you might want to use the following additional techniques.

Creating Types and Members On-Demand

The ProvidedType API has delayed versions of AddMember.

```
type ProvidedType =
    member AddMemberDelayed  : (unit -> MemberInfo)      -> unit
    member AddMembersDelayed : (unit -> MemberInfo list) -> unit
```

These versions are used to create on-demand spaces of types.

Providing Array types and Generic Type Instantiations

You make provided members (whose signatures include array types, byref types, and instantiations of generic types) by using the normal `MakeArrayType`, `MakePointerType`, and `MakeGenericType` on any instance of Type, including `ProvidedTypeDefinitions`.

> **NOTE**
>
> In some cases you may have to use the helper in `ProvidedTypeBuilder.MakeGenericType`. See the Type Provider SDK documentation for more details.

Providing Unit of Measure Annotations

The ProvidedTypes API provides helpers for providing measure annotations. For example, to provide the type `float<kg>`, use the following code:

```
let measures = ProvidedMeasureBuilder.Default
let kg = measures.SI "kilogram"
let m = measures.SI "meter"
let float_kg = measures.AnnotateType(typeof<float>,[kg])
```

To provide the type `Nullable<decimal<kg/m^2>>`, use the following code:

```
let kgpm2 = measures.Ratio(kg, measures.Square m)
let dkgpm2 = measures.AnnotateType(typeof<decimal>,[kgpm2])
let nullableDecimal_kgpm2 = typedefof<System.Nullable<_>>.MakeGenericType [|dkgpm2 |]
```

Accessing Project-Local or Script-Local Resources

Each instance of a type provider can be given a `TypeProviderConfig` value during construction. This value contains the "resolution folder" for the provider (that is, the project folder for the compilation or the directory that contains a script), the list of referenced assemblies, and other information.

Invalidation

Providers can raise invalidation signals to notify the F# language service that the schema assumptions may have changed. When invalidation occurs, a typecheck is redone if the provider is being hosted in Visual Studio. This signal will be ignored when the provider is hosted in F# Interactive or by the F# Compiler (fsc.exe).

Caching Schema Information

Providers must often cache access to schema information. The cached data should be stored by using a file name that's given as a static parameter or as user data. An example of schema caching is the `LocalSchemaFile` parameter in the type providers in the `FSharp.Data.TypeProviders` assembly. In the implementation of these providers, this static parameter directs the type provider to use the schema information in the specified local file instead of accessing the data source over the network. To use cached schema information, you must also set the static parameter `ForceUpdate` to `false`. You could use a similar technique to enable online and offline data access.

Backing Assembly

When you compile a `.dll` or `.exe` file, the backing .dll file for generated types is statically linked into the resulting assembly. This link is created by copying the Intermediate Language (IL) type definitions and any managed resources from the backing assembly into the final assembly. When you use F# Interactive, the backing .dll file isn't copied and is instead loaded directly into the F# Interactive process.

Exceptions and Diagnostics from Type Providers

All uses of all members from provided types may throw exceptions. In all cases, if a type provider throws an exception, the host compiler attributes the error to a specific type provider.

- Type provider exceptions should never result in internal compiler errors.
- Type providers can't report warnings.
- When a type provider is hosted in the F# compiler, an F# development environment, or F# Interactive, all exceptions from that provider are caught. The Message property is always the error text, and no stack trace appears. If you’re going to throw an exception, you can throw the following examples: `System.NotSupportedException`, `System.IO.IOException`, `System.Exception`.

Providing Generated Types

So far, this document has explained how to provide erased types. You can also use the type provider mechanism in F# to provide generated types, which are added as real .NET type definitions into the users' program. You must refer to generated provided types by using a type definition.

```
open Microsoft.FSharp.TypeProviders

type Service = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

The ProvidedTypes-0.2 helper code that is part of the F# 3.0 release has only limited support for providing generated types. The following statements must be true for a generated type definition:

- `isErased` must be set to `false`.
- The generated type must be added to a newly constructed `ProvidedAssembly()`, which represents a container for generated code fragments.
- The provider must have an assembly that has an actual backing .NET .dll file with a matching .dll file on disk.

Rules and Limitations

When you write type providers, keep the following rules and limitations in mind.

Provided types must be reachable

All provided types should be reachable from the non-nested types. The non-nested types are given in the call to the `TypeProviderForNamespaces` constructor or a call to `AddNamespace`. For example, if the provider provides a type `StaticClass.P : T`, you must ensure that T is either a non-nested type or nested under one.

For example, some providers have a static class such as `DataTypes` that contain these `T1, T2, T3, ...` types.

Otherwise, the error says that a reference to type T in assembly A was found, but the type couldn't be found in that assembly. If this error appears, verify that all your subtypes can be reached from the provider types. Note: These `T1, T2, T3...` types are referred to as the *on-the-fly* types. Remember to put them in an accessible namespace or a parent type.

Limitations of the Type Provider Mechanism

The type provider mechanism in F# has the following limitations:

- The underlying infrastructure for type providers in F# doesn't support provided generic types or provided generic methods.
- The mechanism doesn't support nested types with static parameters.

Development Tips

You might find the following tips helpful during the development process:

Run two instances of Visual Studio

You can develop the type provider in one instance and test the provider in the other because the test IDE will take a lock on the .dll file that prevents the type provider from being rebuilt. Thus, you must close the second instance of Visual Studio while the provider is built in the first instance, and then you must reopen the second instance after the provider is built.

Debug type providers by using invocations of fsc.exe

You can invoke type providers by using the following tools:

- fsc.exe (The F# command line compiler)
- fsi.exe (The F# Interactive compiler)
- devenv.exe (Visual Studio)

You can often debug type providers most easily by using fsc.exe on a test script file (for example, script.fsx). You can launch a debugger from a command prompt.

```
devenv /debugexe fsc.exe script.fsx
```

You can use print-to-stdout logging.

See also

- Type Providers
- The Type Provider SDK

Type Provider Security

8/5/2022 • 2 minutes to read • Edit Online

Type providers are assemblies (DLLs) referenced by your F# project or script that contain code to connect to external data sources and surface this type information to the F# type environment. Typically, code in referenced assemblies is only run when you compile and then execute the code (or in the case of a script, send the code to F# Interactive). However, a type provider assembly will run inside Visual Studio when the code is merely browsed in the editor. This happens because type providers need to run to add extra information to the editor, such as Quick Info tooltips, IntelliSense completions, and so on. As a result, there are extra security considerations for type provider assemblies, since they run automatically inside the Visual Studio process.

Security Warning Dialog

When using a particular type provider assembly for the first time, Visual Studio displays a security dialog that warns you that the type provider is about to run. Before Visual Studio loads the type provider, it gives you the opportunity to decide if you trust this particular provider. If you trust the source of the type provider, then select "I trust this type provider." If you do not trust the source of the type provider, then select "I do not trust this type provider." Trusting the provider enables it to run inside Visual Studio and provide IntelliSense and build features. But if the type provider itself is malicious, running its code could compromise your machine.

If your project contains code that references type providers that you chose in the dialog not to trust, then at compile time, the compiler will report an error that indicates that the type provider is untrusted. Any types that are dependent on the untrusted type provider are indicated by red squiggles. It is safe to browse the code in the editor.

If you decide to change the trust setting directly in Visual Studio, perform the following steps.

To change the trust settings for type providers

1. On the `Tools` menu, select `Options`, and expand the `F# Tools` node.
2. Select `Type Providers`, and in the list of type providers, select the check box for type providers you trust, and clear the check box for those you don't trust.

See also

- Type Providers

Troubleshooting Type Providers

8/5/2022 • 2 minutes to read • Edit Online

This topic describes and provides potential solutions for the problems that you are most likely to encounter when you use type providers.

Possible Problems with Type Providers

If you encounter a problem when you work with type providers, you can review the following table for the most common solutions.

PROBLEM	SUGGESTED ACTIONS
Schema Changes. Type providers work best when the data source schema is stable. If you add a data table or column or make another change to that schema, the type provider doesn't automatically recognize these changes.	Clean or rebuild the project. To clean the project, choose **Build**, **Clean** *ProjectName* on the menu bar. To rebuild the project, choose **Build**, **Rebuild** *ProjectName* on the menu bar. These actions reset all type provider state and force the provider to reconnect to the data source and obtain updated schema information.
Connection Failure. The URL or connection string is incorrect, the network is down, or the data source or service is unavailable.	For a web service or OData service, you can try the URL in Internet Explorer to verify whether the URL is correct and the service is available. For a database connection string, you can use the data connection tools in **Server Explorer** to verify whether the connection string is valid and the database is available. After you restore your connection, you should then clean or rebuild the project so that the type provider will reconnect to the network.
Not Valid Credentials. You must have valid permissions for the data source or web service.	For a SQL connection, the username and the password that are specified in the connection string or configuration file must be valid for the database. If you are using Windows Authentication, you must have access to the database. The database administrator can identify what permissions you need for access to each database and each element within a database. For a web service or a data service, you must have appropriate credentials. Most type providers provide a DataContext object, which contains a Credentials property that you can set with the appropriate username and access key.
Not Valid Path. A path to a file was not valid.	Verify whether the path is correct and the file exists. In addition, you must either quote any backslashes in the path appropriately or use a verbatim string or triple-quoted string.

See also

- Type Providers

Compiler Directives

8/5/2022 • 3 minutes to read • Edit Online

This topic describes processor directives and compiler directives.

For F# Interactive (`dotnet fsi`) directives, see Interactive Programming with F#.

Preprocessor Directives

A preprocessor directive is prefixed with the # symbol and appears on a line by itself. It is interpreted by the preprocessor, which runs before the compiler itself.

The following table lists the preprocessor directives that are available in F#.

DIRECTIVE	DESCRIPTION
`#if` *symbol*	Supports conditional compilation. Code in the section after the `#if` is included if the *symbol* is defined. The symbol can also be negated with `!` .
`#else`	Supports conditional compilation. Marks a section of code to include if the symbol used with the previous `#if` is not defined.
`#endif`	Supports conditional compilation. Marks the end of a conditional section of code.
`#` [line] *int*, `#` [line] *int string*, `#` [line] *int verbatim-string*	Indicates the original source code line and file name, for debugging. This feature is provided for tools that generate F# source code.
`#nowarn` *warningcode*	Disables a compiler warning or warnings. To disable a warning, find its number from the compiler output and include it in quotation marks. Omit the "FS" prefix. To disable multiple warning numbers on the same line, include each number in quotation marks, and separate each string by a space. For example:

`#nowarn "9" "40"`

The effect of disabling a warning applies to the entire file, including portions of the file that precede the directive.|

Conditional Compilation Directives

Code that is deactivated by one of these directives appears dimmed in the Visual Studio Code Editor.

> **NOTE**
>
> The behavior of the conditional compilation directives is not the same as it is in other languages. For example, you cannot use Boolean expressions involving symbols, and `true` and `false` have no special meaning. Symbols that you use in the `if` directive must be defined by the command line or in the project settings; there is no `define` preprocessor directive.

The following code illustrates the use of the `#if`, `#else`, and `#endif` directives. In this example, the code contains two versions of the definition of `function1`. When `VERSION1` is defined by using the -define compiler option, the code between the `#if` directive and the `#else` directive is activated. Otherwise, the code between `#else` and `#endif` is activated.

```
#if VERSION1
let function1 x y =
   printfn "x: %d y: %d" x y
   x + 2 * y
#else
let function1 x y =
   printfn "x: %d y: %d" x y
   x - 2*y
#endif

let result = function1 10 20
```

There is no `#define` preprocessor directive in F#. You must use the compiler option or project settings to define the symbols used by the `#if` directive.

Conditional compilation directives can be nested. Indentation is not significant for preprocessor directives.

You can also negate a symbol with `!`. In this example, a string's value is something only when *not* debugging:

```
#if !DEBUG
let str = "Not debugging!"
#else
let str = "Debugging!"
#endif
```

Line Directives

When building, the compiler reports errors in F# code by referencing line numbers on which each error occurs. These line numbers start at 1 for the first line in a file. However, if you are generating F# source code from another tool, the line numbers in the generated code are generally not of interest, because the errors in the generated F# code most likely arise from another source. The `#line` directive provides a way for authors of tools that generate F# source code to pass information about the original line numbers and source files to the generated F# code.

When you use the `#line` directive, file names must be enclosed in quotation marks. Unless the verbatim token (`@`) appears in front of the string, you must escape backslash characters by using two backslash characters instead of one in order to use them in the path. The following are valid line tokens. In these examples, assume that the original file `Script1` results in an automatically generated F# code file when it is run through a tool, and that the code at the location of these directives is generated from some tokens at line 25 in file `Script1`.

```
# 25
#line 25
#line 25 "C:\\Projects\\MyProject\\MyProject\\Script1"
#line 25 @"C:\Projects\MyProject\MyProject\Script1"
# 25 @"C:\Projects\MyProject\MyProject\Script1"
```

These tokens indicate that the F# code generated at this location is derived from some constructs at or near line `25` in `Script1`.

See also

- F# Language Reference
- Compiler Options

Keyword Reference

8/5/2022 • 8 minutes to read • Edit Online

This topic contains links to information about all F# language keywords.

F# Keyword Table

The following table shows all F# keywords in alphabetical order, together with brief descriptions and links to relevant topics that contain more information.

KEYWORD	LINK	DESCRIPTION
`abstract`	Members Abstract Classes	Indicates a method that either has no implementation in the type in which it is declared or that is virtual and has a default implementation.
`and`	`let` Bindings Records Members Constraints	Used in mutually recursive bindings and records, in property declarations, and with multiple constraints on generic parameters.
`as`	Classes Pattern Matching	Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.
`assert`	Assertions	Used to verify code during debugging.
`base`	Classes Inheritance	Used as the name of the base class object.
`begin`	Verbose Syntax	In verbose syntax, indicates the start of a code block.
`class`	Classes	In verbose syntax, indicates the start of a class definition.
`default`	Members	Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.
`delegate`	Delegates	Used to declare a delegate.

KEYWORD	LINK	DESCRIPTION
`do`	do Bindings Loops: `for...to` Expression Loops: `for...in` Expression Loops: `while...do` Expression	Used in looping constructs or to execute imperative code.
`done`	Verbose Syntax	In verbose syntax, indicates the end of a block of code in a looping expression.
`downcast`	Casting and Conversions	Used to convert to a type that is lower in the inheritance chain.
`downto`	Loops: `for...to` Expression	In a `for` expression, used when counting in reverse.
`elif`	Conditional Expressions: `if...then...else`	Used in conditional branching. A short form of `else if`.
`else`	Conditional Expressions: `if...then...else`	Used in conditional branching.
`end`	Structs Discriminated Unions Records Type Extensions Verbose Syntax	In type definitions and type extensions, indicates the end of a section of member definitions. In verbose syntax, used to specify the end of a code block that starts with the `begin` keyword.
`exception`	Exception Handling Exception Types	Used to declare an exception type.
`extern`	External Functions	Indicates that a declared program element is defined in another binary or assembly.
`false`	Primitive Types	Used as a Boolean literal.
`finally`	Exceptions: The `try...finally` Expression	Used together with `try` to introduce a block of code that executes regardless of whether an exception occurs.
`fixed`	Fixed	Used to "pin" a pointer on the stack to prevent it from being garbage collected.
`for`	Loops: `for...to` Expression Loops: for...in Expression	Used in looping constructs.

KEYWORD	LINK	DESCRIPTION
`fun`	Lambda Expressions: The `fun` Keyword	Used in lambda expressions, also known as anonymous functions.
`function`	Match Expressions Lambda Expressions: The fun Keyword	Used as a shorter alternative to the `fun` keyword and a `match` expression in a lambda expression that has pattern matching on a single argument.
`global`	Namespaces	Used to reference the top-level .NET namespace.
`if`	Conditional Expressions: `if...then...else`	Used in conditional branching constructs.
`in`	Loops: for...in Expression Verbose Syntax	Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.
`inherit`	Inheritance	Used to specify a base class or base interface.
`inline`	Functions Inline Functions	Used to indicate a function that should be integrated directly into the caller's code.
`interface`	Interfaces	Used to declare and implement interfaces.
`internal`	Access Control	Used to specify that a member is visible inside an assembly but not outside it.
`lazy`	Lazy Expressions	Used to specify an expression that is to be performed only when a result is needed.
`let`	`let` Bindings	Used to associate, or bind, a name to a value or function.
`let!`	Async expressions Task expressions Computation Expressions	Used in async expressions to bind a name to the result of an asynchronous computation, or, in other computation expressions, used to bind a name to a result, which is of the computation type.
`match`	Match Expressions	Used to branch by comparing a value to a pattern.
`match!`	Computation Expressions	Used to inline a call to a computation expression and pattern match on its result.

KEYWORD	LINK	DESCRIPTION
`member`	Members	Used to declare a property or method in an object type.
`module`	Modules	Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.
`mutable`	let Bindings	Used to declare a variable, that is, a value that can be changed.
`namespace`	Namespaces	Used to associate a name with a group of related types and modules, to logically separate it from other code.
`new`	Constructors Constraints	Used to declare, define, or invoke a constructor that creates or that can create an object. Also used in generic parameter constraints to indicate that a type must have a certain constructor.
`not`	Symbol and Operator Reference Constraints	Not actually a keyword. However, `not struct` in combination is used as a generic parameter constraint.
`null`	Null Values Constraints	Indicates the absence of an object. Also used in generic parameter constraints.
`of`	Discriminated Unions Delegates Exception Types	Used in discriminated unions to indicate the type of categories of values, and in delegate and exception declarations.
`open`	Import Declarations: The `open` Keyword	Used to make the contents of a namespace or module available without qualification.
`or`	Symbol and Operator Reference Constraints	Used with Boolean conditions as a Boolean `or` operator. Equivalent to `||`. Also used in member constraints.
`override`	Members	Used to implement a version of an abstract or virtual method that differs from the base version.
`private`	Access Control	Restricts access to a member to code in the same type or module.
`public`	Access Control	Allows access to a member from outside the type.

KEYWORD	LINK	DESCRIPTION
`rec`	Functions	Used to indicate that a function is recursive.
`return`	[Computation Expressions Async expressions Task expressions	Used to indicate a value to provide as the result of a computation expression.
`return!`	Computation Expressions Async expressions Task expressions	Used to indicate a computation expression that, when evaluated, provides the result of the containing computation expression.
`select`	Query Expressions	Used in query expressions to specify what fields or columns to extract. Note that this is a contextual keyword, which means that it is not actually a reserved word and it only acts like a keyword in appropriate context.
`static`	Members	Used to indicate a method or property that can be called without an instance of a type, or a value member that is shared among all instances of a type.
`struct`	Structs Tuples Constraints	Used to declare a structure type. Used to specify a struct tuple. Also used in generic parameter constraints. Used for OCaml compatibility in module definitions.
`then`	Conditional Expressions: `if...then...else` Constructors	Used in conditional expressions. Also used to perform side effects after object construction.
`to`	Loops: `for...to` Expression	Used in `for` loops to indicate a range.
`true`	Primitive Types	Used as a Boolean literal.
`try`	Exceptions: The try...with Expression Exceptions: The try...finally Expression	Used to introduce a block of code that might generate an exception. Used together with `with` or `finally`.

KEYWORD	LINK	DESCRIPTION
`type`	F# Types Classes Records Structs Enumerations Discriminated Unions Type Abbreviations Units of Measure	Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.
`upcast`	Casting and Conversions	Used to convert to a type that is higher in the inheritance chain.
`use`	Resource Management: The `use` Keyword	Used instead of `let` for values that require `Dispose` to be called to free resources.
`use!`	Computation Expressions Async expressions Task expressions	Used instead of `let!` in async expressions and other computation expressions for values that require `Dispose` to be called to free resources.
`val`	Explicit Fields: The `val` Keyword Signatures Members	Used in a signature to indicate a value, or in a type to declare a member, in limited situations.
`void`	Primitive Types	Indicates the .NET `void` type. Used when interoperating with other .NET languages.
`when`	Constraints	Used for Boolean conditions (*when guards*) on pattern matches and to introduce a constraint clause for a generic type parameter.
`while`	Loops: `while...do` Expression	Introduces a looping construct.
`with`	Match Expressions Object Expressions Copy and Update Record Expressions Type Extensions Exceptions: The `try...with` Expression	Used together with the `match` keyword in pattern matching expressions. Also used in object expressions, record copying expressions, and type extensions to introduce member definitions, and to introduce exception handlers.

KEYWORD	LINK	DESCRIPTION
`yield`	Lists, Arrays, Sequences	Used in a list, array, or sequence expression to produce a value for a sequence. Typically can be omitted, as it is implicit in most situations.
`yield!`	Computation Expressions Async expressions Task expressions	Used in a computation expression to append the result of a given computation expression to a collection of results for the containing computation expression.
`const`	Type Providers	Type Providers allow the use of `const` as a keyword to specify a constant literal as a type parameter argument.

The following tokens are reserved in F# because they are keywords in the OCaml language:

- `asr`
- `land`
- `lor`
- `lsl`
- `lsr`
- `lxor`
- `mod`
- `sig`

If you use the `--mlcompatibility` compiler option, the above keywords are available for use as identifiers.

The following tokens are reserved as keywords for future expansion of F#:

- `break`
- `checked`
- `component`
- `const`
- `constraint`
- `continue`
- `event`
- `external`
- `include`
- `mixin`
- `parallel`
- `process`
- `protected`
- `pure`
- `sealed`
- `tailcall`
- `trait`
- `virtual`

The following tokens were once reserved as keywords but were released in F# 4.1, so now you can use them as identifiers:

KEYWORD	REASON
`method`	the F# community are happy with `member` to introduce methods
`constructor`	the F# community are happy with `new` to introduce constructors
`atomic`	this was related to the fad for transactional memory circa 2006. In F# this would now be a library-defined computation expression
`eager`	this is no longer needed, it was initially designed to be `let eager` to match a potential `let lazy`
`object`	there is no need to reserve this
`recursive`	F# is happy using `rec`
`functor`	If F# added parameterized modules, we would use `module M(args) = ...`
`measure`	There is no specific reason to reserve this these days, the `[<Measure>]` attribute suffices
`volatile`	There is no specific reason to reserve this these days, the `[<Volatile>]` attribute suffices

See also

- F# Language Reference
- Symbol and Operator Reference
- Compiler Options

Verbose Syntax

8/5/2022 • 2 minutes to read • Edit Online

There are two forms of syntax available for many constructs in F#: *verbose syntax* and *lightweight syntax*. The verbose syntax is not as commonly used, but has the advantage of being less sensitive to indentation. The lightweight syntax is shorter and uses indentation to signal the beginning and end of constructs, rather than additional keywords like `begin`, `end`, `in`, and so on. The default syntax is the lightweight syntax. This topic describes the syntax for F# constructs when lightweight syntax is not enabled. Verbose syntax is always enabled, so even if you enable lightweight syntax, you can still use verbose syntax for some constructs.

Table of Constructs

The following table shows the lightweight and verbose syntax for F# language constructs in contexts where there is a difference between the two forms. In this table, angle brackets (<>) enclose user-supplied syntax elements. Refer to the documentation for each language construct for more detailed information about the syntax used within these constructs.

LANGUAGE CONSTRUCT	LIGHTWEIGHT SYNTAX	VERBOSE SYNTAX
compound expressions	`<expression1>` `<expression2>`	`<expression1>; <expression2>`

nested `let` bindings	`let f x =`
	`    let a = 1`
	`    let b = 2`
	`    x + a + b`
		`let f x =`
		`    let a = 1 in`
		`    let b = 2 in`
		`    x + a + b`

code block	`(`
	`    <expression1>`
	`    <expression2>`
	`)`
		`begin`
		`    <expression1>;`
		`    <expression2>;`
		`end`

\`for...do\`	`for counter = start to finish do`
	`    ...`
		`for counter = start to finish do`
		`    ...`
		`done`

\`while...do\`	`while <condition> do`
	`    ...`
		`while <condition> do`
		`    ...`
		`done`

\`for...in\`	for var in start .. finish
	do
	    ...
		for var in start .. finish
		do
		    ...
		done

\`do\`	do
	    ...
		do
		    ...
		in

record	type <record-name> =
	    {
	        <field-declarations>
	    }
	    <value-or-member-
	definitions>
		type <record-name> =
		    {
		        <field-declarations>
		    }
		    with
		        <value-or-member-
		definitions>
		    end

class	type <class-name>(<params>)
	=
	    ...
		type <class-name>(<params>)
		=
		    class
		        ...
		    end

structure	[<StructAttribute>]
	type <structure-name> =
	    ...
		type <structure-name> =
		    struct
		        ...
		    end

discriminated union	type <union-name> =
	    | ...
	    | ...
	    ...
	    <value-or-member
	definitions>
		type <union-name> =
		    | ...
		    | ...
		    ...
		    with
		        <value-or-member-
		definitions>
		    end

interface	type <interface-name> =
	    ...
		type <interface-name> =
		    interface
		        ...
		    end

object expression	`{ new <type-name>`
	`    with`
	`        <value-or-member-`
	`definitions>`
	`        <interface-`
	`implementations>`
	`}`
		`{ new <type-name>`
		`    with`
		`        <value-or-member-`
		`definitions>`
		`    end`
		`    <interface-`
		`implementations>`
		`}`

interface implementation	`interface <interface-name>`
	`    with`
	`        <value-or-member-`
	`definitions>`
		`interface <interface-name>`
		`    with`
		`        <value-or-member-`
		`definitions>`
		`    end`

type extension	`type <type-name>`
	`    with`
	`        <value-or-member-`
	`definitions>`
		`type <type-name>`
		`    with`
		`        <value-or-member-`
		`definitions>`
		`    end`

module	`module <module-name> =`
	`    ...`
		`module <module-name> =`
		`    begin`
		`        ...`
		`    end`

See also

- F# Language Reference
- Compiler Directives
- Code Formatting Guidelines

Symbol and operator reference

8/5/2022 • 10 minutes to read • Edit Online

This article includes tables describing the symbols and operators that are used in F# and provides a brief description of each. Some symbols and operators have two or more entries when used in multiple roles.

Comment, compiler directive and attribute symbols

The following table describes symbols related to comments, compiler directives and attributes.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`(*...*)`		Delimits a comment that could span multiple lines.
`//`		Indicates the beginning of a single-line comment.
`///`	XML Documentation	Indicates an XML comment.
`#`	Compiler Directives	Prefixes a preprocessor or compiler directive.
`[<...>]`	Attributes	Delimits an attribute.

String and identifier symbols

The following table describes symbols related to strings.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`"`	Strings	Delimits a text string.
`@"`	Strings	Starts a verbatim text string, which may include backslashes and other characters.
`"""`	Strings	Delimits a triple-quoted text string, which may include backslashes, double quotation marks and other characters.
`$"`	Interpolated Strings	Starts an interpolated string.
`'`	Literals	Delimits a single-character literal.
``` ``...`` ```		Delimits an identifier that would otherwise not be a legal identifier, such as a language keyword.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`\`	Strings	Escapes the next character; used in character and string literals.

Arithmetic operators

The following table describes the arithmetic operators.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`+`	Arithmetic Operators	• When used as a binary operator, adds the left and right sides. • When used as a unary operator, indicates a positive quantity. (Formally, it produces the same value with the sign unchanged.)
`-`	Arithmetic Operators	• When used as a binary operator, subtracts the right side from the left side. • When used as a unary operator, performs a negation operation.
`*`	Arithmetic Operators Tuples Units of Measure	• When used as a binary operator, multiplies the left and right sides. • In types, indicates pairing in a tuple. • Used in units of measure types.
`/`	Arithmetic Operators Units of Measure	• Divides the left side (numerator) by the right side (denominator). • Used in units of measure types.
`%`	Arithmetic Operators	Computes the integer remainder.
`**`	Arithmetic Operators	Computes the exponentiation operation (`x ** y` means `x` to the power of `y`).

Comparison operators

The following table describes the comparison operators.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`<`	Arithmetic Operators	Computes the less-than operation.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`<>`	Arithmetic Operators	Returns `true` if the left side is not equal to the right side; otherwise, returns false.
`<=`	Arithmetic Operators	Returns `true` if the left side is less than or equal to the right side; otherwise, returns `false`.
`=`	Arithmetic Operators	Returns `true` if the left side is equal to the right side; otherwise, returns `false`.
`>`	Arithmetic Operators	Returns `true` if the left side is greater than the right side; otherwise, returns `false`.
`>=`	Arithmetic Operators	Returns `true` if the left side is greater than or equal to the right side; otherwise, returns `false`.

Boolean operators

The following table describes the arithmetic and boolean operators symbols.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`&&`	Boolean Operators	Computes the Boolean AND operation.
`||`	Boolean Operators	Computes the Boolean OR operation.

Bitwise operators

The following table describes bitwise operators.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`&&&`	Bitwise Operators	Computes the bitwise AND operation.
`<<<`	Bitwise Operators	Shifts bits in the quantity on the left side to the left by the number of bits specified on the right side.
`>>>`	Bitwise Operators	Shifts bits in the quantity on the left side to the right by the number of places specified on the right side.
`^^^`	Bitwise Operators	Computes the bitwise exclusive OR operation.
`|||`	Bitwise Operators	Computes the bitwise OR operation.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`~~~`	Bitwise Operators	Computes the bitwise NOT operation.

Function symbols and operators

The following table describes the operators and symbols related to functions.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`->`	Functions	In function expressions, separates the input pattern from the output expression.
`|>`	Functions	Passes the result of the left side to the function on the right side (forward pipe operator).
`||>`	(||>)<'T1,'T2,'U> Function	Passes the tuple of two arguments on the left side to the function on the right side.
`|||>`	(|||>)<'T1,'T2,'T3,'U> Function	Passes the tuple of three arguments on the left side to the function on the right side.
`>>`	Functions	Composes two functions (forward composition operator).
`<<`	Functions	Composes two functions in reverse order; the second one is executed first (backward composition operator).
`<|`	Functions	Passes the result of the expression on the right side to the function on left side (backward pipe operator).
`<||`	(<||)<'T1,'T2,'U> Function	Passes the tuple of two arguments on the right side to the function on left side.
`<|||`	(<|||)<'T1,'T2,'T3,'U> Function	Passes the tuple of three arguments on the right side to the function on left side.

Type symbols and operators

The following table describes symbols related to type annotation and type tests.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`->`	Functions	In function types, delimits arguments and return values, also yields a result in sequence expressions.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`:`	Functions	In a type annotation, separates a parameter or member name from its type.
`:>`	Casting and Conversions	Converts a type to type that is higher in the hierarchy.
`:?`	Match Expressions	Returns `true` if the value matches the specified type (including if it is a subtype); otherwise, returns `false` (type test operator).
`:?>`	Casting and Conversions	Converts a type to a type that is lower in the hierarchy.
`#`	Flexible Types	When used with a type, indicates a *flexible type*, which refers to a type or any one of its derived types.
`'`	Automatic Generalization	Indicates a generic type parameter.
`<...>`	Automatic Generalization	Delimits type parameters.
`^`	Statically Resolved Type Parameters Strings	• Specifies type parameters that must be resolved at compile time, not at run time. • Concatenates strings.
`{}`	Class or Record	When used with the `type` keyword, delimits a class or record. The type is a class when members are declared or the `class` keyword is used. Otherwise, it's a record.
`{||}`	Anonymous record	Denotes an anonymous record

Symbols used in member lookup and slice expressions

The following table describes additional symbols used in member lookup and slice expressions.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`.`	Members	Accesses a member, and separates individual names in a fully qualified name.
`[...]` or `.[...]`	Arrays Indexed Properties Slice Expressions	Indexes into an array, string or collection, or takes a slice of a collection.

Symbols used in tuple, list, array, unit expressions and patterns

The following table describes symbols related to tuples, lists, unit values and arrays.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`( )`	Unit Type	Represents the single value of the unit type.
`,`	Tuples	Separates the elements of a tuple, or type parameters.
`::`	Lists Match Expressions	• Creates a list. The element on the left side is prepended to the list on the right side. • Used in pattern matching to separate the parts of a list.
`@`	Lists	Concatenates two lists.
`[...]`	Lists	Delimits the elements of a list.
`[|...|]`	Arrays	Delimits the elements of an array.

Symbols used in imperative expressions

The following table describes additional symbols used in expressions.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`<-`	Values	Assigns a value to a variable.
`;`	Verbose Syntax	Separates expressions (used mostly in verbose syntax). Also separates elements of a list or fields of a record.

Additional symbols used in sequences and computation expressions

The following table describes additional symbols used in Sequences and Computation Expressions.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`->`	Sequences	Yields an expression (in sequence expressions); equivalent to the `do yield` keywords.
`!`	Computation Expressions	After a keyword, indicates a modified version of the keyword's behavior as controlled by a computation expression.

Additional symbols used in match patterns

The following table describes symbols related to pattern matching.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`->`	Match Expressions	Used in match expressions.
`&`	Match Expressions	• Computes the address of a mutable value, for use when interoperating with other languages. • Used in AND patterns.
`_`	Match Expressions Generics	• Indicates a wildcard pattern. • Specifies an anonymous generic parameter.
`|`	Match Expressions	Delimits individual match cases, individual discriminated union cases, and enumeration values.

Additional symbols used in declarations

The following table describes symbols related to declarations.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`(|...|)`	Active Patterns	Delimits an active pattern name. Also called *banana clips*.
`?`	Parameters and Arguments	Specifies an optional argument.
`~~`	Operator Overloading	Used to declare an overload for the unary negation operator.
`~-`	Operator Overloading	Used to declare an overload for the unary minus operator.
`~+`	Operator Overloading	Used to declare an overload for the unary plus operator.

Additional symbols used in quotations

The following table describes symbols related to Code Quotations.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`<@...@>`	Code Quotations	Delimits a typed code quotation.
`<@@...@@>`	Code Quotations	Delimits an untyped code quotation.
`%`	Code Quotations	Used for splicing expressions into typed code quotations.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`%%`	Code Quotations	Used for splicing expressions into untyped code quotations.

Dynamic lookup operators

The following table describes additional symbols used in dynamic lookup expressions. They are not generally used in routine F# programming and no implementations of these operator are provided in the F# core library.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`?`		Used as an operator for dynamic method and property calls.
`? ... <- ...`		Used as an operator for setting dynamic properties.

Nullable operators in queries

Nullable Operators are defined for use in Query Expressions. The following table shows these operators.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`%?`	Nullable Operators	Computes the integer remainder, when the right side is a nullable type.
`*?`	Nullable Operators	Multiplies the left and right sides, when the right side is a nullable type.
`+?`	Nullable Operators	Adds the left and right sides, when the right side is a nullable type.
`-?`	Nullable Operators	Subtracts the right side from the left side, when the right side is a nullable type.
`/?`	Nullable Operators	Divides the left side by the right side, when the right side is a nullable type.
`<?`	Nullable Operators	Computes the less than operation, when the right side is a nullable type.
`<>?`	Nullable Operators	Computes the "not equal" operation when the right side is a nullable type.
`<=?`	Nullable Operators	Computes the "less than or equal to" operation when the right side is a nullable type.
`=?`	Nullable Operators	Computes the "equal" operation when the right side is a nullable type.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`>?`	Nullable Operators	Computes the "greater than" operation when the right side is a nullable type.
`>=?`	Nullable Operators	Computes the "greater than or equal" operation when the right side is a nullable type.
`?>=`, `?>`, `?<=`, `?<`, `?=`, `?<>`, `?+`, `?-`, `?*`, `?/`	Nullable Operators	Equivalent to the corresponding operators without the ? prefix, where a nullable type is on the left.
`>=?`, `>?`, `<=?`, `<?`, `=?`, `<>?`, `+?`, `-?`, `*?`, `/?`	Nullable Operators	Equivalent to the corresponding operators without the ? suffix, where a nullable type is on the right.
`?>=?`, `?>?`, `?<=?`, `?<?`, `?=?`, `?<>?`, `?+?`, `?-?`, `?*?`, `?/?`	Nullable Operators	Equivalent to the corresponding operators without the surrounding question marks, where both sides are nullable types.

Reference cell operators (deprecated)

The following table describes symbols related to Reference Cells. The use of these operators generates advisory messages as of F# 6. For more information, see Reference cell operation advisory messages.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`!`	Reference Cells	Dereferences a reference cell.
`:=`	Reference Cells	Assigns a value to a reference cell.

Operator precedence

The following table shows the order of precedence of operators and other expression keywords in F#, in order from lowest precedence to the highest precedence. Also listed is the associativity, if applicable.

OPERATOR	ASSOCIATIVITY
`as`	Right
`when`	Right
`|` (pipe)	Left
`;`	Right
`let`	Nonassociative
`function`, `fun`, `match`, `try`	Nonassociative
`if`	Nonassociative

OPERATOR	ASSOCIATIVITY
`not`	Right
`->`	Right
`:=`	Right
`,`	Nonassociative
`or`, `||`	Left
`&`, `&&`	Left
`:>`, `:?>`	Right
`<` *op*, `>` *op*, `=`, `|` *op*, `&` *op*, `&`, `$` (including `<<<`, `>>>`, `|||`, `&&&`)	Left
`^` *op* (including `^^^`)	Right
`::`	Right
`:?`	Not associative
`-` *op*, `+` *op*	Applies to infix uses of these symbols
`*` *op*, `/` *op*, `%` *op*	Left
`**` *op*	Right
`f x` (function application) (including `lazy x`, `assert x`)	Left
`|` (pattern match)	Right
prefix operators (`+` *op*, `-` *op*, `%`, `%%`, `&`, `&&`, `!` *op*, `~` *op*)	Left
`.`	Left
`f(x)`	Left
`f<` *types* `>`	Left

F# supports custom operator overloading. This means that you can define your own operators. In the previous table, *op* can be any valid (possibly empty) sequence of operator characters, either built-in or user-defined. Thus, you can use this table to determine what sequence of characters to use for a custom operator to achieve the desired level of precedence. Leading `.` characters are ignored when the compiler determines precedence.

See also

- F# Language Reference
- Operator Overloading

Arithmetic Operators

8/5/2022 • 3 minutes to read • Edit Online

This topic describes arithmetic operators that are available in F#.

Summary of Binary Arithmetic Operators

The following table summarizes the binary arithmetic operators that are available for unboxed integral and floating-point types.

BINARY OPERATOR	NOTES
`+` (addition, plus)	Unchecked. Possible overflow condition when numbers are added together and the sum exceeds the maximum absolute value supported by the type.
`-` (subtraction, minus)	Unchecked. Possible underflow condition when unsigned types are subtracted, or when floating-point values are too small to be represented by the type.
`*` (multiplication, times)	Unchecked. Possible overflow condition when numbers are multiplied and the product exceeds the maximum absolute value supported by the type.
`/` (division, divided by)	Division by zero causes a DivideByZeroException for integral types. For floating-point types, division by zero gives you the special floating-point values `+Infinity` or `-Infinity`. There is also a possible underflow condition when a floating-point number is too small to be represented by the type.
`%` (remainder, rem)	Returns the remainder of a division operation. The sign of the result is the same as the sign of the first operand.
`**` (exponentiation, to the power of)	Possible overflow condition when the result exceeds the maximum absolute value for the type. The exponentiation operator works only with floating-point types.

Summary of Unary Arithmetic Operators

The following table summarizes the unary arithmetic operators that are available for integral and floating-point types.

UNARY OPERATOR	NOTES
`+` (positive)	Can be applied to any arithmetic expression. Does not change the sign of the value.
`-` (negation, negative)	Can be applied to any arithmetic expression. Changes the sign of the value.

The behavior at overflow or underflow for integral types is to wrap around. This causes an incorrect result. Integer overflow is a potentially serious problem that can contribute to security issues when software is not written to account for it. If this is a concern for your application, consider using the checked operators in `Microsoft.FSharp.Core.Operators.Checked`.

Summary of Binary Comparison Operators

The following table shows the binary comparison operators that are available for integral and floating-point types. These operators return values of type `bool`.

Floating-point numbers should never be directly compared for equality, because the IEEE floating-point representation does not support an exact equality operation. Two numbers that you can easily verify to be equal by inspecting the code might actually have different bit representations.

OPERATOR	NOTES
`=` (equality, equals)	This is not an assignment operator. It is used only for comparison. This is a generic operator.
`>` (greater than)	This is a generic operator.
`<` (less than)	This is a generic operator.
`>=` (greater than or equals)	This is a generic operator.
`<=` (less than or equals)	This is a generic operator.
`<>` (not equal)	This is a generic operator.

Overloaded and Generic Operators

All of the operators discussed in this topic are defined in the **Microsoft.FSharp.Core.Operators** namespace. Some of the operators are defined by using statically resolved type parameters. This means that there are individual definitions for each specific type that works with that operator. All of the unary and binary arithmetic and bitwise operators are in this category. The comparison operators are generic and therefore work with any type, not just primitive arithmetic types. Discriminated union and record types have their own custom implementations that are generated by the F# compiler. Class types use the method Equals.

The generic operators are customizable. To customize the comparison functions, override Equals to provide your own custom equality comparison, and then implement IComparable. The System.IComparable interface has a single method, the CompareTo method.

Operators and Type Inference

The use of an operator in an expression constrains type inference on that operator. Also, the use of operators prevents automatic generalization, because the use of operators implies an arithmetic type. In the absence of any other information, the F# compiler infers `int` as the type of arithmetic expressions. You can override this behavior by specifying another type. Thus the argument types and return type of `function1` in the following code are inferred to be `int`, but the types for `function2` are inferred to be `float`.

```fsharp
// x, y and return value inferred to be int
// function1: int -> int -> int
let function1 x y = x + y

// x, y and return value inferred to be float
// function2: float -> float -> float
let function2 (x: float) y = x + y
```

See also

- Symbol and Operator Reference
- Operator Overloading
- Bitwise Operators
- Boolean Operators

Boolean Operators

8/5/2022 • 2 minutes to read • Edit Online

This topic describes the support for Boolean operators in F#.

Summary of Boolean Operators

The following table summarizes the Boolean operators that are available in F#. The only type supported by these operators is the `bool` type.

OPERATOR	DESCRIPTION
`not`	Boolean negation
`||`	Boolean OR
`&&`	Boolean AND

The Boolean AND and OR operators perform *short-circuit evaluation*, that is, they evaluate the expression on the right of the operator only when it is necessary to determine the overall result of the expression. The second expression of the `&&` operator is evaluated only if the first expression evaluates to `true`; the second expression of the `||` operator is evaluated only if the first expression evaluates to `false`.

See also

- Bitwise Operators
- Arithmetic Operators
- Symbol and Operator Reference

Bitwise Operators

8/5/2022 • 2 minutes to read • Edit Online

This topic describes bitwise operators that are available in F#.

Summary of Bitwise Operators

The following table describes the bitwise operators that are supported for unboxed integral types in F#.

OPERATOR	NOTES
`&&&`	Bitwise AND operator. Bits in the result have the value 1 if and only if the corresponding bits in both source operands are 1.
`|||`	Bitwise OR operator. Bits in the result have the value 1 if either of the corresponding bits in the source operands are 1.
`^^^`	Bitwise exclusive OR operator. Bits in the result have the value 1 if and only if bits in the source operands have unequal values.
`~~~`	Bitwise negation operator. This is a unary operator and produces a result in which all 0 bits in the source operand are converted to 1 bits and all 1 bits are converted to 0 bits.
`<<<`	Bitwise left-shift operator. The result is the first operand with bits shifted left by the number of bits in the second operand. Bits shifted off the most significant position are not rotated into the least significant position. The least significant bits are padded with zeros. The type of the second argument is `int32`.
`>>>`	Bitwise right-shift operator. The result is the first operand with bits shifted right by the number of bits in the second operand. Bits shifted off the least significant position are not rotated into the most significant position. For unsigned types, the most significant bits are padded with zeros. For signed types with negative values, the most significant bits are padded with ones. The type of the second argument is `int32`.

The following types can be used with bitwise operators: `byte`, `sbyte`, `int16`, `uint16`, `int32 (int)`, `uint32`, `int64`, `uint64`, `nativeint`, and `unativeint`.

See also

- Symbol and Operator Reference
- Arithmetic Operators
- Boolean Operators

Nullable Operators in Queries

8/5/2022 • 3 minutes to read • Edit Online

Nullable operators are binary arithmetic or comparison operators that work with nullable arithmetic types on one or both sides. Nullable types arise when you work with data from sources such as databases that allow nulls in place of actual values. Nullable operators are used in query expressions. In addition to nullable operators for arithmetic and comparison, conversion operators can be used to convert between nullable types. There are also nullable versions of certain query operators.

> **NOTE**
> Nullable operators are generally only used in query expressions. If you don't use query expressions, you don't need to know or use these operators.

Table of Nullable Operators

The following table lists nullable operators supported in F#.

NULLABLE ON LEFT	NULLABLE ON RIGHT	BOTH SIDES NULLABLE
?>=	>=?	?>=?
?>	>?	?>?
?<=	<=?	?<=?
?<	<?	?<?
?=	=?	?=?
?<>	<>?	?<>?
?+	+?	?+?
?-	-?	?-?
?*	*?	?*?
?/	/?	?/?
?%	%?	?%?

Remarks

The nullable operators are included in the NullableOperators module in the namespace FSharp.Linq. The type for nullable data is `System.Nullable<'T>`.

In query expressions, nullable types arise when selecting data from a data source that allows nulls instead of values. In a SQL Server database, each data column in a table has an attribute that indicates whether nulls are allowed. If nulls are allowed, the data returned from the database can contain nulls that cannot be represented by a primitive data type such as `int`, `float`, and so on. Therefore, the data is returned as a `System.Nullable<int>` instead of `int`, and `System.Nullable<float>` instead of `float`. The actual value can be obtained from a `System.Nullable<'T>` object by using the `Value` property, and you can determine if a `System.Nullable<'T>` object has a value by calling the `HasValue` method. Another useful method is the `System.Nullable<'T>.GetValueOrDefault` method, which allows you to get the value or a default value of the appropriate type. The default value is some form of "zero" value, such as 0, 0.0, or `false`.

Nullable types may be converted to non-nullable primitive types using the usual conversion operators such as `int` or `float`. It is also possible to convert from one nullable type to another nullable type by using the conversion operators for nullable types. The appropriate conversion operators have the same name as the standard ones, but they are in a separate module, the Nullable module in the FSharp.Linq namespace. Typically, you open this namespace when working with query expressions. In that case, you can use the nullable conversion operators by adding the prefix `Nullable.` to the appropriate conversion operator, as shown in the following code.

```
open Microsoft.FSharp.Linq

let nullableInt = new System.Nullable<int>(10)

// Use the Nullable.float conversion operator to convert from one nullable type to another nullable type.
let nullableFloat = Nullable.float nullableInt

// Use the regular non-nullable float operator to convert to a non-nullable float.
printfn $"%f{float nullableFloat}"
```

The output is `10.000000`.

Query operators on nullable data fields, such as `sumByNullable`, also exist for use in query expressions. The query operators for non-nullable types are not type-compatible with nullable types, so you must use the nullable version of the appropriate query operator when you are working with nullable data values. For more information, see Query Expressions.

The following example shows the use of nullable operators in an F# query expression. The first query shows how you would write a query without a nullable operator; the second query shows an equivalent query that uses a nullable operator. For the full context, including how to set up the database to use this sample code, see Walkthrough: Accessing a SQL Database by Using Type Providers.

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

[<Generate>]
type dbSchema = SqlDataConnection<"Data Source=MYSERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = dbSchema.GetDataContext()

query {
    for row in db.Table2 do
    where (row.TestData1.HasValue && row.TestData1.Value > 2)
    select row
} |> Seq.iter (fun row -> printfn $"%d{row.TestData1.Value} %s{row.Name}")

query {
    for row in db.Table2 do
    // Use a nullable operator ?>
    where (row.TestData1 ?> 2)
    select row
} |> Seq.iter (fun row -> printfn "%d{row.TestData1.GetValueOrDefault()} %s{row.Name}")
```

See also

- Type Providers
- Query Expressions

Compiler options

8/5/2022 • 9 minutes to read • Edit Online

This article describes compiler command-line options for the F# compiler. The command `dotnet build` invokes the F# compiler on F# project files. F# project files are noted with the `.fsproj` extension.

The compilation environment can also be controlled by setting the project properties. For projects targeting .NET Core, the "Other flags" property, `<OtherFlags>...</OtherFlags>` in `.fsproj`, is used for specifying extra command-line options.

Compiler Options Listed Alphabetically

The following table shows compiler options listed alphabetically. Some of the F# compiler options are similar to the C# compiler options. If that is the case, a link to the C# compiler options topic is provided.

COMPILER OPTION	DESCRIPTION
`--allsigs`	Generates a new (or regenerates an existing) signature file for each source file in the compilation. For more information about signature files, see Signatures.
`-a filename.fs`	Generates a library from the specified file. This option is a short form of `--target:library filename.fs`.
`--baseaddress:address`	Specifies the preferred base address at which to load a DLL. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /baseaddress (C# Compiler Options).
`--codepage:id`	Specifies which code page to use during compilation if the required page isn't the current default code page for the system. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /code pages (C# Compiler Options).
`--consolecolors`	Specifies that errors and warnings use color-coded text on the console.
`--crossoptimize[+|-]`	Enables or disables cross-module optimizations.
`--delaysign[+|-]`	Delay-signs the assembly using only the public portion of the strong name key. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /delaysign (C# Compiler Options).

COMPILER OPTION	DESCRIPTION
`--checked[+|-]`	Enables or disables the generation of overflow checks. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /checked (C# Compiler Options).

`--debug[+|-]`
 	
`-g[+|-]`
 	
`--debug:[full|pdbonly]`
 	
`-g: [full|pdbonly]`
	Enables or disables the generation of debug information, or specifies the type of debug information to generate. The default is `full`, which allows attaching to a running program. Choose `pdbonly` to get limited debugging information stored in a pdb (program database) file.
	
	Equivalent to the C# compiler option of the same name. For more information, see
	
	/debug (C# Compiler Options).

`--define:symbol`
 	
`-d:symbol`
	Defines a symbol for use in conditional compilation.

`--deterministic[+|-]`	Produces a deterministic assembly (including module version GUID and timestamp). This option cannot be used with wildcard version numbers, and only supports embedded and portable debugging types
`--doc:xmldoc-filename`	Instructs the compiler to generate XML documentation comments to the file specified. For more information, see XML Documentation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /doc (C# Compiler Options).
`--fullpaths`	Instructs the compiler to generate fully qualified paths. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /fullpaths (C# Compiler Options).

`--help`
 	
`-?`
	Displays usage information, including a brief description of all the compiler options.

`--highentropyva[+|-]`	Enable or disable high-entropy address space layout randomization (ASLR), an enhanced security feature. The OS randomizes the locations in memory where infrastructure for applications (such as the stack and heap) are loaded. If you enable this option, operating systems can use this randomization to use the full 64-bit address-space on a 64-bit machine.
`--keycontainer:key-container-name`	Specifies a strong name key container.
`--keyfile:filename`	Specifies the name of a public key file for signing the generated assembly.

COMPILER OPTION	DESCRIPTION

`--lib:folder-name`
 	
`-I:folder-name`
	Specifies a directory to be searched for assemblies that are referenced.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /lib (C# Compiler Options).

`--linkresource:resource-info`	Links a specified resource to the assembly. The format of resource-info is `filename[name[public|private]]` Linking a single resource with this option is an alternative to embedding an entire resource file with the `--resource` option. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /linkresource (C# Compiler Options).
`--mlcompatibility`	Ignores warnings that appear when you use features that are designed for compatibility with other versions of ML.
`--noframework`	Disables the default reference to the .NET Framework assembly.
`--nointerfacedata`	Instructs the compiler to omit the resource it normally adds to an assembly that includes F#-specific metadata.
`--nologo`	Doesn't show the banner text when launching the compiler.
`--nooptimizationdata`	Instructs the compiler to only include optimization essential for implementing inlined constructs. Inhibits cross-module inlining but improves binary compatibility.
`--nowin32manifest`	Instructs the compiler to omit the default Win32 manifest.
`--nowarn:warning-number-list`	Disables specific warnings listed by number. Separate each warning number by a comma. You can discover the warning number for any warning from the compilation output. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /nowarn (C# Compiler Options).

`--optimize[+|-] [optimization-option-list]`
 	
`-O[+|-] [optimization-option-list]`
	Enables or disables optimizations. Some optimization options can be disabled or enabled selectively by listing them. These are: `nojitoptimize`, `nojittracking`, `nolocaloptimize`, `nocrossoptimize`, `notailcalls`.

`--out:output-filename`
 	
`-o:output-filename`
	Specifies the name of the compiled assembly or module.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /out (C# Compiler Options).

COMPILER OPTION	DESCRIPTION
`--pathmap:path=sourcePath,...`	Specifies how to map physical paths to source path names output by the compiler. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /pathmap (C# Compiler Options).
`--pdb:pdb-filename`	Names the output debug PDB (program database) file. This option only applies when `--debug` is also enabled. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /pdb (C# Compiler Options).
`--platform:platform-name`	Specifies that the generated code will only run on the specified platform (`x86` , `Itanium` , or `x64`), or, if the platform-name `anycpu` is chosen, specifies that the generated code can run on any platform. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /platform (C# Compiler Options).
`--preferreduilang:lang`	Specifies the preferred output language culture name (for example, `es-ES` , `ja-JP`).
`--quotations-debug`	Specifies that extra debugging information should be emitted for expressions that are derived from F# quotation literals and reflected definitions. The debug information is added to the custom attributes of an F# expression tree node. See Code Quotations and Expr.CustomAttributes.

`--reference:assembly-filename`
 	
`-r:assembly-filename`
	Makes code from an F# or .NET Framework assembly available to the code being compiled.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /reference (C# Compiler Options).

`--resource:resource-filename`	Embeds a managed resource file into the generated assembly. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /resource (C# Compiler Options).
`--sig:signature-filename`	Generates a signature file based on the generated assembly. For more information about signature files, see Signatures.
`--simpleresolution`	Specifies that assembly references should be resolved using directory-based Mono rules rather than MSBuild resolution. The default is to use MSBuild resolution except when running under Mono.
`--standalone`	Specifies to produce an assembly that contains all of its dependencies so that it runs by itself without the need for additional assemblies, such as the F# library.

COMPILER OPTION	DESCRIPTION
`--staticlink:assembly-name`	Statically links the given assembly and all referenced DLLs that depend on this assembly. Use the assembly name, not the DLL name.
`--subsystemversion`	Specifies the version of the OS subsystem to be used by the generated executable. Use 6.02 for Windows 8.1, 6.01 for Windows 7, 6.00 for Windows Vista. This option only applies to executables, not DLLs, and need only be used if your application depends on specific security features available only on certain versions of the OS. If this option is used, and a user attempts to execute your application on a lower version of the OS, it will fail with an error message.
`--tailcalls[+|-]`	Enables or disables the use of the tail IL instruction, which causes the stack frame to be reused for tail recursive functions. This option is enabled by default.
`--target:[exe|winexe|library|module] filename`	Specifies the type and file name of the generated compiled code. • `exe` means a console application. • `winexe` means a Windows application, which differs from the console application in that it does not have standard input/output streams (stdin, stdout, and stderr) defined. • `library` is an assembly without an entry point. • `module` is a .NET Framework module (.netmodule), which can later be combined with other modules into an assembly. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /target (C# Compiler Options).
`--times`	Displays timing information for compilation.
`--utf8output`	Enables printing compiler output in the UTF-8 encoding.
`--warn:warning-level`	Sets a warning level (0 to 5). The default level is 3. Each warning is given a level based on its severity. Level 5 gives more, but less severe, warnings than level 1. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /warn (C# Compiler Options).
`--warnon:warning-number-list`	Enable specific warnings that might be off by default or disabled by another command-line option.

COMPILER OPTION	DESCRIPTION
`--warnaserror[+|-] [warning-number-list]`	Enables or disables the option to report warnings as errors. You can provide specific warning numbers to be disabled or enabled. Options later in the command line override options earlier in the command line. For example, to specify the warnings that you don't want reported as errors, specify `--warnaserror+` `--warnaserror-:warning-number-list`. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /warnaserror (C# Compiler Options).
`--win32manifest:manifest-filename`	Adds a Win32 manifest file to the compilation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /win32manifest (C# Compiler Options).
`--win32res:resource-filename`	Adds a Win32 resource file to the compilation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /win32res ((C#) Compiler Options).

Opt-in warnings

The F# compiler supports several opt-in warnings:

NUMBER	SUMMARY	LEVEL	DESCRIPTION
21	Recursion checked at run time	5	Warn when a recursive use is checked for initialization-soundness at run time.
22	Bindings executed out of order	5	Warn when a recursive binding may be executed out-of-order because of a forward reference.
52	Implicit copies of structs	5	Warn when an immutable struct is copied to ensure the original is not mutated by an operation.
1178	Implicit equality/comparison	5	Warn when an F# type declaration is implicitly inferred to be `NoEquality` or `NoComparison` but the attribute is not present on the type.
1182	Unused variables	n/a	Warn for unused variables.
3180	Implicit heap allocations	n/a	Warn when a mutable local is implicitly allocated as a reference cell because it has been captured by a closure.

NUMBER	SUMMARY	LEVEL	DESCRIPTION
3366	Index notation	n/a	Warn when the F# 5 index notation `expr.[idx]` is used.
3517	InlineIfLambda failure	n/a	Warn when the F# optimizer fails to inline an `InlineIfLambda` value, for example if a computed function value has been provided instead of an explicit lambda.
3388	Additional implicit upcast	n/a	Warn when an additional upcast is implicitly used, added in F# 6.
3389	Implicit widening	n/a	Warn when an implicit numeric widening is used.
3390	`op_Implicit` conversion	n/a	Warn when a .NET implicit conversion is used at a method argument.

You can enable these warnings by using `/warnon:NNNN` or `<WarnOn>NNNN</WarnOn>` where `NNNN` is the relevant warning number.

Related articles

TITLE	DESCRIPTION
F# Interactive Options	Describes command-line options supported by the F# interpreter, fsi.exe.
Project Properties Reference	Describes the UI for projects, including project property pages that provide build options.

F# Interactive options

8/5/2022 • 5 minutes to read • Edit Online

This article describes the command-line options supported by F# Interactive, `fsi.exe`. F# Interactive accepts many of the same command-line options as the F# compiler, but also accepts some additional options.

Use F# Interactive for scripting

F# Interactive, `dotnet fsi`, can be launched interactively, or it can be launched from the command line to run a script. The command-line syntax is

```
dotnet fsi [options] [ script-file [arguments] ]
```

The file extension for F# script files is `.fsx`.

Table of F# Interactive Options

The following table summarizes the options supported by F# Interactive. You can set these options on the command line or through the Visual Studio IDE. To set these options in the Visual Studio IDE, open the **Tools** menu, select **Options**, expand the **F# Tools** node, and then select **F# Interactive**.

Where lists appear in F# Interactive option arguments, list elements are separated by semicolons (`;`).

OPTION	DESCRIPTION
--	Used to instruct F# Interactive to treat remaining arguments as command-line arguments to the F# program or script, which you can access in code by using the list **fsi.CommandLineArgs**.
--**checked**[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--**codepage**:<**int**>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--**consolecolors**[+|-]	Outputs warning and error messages in color.
--**crossoptimize**[+|-]	Enable or disable cross-module optimizations.
--**debug**[+|-] --**debug**:[**full**|**pdbonly**|**portable**|**embedded**] -**g**[+|-] -**g**:[**full**|**pdbonly**|**portable**|**embedded**]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--**define**:<**string**>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.

OPTION	DESCRIPTION
--deterministic[+|-]	Produces a deterministic assembly (including module version GUID and timestamp).
--exec	Instructs F# interactive to exit after loading the files or running the script file given on the command line.
--fullpaths	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--gui[+|-]	Enables or disables the Windows Forms event loop. The default is enabled.
--help **-?**	Used to display the command-line syntax and a brief description of each option.
--lib:<folder-list> **-I:<folder-list>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--load:<filename>	Compiles the given source code at startup and loads the compiled F# constructs into the session.
--mlcompatibility	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--noframework	Same as the **fsc.exe** compiler option. For more information, see Compiler Options
--nologo	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--nowarn:<warning-list>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--optimize[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--preferreduilang:<lang>	Specifies the preferred output language culture name (for example, es-ES, ja-JP).
--quiet	Suppress F# Interactive's output to the **stdout** stream.
--quotations-debug	Specifies that extra debugging information should be emitted for expressions that are derived from F# quotation literals and reflected definitions. The debug information is added to the custom attributes of an F# expression tree node. See Code Quotations and Expr.CustomAttributes.
--readline[+|-]	Enable or disable tab completion in interactive mode.
--reference:<filename> **-r:<filename>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.

OPTION	DESCRIPTION
--**tailcalls**[+|-]	Enable or disable the use of the tail IL instruction, which causes the stack frame to be reused for tail recursive functions. This option is enabled by default.
--**targetprofile:<string>**	Specifies target framework profile of this assembly. Valid values are `mscorlib`, `netcore`, or `netstandard`. The default is `mscorlib`.
--**use:<filename>**	Tells the interpreter to use the given file on startup as initial input.
--**utf8output**	Same as the fsc.exe compiler option. For more information, see Compiler Options.
--**warn:<warning-level>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--**warnaserror**[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--**warnaserror**[+|-]:**<int-list>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.

F# Interactive structured printing

F# Interactive (`dotnet fsi`) uses an extended version of structured plain text formatting to report values.

1. All features of `%A` plain text formatting are supported, and some are additionally customizable.
2. Printing is colorized if colors are supported by the output console.
3. A limit is placed on the length of strings shown, unless you explicitly evaluate that string.
4. A set of user-definable settings is available via the `fsi` object.

The available settings to customize plain text printing for reported values are:

```
open System.Globalization

fsi.FormatProvider <- CultureInfo("de-DE")  // control the default culture for primitives

fsi.PrintWidth <- 120        // Control the width used for structured printing

fsi.PrintDepth <- 10         // Control the maximum depth of nested printing

fsi.PrintLength <- 10        // Control the length of lists and arrays

fsi.PrintSize <- 100         // Control the maximum overall object count

fsi.ShowProperties <- false  // Control whether properties of .NET objects are shown by default

fsi.ShowIEnumerable <- false // Control whether sequence values are expanded by default

fsi.ShowDeclarationValues <- false // Control whether values are shown for declaration outputs
```

Customize with `AddPrinter` and `AddPrintTransformer`

Printing in F# Interactive outputs can be customized by using `fsi.AddPrinter` and `fsi.AddPrintTransformer`. The first function gives text to replace the printing of an object. The second function returns a surrogate object to display instead. For example, consider the following F# code:

```
open System

fsi.AddPrinter<DateTime>(fun dt -> dt.ToString("s"))

type DateAndLabel =
    { Date: DateTime
      Label: string  }

let newYearsDay1999 =
    { Date = DateTime(1999, 1, 1)
      Label = "New Year" }
```

If you execute the example in F# Interactive, it outputs based on the formatting option set. In this case, it affects the formatting of date and time:

```
type DateAndLabel =
  { Date: DateTime
    Label: string }
val newYearsDay1999 : DateAndLabel = { Date = 1999-01-01T00:00:00
                                       Label = "New Year" }
```

`fsi.AddPrintTransformer` can be used to give a surrogate object for printing:

```
type MyList(values: int list) =
    member _.Values = values

fsi.AddPrintTransformer(fun (x:MyList) -> box x.Values)

let x = MyList([1..10])
```

This outputs:

```
val x : MyList = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

If the transformer function passed to `fsi.AddPrintTransformer` returns `null`, then the print transformer is ignored. This can be used to filter any input value by starting with type `obj`. For example:

```
fsi.AddPrintTransformer(fun (x:obj) ->
    match x with
    | :? string as s when s = "beep" -> box ["quack"; "quack"; "quack"]
    | _ -> null)

let y = "beep"
```

This outputs:

```
val y : string = ["quack"; "quack"; "quack"]
```

Related topics

TITLE	DESCRIPTION
Compiler Options	Describes command-line options available for the F# compiler, **fsc.exe**.

F# compiler messages

8/5/2022 • 2 minutes to read • Edit Online

This section details compiler errors and warnings that the F# compiler will emit for certain constructs. The default sets of errors can be changed by:

- Treating specific warnings as if they were errors by using the `-warnaserror+` compiler option,
- Ignoring specific warnings by using the `-nowarn` compiler option

If a particular warning or error is not yet recorded in this section:

- Go to the end of this page and send feedback that includes the number or text of the error, or
- Add it yourself by following the instructions in create-new-fsharp-compiler-message.fsx and opening a pull request for this repository.

See also

- F# Compiler Options

Tour of F#

8/5/2022 • 30 minutes to read • Edit Online

The best way to learn about F# is to read and write F# code. This article will act as a tour through some of the key features of F# and give you some code snippets that you can execute on your machine. To learn about setting up a development environment, check out Getting Started.

There are two primary concepts in F#: functions and types. This tour emphasizes features of the language that fall into these two concepts.

Executing the code online

If you don't have F# installed on your machine, you can execute all of the samples in your browser with Try F# in Fable. Fable is a dialect of F# that executes directly in your browser. To view the samples that follow in the REPL, check out **Samples > Learn > Tour of F#** in the left-hand menu bar of the Fable REPL.

Functions and Modules

The most fundamental pieces of any F# program are ***functions*** organized into ***modules***. Functions perform work on inputs to produce outputs, and they are organized under Modules, which are the primary way you group things in F#. They are defined using the `let` binding, which give the function a name and define its arguments.

```
module BasicFunctions =

    /// You use 'let' to define a function. This one accepts an integer argument and returns an integer.
    /// Parentheses are optional for function arguments, except for when you use an explicit type 
annotation.
    let sampleFunction1 x = x*x + 3

    /// Apply the function, naming the function return result using 'let'.
    /// The variable type is inferred from the function return type.
    let result1 = sampleFunction1 4573

    // This line uses '%d' to print the result as an integer. This is type-safe.
    // If 'result1' were not of type 'int', then the line would fail to compile.
    printfn $"The result of squaring the integer 4573 and adding 3 is %d{result1}"

    /// When needed, annotate the type of a parameter name using '(argument:type)'.  Parentheses are 
required.
    let sampleFunction2 (x:int) = 2*x*x - x/5 + 3

    let result2 = sampleFunction2 (7 + 4)
    printfn $"The result of applying the 2nd sample function to (7 + 4) is %d{result2}"

    /// Conditionals use if/then/elif/else.
    ///
    /// Note that F# uses white space indentation-aware syntax, similar to languages like Python.
    let sampleFunction3 x =
        if x < 100.0 then
            2.0*x*x - x/5.0 + 3.0
        else
            2.0*x*x + x/5.0 - 37.0

    let result3 = sampleFunction3 (6.5 + 4.5)

    // This line uses '%f' to print the result as a float.  As with '%d' above, this is type-safe.
    printfn $"The result of applying the 3rd sample function to (6.5 + 4.5) is %f{result3}"
```

`let` bindings are also how you bind a value to a name, similar to a variable in other languages. `let` bindings are ***immutable*** by default, which means that once a value or function is bound to a name, it cannot be changed in-place. This is in contrast to variables in other languages, which are ***mutable***, meaning their values can be changed at any point in time. If you require a mutable binding, you can use `let mutable ...` syntax.

```
module Immutability =

    /// Binding a value to a name via 'let' makes it immutable.
    ///
    /// The second line of code compiles, but 'number' from that point onward will shadow the previous 
definition.
    /// There is no way to access the previous definition of 'number' due to shadowing.
    let number = 2
    // let number = 3

    /// A mutable binding.  This is required to be able to mutate the value of 'otherNumber'.
    let mutable otherNumber = 2

    printfn $"'otherNumber' is {otherNumber}"

    // When mutating a value, use '<-' to assign a new value.
    //
    // Note that '=' is not the same as this.  Outside binding values via 'let', '=' is used to test 
equality.
    otherNumber <- otherNumber + 1

    printfn $"'otherNumber' changed to be {otherNumber}"
```

Numbers, Booleans, and Strings

As a .NET language, F# supports the same underlying primitive types that exist in .NET.

Here is how various numeric types are represented in F#:

```
module IntegersAndNumbers =

    /// This is a sample integer.
    let sampleInteger = 176

    /// This is a sample floating point number.
    let sampleDouble = 4.1

    /// This computed a new number by some arithmetic.  Numeric types are converted using
    /// functions 'int', 'double' and so on.
    let sampleInteger2 = (sampleInteger/4 + 5 - 7) * 4 + int sampleDouble

    /// This is a list of the numbers from 0 to 99.
    let sampleNumbers = [ 0 .. 99 ]

    /// This is a list of all tuples containing all the numbers from 0 to 99 and their squares.
    let sampleTableOfSquares = [ for i in 0 .. 99 -> (i, i*i) ]

    // The next line prints a list that includes tuples, using an interpolated string.
    printfn $"The table of squares from 0 to 99 is:\n{sampleTableOfSquares}"
```

Here's what Boolean values and performing basic conditional logic looks like:

```
module Booleans =

    /// Booleans values are 'true' and 'false'.
    let boolean1 = true
    let boolean2 = false

    /// Operators on booleans are 'not', '&&' and '||'.
    let boolean3 = not boolean1 && (boolean2 || false)

    // This line uses '%b'to print a boolean value.  This is type-safe.
    printfn $"The expression 'not boolean1 && (boolean2 || false)' is %b{boolean3}"
```

And here's what basic string manipulation looks like:

```
module StringManipulation =

    /// Strings use double quotes.
    let string1 = "Hello"
    let string2  = "world"

    /// Strings can also use @ to create a verbatim string literal.
    /// This will ignore escape characters such as '\', '\n', '\t', etc.
    let string3 = @"C:\Program Files\"

    /// String literals can also use triple-quotes.
    let string4 = """The computer said "hello world" when I told it to!"""

    /// String concatenation is normally done with the '+' operator.
    let helloWorld = string1 + " " + string2

    // This line uses '%s' to print a string value.  This is type-safe.
    printfn "%s" helloWorld

    /// Substrings use the indexer notation.  This line extracts the first 7 characters as a substring.
    /// Note that like many languages, Strings are zero-indexed in F#.
    let substring = helloWorld[0..6]
    printfn $"{substring}"
```

Tuples

Tuples are a big deal in F#. They are a grouping of unnamed but ordered values that can be treated as values themselves. Think of them as values which are aggregated from other values. They have many uses, such as conveniently returning multiple values from a function, or grouping values for some ad-hoc convenience.

```
module Tuples =

    /// A simple tuple of integers.
    let tuple1 = (1, 2, 3)

    /// A function that swaps the order of two values in a tuple.
    ///
    /// F# Type Inference will automatically generalize the function to have a generic type,
    /// meaning that it will work with any type.
    let swapElems (a, b) = (b, a)

    printfn $"The result of swapping (1, 2) is {(swapElems (1,2))}"

    /// A tuple consisting of an integer, a string,
    /// and a double-precision floating point number.
    let tuple2 = (1, "fred", 3.1415)

    printfn $"tuple1: {tuple1}\ttuple2: {tuple2}"
```

You can also create `struct` tuples. These also interoperate fully with C#7/Visual Basic 15 tuples, which are also `struct` tuples:

```
/// Tuples are normally objects, but they can also be represented as structs.
///
/// These interoperate completely with structs in C# and Visual Basic.NET; however,
/// struct tuples are not implicitly convertible with object tuples (often called reference tuples).
///
/// The second line below will fail to compile because of this.  Uncomment it to see what happens.
let sampleStructTuple = struct (1, 2)
//let thisWillNotCompile: (int*int) = struct (1, 2)

// Although you can
let convertFromStructTuple (struct(a, b)) = (a, b)
let convertToStructTuple (a, b) = struct(a, b)

printfn $"Struct Tuple: {sampleStructTuple}\nReference tuple made from the Struct Tuple: {(sampleStructTuple
|> convertFromStructTuple)}"
```

It's important to note that because `struct` tuples are value types, they cannot be implicitly converted to reference tuples, or vice versa. You must explicitly convert between a reference and struct tuple.

Pipelines

The pipe operator `|>` is used extensively when processing data in F#. This operator allows you to establish "pipelines" of functions in a flexible manner. The following example walks through how you can take advantage of these operators to build a simple functional pipeline:

```
module PipelinesAndComposition =

    /// Squares a value.
    let square x = x * x

    /// Adds 1 to a value.
    let addOne x = x + 1

    /// Tests if an integer value is odd via modulo.
    ///
    /// '<>' is a binary comparison operator that means "not equal to".
    let isOdd x = x % 2 <> 0

    /// A list of 5 numbers.  More on lists later.
    let numbers = [ 1; 2; 3; 4; 5 ]

    /// Given a list of integers, it filters out the even numbers,
    /// squares the resulting odds, and adds 1 to the squared odds.
    let squareOddValuesAndAddOne values =
        let odds = List.filter isOdd values
        let squares = List.map square odds
        let result = List.map addOne squares
        result

    printfn $"processing {numbers} through 'squareOddValuesAndAddOne' produces: {squareOddValuesAndAddOne
numbers}"

    /// A shorter way to write 'squareOddValuesAndAddOne' is to nest each
    /// sub-result into the function calls themselves.
    ///
    /// This makes the function much shorter, but it's difficult to see the
    /// order in which the data is processed.
    let squareOddValuesAndAddOneNested values =
        List.map addOne (List.map square (List.filter isOdd values))

    printfn $"processing {numbers} through 'squareOddValuesAndAddOneNested' produces:
{squareOddValuesAndAddOneNested numbers}"

    /// A preferred way to write 'squareOddValuesAndAddOne' is to use F# pipe operators.
```

```
    /// This allows you to avoid creating intermediate results, but is much more readable
    /// than nesting function calls like 'squareOddValuesAndAddOneNested'
    let squareOddValuesAndAddOnePipeline values =
        values
        |> List.filter isOdd
        |> List.map square
        |> List.map addOne

    printfn $"processing {numbers} through 'squareOddValuesAndAddOnePipeline' produces:
{squareOddValuesAndAddOnePipeline numbers}"

    /// You can shorten 'squareOddValuesAndAddOnePipeline' by moving the second `List.map` call
    /// into the first, using a Lambda Function.
    ///
    /// Note that pipelines are also being used inside the lambda function.  F# pipe operators
    /// can be used for single values as well.  This makes them very powerful for processing data.
    let squareOddValuesAndAddOneShorterPipeline values =
        values
        |> List.filter isOdd
        |> List.map(fun x -> x |> square |> addOne)

    printfn $"processing {numbers} through 'squareOddValuesAndAddOneShorterPipeline' produces:
{squareOddValuesAndAddOneShorterPipeline numbers}"

    /// Lastly, you can eliminate the need to explicitly take 'values' in as a parameter by using '>>'
    /// to compose the two core operations: filtering out even numbers, then squaring and adding one.
    /// Likewise, the 'fun x -> ...' bit of the lambda expression is also not needed, because 'x' is simply
    /// being defined in that scope so that it can be passed to a functional pipeline.  Thus, '>>' can be
used
    /// there as well.
    ///
    /// The result of 'squareOddValuesAndAddOneComposition' is itself another function which takes a
    /// list of integers as its input.  If you execute 'squareOddValuesAndAddOneComposition' with a list
    /// of integers, you'll notice that it produces the same results as previous functions.
    ///
    /// This is using what is known as function composition.  This is possible because functions in F#
    /// use Partial Application and the input and output types of each data processing operation match
    /// the signatures of the functions we're using.
    let squareOddValuesAndAddOneComposition =
        List.filter isOdd >> List.map (square >> addOne)

    printfn $"processing {numbers} through 'squareOddValuesAndAddOneComposition' produces:
{squareOddValuesAndAddOneComposition numbers}"
```

The previous sample made use of many features of F#, including list processing functions, first-class functions, and partial application. Although these are advanced concepts, it should be clear how easily functions can be used to process data when building pipelines.

Lists, Arrays, and Sequences

Lists, Arrays, and Sequences are three primary collection types in the F# core library.

Lists are ordered, immutable collections of elements of the same type. They are singly linked lists, which means they are meant for enumeration, but a poor choice for random access and concatenation if they're large. This is in contrast to Lists in other popular languages, which typically do not use a singly linked list to represent Lists.

```fsharp
module Lists =

    /// Lists are defined using [ ... ].  This is an empty list.
    let list1 = [ ]

    /// This is a list with 3 elements.  ';' is used to separate elements on the same line.
    let list2 = [ 1; 2; 3 ]

    /// You can also separate elements by placing them on their own lines.
    let list3 = [
        1
        2
        3
    ]

    /// This is a list of integers from 1 to 1000
    let numberList = [ 1 .. 1000 ]

    /// Lists can also be generated by computations. This is a list containing
    /// all the days of the year.
    ///
    /// 'yield' is used for on-demand evaluation. More on this later in Sequences.
    let daysList =
        [ for month in 1 .. 12 do
              for day in 1 .. System.DateTime.DaysInMonth(2017, month) do
                  yield System.DateTime(2017, month, day) ]

    // Print the first 5 elements of 'daysList' using 'List.take'.
    printfn $"The first 5 days of 2017 are: {daysList |> List.take 5}"

    /// Computations can include conditionals.  This is a list containing the tuples
    /// which are the coordinates of the black squares on a chess board.
    let blackSquares =
        [ for i in 0 .. 7 do
              for j in 0 .. 7 do
                  if (i+j) % 2 = 1 then
                      yield (i, j) ]

    /// Lists can be transformed using 'List.map' and other functional programming combinators.
    /// This definition produces a new list by squaring the numbers in numberList, using the pipeline
    /// operator to pass an argument to List.map.
    let squares =
        numberList
        |> List.map (fun x -> x*x)

    /// There are many other list combinations. The following computes the sum of the squares of the
    /// numbers divisible by 3.
    let sumOfSquares =
        numberList
        |> List.filter (fun x -> x % 3 = 0)
        |> List.sumBy (fun x -> x * x)

    printfn $"The sum of the squares of numbers up to 1000 that are divisible by 3 is: %d{sumOfSquares}"
```

Arrays are fixed-size, *mutable* collections of elements of the same type. They support fast random access of elements, and are faster than F# lists because they are just contiguous blocks of memory.

```
module Arrays =

    /// This is The empty array.  Note that the syntax is similar to that of Lists, but uses `[| ... |]`
instead.
    let array1 = [| |]

    /// Arrays are specified using the same range of constructs as lists.
    let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

    /// This is an array of numbers from 1 to 1000.
    let array3 = [| 1 .. 1000 |]

    /// This is an array containing only the words "hello" and "world".
    let array4 =
        [| for word in array2 do
               if word.Contains("l") then
                   yield word |]

    /// This is an array initialized by index and containing the even numbers from 0 to 2000.
    let evenNumbers = Array.init 1001 (fun n -> n * 2)

    /// Sub-arrays are extracted using slicing notation.
    let evenNumbersSlice = evenNumbers[0..500]

    /// You can loop over arrays and lists using 'for' loops.
    for word in array4 do
        printfn $"word: {word}"

    // You can modify the contents of an array element by using the left arrow assignment operator.
    //
    // To learn more about this operator, see: https://docs.microsoft.com/dotnet/fsharp/language-
reference/values/index#mutable-variables
    array2[1] <- "WORLD!"

    /// You can transform arrays using 'Array.map' and other functional programming operations.
    /// The following calculates the sum of the lengths of the words that start with 'h'.
    ///
    /// Note that in this case, similar to Lists, array2 is not mutated by Array.filter.
    let sumOfLengthsOfWords =
        array2
        |> Array.filter (fun x -> x.StartsWith "h")
        |> Array.sumBy (fun x -> x.Length)

    printfn $"The sum of the lengths of the words in Array 2 is: %d{sumOfLengthsOfWords}"
```

Sequences are a logical series of elements, all of the same type. These are a more general type than Lists and Arrays, capable of being your "view" into any logical series of elements. They also stand out because they can be ***lazy***, which means that elements can be computed only when they are needed.

```
module Sequences =

    /// This is the empty sequence.
    let seq1 = Seq.empty

    /// This a sequence of values.
    let seq2 = seq { yield "hello"; yield "world"; yield "and"; yield "hello"; yield "world"; yield "again"
}

    /// This is an on-demand sequence from 1 to 1000.
    let numbersSeq = seq { 1 .. 1000 }

    /// This is a sequence producing the words "hello" and "world"
    let seq3 =
        seq { for word in seq2 do
                  if word.Contains("l") then
                      yield word }

    /// This is a sequence producing the even numbers up to 2000.
    let evenNumbers = Seq.init 1001 (fun n -> n * 2)

    let rnd = System.Random()

    /// This is an infinite sequence which is a random walk.
    /// This example uses yield! to return each element of a subsequence.
    let rec randomWalk x =
        seq { yield x
              yield! randomWalk (x + rnd.NextDouble() - 0.5) }

    /// This example shows the first 100 elements of the random walk.
    let first100ValuesOfRandomWalk =
        randomWalk 5.0
        |> Seq.truncate 100
        |> Seq.toList

    printfn $"First 100 elements of a random walk: {first100ValuesOfRandomWalk}"
```

Recursive Functions

Processing collections or sequences of elements is typically done with recursion in F#. Although F# has support for loops and imperative programming, recursion is preferred because it is easier to guarantee correctness.

> **NOTE**
>
> The following example makes use of the pattern matching via the `match` expression. This fundamental construct is covered later in this article.

```
module RecursiveFunctions =

    /// This example shows a recursive function that computes the factorial of an
    /// integer. It uses 'let rec' to define a recursive function.
    let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

    printfn $"Factorial of 6 is: %d{factorial 6}"

    /// Computes the greatest common factor of two integers.
    ///
    /// Since all of the recursive calls are tail calls,
    /// the compiler will turn the function into a loop,
    /// which improves performance and reduces memory consumption.
    let rec greatestCommonFactor a b =
        if a = 0 then b
        elif a < b then greatestCommonFactor a (b - a)
        else greatestCommonFactor (a - b) b

    printfn $"The Greatest Common Factor of 300 and 620 is %d{greatestCommonFactor 300 620}"

    /// This example computes the sum of a list of integers using recursion.
    ///
    /// '::' is used to split a list into the head and tail of the list,
    /// the head being the first element and the tail being the rest of the list.
    let rec sumList xs =
        match xs with
        | []    -> 0
        | y::ys -> y + sumList ys

    /// This makes 'sumList' tail recursive, using a helper function with a result accumulator.
    let rec private sumListTailRecHelper accumulator xs =
        match xs with
        | []    -> accumulator
        | y::ys -> sumListTailRecHelper (accumulator+y) ys

    /// This invokes the tail recursive helper function, providing '0' as a seed accumulator.
    /// An approach like this is common in F#.
    let sumListTailRecursive xs = sumListTailRecHelper 0 xs

    let oneThroughTen = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

    printfn $"The sum 1-10 is %d{sumListTailRecursive oneThroughTen}"
```

F# also has full support for Tail Call Optimization, which is a way to optimize recursive calls so that they are just as fast as a loop construct.

Record and Discriminated Union Types

Record and Union types are two fundamental data types used in F# code, and are generally the best way to represent data in an F# program. Although this makes them similar to classes in other languages, one of their primary differences is that they have structural equality semantics. This means that they are "natively" comparable and equality is straightforward - just check if one is equal to the other.

Records are an aggregate of named values, with optional members (such as methods). If you're familiar with C# or Java, then these should feel similar to POCOs or POJOs - just with structural equality and less ceremony.

```fsharp
module RecordTypes =

    /// This example shows how to define a new record type.
    type ContactCard =
        { Name     : string
          Phone    : string
          Verified : bool }

    /// This example shows how to instantiate a record type.
    let contact1 =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false }

    /// You can also do this on the same line with ';' separators.
    let contactOnSameLine = { Name = "Alf"; Phone = "(206) 555-0157"; Verified = false }

    /// This example shows how to use "copy-and-update" on record values. It creates
    /// a new record value that is a copy of contact1, but has different values for
    /// the 'Phone' and 'Verified' fields.
    ///
    /// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/copy-and-update-
record-expressions
    let contact2 =
        { contact1 with
            Phone = "(206) 555-0112"
            Verified = true }

    /// This example shows how to write a function that processes a record value.
    /// It converts a 'ContactCard' object to a string.
    let showContactCard (c: ContactCard) =
        c.Name + " Phone: " + c.Phone + (if not c.Verified then " (unverified)" else "")

    printfn $"Alf's Contact Card: {showContactCard contact1}"

    /// This is an example of a Record with a member.
    type ContactCardAlternate =
        { Name     : string
          Phone    : string
          Address  : string
          Verified : bool }

        /// Members can implement object-oriented members.
        member this.PrintedContactCard =
            this.Name + " Phone: " + this.Phone + (if not this.Verified then " (unverified)" else "") +
this.Address

    let contactAlternate =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false
          Address = "111 Alf Street" }

    // Members are accessed via the '.' operator on an instantiated type.
    printfn $"Alf's alternate contact card is {contactAlternate.PrintedContactCard}"
```

You can also represent Records as structs. This is done with the `[<Struct>]` attribute:

```fsharp
[<Struct>]
type ContactCardStruct =
    { Name     : string
      Phone    : string
      Verified : bool }
```

Discriminated Unions (DUs) are values that could be a number of named forms or cases. Data stored in the type can be one of several distinct values.

```
module DiscriminatedUnions =

    /// The following represents the suit of a playing card.
    type Suit =
        | Hearts
        | Clubs
        | Diamonds
        | Spades

    /// A Discriminated Union can also be used to represent the rank of a playing card.
    type Rank =
        /// Represents the rank of cards 2 .. 10
        | Value of int
        | Ace
        | King
        | Queen
        | Jack

        /// Discriminated Unions can also implement object-oriented members.
        static member GetAllRanks() =
            [ yield Ace
              for i in 2 .. 10 do yield Value i
              yield Jack
              yield Queen
              yield King ]

    /// This is a record type that combines a Suit and a Rank.
    /// It's common to use both Records and Discriminated Unions when representing data.
    type Card = { Suit: Suit; Rank: Rank }

    /// This computes a list representing all the cards in the deck.
    let fullDeck =
        [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
              for rank in Rank.GetAllRanks() do
                  yield { Suit=suit; Rank=rank } ]

    /// This example converts a 'Card' object to a string.
    let showPlayingCard (c: Card) =
        let rankString =
            match c.Rank with
            | Ace -> "Ace"
            | King -> "King"
            | Queen -> "Queen"
            | Jack -> "Jack"
            | Value n -> string n
        let suitString =
            match c.Suit with
            | Clubs -> "clubs"
            | Diamonds -> "diamonds"
            | Spades -> "spades"
            | Hearts -> "hearts"
        rankString  + " of " + suitString

    /// This example prints all the cards in a playing deck.
    let printAllCards() =
        for card in fullDeck do
            printfn $"{showPlayingCard card}"
```

You can also use DUs as *Single-Case Discriminated Unions*, to help with domain modeling over primitive types. Often, strings and other primitive types are used to represent something, and are thus given a particular meaning. However, using only the primitive representation of the data can result in mistakenly assigning an incorrect value! Representing each type of information as a distinct single-case union can enforce correctness in this scenario.

```
// Single-case DUs are often used for domain modeling.  This can buy you extra type safety
// over primitive types such as strings and ints.
//
// Single-case DUs cannot be implicitly converted to or from the type they wrap.
// For example, a function which takes in an Address cannot accept a string as that input,
// or vice versa.
type Address = Address of string
type Name = Name of string
type SSN = SSN of int

// You can easily instantiate a single-case DU as follows.
let address = Address "111 Alf Way"
let name = Name "Alf"
let ssn = SSN 1234567890

/// When you need the value, you can unwrap the underlying value with a simple function.
let unwrapAddress (Address a) = a
let unwrapName (Name n) = n
let unwrapSSN (SSN s) = s

// Printing single-case DUs is simple with unwrapping functions.
printfn $"Address: {address |> unwrapAddress}, Name: {name |> unwrapName}, and SSN: {ssn |> unwrapSSN}"
```

As the above sample demonstrates, to get the underlying value in a single-case Discriminated Union, you must explicitly unwrap it.

Additionally, DUs also support recursive definitions, allowing you to easily represent trees and inherently recursive data. For example, here's how you can represent a Binary Search Tree with `exists` and `insert` functions.

```
/// Discriminated Unions also support recursive definitions.
///
/// This represents a Binary Search Tree, with one case being the Empty tree,
/// and the other being a Node with a value and two subtrees.
///
/// Note 'T here is a type parameter, indicating that 'BST' is a generic type.
/// More on generics later.
type BST<'T> =
    | Empty
    | Node of value:'T * left: BST<'T> * right: BST<'T>

/// Check if an item exists in the binary search tree.
/// Searches recursively using Pattern Matching.  Returns true if it exists; otherwise, false.
let rec exists item bst =
    match bst with
    | Empty -> false
    | Node (x, left, right) ->
        if item = x then true
        elif item < x then (exists item left) // Check the left subtree.
        else (exists item right) // Check the right subtree.

/// Inserts an item in the Binary Search Tree.
/// Finds the place to insert recursively using Pattern Matching, then inserts a new node.
/// If the item is already present, it does not insert anything.
let rec insert item bst =
    match bst with
    | Empty -> Node(item, Empty, Empty)
    | Node(x, left, right) as node ->
        if item = x then node // No need to insert, it already exists; return the node.
        elif item < x then Node(x, insert item left, right) // Call into left subtree.
        else Node(x, left, insert item right) // Call into right subtree.
```

Because DUs allow you to represent the recursive structure of the tree in the data type, operating on this recursive structure is straightforward and guarantees correctness. It is also supported in pattern matching, as shown below.

Pattern Matching

Pattern Matching is the F# feature that enables correctness for operating on F# types. In the above samples, you probably noticed quite a bit of `match x with ...` syntax. This construct allows the compiler, which can understand the "shape" of data types, to force you to account for all possible cases when using a data type through what is known as Exhaustive Pattern Matching. This is incredibly powerful for correctness, and can be cleverly used to "lift" what would normally be a run-time concern into a compile-time concern.

```
module PatternMatching =

    /// A record for a person's first and last name
    type Person = {
        First : string
        Last  : string
    }

    /// A Discriminated Union of 3 different kinds of employees
    type Employee =
        | Engineer of engineer: Person
        | Manager of manager: Person * reports: List<Employee>
        | Executive of executive: Person * reports: List<Employee> * assistant: Employee

    /// Count everyone underneath the employee in the management hierarchy,
    /// including the employee. The matches bind names to the properties
    /// of the cases so that those names can be used inside the match branches.
    /// Note that the names used for binding do not need to be the same as the
    /// names given in the DU definition above.
    let rec countReports(emp : Employee) =
        1 + match emp with
            | Engineer(person) ->
                0
            | Manager(person, reports) ->
                reports |> List.sumBy countReports
            | Executive(person, reports, assistant) ->
                (reports |> List.sumBy countReports) + countReports assistant
```

Something you may have noticed is the use of the `_` pattern. This is known as the Wildcard Pattern, which is a way of saying "I don't care what something is". Although convenient, you can accidentally bypass Exhaustive Pattern Matching and no longer benefit from compile-time enforcements if you aren't careful in using `_`. It is best used when you don't care about certain pieces of a decomposed type when pattern matching, or the final clause when you have enumerated all meaningful cases in a pattern matching expression.

In the following example, the `_` case is used when a parse operation fails.

```
/// Find all managers/executives named "Dave" who do not have any reports.
/// This uses the 'function' shorthand to as a lambda expression.
let findDaveWithOpenPosition(emps : List<Employee>) =
    emps
    |> List.filter(function
                  | Manager({First = "Dave"}, []) -> true // [] matches an empty list.
                  | Executive({First = "Dave"}, [], _) -> true
                  | _ -> false) // '_' is a wildcard pattern that matches anything.
                                // This handles the "or else" case.

/// You can also use the shorthand function construct for pattern matching,
/// which is useful when you're writing functions which make use of Partial Application.
let private parseHelper (f: string -> bool * 'T) = f >> function
    | (true, item) -> Some item
    | (false, _) -> None

let parseDateTimeOffset = parseHelper DateTimeOffset.TryParse

let result = parseDateTimeOffset "1970-01-01"
match result with
| Some dto -> printfn "It parsed!"
| None -> printfn "It didn't parse!"

// Define some more functions which parse with the helper function.
let parseInt = parseHelper Int32.TryParse
let parseDouble = parseHelper Double.TryParse
let parseTimeSpan = parseHelper TimeSpan.TryParse
```

Active Patterns are another powerful construct to use with pattern matching. They allow you to partition input data into custom forms, decomposing them at the pattern match call site. They can also be parameterized, thus allowing to define the partition as a function. Expanding the previous example to support Active Patterns looks something like this:

```
let (|Int|_|) = parseInt
let (|Double|_|) = parseDouble
let (|Date|_|) = parseDateTimeOffset
let (|TimeSpan|_|) = parseTimeSpan

/// Pattern Matching via 'function' keyword and Active Patterns often looks like this.
let printParseResult = function
    | Int x -> printfn $"%d{x}"
    | Double x -> printfn $"%f{x}"
    | Date d -> printfn $"%O{d}"
    | TimeSpan t -> printfn $"%O{t}"
    | _ -> printfn "Nothing was parse-able!"

// Call the printer with some different values to parse.
printParseResult "12"
printParseResult "12.045"
printParseResult "12/28/2016"
printParseResult "9:01PM"
printParseResult "banana!"
```

Options

One special case of Discriminated Union types is the Option Type, which is so useful that it's a part of the F# core library.

The Option Type is a type that represents one of two cases: a value, or nothing at all. It is used in any scenario where a value may or may not result from a particular operation. This then forces you to account for both cases, making it a compile-time concern rather than a runtime concern. These are often used in APIs where `null` is used to represent "nothing" instead, thus eliminating the need to worry about `NullReferenceException` in many circumstances.

```
module OptionValues =

    /// First, define a zip code defined via Single-case Discriminated Union.
    type ZipCode = ZipCode of string

    /// Next, define a type where the ZipCode is optional.
    type Customer = { ZipCode: ZipCode option }

    /// Next, define an interface type that represents an object to compute the shipping zone for the
customer's zip code,
    /// given implementations for the 'getState' and 'getShippingZone' abstract methods.
    type IShippingCalculator =
        abstract GetState : ZipCode -> string option
        abstract GetShippingZone : string -> int

    /// Next, calculate a shipping zone for a customer using a calculator instance.
    /// This uses combinators in the Option module to allow a functional pipeline for
    /// transforming data with Optionals.
    let CustomerShippingZone (calculator: IShippingCalculator, customer: Customer) =
        customer.ZipCode
        |> Option.bind calculator.GetState
        |> Option.map calculator.GetShippingZone
```

Units of Measure

F#'s type system includes the ability to provide context for numeric literals through Units of Measure. Units of measure allow you to associate a numeric type to a unit, such as Meters, and have functions perform work on units rather than numeric literals. This enables the compiler to verify that the types of numeric literals passed in make sense under a certain context, thus eliminating run-time errors associated with that kind of work.

```
module UnitsOfMeasure =

    /// First, open a collection of common unit names
    open Microsoft.FSharp.Data.UnitSystems.SI.UnitNames

    /// Define a unitized constant
    let sampleValue1 = 1600.0<meter>

    /// Next, define a new unit type
    [<Measure>]
    type mile =
        /// Conversion factor mile to meter.
        static member asMeter = 1609.34<meter/mile>

    /// Define a unitized constant
    let sampleValue2  = 500.0<mile>

    /// Compute  metric-system constant
    let sampleValue3 = sampleValue2 * mile.asMeter

    // Values using Units of Measure can be used just like the primitive numeric type for things like
printing.
    printfn $"After a %f{sampleValue1} race I would walk %f{sampleValue2} miles which would be
%f{sampleValue3} meters"
```

The F# Core library defines many SI unit types and unit conversions. To learn more, check out the FSharp.Data.UnitSystems.SI.UnitSymbols Namespace.

Object Programming

F# has full support for object programming through classes, Interfaces, Abstract Classes, Inheritance, and so on.

Classes are types that represent .NET objects, which can have properties, methods, and events as its Members.

```
module DefiningClasses =

    /// A simple two-dimensional Vector class.
    ///
    /// The class's constructor is on the first line,
    /// and takes two arguments: dx and dy, both of type 'double'.
    type Vector2D(dx : double, dy : double) =

        /// This internal field stores the length of the vector, computed when the
        /// object is constructed
        let length = sqrt (dx*dx + dy*dy)

        // 'this' specifies a name for the object's self-identifier.
        // In instance methods, it must appear before the member name.
        member this.DX = dx

        member this.DY = dy

        member this.Length = length

        /// This member is a method.  The previous members were properties.
        member this.Scale(k) = Vector2D(k * this.DX, k * this.DY)

    /// This is how you instantiate the Vector2D class.
    let vector1 = Vector2D(3.0, 4.0)

    /// Get a new scaled vector object, without modifying the original object.
    let vector2 = vector1.Scale(10.0)

    printfn $"Length of vector1: %f{vector1.Length}\nLength of vector2: %f{vector2.Length}"
```

Defining generic classes is also straightforward.

```
module DefiningGenericClasses =

    type StateTracker<'T>(initialElement: 'T) =

        /// This internal field store the states in a list.
        let mutable states = [ initialElement ]

        /// Add a new element to the list of states.
        member this.UpdateState newState =
            states <- newState :: states  // use the '<-' operator to mutate the value.

        /// Get the entire list of historical states.
        member this.History = states

        /// Get the latest state.
        member this.Current = states.Head

    /// An 'int' instance of the state tracker class. Note that the type parameter is inferred.
    let tracker = StateTracker 10

    // Add a state
    tracker.UpdateState 17
```

To implement an Interface, you can use either `interface ... with` syntax or an Object Expression.

```
module ImplementingInterfaces =

    /// This is a type that implements IDisposable.
    type ReadFile() =

        let file = new System.IO.StreamReader("readme.txt")

        member this.ReadLine() = file.ReadLine()

        // This is the implementation of IDisposable members.
        interface System.IDisposable with
            member this.Dispose() = file.Close()


    /// This is an object that implements IDisposable via an Object Expression
    /// Unlike other languages such as C# or Java, a new type definition is not needed
    /// to implement an interface.
    let interfaceImplementation =
        { new System.IDisposable with
            member this.Dispose() = printfn "disposed" }
```

Which Types to Use

The presence of Classes, Records, Discriminated Unions, and Tuples leads to an important question: which should you use? Like most everything in life, the answer depends on your circumstances.

Tuples are great for returning multiple values from a function, and using an ad-hoc aggregate of values as a value itself.

Records are a "step up" from Tuples, having named labels and support for optional members. They are great for a low-ceremony representation of data in-transit through your program. Because they have structural equality, they are easy to use with comparison.

Discriminated Unions have many uses, but the core benefit is to be able to utilize them in conjunction with Pattern Matching to account for all possible "shapes" that a data can have.

Classes are great for a huge number of reasons, such as when you need to represent information and also tie that information to functionality. As a rule of thumb, when you have functionality that is conceptually tied to some data, using Classes and the principles of Object-Oriented Programming is a significant benefit. Classes are also the preferred data type when interoperating with C# and Visual Basic, as these languages use classes for nearly everything.

Next Steps

Now that you've seen some of the primary features of the language, you should be ready to write your first F# programs! Check out Getting Started to learn how to set up your development environment and write some code.

Also, check out the F# Language Reference to see a comprehensive collection of conceptual content on F#.

Introduction to Functional Programming Concepts in F#

8/5/2022 • 8 minutes to read • Edit Online

Functional programming is a style of programming that emphasizes the use of functions and immutable data. Typed functional programming is when functional programming is combined with static types, such as with F#. In general, the following concepts are emphasized in functional programming:

- Functions as the primary constructs you use
- Expressions instead of statements
- Immutable values over variables
- Declarative programming over imperative programming

Throughout this series, you'll explore concepts and patterns in functional programming using F#. Along the way, you'll learn some F# too.

Terminology

Functional programming, like other programming paradigms, comes with a vocabulary that you will eventually need to learn. Here are some common terms you'll see all of the time:

- **Function** - A function is a construct that will produce an output when given an input. More formally, it *maps* an item from one set to another set. This formalism is lifted into the concrete in many ways, especially when using functions that operate on collections of data. It is the most basic (and important) concept in functional programming.
- **Expression** - An expression is a construct in code that produces a value. In F#, this value must be bound or explicitly ignored. An expression can be trivially replaced by a function call.
- **Purity** - Purity is a property of a function such that its return value is always the same for the same arguments, and that its evaluation has no side effects. A pure function depends entirely on its arguments.
- **Referential Transparency** - Referential Transparency is a property of expressions such that they can be replaced with their output without affecting a program's behavior.
- **Immutability** - Immutability means that a value cannot be changed in-place. This is in contrast with variables, which can change in place.

Examples

The following examples demonstrate these core concepts.

Functions

The most common and fundamental construct in functional programming is the function. Here's a simple function that adds 1 to an integer:

```
let addOne x = x + 1
```

Its type signature is as follows:

```
val addOne: x:int -> int
```

The signature can be read as, "`addOne` accepts an `int` named `x` and will produce an `int`". More formally, `addOne` is *mapping* a value from the set of integers to the set of integers. The `->` token signifies this mapping. In F#, you can usually look at the function signature to get a sense for what it does.

So, why is the signature important? In typed functional programming, the implementation of a function is often less important than the actual type signature! The fact that `addOne` adds the value 1 to an integer is interesting at run time, but when you are constructing a program, the fact that it accepts and returns an `int` is what informs how you will actually use this function. Furthermore, once you use this function correctly (with respect to its type signature), diagnosing any problems can be done only within the body of the `addOne` function. This is the impetus behind typed functional programming.

Expressions

Expressions are constructs that evaluate to a value. In contrast to statements, which perform an action, expressions can be thought of performing an action that gives back a value. Expressions are almost always used in functional programming instead of statements.

Consider the previous function, `addOne`. The body of `addOne` is an expression:

```
// 'x + 1' is an expression!
let addOne x = x + 1
```

It is the result of this expression that defines the result type of the `addOne` function. For example, the expression that makes up this function could be changed to be a different type, such as a `string`:

```
let addOne x = x.ToString() + "1"
```

The signature of the function is now:

```
val addOne: x:'a -> string
```

Since any type in F# can have `ToString()` called on it, the type of `x` has been made generic (called Automatic Generalization), and the resultant type is a `string`.

Expressions are not just the bodies of functions. You can have expressions that produce a value you use elsewhere. A common one is `if`:

```
// Checks if 'x' is odd by using the mod operator
let isOdd x = x % 2 <> 0

let addOneIfOdd input =
    let result =
        if isOdd input then
            input + 1
        else
            input

    result
```

The `if` expression produces a value called `result`. Note that you could omit `result` entirely, making the `if` expression the body of the `addOneIfOdd` function. The key thing to remember about expressions is that they produce a value.

There is a special type, `unit`, that is used when there is nothing to return. For example, consider this simple function:

```
let printString (str: string) =
    printfn $"String is: {str}"
```

The signature looks like this:

```
val printString: str:string -> unit
```

The `unit` type indicates that there is no actual value being returned. This is useful when you have a routine that must "do work" despite having no value to return as a result of that work.

This is in sharp contrast to imperative programming, where the equivalent `if` construct is a statement, and producing values is often done with mutating variables. For example, in C#, the code might be written like this:

```
bool IsOdd(int x) => x % 2 != 0;

int AddOneIfOdd(int input)
{
    var result = input;

    if (IsOdd(input))
    {
        result = input + 1;
    }

    return result;
}
```

It's worth noting that C# and other C-style languages do support the ternary expression, which allows for expression-based conditional programming.

In functional programming, it is rare to mutate values with statements. Although some functional languages support statements and mutation, it is not common to use these concepts in functional programming.

Pure functions

As previously mentioned, pure functions are functions that:

- Always evaluate to the same value for the same input.
- Have no side effects.

It is helpful to think of mathematical functions in this context. In mathematics, functions depend only on their arguments and do not have any side effects. In the mathematical function `f(x) = x + 1`, the value of `f(x)` depends only on the value of `x`. Pure functions in functional programming are the same way.

When writing a pure function, the function must depend only on its arguments and not perform any action that results in a side effect.

Here is an example of a non-pure function because it depends on global, mutable state:

```
let mutable value = 1

let addOneToValue x = x + value
```

The `addOneToValue` function is clearly impure, because `value` could be changed at any time to have a different value than 1. This pattern of depending on a global value is to be avoided in functional programming.

Here is another example of a non-pure function, because it performs a side effect:

```
let addOneToValue x =
    printfn $"x is %d{x}"
    x + 1
```

Although this function does not depend on a global value, it writes the value of `x` to the output of the program. Although there is nothing inherently wrong with doing this, it does mean that the function is not pure. If another part of your program depends on something external to the program, such as the output buffer, then calling this function can affect that other part of your program.

Removing the `printfn` statement makes the function pure:

```
let addOneToValue x = x + 1
```

Although this function is not inherently *better* than the previous version with the `printfn` statement, it does guarantee that all this function does is return a value. Calling this function any number of times produces the same result: it just produces a value. The predictability given by purity is something many functional programmers strive for.

Immutability

Finally, one of the most fundamental concepts of typed functional programming is immutability. In F#, all values are immutable by default. That means they cannot be mutated in-place unless you explicitly mark them as mutable.

In practice, working with immutable values means that you change your approach to programming from, "I need to change something", to "I need to produce a new value".

For example, adding 1 to a value means producing a new value, not mutating the existing one:

```
let value = 1
let secondValue = value + 1
```

In F#, the following code does **not** mutate the `value` function; instead, it performs an equality check:

```
let value = 1
value = value + 1 // Produces a 'bool' value!
```

Some functional programming languages do not support mutation at all. In F#, it is supported, but it is not the default behavior for values.

This concept extends even further to data structures. In functional programming, immutable data structures such as sets (and many more) have a different implementation than you might initially expect. Conceptually, something like adding an item to a set does not change the set, it produces a *new* set with the added value. Under the covers, this is often accomplished by a different data structure that allows for efficiently tracking a value so that the appropriate representation of the data can be given as a result.

This style of working with values and data structures is critical, as it forces you to treat any operation that modifies something as if it creates a new version of that thing. This allows for things like equality and comparability to be consistent in your programs.

Next steps

The next section will thoroughly cover functions, exploring different ways you can use them in functional programming.

Using functions in F# explores functions deeply, showing how you can use them in various contexts.

Further reading

The Thinking Functionally series is another great resource to learn about functional programming with F#. It covers fundamentals of functional programming in a pragmatic and easy-to-read way, using F# features to illustrate the concepts.

Async programming in F#

8/5/2022 • 13 minutes to read • Edit Online

Asynchronous programming is a mechanism that is essential to modern applications for diverse reasons. There are two primary use cases that most developers will encounter:

- Presenting a server process that can service a significant number of concurrent incoming requests, while minimizing the system resources occupied while request processing awaits inputs from systems or services external to that process
- Maintaining a responsive UI or main thread while concurrently progressing background work

Although background work often does involve the utilization of multiple threads, it's important to consider the concepts of asynchrony and multi-threading separately. In fact, they are separate concerns, and one does not imply the other. This article describes the separate concepts in more detail.

Asynchrony defined

The previous point - that asynchrony is independent of the utilization of multiple threads - is worth explaining a bit further. There are three concepts that are sometimes related, but strictly independent of one another:

- Concurrency; when multiple computations execute in overlapping time periods.
- Parallelism; when multiple computations or several parts of a single computation run at exactly the same time.
- Asynchrony; when one or more computations can execute separately from the main program flow.

All three are orthogonal concepts, but can be easily conflated, especially when they are used together. For example, you may need to execute multiple asynchronous computations in parallel. This relationship does not mean that parallelism or asynchrony imply one another.

If you consider the etymology of the word "asynchronous", there are two pieces involved:

- "a", meaning "not".
- "synchronous", meaning "at the same time".

When you put these two terms together, you'll see that "asynchronous" means "not at the same time". That's it! There is no implication of concurrency or parallelism in this definition. This is also true in practice.

In practical terms, asynchronous computations in F# are scheduled to execute independently of the main program flow. This independent execution doesn't imply concurrency or parallelism, nor does it imply that a computation always happens in the background. In fact, asynchronous computations can even execute synchronously, depending on the nature of the computation and the environment the computation is executing in.

The main takeaway you should have is that asynchronous computations are independent of the main program flow. Although there are few guarantees about when or how an asynchronous computation executes, there are some approaches to orchestrating and scheduling them. The rest of this article explores core concepts for F# asynchrony and how to use the types, functions, and expressions built into F#.

Core concepts

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

- The `Async<'T>` type with `async { }` computation expression, which represents a composable asynchronous computation that can be started to form a task.
- The `Task<'T>` type, with `task { }` computation expression, which represents an executing .NET task.

In general, you should use `async { }` programming in F# unless you frequently need to create or consume .NET tasks.

Core concepts of async

You can see the basic concepts of "async" programming in the following example:

```
open System
open System.IO

// Perform an asynchronous read of a file using 'async'
let printTotalFileBytesUsingAsync (path: string) =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    printTotalFileBytesUsingAsync "path-to-file.txt"
    |> Async.RunSynchronously

    Console.Read() |> ignore
    0
```

In the example, the `printTotalFileBytesUsingAsync` function is of type `string -> Async<unit>`. Calling the function does not actually execute the asynchronous computation. Instead, it returns an `Async<unit>` that acts as a *specification* of the work that is to execute asynchronously. It calls `Async.AwaitTask` in its body, which converts the result of ReadAllBytesAsync to an appropriate type.

Another important line is the call to `Async.RunSynchronously`. This is one of the Async module starting functions that you'll need to call if you want to actually execute an F# asynchronous computation.

This is a fundamental difference with the C#/Visual Basic style of `async` programming. In F#, asynchronous computations can be thought of as **Cold tasks**. They must be explicitly started to actually execute. This has some advantages, as it allows you to combine and sequence asynchronous work much more easily than in C# or Visual Basic.

Combine asynchronous computations

Here is an example that builds upon the previous one by combining computations:

```
open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    argv
    |> Seq.map printTotalFileBytes
    |> Async.Parallel
    |> Async.Ignore
    |> Async.RunSynchronously

    0
```

As you can see, the `main` function has quite a few more elements. Conceptually, it does the following:

1. Transform the command-line arguments into a sequence of `Async<unit>` computations with `Seq.map`.
2. Create an `Async<'T[]>` that schedules and runs the `printTotalFileBytes` computations in parallel when it runs.
3. Create an `Async<unit>` that will run the parallel computation and ignore its result (which is a `unit[]`).
4. Explicitly run the overall composed computation with `Async.RunSynchronously`, blocking until it completes.

When this program runs, `printTotalFileBytes` runs in parallel for each command-line argument. Because asynchronous computations execute independently of program flow, there is no defined order in which they print their information and finish executing. The computations will be scheduled in parallel, but their order of execution is not guaranteed.

Sequence asynchronous computations

Because `Async<'T>` is a specification of work rather than an already-running task, you can perform more intricate transformations easily. Here is an example that sequences a set of Async computations so they execute one after another.

```
let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    argv
    |> Seq.map printTotalFileBytes
    |> Async.Sequential
    |> Async.Ignore
    |> Async.RunSynchronously
    |> ignore
```

This will schedule `printTotalFileBytes` to execute in the order of the elements of `argv` rather than scheduling them in parallel. Because each successive operation will not be scheduled until after the preceding computation has finished executing, the computations are sequenced such that there is no overlap in their execution.

Important Async module functions

When you write async code in F#, you'll usually interact with a framework that handles scheduling of computations for you. However, this is not always the case, so it is good to understand the various functions that can be used to schedule asynchronous work.

Because F# asynchronous computations are a *specification* of work rather than a representation of work that is already executing, they must be explicitly started with a starting function. There are many Async starting methods that are helpful in different contexts. The following section describes some of the more common starting functions.

Async.StartChild

Starts a child computation within an asynchronous computation. This allows multiple asynchronous computations to be executed concurrently. The child computation shares a cancellation token with the parent computation. If the parent computation is canceled, the child computation is also canceled.

Signature:

```
computation: Async<'T> * ?millisecondsTimeout: int -> Async<Async<'T>>
```

When to use:

- When you want to execute multiple asynchronous computations concurrently rather than one at a time, but not have them scheduled in parallel.
- When you wish to tie the lifetime of a child computation to that of a parent computation.

What to watch out for:

- Starting multiple computations with `Async.StartChild` isn't the same as scheduling them in parallel. If you wish to schedule computations in parallel, use `Async.Parallel`.
- Canceling a parent computation will trigger cancellation of all child computations it started.

Async.StartImmediate

Runs an asynchronous computation, starting immediately on the current operating system thread. This is helpful if you need to update something on the calling thread during the computation. For example, if an asynchronous computation must update a UI (such as updating a progress bar), then `Async.StartImmediate` should be used.

Signature:

```
computation: Async<unit> * ?cancellationToken: CancellationToken -> unit
```

When to use:

- When you need to update something on the calling thread in the middle of an asynchronous computation.

What to watch out for:

- Code in the asynchronous computation will run on whatever thread one happens to be scheduled on. This can be problematic if that thread is in some way sensitive, such as a UI thread. In such cases, `Async.StartImmediate` is likely inappropriate to use.

Async.StartAsTask

Executes a computation in the thread pool. Returns a Task<TResult> that will be completed on the corresponding state once the computation terminates (produces the result, throws exception, or gets canceled).

If no cancellation token is provided, then the default cancellation token is used.

Signature:

```
computation: Async<'T> * ?taskCreationOptions: TaskCreationOptions * ?cancellationToken: CancellationToken -
> Task<'T>
```

When to use:

- When you need to call into a .NET API that yields a Task<TResult> to represent the result of an asynchronous computation.

What to watch out for:

- This call will allocate an additional `Task` object, which can increase overhead if it is used often.

Async.Parallel

Schedules a sequence of asynchronous computations to be executed in parallel, yielding an array of results in the order they were supplied. The degree of parallelism can be optionally tuned/throttled by specifying the `maxDegreeOfParallelism` parameter.

Signature:

```
computations: seq<Async<'T>> * ?maxDegreeOfParallelism: int -> Async<'T[]>
```

When to use it:

- If you need to run a set of computations at the same time and have no reliance on their order of execution.
- If you don't require results from computations scheduled in parallel until they have all completed.

What to watch out for:

- You can only access the resulting array of values once all computations have finished.
- Computations will be run whenever they end up getting scheduled. This behavior means you cannot rely on their order of their execution.

Async.Sequential

Schedules a sequence of asynchronous computations to be executed in the order that they are passed. The first computation will be executed, then the next, and so on. No computations will be executed in parallel.

Signature:

```
computations: seq<Async<'T>> -> Async<'T[]>
```

When to use it:

- If you need to execute multiple computations in order.

What to watch out for:

- You can only access the resulting array of values once all computations have finished.
- Computations will be run in the order that they are passed to this function, which can mean that more time will elapse before the results are returned.

Async.AwaitTask

Returns an asynchronous computation that waits for the given Task<TResult> to complete and returns its result as an `Async<'T>`

Signature:

```
task: Task<'T> -> Async<'T>
```

When to use:

- When you are consuming a .NET API that returns a Task<TResult> within an F# asynchronous computation.

What to watch out for:

- Exceptions are wrapped in AggregateException following the convention of the Task Parallel Library; this behavior is different from how F# async generally surfaces exceptions.

Async.Catch

Creates an asynchronous computation that executes a given `Async<'T>`, returning an `Async<Choice<'T, exn>>`. If the given `Async<'T>` completes successfully, then a `Choice1Of2` is returned with the resultant value. If an exception is thrown before it completes, then a `Choice2of2` is returned with the raised exception. If it is used on an asynchronous computation that is itself composed of many computations, and one of those computations throws an exception, the encompassing computation will be stopped entirely.

Signature:

```
computation: Async<'T> -> Async<Choice<'T, exn>>
```

When to use:

- When you are performing asynchronous work that may fail with an exception and you want to handle that exception in the caller.

What to watch out for:

- When using combined or sequenced asynchronous computations, the encompassing computation will fully stop if one of its "internal" computations throws an exception.

Async.Ignore

Creates an asynchronous computation that runs the given computation but drops its result.

Signature:

```
computation: Async<'T> -> Async<unit>
```

When to use:

- When you have an asynchronous computation whose result is not needed. This is analogous to the `ignore` function for non-asynchronous code.

What to watch out for:

- If you must use `Async.Ignore` because you wish to use `Async.Start` or another function that requires `Async<unit>`, consider if discarding the result is okay. Avoid discarding results just to fit a type signature.

Async.RunSynchronously

Runs an asynchronous computation and awaits its result on the calling thread. Propagates an exception should the computation yield one. This call is blocking.

Signature:

```
computation: Async<'T> * ?timeout: int * ?cancellationToken: CancellationToken -> 'T
```

When to use it:

- If you need it, use it only once in an application - at the entry point for an executable.
- When you don't care about performance and want to execute a set of other asynchronous operations at once.

What to watch out for:

- Calling `Async.RunSynchronously` blocks the calling thread until the execution completes.

Async.Start

Starts an asynchronous computation that returns `unit` in the thread pool. Doesn't wait for its completion and/or observe an exception outcome. Nested computations started with `Async.Start` are started independently of the parent computation that called them; their lifetime is not tied to any parent computation. If the parent computation is canceled, no child computations are canceled.

Signature:

```
computation: Async<unit> * ?cancellationToken: CancellationToken -> unit
```

Use only when:

- You have an asynchronous computation that doesn't yield a result and/or require processing of one.
- You don't need to know when an asynchronous computation completes.
- You don't care which thread an asynchronous computation runs on.
- You don't have any need to be aware of or report exceptions resulting from the execution.

What to watch out for:

- Exceptions raised by computations started with `Async.Start` aren't propagated to the caller. The call stack will be completely unwound.
- Any work (such as calling `printfn`) started with `Async.Start` won't cause the effect to happen on the main thread of a program's execution.

Interoperate with .NET

If using `async { }` programming, you may need to interoperate with a .NET library or C# codebase that uses async/await-style asynchronous programming. Because C# and the majority of .NET libraries use the Task<TResult> and Task types as their core abstractions this may change how you write your F# asynchronous code.

One option is to switch to writing .NET tasks directly using `task { }`. Alternatively, you can use the `Async.AwaitTask` function to await a .NET asynchronous computation:

```
let getValueFromLibrary param =
    async {
        let! value = DotNetLibrary.GetValueAsync param |> Async.AwaitTask
        return value
    }
```

You can use the `Async.StartAsTask` function to pass an asynchronous computation to a .NET caller:

```
let computationForCaller param =
    async {
        let! result = getAsyncResult param
        return result
    } |> Async.StartAsTask
```

To work with APIs that use Task (that is, .NET async computations that do not return a value), you may need to add an additional function that will convert an `Async<'T>` to a Task:

```
module Async =
    // Async<unit> -> Task
    let startTaskFromAsyncUnit (comp: Async<unit>) =
        Async.StartAsTask comp :> Task
```

There is already an `Async.AwaitTask` that accepts a Task as input. With this and the previously defined `startTaskFromAsyncUnit` function, you can start and await Task types from an F# async computation.

Writing .NET tasks directly in F#

In F#, you can write tasks directly using `task { }`, for example:

```
open System
open System.IO

/// Perform an asynchronous read of a file using 'task'
let printTotalFileBytesUsingTasks (path: string) =
    task {
        let! bytes = File.ReadAllBytesAsync(path)
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    let task = printTotalFileBytesUsingTasks "path-to-file.txt"
    task.Wait()

    Console.Read() |> ignore
    0
```

In the example, the `printTotalFileBytesUsingTasks` function is of type `string -> Task<unit>`. Calling the function starts to execute the task. The call to `task.Wait()` waits for the task to complete.

Relationship to multi-threading

Although threading is mentioned throughout this article, there are two important things to remember:

1. There is no affinity between an asynchronous computation and a thread, unless explicitly started on the current thread.
2. Asynchronous programming in F# is not an abstraction for multi-threading.

For example, a computation may actually run on its caller's thread, depending on the nature of the work. A computation could also "jump" between threads, borrowing them for a small amount of time to do useful work in between periods of "waiting" (such as when a network call is in transit).

Although F# provides some abilities to start an asynchronous computation on the current thread (or explicitly not on the current thread), asynchrony generally is not associated with a particular threading strategy.

See also

- The F# Asynchronous Programming Model
- Leo Gorodinski's F# Async Guide
- F# for fun and profit's Asynchronous Programming guide
- Async in C# and F#: Asynchronous gotchas in C#

Using functions in F#

8/5/2022 • 22 minutes to read • Edit Online

A simple function definition resembles the following:

```
let f x = x + 1
```

In the previous example, the function name is `f`, the argument is `x`, which has type `int`, the function body is `x + 1`, and the return value is of type `int`.

A defining characteristic of F# is that functions have first-class status. You can do with a function whatever you can do with values of the other built-in types, with a comparable degree of effort.

- You can give function values names.
- You can store functions in data structures, such as in a list.
- You can pass a function as an argument in a function call.
- You can return a function from a function call.

Give the Value a Name

If a function is a first-class value, you must be able to name it, just as you can name integers, strings, and other built-in types. This is referred to in functional programming literature as binding an identifier to a value. F# uses `let` bindings to bind names to values: `let <identifier> = <value>`. The following code shows two examples.

```
// Integer and string.
let num = 10
let str = "F#"
```

You can name a function just as easily. The following example defines a function named `squareIt` by binding the identifier `squareIt` to the lambda expression `fun n -> n * n`. Function `squareIt` has one parameter, `n`, and it returns the square of that parameter.

```
let squareIt = fun n -> n * n
```

F# provides the following more concise syntax to achieve the same result with less typing.

```
let squareIt2 n = n * n
```

The examples that follow mostly use the first style, `let <function-name> = <lambda-expression>`, to emphasize the similarities between the declaration of functions and the declaration of other types of values. However, all the named functions can also be written with the concise syntax. Some of the examples are written in both ways.

Store the Value in a Data Structure

A first-class value can be stored in a data structure. The following code shows examples that store values in lists and in tuples.

```
// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )
```

To verify that a function name stored in a tuple does in fact evaluate to a function, the following example uses the `fst` and `snd` operators to extract the first and second elements from tuple `funAndArgTuple`. The first element in the tuple is `squareIt` and the second element is `num`. Identifier `num` is bound in a previous example to integer 10, a valid argument for the `squareIt` function. The second expression applies the first element in the tuple to the second element in the tuple: `squareIt num`.

```
// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))
```

Similarly, just as identifier `num` and integer 10 can be used interchangeably, so can identifier `squareIt` and lambda expression `fun n -> n * n`.

```
// Make a tuple of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))
```

Pass the Value as an Argument

If a value has first-class status in a language, you can pass it as an argument to a function. For example, it is common to pass integers and strings as arguments. The following code shows integers and strings passed as arguments in F#.

```
// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)
```

If functions have first-class status, you must be able to pass them as arguments in the same way. Remember that this is the first characteristic of higher-order functions.

In the following example, function `applyIt` has two parameters, `op` and `arg`. If you send in a function that has one parameter for `op` and an appropriate argument for the function to `arg`, the function returns the result of applying `op` to `arg`. In the following example, both the function argument and the integer argument are sent in the same way, by using their names.

```
// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)
```

The ability to send a function as an argument to another function underlies common abstractions in functional programming languages, such as map or filter operations. A map operation, for example, is a higher-order function that captures the computation shared by functions that step through a list, do something to each element, and then return a list of the results. You might want to increment each element in a list of integers, or to square each element, or to change each element in a list of strings to uppercase. The error-prone part of the computation is the recursive process that steps through the list and builds a list of the results to return. That part is captured in the mapping function. All you have to write for a particular application is the function that you want to apply to each list element individually (adding, squaring, changing case). That function is sent as an argument to the mapping function, just as `squareIt` is sent to `applyIt` in the previous example.

F# provides map methods for most collection types, including lists, arrays, and sequences. The following examples use lists. The syntax is `List.map <the function> <the list>`.

```
// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot
```

For more information, see Lists.

Return the Value from a Function Call

Finally, if a function has first-class status in a language, you must be able to return it as the value of a function call, just as you return other types, such as integers and strings.

The following function calls return integers and display them.

```
// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)
```

The following function call returns a string.

```
// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()
```

The following function call, declared inline, returns a Boolean value. The value displayed is `True`.

```
System.Console.WriteLine((fun n -> n % 2 = 1) 15)
```

The ability to return a function as the value of a function call is the second characteristic of higher-order functions. In the following example, `checkFor` is defined to be a function that takes one argument, `item`, and returns a new function as its value. The returned function takes a list as its argument, `lst`, and searches for `item` in `lst`. If `item` is present, the function returns `true`. If `item` is not present, the function returns `false`. As in the previous section, the following code uses a provided list function, List.exists, to search the list.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

The following code uses `checkFor` to create a new function that takes one argument, a list, and searches for 7 in the list.

```
// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)
```

The following example uses the first-class status of functions in F# to declare a function, `compose`, that returns a composition of two function arguments.

```
// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                        op1 (op2 n)
    funToReturn
```

NOTE

For an even shorter version, see the following section, "Curried Functions."

The following code sends two functions as arguments to `compose`, both of which take a single argument of the same type. The return value is a new function that is a composition of the two function arguments.

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

NOTE

F# provides two operators, `<<` and `>>`, that compose functions. For example, `let squareAndDouble2 = doubleIt << squareIt` is equivalent to `let squareAndDouble = compose doubleIt squareIt` in the previous example.

The following example of returning a function as the value of a function call creates a simple guessing game. To create a game, call `makeGame` with the value that you want someone to guess sent in for `target`. The return value from function `makeGame` is a function that takes one argument (the guess) and reports whether the guess is correct.

```
let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                   if guess = target then
                      System.Console.WriteLine("You win!")
                   else
                      System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game
```

The following code calls `makeGame`, sending the value `7` for `target`. Identifier `playGame` is bound to the returned lambda expression. Therefore, `playGame` is a function that takes as its one argument a value for `guess`.

```
let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!
```

Curried Functions

Many of the examples in the previous section can be written more concisely by taking advantage of the implicit *currying* in F# function declarations. Currying is a process that transforms a function that has more than one parameter into a series of embedded functions, each of which has a single parameter. In F#, functions that have more than one parameter are inherently curried. For example, `compose` from the previous section can be written as shown in the following concise style, with three parameters.

```
let compose4 op1 op2 n = op1 (op2 n)
```

However, the result is a function of one parameter that returns a function of one parameter that in turn returns another function of one parameter, as shown in `compose4curried`.

```
let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)
```

You can access this function in several ways. Each of the following examples returns and displays 18. You can replace `compose4` with `compose4curried` in any of the examples.

```
// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)
```

To verify that the function still works as it did before, try the original test cases again.

```
let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)
```

NOTE

You can restrict currying by enclosing parameters in tuples. For more information, see "Parameter Patterns" in Parameters and Arguments.

The following example uses implicit currying to write a shorter version of `makeGame`. The details of how `makeGame` constructs and returns the `game` function are less explicit in this format, but you can verify by using the original test cases that the result is the same.

```
let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

For more information about currying, see "Partial Application of Arguments" in Functions.

Identifier and Function Definition Are Interchangeable

The variable name `num` in the previous examples evaluates to the integer 10, and it is no surprise that where `num` is valid, 10 is also valid. The same is true of function identifiers and their values: anywhere the name of the function can be used, the lambda expression to which it is bound can be used.

The following example defines a `Boolean` function called `isNegative`, and then uses the name of the function and the definition of the function interchangeably. The next three examples all return and display `False`.

```
let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)
```

To take it one step further, substitute the value that `applyIt` is bound to for `applyIt`.

```
System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)
```

Functions Are First-Class Values in F#

The examples in the previous sections demonstrate that functions in F# satisfy the criteria for being first-class values in F#:

- You can bind an identifier to a function definition.

```
let squareIt = fun n -> n * n
```

- You can store a function in a data structure.

```
let funTuple2 = ( BMICalculator, fun n -> n * n )
```

- You can pass a function as an argument.

```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]
```

- You can return a function as the value of a function call.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

For more information about F#, see the F# Language Reference.

Example

Description

The following code contains all the examples in this topic.

Code

```
// ** GIVE THE VALUE A NAME **


// Integer and string.
let num = 10
let str = "F#"



let squareIt = fun n -> n * n



let squareIt2 n = n * n



// ** STORE THE VALUE IN A DATA STRUCTURE **


// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
```

```
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )



// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))



// Make a list of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))



// ** PASS THE VALUE AS AN ARGUMENT **


// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)



// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
```

```
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)



// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot



// ** RETURN THE VALUE FROM A FUNCTION CALL **


// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)


// The following function call returns a string:

// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()



System.Console.WriteLine((fun n -> n % 2 = 1) 15)



let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn



// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
```

```
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)
```

```
// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                        op1 (op2 n)
    funToReturn
```

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

```
let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                  if guess = target then
                     System.Console.WriteLine("You win!")
                  else
                     System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game
```

```
let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
```

```
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!



// ** CURRIED FUNCTIONS **


let compose4 op1 op2 n = op1 (op2 n)



let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)



// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)



let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)



let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

```
// ** IDENTIFIER AND FUNCTION DEFINITION ARE INTERCHANGEABLE **


let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)



System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)



// ** FUNCTIONS ARE FIRST-CLASS VALUES IN F# **

//let squareIt = fun n -> n * n


let funTuple2 = ( BMICalculator, fun n -> n * n )



let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]


//let checkFor item =
//    let functionToReturn = fun lst ->
//                           List.exists (fun a -> a = item) lst
//    functionToReturn
```

See also

- Lists
- Tuples
- Functions
- `let` Bindings
- Lambda Expressions: The `fun` Keyword

What's new in F# 6

8/5/2022 • 16 minutes to read • Edit Online

F# 6 adds several improvements to the F# language and F# Interactive. It is released with **.NET 6**.

You can download the latest .NET SDK from the .NET downloads page.

Get started

F# 6 is available in all .NET Core distributions and Visual Studio tooling. For more information, see Get started with F#.

task {…}

F# 6 includes native support for authoring .NET tasks in F# code. For example, consider the following F# code to create a .NET-compatible task:

```
let readFilesTask (path1, path2) =
   async {
        let! bytes1 = File.ReadAllBytesAsync(path1) |> Async.AwaitTask
        let! bytes2 = File.ReadAllBytesAsync(path2) |> Async.AwaitTask
        return Array.append bytes1 bytes2
   } |> Async.StartAsTask
```

Using F# 6, this code can be rewritten as follows.

```
let readFilesTask (path1, path2) =
   task {
        let! bytes1 = File.ReadAllBytesAsync(path1)
        let! bytes2 = File.ReadAllBytesAsync(path2)
        return Array.append bytes1 bytes2
   }
```

Task support was available for F# 5 through the excellent TaskBuilder.fs and Ply libraries. It should be straightforward to migrate code to the built-in support. However, there are some differences: namespaces and type inference differ slightly between the built-in support and these libraries, and some additional type annotations may be needed. If necessary, you can still use these community libraries with F# 6 if you reference them explicitly and open the correct namespaces in each file.

Using `task {…}` is very similar to using `async {…}`. Using `task {…}` has several advantages over `async {…}`:

- The performance of `task {…}` is much better.
- Debugging stepping and stack traces for `task {…}` is better.
- Interoperating with .NET packages that expect or produce tasks is easier.

If you're familiar with `async {…}`, there are some differences to be aware of:

- `task {…}` immediately executes the task to the first await point.
- `task {…}` does not implicitly propagate a cancellation token.
- `task {…}` does not perform implicit cancellation checks.
- `task {…}` does not support asynchronous tailcalls. This means using `return! ..` recursively may result in stack overflows if there are no intervening asynchronous waits.

In general, you should consider using `task {…}` over `async {…}` in new code if you're interoperating with .NET libraries that use tasks, and if you don't rely on asynchronous code tailcalls or implicit cancellation token propagation. In existing code, you should only switch to `task {…}` once you have reviewed your code to ensure you are not relying on the previously mentioned characteristics of `async {…}`.

This feature implements F# RFC FS-1097.

Simpler indexing syntax with `expr[idx]`

F# 6 allows the syntax `expr[idx]` for indexing and slicing collections.

Up to and including F# 5, F# has used `expr.[idx]` as indexing syntax. Allowing the use of `expr[idx]` is based on repeated feedback from those learning F# or seeing F# for the first time that the use of dot-notation indexing comes across as an unnecessary divergence from standard industry practice.

This is not a breaking change because by default, no warnings are emitted on the use of `expr.[idx]`. However, some informational messages that suggest code clarifications are emitted. You can optionally activate further informational messages as well. For example, you can activate an optional informational warning (`/warnon:3566`) to start reporting uses of the `expr.[idx]` notation. For more information, see Indexer Notation.

In new code, we recommend the systematic use of `expr[idx]` as the indexing syntax.

This feature implements F# RFC FS-1110.

Struct representations for partial active patterns

F# 6 augments the "active patterns" feature with optional struct representations for partial active patterns. This allows you to use an attribute to constrain a partial active pattern to return a value option:

```
[<return: Struct>]
let (|Int|_|) str =
   match System.Int32.TryParse(str) with
   | true, int -> ValueSome(int)
   | _ -> ValueNone
```

The use of the attribute is required. At usage sites, code doesn't change. The net result is that allocations are reduced.

This feature implements F# RFC FS-1039.

Overloaded custom operations in computation expressions

F# 6 lets you consume interfaces with default implementations.

Consider the following use of a computation expression builder `content`:

```
let mem = new System.IO.MemoryStream("Stream"B)
let content = ContentBuilder()
let ceResult =
    content {
        body "Name"
        body (ArraySegment<_>("Email"B, 0, 5))
        body "Password"B 2 4
        body "BYTES"B
        body mem
        body "Description" "of" "content"
    }
```

Here the `body` custom operation takes a varying number of arguments of different types. This is supported by the implementation of the following builder, which uses overloading:

```
type Content = ArraySegment<byte> list

type ContentBuilder() =
    member _.Run(c: Content) =
        let crlf = "\r\n"B
        [|for part in List.rev c do
            yield! part.Array[part.Offset..(part.Count+part.Offset-1)]
            yield! crlf |]

    member _.Yield(_) = []

    [<CustomOperation("body")>]
    member _.Body(c: Content, segment: ArraySegment<byte>) =
        segment::c

    [<CustomOperation("body")>]
    member _.Body(c: Content, bytes: byte[]) =
        ArraySegment<byte>(bytes, 0, bytes.Length)::c

    [<CustomOperation("body")>]
    member _.Body(c: Content, bytes: byte[], offset, count) =
        ArraySegment<byte>(bytes, offset, count)::c

    [<CustomOperation("body")>]
    member _.Body(c: Content, content: System.IO.Stream) =
        let mem = new System.IO.MemoryStream()
        content.CopyTo(mem)
        let bytes = mem.ToArray()
        ArraySegment<byte>(bytes, 0, bytes.Length)::c

    [<CustomOperation("body")>]
    member _.Body(c: Content, [<ParamArray>] contents: string[]) =
        List.rev [for c in contents -> let b = Text.Encoding.ASCII.GetBytes c in ArraySegment<_>
(b,0,b.Length)] @ c
```

This feature implements F# RFC FS-1056.

“as” patterns

In F# 6, the right-hand side of an `as` pattern can now itself be a pattern. This is important when a type test has given a stronger type to an input. For example, consider the following code:

```
type Pair = Pair of int * int

let analyzeObject (input: obj) =
    match input with
    | :? (int * int) as (x, y) -> printfn $"A tuple: {x}, {y}"
    | :? Pair as Pair (x, y) -> printfn $"A DU: {x}, {y}"
    | _ -> printfn "Nope"

let input = box (1, 2)
```

In each pattern case, the input object is type-tested. The right-hand side of the `as` pattern is now allowed to be a further pattern, which can itself match the object at the stronger type.

This feature implements F# RFC FS-1105.

Indentation syntax revisions

F# 6 removes a number of inconsistencies and limitations in its use of indentation-aware syntax. See RFC FS-1108. This resolves 10 significant issues highlighted by F# users since F# 4.0.

For example, in F# 5 the following code was allowed:

```
let c = (
    printfn "aaaa"
    printfn "bbbb"
)
```

However, the following code was not allowed (it produced a warning):

```
let c = [
    1
    2
]
```

In F# 6, both are allowed. This makes F# simpler and easier to learn. The F# community contributor Hadrian Tang has led the way on this, including remarkable and highly valuable systematic testing of the feature.

This feature implements F# RFC FS-1108.

Additional implicit conversions

In F# 6, we've activated support for additional “implicit” and “type-directed” conversions, as described in RFC FS-1093.

This change brings three advantages:

1. Fewer explicit upcasts are required
2. Fewer explicit integer conversions are required
3. First-class support for .NET-style implicit conversions is added

This feature implements F# RFC FS-1093.

Additional implicit upcast conversions

F# 6 implements additional implicit upcast conversions. For example, in F# 5 and earlier versions, upcasts were needed for the return expression when implementing a function where the expressions had different subtypes on different branches, even when a type annotation was present. Consider the following F# 5 code:

```
open System
open System.IO

let findInputSource () : TextReader =
    if DateTime.Now.DayOfWeek = DayOfWeek.Monday then
        // On Monday a TextReader
        Console.In
    else
        // On other days a StreamReader
        File.OpenText("path.txt") :> TextReader
```

Here the branches of the conditional compute a `TextReader` and `StreamReader` respectively, and the upcast was added to make both branches have type StreamReader. In F# 6, these upcasts are now added automatically. This means the code is simpler:

```
let findInputSource () : TextReader =
    if DateTime.Now.DayOfWeek = DayOfWeek.Monday then
        // On Monday a TextReader
        Console.In
    else
        // On other days a StreamReader
        File.OpenText("path.txt")
```

You may optionally enable the warning `/warnon:3388` to show a warning at every point an additional implicit upcast is used, as described in Optional warnings for implicit conversions.

Implicit integer conversions

In F# 6, 32-bit integers are widened to 64-bit integers when both types are known. For example, consider a typical API shape:

```
type Tensor(…) =
    static member Create(sizes: seq<int64>) = Tensor(…)
```

In F# 5, integer literals for int64 must be used:

```
Tensor.Create([100L; 10L; 10L])
```

or

```
Tensor.Create([int64 100; int64 10; int64 10])
```

In F# 6, widening happens automatically for `int32` to `int64`, `int32` to `nativeint`, and `int32` to `double`, when both source and destination type are known during type inference. So in cases such as the previous examples, `int32` literals can be used:

```
Tensor.Create([100; 10; 10])
```

Despite this change, F# continues to use explicit widening of numeric types in most cases. For example, implicit widening does not apply to other numeric types, such as `int8` or `int16`, or from `float32` to `float64`, or when either source or destination type is unknown. You can also optionally enable the warning `/warnon:3389` to show a warning at every point implicit numeric widening is used, as described in Optional warnings for implicit conversions.

First-class support for .NET-style implicit conversions

In F# 6, .NET “op_Implicit” conversions are applied automatically in F# code when calling methods. For example, in F# 5 it was necessary to use `XName.op_Implicit` when working with .NET APIs for XML:

```
open System.Xml.Linq
let purchaseOrder = XElement.Load("PurchaseOrder.xml")
let partNos = purchaseOrder.Descendants(XName.op_Implicit "Item")
```

In F# 6, `op_Implicit` conversions are applied automatically for argument expressions when types are available for source expression and target type:

```
open System.Xml.Linq
let purchaseOrder = XElement.Load("PurchaseOrder.xml")
let partNos = purchaseOrder.Descendants("Item")
```

You may optionally enable the warning `/warnon:3395` to show a warning at every point `op_Implicit` conversions widening is used at method arguments, as described in Optional warnings for implicit conversions.

> **NOTE**
>
> In the first release of F# 6, this warning number was `/warnon:3390`. Due to a conflict, the warning number was later updated to `/warnon:3395`.

Optional warnings for implicit conversions

Type-directed and implicit conversions can interact poorly with type inference and lead to code that's harder to understand. For this reason, some mitigations exist to help ensure this feature is not abused in F# code. First, both source and destination type must be strongly known, with no ambiguity or additional type inference arising. Secondly, opt-in warnings can be activated to report any use of implicit conversions, with one warning on by default:

- `/warnon:3388` (additional implicit upcast)
- `/warnon:3389` (implicit numeric widening)
- `/warnon:3391` (op_Implicit at non-method arguments, on by default)
- `/warnon:3395` (op_Implicit at method arguments)

If your team wants to ban all uses of implicit conversions, you can also specify `/warnaserror:3388`, `/warnaserror:3389`, `/warnaserror:3391`, and `/warnaserror:3395`.

Formatting for binary numbers

F# 6 adds the `%B` pattern to the available format specifiers for binary number formats. Consider the following F# code:

```
printf "%o" 123
printf "%B" 123
```

This code prints the following output:

```
173
1111011
```

This feature implements F# RFC FS-1100.

Discards on use bindings

F# 6 allows `_` to be used in a `use` binding, for example:

```
let doSomething () =
    use _ = System.IO.File.OpenText("input.txt")
    printfn "reading the file"
```

This feature implements F# RFC FS-1102.

InlineIfLambda

The F# compiler includes an optimizer that performs inlining of code. In F# 6 we've added a new declarative feature that allows code to optionally indicate that, if an argument is determined to be a lambda function, then that argument should itself always be inlined at call sites.

For example, consider the following `iterateTwice` function to traverse an array:

```
let inline iterateTwice ([<InlineIfLambda>] action) (array: 'T[]) =
    for j = 0 to array.Length-1 do
        action array[j]
    for j = 0 to array.Length-1 do
        action array[j]
```

If the call site is:

```
let arr = [| 1.. 100 |]
let mutable sum = 0
arr  |> iterateTwice (fun x ->
    sum <- sum + x)
```

Then after inlining and other optimizations, the code becomes:

```
let arr = [| 1.. 100 |]
let mutable sum = 0
for j = 0 to array.Length-1 do
    sum <- array[i] + x
for j = 0 to array.Length-1 do
    sum <- array[i] + x
```

Unlike previous versions of F#, this optimization is applied regardless of the size of the lambda expression involved. This feature can also be used to implement loop unrolling and similar transformations more reliably.

An opt-in warning (`/warnon:3517` , off by default) can be turned on to indicate places in your code where `InlineIfLambda` arguments are not bound to lambda expressions at call sites. In normal situations, this warning should not be enabled. However, in certain kinds of high-performance programming, it can be useful to ensure all code is inlined and flattened.

This feature implements F# RFC FS-1098.

Resumable code

The `task {…}` support of F# 6 is built on a foundation called *resumable code* RFC FS-1087. Resumable code is a technical feature that can be used to build many kinds of high-performance asynchronous and yielding state machines.

Additional collection functions

FSharp.Core 6.0.0 adds five new operations to the core collection functions. These functions are:

- List/Array/Seq.insertAt
- List/Array/Seq.removeAt
- List/Array/Seq.updateAt
- List/Array/Seq.insertManyAt
- List/Array/Seq.removeManyAt

These functions all perform copy-and-update operations on the corresponding collection type or sequence. This type of operation is a form of a “functional update”. For examples of using these functions, see the corresponding documentation, for example, List.insertAt.

As an example, consider the model, message, and update logic for a simple "Todo List" application written in the Elmish style. Here the user interacts with the application, generating messages, and the `update` function processes these messages, producing a new model:

```
type Model =
    { ToDo: string list }

type Message =
    | InsertToDo of index: int * what: string
    | RemoveToDo of index: int
    | LoadedToDos of index: int * what: string list

let update (model: Model) (message: Message) =
    match message with
    | InsertToDo (index, what) ->
        { model with ToDo = model.ToDo |> List.insertAt index what }
    | RemoveToDo index ->
        { model with ToDo = model.ToDo |> List.removeAt index }
    | LoadedToDos (index, what) ->
        { model with ToDo = model.ToDo |> List.insertManyAt index what }
```

With these new functions, the logic is clear and simple and relies only on immutable data.

This feature implements F# RFC FS-1113.

Map has Keys and Values

In FSharp.Core 6.0.0, the `Map` type now supports the Keys and Values properties. These properties do not copy the underlying collection.

This feature is documented in F# RFC FS-1113.

Additional intrinsics for NativePtr

FSharp.Core 6.0.0 adds new intrinsics to the NativePtr module:

- `NativePtr.nullPtr`
- `NativePtr.isNullPtr`
- `NativePtr.initBlock`
- `NativePtr.clear`
- `NativePtr.copy`
- `NativePtr.copyBlock`
- `NativePtr.ofILSigPtr`
- `NativePtr.toILSigPtr`

As with other functions in `NativePtr`, these functions are inlined, and their use emits warnings unless `/nowarn:9` is used. The use of these functions is restricted to unmanaged types.

This feature is documented in F# RFC FS-1109.

Additional numeric types with unit annotations

In F# 6, the following types or type abbreviation aliases now support unit-of-measure annotations. The new additions are shown in bold:

F# ALIAS	CLR TYPE
`float32` / `single`	`System.Single`
`float` / `double`	`System.Double`
`decimal`	`System.Decimal`
`sbyte` / `int8`	`System.SByte`
`int16`	`System.Int16`
`int` / `int32`	`System.Int32`
`int64`	`System.Int64`
`byte` / `uint8`	`System.Byte`
`uint16`	`System.UInt16`
`uint` / `uint32`	`System.UInt32`
`uint64`	`System.UIn64`
`nativeint`	`System.IntPtr`
`unativeint`	`System.UIntPtr`

For example, you can annotate an unsigned integer as follows:

```
[<Measure>]
type days

let better_age = 3u<days>
```

This feature is documented in F# RFC FS-1091.

Informational warnings for rarely used symbolic operators

F# 6 adds soft guidance that de-normalizes the use of `:=`, `!`, `incr`, and `decr` in F# 6 and beyond. Using these operators and functions produces informational messages that ask you to replace your code with explicit use of the `Value` property.

In F# programming, reference cells can be used for heap-allocated mutable registers. While they are occasionally useful, they're rarely needed in modern F# coding, because `let mutable` can be used instead. The F# core library includes two operators `:=` and `!` and two functions `incr` and `decr` specifically related to reference calls. The presence of these operators makes reference cells more central to F# programming than they need to be, requiring all F# programmers to know these operators. Further, the `!` operator can be easily confused with the `not` operation in C# and other languages, a potentially subtle source of bugs when translating code.

The rationale for this change is to reduce the number of operators the F# programmer needs to know, and thus simplify F# for beginners.

For example, consider the following F# 5 code:

```
let r = ref 0

let doSomething() =
    printfn "doing something”
    r := !r + 1
```

First, reference cells are rarely needed in modern F# coding, as `let mutable` can normally be used instead:

```
let mutable r = 0

let doSomething() =
    printfn "doing something”
    r <- r + 1
```

If you use reference cells, F# 6 emits an informational warning asking you to change the last line to `r.Value <- r.Value + 1`, and linking you to further guidance on the appropriate use of reference cells.

```
let r = ref 0

let doSomething() =
    printfn "doing something”
    r.Value <- r.Value + 1
```

These messages are not warnings; they are "informational messages" shown in the IDE and compiler output. F# remains backwards-compatible.

This feature implements F# RFC FS-1111.

F# tooling: .NET 6 the default for scripting in Visual Studio

If you open or execute an F# Script (`.fsx`) in Visual Studio, by default the script will be analyzed and executed using .NET 6 with 64-bit execution. This functionality has been in preview in the later releases of Visual Studio 2019 and is now enabled by default.

To enable .NET Framework scripting, select **Tools** > **Options** > **F# Tools** > **F# Interactive**. Set **Use .NET Core Scripting** to **false**, and then restart the F# Interactive window. This setting affects both script editing and script execution. To enable 32-bit execution for .NET Framework scripting, also set **64-bit F# Interactive** to **false**. There is no 32-bit option for .NET Core scripting.

F# tooling: Pin the SDK version of your F# scripts

If you execute a script using `dotnet fsi` in a directory containing a *global.json* file with a .NET SDK setting, then the listed version of the .NET SDK will be used to execute and edit the script. This feature has been available in the later versions of F# 5.

For example, assume there's a script in a directory with the following *global.json* file specifying a .NET SDK version policy:

```
{
  "sdk": {
    "version": "5.0.200",
    "rollForward": "minor"
  }
}
```

If you now execute the script using `dotnet fsi`, from this directory, the SDK version will be respected. This is a powerful feature that lets you "lock down" the SDK used to compile, analyze, and execute your scripts.

If you open and edit your script in Visual Studio and other IDEs, the tooling will respect this setting when analyzing and checking your script. If the SDK is not found, you will need to install it on your development machine.

On Linux and other Unix systems, you can combine this with a shebang to also specify a language version for direct execution of the script. A simple shebang for `script.fsx` is:

```
#!/usr/bin/env -S dotnet fsi

printfn "Hello, world"
```

Now the script can be executed directly with `script.fsx`. You can combine this with a specific, non-default language version like this:

```
#!/usr/bin/env -S dotnet fsi --langversion:5.0
```

> **NOTE**
> This setting is ignored by editing tools, which will analyze the script assuming latest language version.

Removing legacy features

Since F# 2.0, some deprecated legacy features have long given warnings. Using these features in F# 6 gives errors unless you explicitly use `/langversion:5.0`. The features that give errors are:

- Multiple generic parameters using a postfix type name, for example `(int, int) Dictionary`. This becomes an error in F# 6. The standard syntax `Dictionary<int,int>` should be used instead.
- `#indent "off"`. This becomes an error.
- `x.(expr)`. This becomes an error.
- `module M = struct … end` . This becomes an error.
- Use of inputs `*.ml` and `*.mli`. This becomes an error.
- Use of `(*IF-CAML*)` or `(*IF-OCAML*)`. This becomes an error.
- Use of `land`, `lor`, `lxor`, `lsl`, `lsr`, or `asr` as infix operators. These are infix keywords in F# because they were infix keywords in OCaml and are not defined in FSharp.Core. Using these keywords will now emit a warning.

This implements F# RFC FS-1114.

What's new in F# 5

8/5/2022 • 12 minutes to read • Edit Online

F# 5 adds several improvements to the F# language and F# Interactive. It is released with **.NET 5**.

You can download the latest .NET SDK from the .NET downloads page.

Get started

F# 5 is available in all .NET Core distributions and Visual Studio tooling. For more information, see Get started with F# to learn more.

Package references in F# scripts

F# 5 brings support for package references in F# scripts with `#r "nuget:..."` syntax. For example, consider the following package reference:

```
#r "nuget: Newtonsoft.Json"

open Newtonsoft.Json

let o = {| X = 2; Y = "Hello" |}

printfn $"{JsonConvert.SerializeObject o}"
```

You can also supply an explicit version after the name of the package like this:

```
#r "nuget: Newtonsoft.Json,11.0.1"
```

Package references support packages with native dependencies, such as ML.NET.

Package references also support packages with special requirements about referencing dependent `.dll`s. For example, the FParsec package used to require that users manually ensure that its dependent `FParsecCS.dll` was referenced first before `FParsec.dll` was referenced in F# Interactive. This is no longer needed, and you can reference the package as follows:

```
#r "nuget: FParsec"

open FParsec

let test p str =
    match run p str with
    | Success(result, _, _)   -> printfn $"Success: {result}"
    | Failure(errorMsg, _, _) -> printfn $"Failure: {errorMsg}"

test pfloat "1.234"
```

This feature implements F# Tooling RFC FST-1027. For more information on package references, see the F# Interactive tutorial.

String interpolation

F# interpolated strings are fairly similar to C# or JavaScript interpolated strings, in that they let you write code in "holes" inside of a string literal. Here's a basic example:

```
let name = "Phillip"
let age = 29
printfn $"Name: {name}, Age: {age}"

printfn $"I think {3.0 + 0.14} is close to {System.Math.PI}!"
```

However, F# interpolated strings also allow for typed interpolations, just like the `sprintf` function, to enforce that an expression inside of an interpolated context conforms to a particular type. It uses the same format specifiers.

```
let name = "Phillip"
let age = 29

printfn $"Name: %s{name}, Age: %d{age}"

// Error: type mismatch
printfn $"Name: %s{age}, Age: %d{name}"
```

In the preceding typed interpolation example, the `%s` requires the interpolation to be of type `string`, whereas the `%d` requires the interpolation to be an `integer`.

Additionally, any arbitrary F# expression (or expressions) can be placed in side of an interpolation context. It is even possible to write a more complicated expression, like so:

```
let str =
    $"""The result of squaring each odd item in {[1..10]} is:
{
    let square x = x * x
    let isOdd x = x % 2 <> 0
    let oddSquares xs =
        xs
        |> List.filter isOdd
        |> List.map square
    oddSquares [1..10]
}
"""
```

Although we don't recommend doing this too much in practice.

This feature implements F# RFC FS-1001.

Support for nameof

F# 5 supports the `nameof` operator, which resolves the symbol it's being used for and produces its name in F# source. This is useful in various scenarios, such as logging, and protects your logging against changes in source code.

```
let months =
    [
        "January"; "February"; "March"; "April";
        "May"; "June"; "July"; "August"; "September";
        "October"; "November"; "December"
    ]

let lookupMonth month =
    if (month > 12 || month < 1) then
        invalidArg (nameof month) (sprintf "Value passed in was %d." month)

    months[month-1]

printfn $"{lookupMonth 12}"
printfn $"{lookupMonth 1}"
printfn $"{lookupMonth 13}"
```

The last line will throw an exception and "month" will be shown in the error message.

You can take a name of nearly every F# construct:

```
module M =
    let f x = nameof x

printfn $"{M.f 12}"
printfn $"{nameof M}"
printfn $"{nameof M.f}"
```

Three final additions are changes to how operators work: the addition of the `nameof<'type-parameter>` form for generic type parameters, and the ability to use `nameof` as a pattern in a pattern match expression.

Taking a name of an operator gives its source string. If you need the compiled form, use the compiled name of an operator:

```
nameof(+) // "+"
nameof op_Addition // "op_Addition"
```

Taking the name of a type parameter requires a slightly different syntax:

```
type C<'TType> =
    member _.TypeName = nameof<'TType>
```

This is similar to the `typeof<'T>` and `typedefof<'T>` operators.

F# 5 also adds support for a `nameof` pattern that can be used in `match` expressions:

```
[<Struct; IsByRefLike>]
type RecordedEvent = { EventType: string; Data: ReadOnlySpan<byte> }

type MyEvent =
    | AData of int
    | BData of string

let deserialize (e: RecordedEvent) : MyEvent =
    match e.EventType with
    | nameof AData -> AData (JsonSerializer.Deserialize<int> e.Data)
    | nameof BData -> BData (JsonSerializer.Deserialize<string> e.Data)
    | t -> failwithf "Invalid EventType: %s" t
```

The preceding code uses 'nameof' instead of the string literal in the match expression.

This feature implements F# RFC FS-1003.

Open type declarations

F# 5 also adds support for open type declarations. An open type declaration is like opening a static class in C#, except with some different syntax and some slightly different behavior to fit F# semantics.

With open type declarations, you can `open` any type to expose static contents inside of it. Additionally, you can `open` F#-defined unions and records to expose their contents. For example, this can be useful if you have a union defined in a module and want to access its cases, but don't want to open the entire module.

```
open type System.Math

let x = Min(1.0, 2.0)

module M =
    type DU = A | B | C

    let someOtherFunction x = x + 1

// Open only the type inside the module
open type M.DU

printfn $"{A}"
```

Unlike C#, when you `open type` on two types that expose a member with the same name, the member from the last type being `open`ed shadows the other name. This is consistent with F# semantics around shadowing that exist already.

This feature implements F# RFC FS-1068.

Consistent slicing behavior for built-in data types

Behavior for slicing the built-in `FSharp.Core` data types (array, list, string, 2D array, 3D array, 4D array) used to not be consistent prior to F# 5. Some edge-case behavior threw an exception and some wouldn't. In F# 5, all built-in types now return empty slices for slices that are impossible to generate:

```
let l = [ 1..10 ]
let a = [| 1..10 |]
let s = "hello!"

// Before: would return empty list
// F# 5: same
let emptyList = l[-2..(-1)]

// Before: would throw exception
// F# 5: returns empty array
let emptyArray = a[-2..(-1)]

// Before: would throw exception
// F# 5: returns empty string
let emptyString = s[-2..(-1)]
```

This feature implements F# RFC FS-1077.

Fixed-index slices for 3D and 4D arrays in FSharp.Core

F# 5 brings support for slicing with a fixed index in the built-in 3D and 4D array types.

To illustrate this, consider the following 3D array:

z = 0

X\Y	0	1
0	0	1
1	2	3

z = 1

X\Y	0	1
0	4	5
1	6	7

What if you wanted to extract the slice `[| 4; 5 |]` from the array? This is now very simple!

```
// First, create a 3D array to slice

let dim = 2
let m = Array3D.zeroCreate<int> dim dim dim

let mutable count = 0

for z in 0..dim-1 do
    for y in 0..dim-1 do
        for x in 0..dim-1 do
            m[x,y,z] <- count
            count <- count + 1

// Now let's get the [4;5] slice!
m[*, 0, 1]
```

This feature implements F# RFC FS-1077b.

F# quotations improvements

F# code quotations now have the ability to retain type constraint information. Consider the following example:

```
open FSharp.Linq.RuntimeHelpers

let eval q = LeafExpressionConverter.EvaluateQuotation q

let inline negate x = -x
// val inline negate: x: ^a ->  ^a when  ^a : (static member ( ~- ) :  ^a ->  ^a)

<@ negate 1.0 @>  |> eval
```

The constraint generated by the `inline` function is retained in the code quotation. The `negate` function's quoted form can now be evaluated.

This feature implements F# RFC FS-1071.

Applicative Computation Expressions

Computation expressions (CEs) are used today to model "contextual computations", or in more functional programming-friendly terminology, monadic computations.

F# 5 introduces applicative CEs, which offer a different computational model. Applicative CEs allow for more efficient computations provided that every computation is independent, and their results are accumulated at the end. When computations are independent of one another, they are also trivially parallelizable, allowing CE authors to write more efficient libraries. This benefit comes at a restriction, though: computations that depend on previously computed values are not allowed.

The follow example shows a basic applicative CE for the `Result` type.

```
// First, define a 'zip' function
module Result =
    let zip x1 x2 =
        match x1,x2 with
        | Ok x1res, Ok x2res -> Ok (x1res, x2res)
        | Error e, _ -> Error e
        | _, Error e -> Error e

// Next, define a builder with 'MergeSources' and 'BindReturn'
type ResultBuilder() =
    member _.MergeSources(t1: Result<'T,'U>, t2: Result<'T1,'U>) = Result.zip t1 t2
    member _.BindReturn(x: Result<'T,'U>, f) = Result.map f x

let result = ResultBuilder()

let run r1 r2 r3 =
    // And here is our applicative!
    let res1: Result<int, string> =
        result {
            let! a = r1
            and! b = r2
            and! c = r3
            return a + b - c
        }

    match res1 with
    | Ok x -> printfn $"{nameof res1} is: %d{x}"
    | Error e -> printfn $"{nameof res1} is: {e}"

let printApplicatives () =
    let r1 = Ok 2
    let r2 = Ok 3 // Error "fail!"
    let r3 = Ok 4

    run r1 r2 r3
    run r1 (Error "failure!") r3
```

If you're a library author who exposes CEs in their library today, there are some additional considerations you'll need to be aware of.

This feature implements F# RFC FS-1063.

Interfaces can be implemented at different generic instantiations

You can now implement the same interface at different generic instantiations:

```fsharp
type IA<'T> =
    abstract member Get : unit -> 'T

type MyClass() =
    interface IA<int> with
        member x.Get() = 1
    interface IA<string> with
        member x.Get() = "hello"

let mc = MyClass()
let iaInt = mc :> IA<int>
let iaString = mc :> IA<string>

iaInt.Get() // 1
iaString.Get() // "hello"
```

This feature implements F# RFC FS-1031.

Default interface member consumption

F# 5 lets you consume interfaces with default implementations.

Consider an interface defined in C# like this:

```csharp
using System;

namespace CSharp
{
    public interface MyDim
    {
        public int Z => 0;
    }
}
```

You can consume it in F# through any of the standard means of implementing an interface:

```fsharp
open CSharp

// You can implement the interface via a class
type MyType() =
    member _.M() = ()

    interface MyDim

let md = MyType() :> MyDim
printfn $"DIM from C#: %d{md.Z}"

// You can also implement it via an object expression
let md' = { new MyDim }
printfn $"DIM from C# but via Object Expression: %d{md'.Z}"
```

This lets you safely take advantage of C# code and .NET components written in modern C# when they expect users to be able to consume a default implementation.

This feature implements F# RFC FS-1074.

Simplified interop with nullable value types

Nullable (value) types (called Nullable Types historically) have long been supported by F#, but interacting with them has traditionally been somewhat of a pain since you'd have to construct a `Nullable` or

`Nullable<SomeType>` wrapper every time you wanted to pass a value. Now the compiler will implicitly convert a value type into a `Nullable<ThatValueType>` if the target type matches. The following code is now possible:

```
#r "nuget: Microsoft.Data.Analysis"

open Microsoft.Data.Analysis

let dateTimes = PrimitiveDataFrameColumn<DateTime>("DateTimes")

// The following line used to fail to compile
dateTimes.Append(DateTime.Parse("2019/01/01"))

// The previous line is now equivalent to this line
dateTimes.Append(Nullable<DateTime>(DateTime.Parse("2019/01/01")))
```

This feature implements F# RFC FS-1075.

Preview: reverse indexes

F# 5 also introduces a preview for allowing reverse indexes. The syntax is `^idx`. Here's how you can an element 1 value from the end of a list:

```
let xs = [1..10]

// Get element 1 from the end:
xs[^1]

// From the end slices

let lastTwoOldStyle = xs[(xs.Length-2)..]

let lastTwoNewStyle = xs[^1..]

lastTwoOldStyle = lastTwoNewStyle // true
```

You can also define reverse indexes for your own types. To do so, you'll need to implement the following method:

```
GetReverseIndex: dimension: int -> offset: int
```

Here's an example for the `Span<'T>` type:

```
open System

type Span<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

    member sp.GetReverseIndex(_, offset: int) =
        sp.Length - offset

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn $"{arr}"

let run () =
    let sp = [| 1; 2; 3; 4; 5 |].AsSpan()

    // Pre-# 5.0 slicing on a Span<'T>
    printSpan sp[0..] // [|1; 2; 3; 4; 5|]
    printSpan sp[..3] // [|1; 2; 3|]
    printSpan sp[1..3] // |2; 3|]

    // Same slices, but only using from-the-end index
    printSpan sp[..^0] // [|1; 2; 3; 4; 5|]
    printSpan sp[..^2] // [|1; 2; 3|]
    printSpan sp[^4..^2] // [|2; 3|]

run() // Prints the same thing twice
```

This feature implements F# RFC FS-1076.

Preview: overloads of custom keywords in computation expressions

Computation expressions are a powerful feature for library and framework authors. They allow you to greatly improve the expressiveness of your components by letting you define well-known members and form a DSL for the domain you're working in.

F# 5 adds preview support for overloading custom operations in Computation Expressions. It allows the following code to be written and consumed:

```
open System

type InputKind =
    | Text of placeholder:string option
    | Password of placeholder: string option

type InputOptions =
  { Label: string option
    Kind : InputKind
    Validators : (string -> bool) array }

type InputBuilder() =
    member t.Yield(_) =
      { Label = None
        Kind = Text None
        Validators = [||] }

    [<CustomOperation("text")>]
    member this.Text(io, ?placeholder) =
        { io with Kind = Text placeholder }

    [<CustomOperation("password")>]
    member this.Password(io, ?placeholder) =
        { io with Kind = Password placeholder }

    [<CustomOperation("label")>]
    member this.Label(io, label) =
        { io with Label = Some label }

    [<CustomOperation("with_validators")>]
    member this.Validators(io, [<ParamArray>] validators) =
        { io with Validators = validators }

let input = InputBuilder()

let name =
    input {
    label "Name"
    text
    with_validators
        (String.IsNullOrWhiteSpace >> not)
    }

let email =
    input {
    label "Email"
    text "Your email"
    with_validators
        (String.IsNullOrWhiteSpace >> not)
        (fun s -> s.Contains "@")
    }

let password =
    input {
    label "Password"
    password "Must contains at least 6 characters, one number and one uppercase"
    with_validators
        (String.exists Char.IsUpper)
        (String.exists Char.IsDigit)
        (fun s -> s.Length >= 6)
    }
```

Prior to this change, you could write the `InputBuilder` type as it is, but you couldn't use it the way it's used in the example. Since overloads, optional parameters, and now `System.ParamArray` types are allowed, everything just works as you'd expect it to.

This feature implements F# RFC FS-1056.

What's new in F# 4.7

8/5/2022 • 2 minutes to read • Edit Online

F# 4.7 adds multiple improvements to the F# language.

Get started

F# 4.7 is available in all .NET Core distributions and Visual Studio tooling. Get started with F# to learn more.

Language version

The F# 4.7 compiler introduces the ability to set your effective language version via a property in your project file:

```
<PropertyGroup>
    <LangVersion>preview</LangVersion>
</PropertyGroup>
```

You can set it to the values `4.6`, `4.7`, `latest`, and `preview`. The default is `latest`.

If you set it to `preview`, your compiler will activate all F# preview features that are implemented in your compiler.

Implicit yields

You no longer need to apply the `yield` keyword in arrays, lists, sequences, or computation expressions where the type can be inferred. In the following example, both expressions required the `yield` statement for each entry prior to F# 4.7:

```
let s = seq { 1; 2; 3; 4; 5 }

let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]
```

If you introduce a single `yield` keyword, every other item must also have `yield` applied to it.

Implicit yields are not activated when used in an expression that also uses `yield!` to do something like flatten a sequence. You must continue to use `yield` today in these cases.

Wildcard identifiers

In F# code involving classes, the self-identifier needs to always be explicit in member declarations. But in cases where the self-identifier is never used, it has traditionally been convention to use a double-underscore to indicate a nameless self-identifiers. You can now use a single underscore:

```
type C() =
    member _.M() = ()
```

This also applies for `for` loops:

```
for _ in 1..10 do printfn "Hello!"
```

Indentation relaxations

Prior to F# 4.7, the indentation requirements for primary constructor and static member arguments required excessive indentation. They now only require a single indentation scope:

```
type OffsideCheck(a:int,
    b:int, c:int,
    d:int) = class end

type C() =
    static member M(a:int,
        b:int, c:int,
        d:int) = 1
```

What's new in F# 4.6

8/5/2022 • 2 minutes to read • Edit Online

F# 4.6 adds multiple improvements to the F# language.

Get started

F# 4.6 is available in all .NET Core distributions and Visual Studio tooling. Get started with F# to learn more.

Anonymous records

Anonymous records are a new kind of F# type introduced in F# 4.6. They are simple aggregates of named values that don't need to be declared before use. You can declare them as either structs or reference types. They're reference types by default.

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

They can also be declared as structs for when you want to group value types and are operating in performance-sensitive scenarios:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    struct {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

They're quite powerful and can be used in numerous scenarios. Learn more at Anonymous records.

ValueOption functions

The ValueOption type added in F# 4.5 now has "module-bound function parity" with the Option type. Some of the more commonly-used examples are as follows:

```
// Multiply a value option by 2 if it has  value
let xOpt = ValueSome 99
let result = xOpt |> ValueOption.map (fun v -> v * 2)

// Reverse a string if it exists
let strOpt = ValueSome "Mirror image"
let reverse (str: string) =
    match str with
    | null
    | "" -> ValueNone
    | s ->
        str.ToCharArray()
        |> Array.rev
        |> string
        |> ValueSome

let reversedString = strOpt |> ValueOption.bind reverse
```

This allows for ValueOption to be used just like Option in scenarios where having a value type improves performance.

What's new in F# 4.5

8/5/2022 • 3 minutes to read • Edit Online

F# 4.5 adds multiple improvements to the F# language. Many of these features were added together to enable you to write efficient code in F# while also ensuring this code is safe. Doing so means adding a few concepts to the language and a significant amount of compiler analysis when using these constructs.

Get started

F# 4.5 is available in all .NET Core distributions and Visual Studio tooling. Get started with F# to learn more.

Span and byref-like structs

The Span<T> type introduced in .NET Core allows you to represent buffers in memory in a strongly typed manner, which is now allowed in F# starting with F# 4.5. The following example shows how you can re-use a function operating on a Span<T> with different buffer representations:

```
let safeSum (bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes[i]
    sum

// managed memory
let arrayMemory = Array.zeroCreate<byte>(100)
let arraySpan = new Span<byte>(arrayMemory)

safeSum(arraySpan) |> printfn "res = %d"

// native memory
let nativeMemory = Marshal.AllocHGlobal(100);
let nativeSpan = new Span<byte>(nativeMemory.ToPointer(), 100)

safeSum(nativeSpan) |> printfn "res = %d"
Marshal.FreeHGlobal(nativeMemory)

// stack memory
let mem = NativePtr.stackalloc<byte>(100)
let mem2 = mem |> NativePtr.toVoidPtr
let stackSpan = Span<byte>(mem2, 100)

safeSum(stackSpan) |> printfn "res = %d"
```

An important aspect to this is that Span and other byref-like structs have very rigid static analysis performed by the compiler that restrict their usage in ways you might find to be unexpected. This is the fundamental tradeoff between performance, expressiveness, and safety that is introduced in F# 4.5.

Revamped byrefs

Prior to F# 4.5, Byrefs in F# were unsafe and unsound for numerous applications. Soundness issues around byrefs have been addressed in F# 4.5 and the same static analysis done for span and byref-like structs was also applied.

inref<'T> and outref<'T>

To represent the notion of a read-only, write-only, and read/write managed pointer, F# 4.5 introduces the `inref<'T>`, `outref<'T>` types to represent read-only and write-only pointers, respectively. Each have different semantics. For example, you cannot write to an `inref<'T>`:

```
let f (dt: inref<DateTime>) =
    dt <- DateTime.Now // ERROR - cannot write to an inref!
```

By default, type inference will infer managed pointers as `inref<'T>` to be in line with the immutable nature of F# code, unless something has already been declared as mutable. To make something writable, you'll need to declare a type as `mutable` before passing its address to a function or member that manipulates it. To learn more, see Byrefs.

Readonly structs

Starting with F# 4.5, you can annotate a struct with IsReadOnlyAttribute as such:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

This disallows you from declaring a mutable member in the struct and emits metadata that allows F# and C# to treat it as readonly when consumed from an assembly. To learn more, see ReadOnly structs.

Void pointers

The `voidptr` type is added to F# 4.5, as are the following functions:

- `NativePtr.ofVoidPtr` to convert a void pointer into a native int pointer
- `NativePtr.toVoidPtr` to convert a native int pointer to a void pointer

This is helpful when interoperating with a native component that makes use of void pointers.

The `match!` keyword

The `match!` keyword enhances pattern matching when inside a computation expression:

```
// Code that returns an asynchronous option
let checkBananaAsync (s: string) =
    async {
        if s = "banana" then
            return Some s
        else
            return None
    }

// Now you can use 'match!'
let funcWithString (s: string) =
    async {
        match! checkBananaAsync s with
        | Some bananaString -> printfn "It's banana!"
        | None -> printfn "%s" s
}
```

This allows you to shorten code that often involves mixing options (or other types) with computation expressions such as async. To learn more, see match!.

Relaxed upcasting requirements in array, list, and sequence expressions

Mixing types where one may inherit from another inside of array, list, and sequence expressions has traditionally required you to upcast any derived type to its parent type with `:>` or `upcast`. This is now relaxed, demonstrated as follows:

```
let x0 : obj list  = [ "a" ] // ok pre-F# 4.5
let x1 : obj list  = [ "a"; "b" ] // ok pre-F# 4.5
let x2 : obj list  = [ yield "a" :> obj ] // ok pre-F# 4.5

let x3 : obj list  = [ yield "a" ] // Now ok for F# 4.5, and can replace x2
```

Indentation relaxation for array and list expressions

Prior to F# 4.5, you needed to excessively indent array and list expressions when passed as arguments to method calls. This is no longer required:

```
module NoExcessiveIndenting =
    System.Console.WriteLine(format="{0}", arg = [|
        "hello"
    |])
    System.Console.WriteLine([|
        "hello"
    |])
```

F# Development Tools

8/5/2022 • 2 minutes to read • Edit Online

This article describes some of the primary development tools used with F#.

.NET Command-line Tools

You can install command-line tools for F# in multiple ways, depending on your environment. See Install F#.

Integrated Development Environments (IDEs)

F# with Visual Studio

F# can be installed as part of Visual Studio. See Getting Started with F# in Visual Studio.

F# with Visual Studio Code

F# can be installed as part of Visual Studio Code. See Getting Started with F# in Visual Studio Code.

F# with Visual Studio for Mac

F# can be installed as part of Visual Studio for Mac. See Getting Started with F# in Visual Studio for Mac.

Other development environments

Other IDEs are available for F#, see F# Tools

Community Tools

Many tools and libraries for F# are provided by the F# community. These include:

- Fantomas - The F# code formatting tool
- FSharpLint - An F# code checking tool
- FAKE - An F# build automation tool

For more comprehensive lists, see the F# Software Foundation's Guide to F# Community Projects, or search on the web.

Interactive programming with F#

8/5/2022 • 8 minutes to read • Edit Online

F# Interactive (dotnet fsi) is used to run F# code interactively at the console, or to execute F# scripts. In other words, F# interactive executes a REPL (Read, Evaluate, Print Loop) for F#.

To run F# Interactive from the console, run `dotnet fsi`. You will find `dotnet fsi` in any .NET SDK.

For information about available command-line options, see F# Interactive Options.

Executing code directly in F# Interactive

Because F# Interactive is a REPL (read-eval-print loop), you can execute code interactively in it. Here is an example of an interactive session after executing `dotnet fsi` from the command line:

```
Microsoft (R) F# Interactive version 11.0.0.0 for F# 5.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> let square x = x *  x;;
val square : x:int -> int

> square 12;;
val it : int = 144

> printfn "Hello, FSI!"
- ;;
Hello, FSI!
val it : unit = ()
```

You'll notice two main things:

1. All code must be terminated with a double semicolon (`;;`) to be evaluated
2. Code is evaluated and stored in an `it` value. You can reference `it` interactively.

F# Interactive also supports multi-line input. You just need to terminate your submission with a double semicolon (`;;`). Consider the following snippet that has been pasted into and evaluated by F# Interactive:

```
> let getOddSquares xs =
-     xs
-     |> List.filter (fun x -> x % 2 <> 0)
-     |> List.map (fun x -> x * x)
-
- printfn "%A" (getOddSquares [1..10]);;
[1; 9; 25; 49; 81]
val getOddSquares : xs:int list -> int list
val it : unit = ()

>
```

The code's formatting is preserved, and there is a double semicolon (`;;`) terminating the input. F# Interactive then evaluated the code and printed the results!

Scripting with F#

Evaluating code interactively in F# Interactive can be a great learning tool, but you'll quickly find that it's not as productive as writing code in a normal editor. To support normal code editing, you can write F# scripts.

Scripts use the file extension **.fsx**. Instead of compiling source code and then later running the compiled assembly, you can just run **dotnet fsi** and specify the filename of the script of F# source code, and F# interactive reads the code and executes it in real time. For example, consider the following script called `Script.fsx`:

```
let getOddSquares xs =
    xs
    |> List.filter (fun x -> x % 2 <> 0)
    |> List.map (fun x -> x * x)

printfn "%A" (getOddSquares [1..10])
```

When this file is created in your machine, you can run it with `dotnet fsi` and see the output directly in your terminal window:

```
dotnet fsi Script.fsx
[1; 9; 25; 49; 81]
```

F# scripting is natively supported in Visual Studio, Visual Studio Code, and Visual Studio for Mac.

Referencing packages in F# Interactive

> **NOTE**
>
> Package management system is extensible.

F# Interactive supports referencing NuGet packages with the `#r "nuget:"` syntax and an optional version:

```
#r "nuget: Newtonsoft.Json"
open Newtonsoft.Json

let data = {| Name = "Don Syme"; Occupation = "F# Creator" |}
JsonConvert.SerializeObject(data)
```

If a version is not specified, the highest available non-preview package is taken. To reference a specific version, introduce the version via a comma. This can be handy when referencing a preview version of a package. For example, consider this script using a preview version of DiffSharp:

```
#r "nuget: DiffSharp-lite, 1.0.0-preview-328097867"
open DiffSharp

// A 1D tensor
let t1 = dsharp.tensor [ 0.0 .. 0.2 .. 1.0 ]

// A 2x2 tensor
let t2 = dsharp.tensor [ [ 0; 1 ]; [ 2; 2 ] ]

// Define a scalar-to-scalar function
let f (x: Tensor) = sin (sqrt x)

printfn $"{f (dsharp.tensor 1.2)}"
```

Specifying a package source

You can also specify a package source with the `#i` command. The following example specifies a remote and a local source:

```
#i "nuget: https://my-remote-package-source/index.json"
#i """nuget: C:\path\to\my\local\source"""
```

This will tell the resolution engine under the covers to also take into account the remote and/or local sources added to a script.

You can specify as many package references as you like in a script.

> **NOTE**
>
> There's currently a limitation for scripts that use framework references (e.g. `Microsoft.NET.Sdk.Web` or `Microsoft.NET.Sdk.WindowsDesktop`). Packages like Saturn, Giraffe, WinForms are not available. This is being tracked in issue #9417.

Referencing assemblies on disk with F# interactive

Alternatively, if you have an assembly on disk and wish to reference that in a script, you can use the `#r` syntax to specify an assembly. Consider the following code in a project compiled into `MyAssembly.dll`:

```
// MyAssembly.fs
module MyAssembly
let myFunction x y = x + 2 * y
```

Once compiled, you can reference it in a file called `Script.fsx` like so:

```
#r "path/to/MyAssembly.dll"

printfn $"{MyAssembly.myFunction 10 40}"
```

The output is as follows:

```
dotnet fsi Script.fsx
90
```

You can specify as many assembly references as you like in a script.

Loading other scripts

When scripting, it can often be helpful to use different scripts for different tasks. Sometimes you may want to reuse code from one script in another. Rather than copy-pasting its contents into your file, you can simply load and evaluate it with `#load`.

Consider the following `Script1.fsx`:

```
let square x = x * x
```

And the consuming file, `Script2.fsx`:

```
#load "Script1.fsx"
open Script1

printfn $"%d{square 12}"
```

You can evaluate `Script2.fsx` like so:

```
dotnet fsi Script2.fsx
144
```

You can specify as many `#load` directives as you like in a script.

> **NOTE**
>
> The `open Script1` declaration is required. This is because constructs in an F# script are compiled into a top-level module that is the name of the script file it is in. If the script file has a lowercase name such as `script3.fsx` then the implied module name is automatically capitalized, and you will need to use `open Script3`. If you would like a loadable-script to define constructs in a specific namespace of module you can include a namespace of module declaration, for example:
>
> ```
> module MyScriptLibrary
> ```

Using the `fsi` object in F# code

F# scripts have access to a custom `fsi` object that represents the F# Interactive session. It allows you to customize things like output formatting. It is also how you can access command-line arguments.

The following example shows how to get and use command-line arguments:

```
let args = fsi.CommandLineArgs

for arg in args do
    printfn $"{arg}"
```

When evaluated, it prints all arguments. The first argument is always the name of the script that is evaluated:

```
dotnet fsi Script1.fsx hello world from fsi
Script1.fsx
hello
world
from
fsi
```

You can also use `System.Environment.GetCommandLineArgs()` to access the same arguments.

F# Interactive directive reference

The `#r` and `#load` directives seen previously are only available in F# Interactive. There are several directives only available in F# Interactive:

DIRECTIVE	DESCRIPTION
`#r "nuget:..."`	References a package from NuGet

DIRECTIVE	DESCRIPTION
`#r "assembly-name.dll"`	References an assembly on disk
`#load "file-name.fsx"`	Reads a source file, compiles it, and runs it.
`#help`	Displays information about available directives.
`#I`	Specifies an assembly search path in quotation marks.
`#quit`	Terminates an F# Interactive session.
`#time "on"` or `#time "off"`	By itself, `#time` toggles whether to display performance information. When it is `"on"`, F# Interactive measures real time, CPU time, and garbage collection information for each section of code that is interpreted and executed.

When you specify files or paths in F# Interactive, a string literal is expected. Therefore, files and paths must be in quotation marks, and the usual escape characters apply. You can use the `@` character to cause F# Interactive to interpret a string that contains a path as a verbatim string. This causes F# Interactive to ignore any escape characters.

Interactive and compiled preprocessor directives

When you compile code in F# Interactive, whether you are running interactively or running a script, the symbol **INTERACTIVE** is defined. When you compile code in the compiler, the symbol **COMPILED** is defined. Thus, if code needs to be different in compiled and interactive modes, you can use these preprocessor directives for conditional compilation to determine which to use. For example:

```
#if INTERACTIVE
// Some code that executes only in FSI
// ...
#endif
```

Using F# Interactive in Visual Studio

To run F# Interactive through Visual Studio, you can click the appropriate toolbar button labeled **F# Interactive**, or use the keys **Ctrl+Alt+F**. Doing this will open the interactive window, a tool window running an F# Interactive session. You can also select some code that you want to run in the interactive window and hit the key combination **Alt+Enter**. F# Interactive starts in a tool window labeled **F# Interactive**. When you use this key combination, make sure that the editor window has the focus.

Whether you are using the console or Visual Studio, a command prompt appears and the interpreter awaits your input. You can enter code just as you would in a code file. To compile and execute the code, enter two semicolons (;;) to terminate a line or several lines of input.

F# Interactive attempts to compile the code and, if successful, it executes the code and prints the signature of the types and values that it compiled. If errors occur, the interpreter prints the error messages.

Code entered in the same session has access to any constructs entered previously, so you can build up programs. An extensive buffer in the tool window allows you to copy the code into a file if needed.

When run in Visual Studio, F# Interactive runs independently of your project, so, for example, you cannot use constructs defined in your project in F# Interactive unless you copy the code for the function into the interactive window.

You can control the F# Interactive command-line arguments (options) by adjusting the settings. On the **Tools** menu, select **Options...**, and then expand **F# Tools**. The two settings that you can change are the F# Interactive options and the **64-bit F# Interactive** setting, which is relevant only if you are running F# Interactive on a 64-bit machine. This setting determines whether you want to run the dedicated 64-bit version of **fsi.exe** or **fsianycpu.exe**, which uses the machine architecture to determine whether to run as a 32-bit or 64-bit process.

Related articles

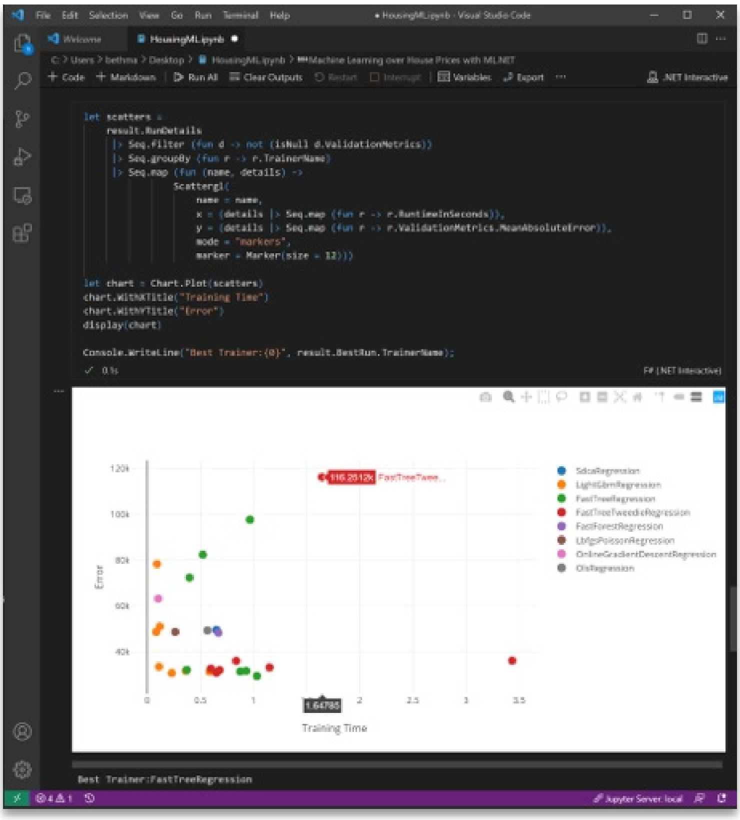
TITLE	DESCRIPTION
F# Interactive Options	Describes command-line syntax and options for the F# Interactive, fsi.exe.

F# notebooks

8/5/2022 • 2 minutes to read • Edit Online

F# is well suited to notebook programming because of its ordered declarations and scripting constructs.

.NET Interactive F# notebooks can be used with Jupyter, Visual Studio Code, and Visual Studio.



See also

- Machine Learning with F#
- .NET Interactive
- A Guide to Data Access with F#
- A Guide to Data Science with F#

F# for JavaScript

8/5/2022 • 2 minutes to read • Edit Online

F# can execute as JavaScript code through two community-provided open source toolchains. This allows F# code to be used for client-side and full-stack web development.

Fable

Fable is a compiler that brings F# into the JavaScript ecosystem. It generates modern JavaScript output, interoperates with JavaScript packages, and supports multiple development models including React.

WebSharper

WebSharper - Provides full-stack, functional reactive web programming for .NET, allowing you to develop microservices, client-server web applications, reactive SPAs, and more in C# or F#.

See also

- F# for Web Development
- A Guide to Web Programming with F#

F# style guide

8/5/2022 • 2 minutes to read • Edit Online

The following articles describe guidelines for formatting F# code and topical guidance for features of the language and how they should be used.

This guidance has been formulated based on the use of F# in large codebases with a diverse group of programmers. This guidance generally leads to successful use of F# and minimizes frustrations when requirements for programs change over time.

Five principles of good F# code

Keep the following principles in mind any time you write F# code, especially in systems that will change over time. Every piece of guidance in further articles stems from these five points.

1. **Good F# code is succinct, expressive, and composable**

   F# has many features that allow you to express actions in fewer lines of code and reuse generic functionality. The F# core library also contains many useful types and functions for working with common collections of data. Composition of your own functions and those in the F# core library (or other libraries) is a part of routine idiomatic F# programming. As a general rule, if you can express a solution to a problem in fewer lines of code, other developers (or your future self) will be appreciative. It's also highly recommended that you use a library such as FSharp.Core, the vast .NET libraries that F# runs on, or a third-party package on NuGet when you need to do a nontrivial task.

2. **Good F# code is interoperable**

   Interoperation can take multiple forms, including consuming code in different languages. The boundaries of your code that other callers interoperate with are critical pieces to get right, even if the callers are also in F#. When writing F#, you should always be thinking about how other code will call into the code you're writing, including if they do so from another language like C#. The F# Component Design Guidelines describe interoperability in detail.

3. **Good F# code makes use of object programming, not object orientation**

   F# has full support for programming with objects in .NET, including classes, interfaces, access modifiers, abstract classes, and so on. For more complicated functional code, such as functions that must be context-aware, objects can easily encapsulate contextual information in ways that functions cannot. Features such as optional parameters and careful use of overloading can make consumption of this functionality easier for callers.

4. **Good F# code performs well without exposing mutation**

   It's no secret that to write high-performance code, you must use mutation. It's how computers work, after all. Such code is often error-prone and difficult to get right. Avoid exposing mutation to callers. Instead, build a functional interface that hides a mutation-based implementation when performance is critical.

5. **Good F# code is toolable**

   Tools are invaluable for working in large codebases, and you can write F# code such that it can be used more effectively with F# language tooling. One example is making sure you don't overdo it with a point-free style of programming, so that intermediate values can be inspected with a debugger. Another example is using XML documentation comments describing constructs such that tooltips in editors can display those comments at the call site. Always think about how your code will be read, navigated, debugged, and manipulated by other programmers with their tools.

Next steps

The F# code formatting guidelines provide guidance on how to format code so that it is easy to read.

The F# coding conventions provide guidance for F# programming idioms that will help the long-term maintenance of larger F# codebases.

The F# component design guidelines provide guidance for authoring F# components, such as libraries.

F# code formatting guidelines

8/5/2022 • 35 minutes to read • Edit Online

This article offers guidelines for how to format your code so that your F# code is:

- More legible
- In accordance with conventions applied by formatting tools in Visual Studio Code and other editors
- Similar to other code online

See also Coding conventions and Component design guidelines, which also covers naming conventions.

Automatic code formatting

The Fantomas code formatter is the F# community standard tool for automatic code formatting. The default settings correspond to this style guide.

We strongly recommend the use of this code formatter. Within F# teams, code formatting specifications should be agreed and codified in terms of an agreed settings file for the code formatter checked into the team repository.

General rules for formatting

F# uses significant white space by default and is white space sensitive. The following guidelines are intended to provide guidance as to how to juggle some challenges this can impose.

Use spaces not tabs

When indentation is required, you must use spaces, not tabs. F# code doesn't use tabs, and the compiler will give an error if a tab character is encountered outside a string literal or comment.

Use consistent indentation

When indenting, at least one space is required. Your organization can create coding standards to specify the number of spaces to use for indentation; two, three, or four spaces of indentation at each level where indentation occurs is typical.

We recommend four spaces per indentation.

That said, indentation of programs is a subjective matter. Variations are OK, but the first rule you should follow is *consistency of indentation*. Choose a generally accepted style of indentation and use it systematically throughout your codebase.

Avoid formatting that is sensitive to name length

Seek to avoid indentation and alignment that is sensitive to naming:

```
// ✔️ OK
let myLongValueName =
    someExpression
    |> anotherExpression

// ❌ Not OK
let myLongValueName = someExpression
                      |> anotherExpression

// ✔️ OK
let myOtherVeryLongValueName =
    match
        someVeryLongExpressionWithManyParameters
            parameter1
            parameter2
            parameter3
        with
    | Some _ -> ()
    | ...

// ❌ Not OK
let myOtherVeryLongValueName =
    match someVeryLongExpressionWithManyParameters parameter1
                                                   parameter2
                                                   parameter3 with
    | Some _ -> ()
    | ...

// ❌ Still Not OK
let myOtherVeryLongValueName =
    match someVeryLongExpressionWithManyParameters
              parameter1
              parameter2
              parameter3 with
    | Some _ -> ()
    | ...
```

The primary reasons for avoiding this are:

- Important code is moved far to the right
- There's less width left for the actual code
- Renaming can break the alignment

Avoid extraneous white space

Avoid extraneous white space in F# code, except where described in this style guide.

```
// ✔️ OK
spam (ham 1)

// ❌ Not OK
spam ( ham 1 )
```

Formatting comments

Prefer multiple double-slash comments over block comments.

```
// Prefer this style of comments when you want
// to express written ideas on multiple lines.

(*
    Block comments can be used, but use sparingly.
    They are useful when eliding code sections.
*)
```

Comments should capitalize the first letter and be well-formed phrases or sentences.

```
// ✔️ A good comment.
let f x = x + 1 // Increment by one.

// ❌ two poor comments
let f x = x + 1 // plus one
```

For formatting XML doc comments, see "Formatting declarations" below.

Formatting expressions

This section discusses formatting expressions of different kinds.

Formatting string expressions

String literals and interpolated strings can just be left on a single line, regardless of how long the line is.

```
let serviceStorageConnection =

$"DefaultEndpointsProtocol=https;AccountName=%s{serviceStorageAccount.Name};AccountKey=%s{serviceStorageAcco
untKey.Value}"
```

Multi-line interpolated expressions are discouraged. Instead, bind the expression result to a value and use that in the interpolated string.

Formatting tuple expressions

A tuple instantiation should be parenthesized, and the delimiting commas within it should be followed by a single space, for example: `(1, 2)`, `(x, y, z)`.

```
// ✔️ OK
let pair = (1, 2)
let triples = [ (1, 2, 3); (11, 12, 13) ]
```

It's commonly accepted to omit parentheses in pattern matching of tuples:

```
// ✔️ OK
let (x, y) = z
let x, y = z

// ✔️ OK
match x, y with
| 1, _ -> 0
| x, 1 -> 0
| x, y -> 1
```

It's also commonly accepted to omit parentheses if the tuple is the return value of a function:

```
// ✔️ OK
let update model msg =
    match msg with
    | 1 -> model + 1, []
    | _ -> model, [ msg ]
```

In summary, prefer parenthesized tuple instantiations, but when using tuples for pattern matching or a return value, it's considered fine to avoid parentheses.

Formatting application expressions

When formatting a function or method application, arguments are provided on the same line when line-width allows:

```
// ✔️ OK
someFunction1 x.IngredientName x.Quantity
```

Omit parentheses unless arguments require them:

```
// ✔️ OK
someFunction1 x.IngredientName

// ❌ Not preferred - parentheses should be omitted unless required
someFunction1 (x.IngredientName)

// ✔️ OK - parentheses are required
someFunction1 (convertVolumeToLiter x)
```

Don't omit spaces when invoking with multiple curried arguments:

```
// ✔️ OK
someFunction1 (convertVolumeToLiter x) (convertVolumeUSPint x)
someFunction2 (convertVolumeToLiter y) y
someFunction3 z (convertVolumeUSPint z)

// ❌ Not preferred - spaces should not be omitted between arguments
someFunction1(convertVolumeToLiter x)(convertVolumeUSPint x)
someFunction2(convertVolumeToLiter y) y
someFunction3 z(convertVolumeUSPint z)
```

In default formatting conventions, a space is added when applying lower-case functions to tupled or parenthesized arguments (even when a single argument is used):

```
// ✔️ OK
someFunction2 ()

// ✔️ OK
someFunction3 (x.Quantity1 + x.Quantity2)

// ❌ Not OK, formatting tools will add the extra space by default
someFunction2()

// ❌ Not OK, formatting tools will add the extra space by default
someFunction3(x.IngredientName, x.Quantity)
```

In default formatting conventions, no space is added when applying capitalized methods to tupled arguments. This is because these are often used with fluent programming:

```
// ✔️ OK - Methods accepting parenthesize arguments are applied without a space
SomeClass.Invoke()

// ✔️ OK - Methods accepting tuples are applied without a space
String.Format(x.IngredientName, x.Quantity)

// ❌ Not OK, formatting tools will remove the extra space by default
SomeClass.Invoke ()

// ❌ Not OK, formatting tools will remove the extra space by default
String.Format (x.IngredientName, x.Quantity)
```

You may need to pass arguments to a function on a new line as a matter of readability or because the list of arguments or the argument names are too long. In that case, indent one level:

```
// ✔️ OK
someFunction2
    x.IngredientName x.Quantity

// ✔️ OK
someFunction3
    x.IngredientName1 x.Quantity2
    x.IngredientName2 x.Quantity2

// ✔️ OK
someFunction4
    x.IngredientName1
    x.Quantity2
    x.IngredientName2
    x.Quantity2

// ✔️ OK
someFunction5
    (convertVolumeToLiter x)
    (convertVolumeUSPint x)
    (convertVolumeImperialPint x)
```

When the function takes a single multi-line tupled argument, place each argument on a new line:

```
// ✔️ OK
someTupledFunction (
    478815516,
    "A very long string making all of this multi-line",
    1515,
    false
)

// OK, but formatting tools will reformat to the above
someTupledFunction
    (478815516,
     "A very long string making all of this multi-line",
     1515,
     false)
```

If argument expressions are short, separate arguments with spaces and keep it in one line.

```
// ✔️ OK
let person = new Person(a1, a2)

// ✔️ OK
let myRegexMatch = Regex.Match(input, regex)

// ✔️ OK
let untypedRes = checker.ParseFile(file, source, opts)
```

If argument expressions are long, use newlines and indent one level, rather than indenting to the left-parenthesis.

```
// ✔️ OK
let person =
    new Person(
        argument1,
        argument2
    )

// ✔️ OK
let myRegexMatch =
    Regex.Match(
        "my longer input string with some interesting content in it",
        "myRegexPattern"
    )

// ✔️ OK
let untypedRes =
    checker.ParseFile(
        fileName,
        sourceText,
        parsingOptionsWithDefines
    )

// ❌ Not OK, formatting tools will reformat to the above
let person =
    new Person(argument1,
               argument2)

// ❌ Not OK, formatting tools will reformat to the above
let untypedRes =
    checker.ParseFile(fileName,
                      sourceText,
                      parsingOptionsWithDefines)
```

The same rules apply even if there is only a single multi-line argument, including multi-line strings:

```
// ✔️ OK
let poemBuilder = StringBuilder()
poemBuilder.AppendLine(
    """
The last train is nearly due
The Underground is closing soon
And in the dark, deserted station
Restless in anticipation
A man waits in the shadows
    """
)

Option.traverse(
    create
    >> Result.setError [ invalidHeader "Content-Checksum" ]
)
```

Formatting pipeline expressions

When using multiple lines, pipeline `|>` operators should go underneath the expressions they operate on.

```
// ✔️ OK
let methods2 =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun assm -> assm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// ❌ Not OK, add a line break after "=" and put multi-line pipelines on multiple lines.
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
            |> List.ofArray
            |> List.map (fun assm -> assm.GetTypes())
            |> Array.concat
            |> List.ofArray
            |> List.map (fun t -> t.GetMethods())
            |> Array.concat

// ❌ Not OK either
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
               |> List.ofArray
               |> List.map (fun assm -> assm.GetTypes())
               |> Array.concat
               |> List.ofArray
               |> List.map (fun t -> t.GetMethods())
               |> Array.concat
```

Formatting lambda expressions

When a lambda expression is used as an argument in a multi-line expression, and is followed by other arguments, place the body of a lambda expression on a new line, indented by one level:

```
// ✔️ OK
let printListWithOffset a list1 =
    List.iter
        (fun elem ->
             printfn $"A very long line to format the value: %d{a + elem}")
        list1
```

If the lambda argument is the last argument in a function application, place all arguments until the arrow on the same line.

```fsharp
// ✔️ OK
Target.create "Build" (fun ctx ->
    // code
    // here
    ())

// ✔️ OK
let printListWithOffsetPiped a list1 =
    list1
    |> List.map (fun x -> x + 1)
    |> List.iter (fun elem ->
        printfn $"A very long line to format the value: %d{a + elem}")
```

Treat match lambda's in a similar fashion.

```fsharp
// ✔️ OK
functionName arg1 arg2 arg3 (function
    | Choice1of2 x -> 1
    | Choice2of2 y -> 2)
```

When there are many leading or multi-line arguments before the lambda indent all arguments with one level.

```fsharp
// ✔️ OK
functionName
    arg1
    arg2
    arg3
    (fun arg4 ->
        bodyExpr)

// ✔️ OK
functionName
    arg1
    arg2
    arg3
    (function
     | Choice1of2 x -> 1
     | Choice2of2 y -> 2)
```

If the body of a lambda expression is multiple lines long, you should consider refactoring it into a locally scoped function.

When pipelines include lambda expressions, each lambda expression is typically the last argument at each stage of the pipeline:

```fsharp
// ✔️ OK, with 4 spaces indentation
let printListWithOffsetPiped list1 =
    list1
    |> List.map (fun elem -> elem + 1)
    |> List.iter (fun elem ->
        // one indent starting from the pipe
        printfn $"A very long line to format the value: %d{elem}")

// ✔️ OK, with 2 spaces indentation
let printListWithOffsetPiped list1 =
  list1
  |> List.map (fun elem -> elem + 1)
  |> List.iter (fun elem ->
    // one indent starting from the pipe
    printfn $"A very long line to format the value: %d{elem}")
```

Formatting arithmetic and binary expressions

Always use white space around binary arithmetic expressions:

```fsharp
// ✔️ OK
let subtractThenAdd x = x - 1 + 3
```

Failing to surround a binary `-` operator, when combined with certain formatting choices, could lead to interpreting it as a unary `-`. Unary `-` operators should always be immediately followed by the value they negate:

```fsharp
// ✔️ OK
let negate x = -x

// ❌ Not OK
let negateBad x = - x
```

Adding a white-space character after the `-` operator can lead to confusion for others.

Separate binary operators by spaces. Infix expressions are OK to lineup on same column:

```fsharp
// ✔️ OK
let function1 () =
    acc +
    (someFunction
         x.IngredientName x.Quantity)

// ✔️ OK
let function1 arg1 arg2 arg3 arg4 =
    arg1 + arg2 +
    arg3 + arg4
```

This rule also applies to units of measures in types and constant annotations:

```fsharp
// ✔️ OK
type Test =
    { WorkHoursPerWeek: uint<hr / (staff weeks)> }
    static member create = { WorkHoursPerWeek = 40u<hr / (staff weeks)> }

// ❌ Not OK
type Test =
    { WorkHoursPerWeek: uint<hr/(staff weeks)> }
    static member create = { WorkHoursPerWeek = 40u<hr/(staff weeks)> }
```

The following operators are defined in the F# standard library and should be used instead of defining equivalents. Using these operators is recommended as it tends to make code more readable and idiomatic. The following list summarizes the recommended F# operators.

```
// ✔️ OK
x |> f // Forward pipeline
f >> g // Forward composition
x |> ignore // Discard away a value
x + y // Overloaded addition (including string concatenation)
x - y // Overloaded subtraction
x * y // Overloaded multiplication
x / y // Overloaded division
x % y // Overloaded modulus
x && y // Lazy/short-cut "and"
x || y // Lazy/short-cut "or"
x <<< y // Bitwise left shift
x >>> y // Bitwise right shift
x ||| y // Bitwise or, also for working with “flags” enumeration
x &&& y // Bitwise and, also for working with “flags” enumeration
x ^^^ y // Bitwise xor, also for working with “flags” enumeration
```

Formatting range operator expressions

Only add spaces around the `..` when all expressions are non-atomic. Integers and single word identifiers are considered atomic.

```
// ✔️ OK
let a = [ 2..7 ] // integers
let b = [ one..two ] // identifiers
let c = [ ..9 ] // also when there is only one expression
let d = [ 0.7 .. 9.2 ] // doubles
let e = [ 2L .. number / 2L ] // complex expression
let f = [| A.B .. C.D |] // identifiers with dots
let g = [ .. (39 - 3) ] // complex expression
let h = [| 1 .. MyModule.SomeConst |] // not all expressions are atomic

for x in 1..2 do
    printfn " x = %d" x

let s = seq { 0..10..100 }

// ❌ Not OK
let a = [ 2 .. 7 ]
let b = [ one .. two ]
```

These rules also apply to slicing:

```
// ✔️ OK
arr[0..10]
list[..^1]
```

Formatting if expressions

Indentation of conditionals depends on the size and complexity of the expressions that make them up. Write them on one line when:

- `cond`, `e1`, and `e2` are short
- `e1` and `e2` are not `if/then/else` expressions themselves.

```
// ✔️ OK
if cond then e1 else e2
```

If the else expression is absent, it's recommended to never write the entire expression in one line. This is to differentiate the imperative code from the functional.

```
// ✔️ OK
if a then
    ()

// ❌ Not OK, code formatters will reformat to the above by default
if a then ()
```

If any of the expressions are multi-line or `if/then/else` expressions.

```
// ✔️ OK
if cond then
    e1
else
    e2
```

Multiple conditionals with `elif` and `else` are indented at the same scope as the `if` when they follow the rules of the one line `if/then/else` expressions.

```
// ✔️ OK
if cond1 then e1
elif cond2 then e2
elif cond3 then e3
else e4
```

If any of the conditions or expressions is multi-line, the entire `if/then/else` expression is multi-line:

```
// ✔️ OK
if cond1 then
    e1
elif cond2 then
    e2
elif cond3 then
    e3
else
    e4
```

If a condition is multiline or exceeds the default tolerance of the single-line, the condition expression should use one indentation and a new line. The `if` and `then` keyword should align when encapsulating the long condition expression.

```
// ✔️ OK, but better to refactor, see below
if
    complexExpression a b && env.IsDevelopment()
    || someFunctionToCall
        aVeryLongParameterNameOne
        aVeryLongParameterNameTwo
        aVeryLongParameterNameThree
then
        e1
    else
        e2

// ✔️The same applies to nested `elif` or `else if` expressions
if a then
    b
elif
    someLongFunctionCall
        argumentOne
        argumentTwo
        argumentThree
        argumentFour
then
    c
else if
    someOtherLongFunctionCall
        argumentOne
        argumentTwo
        argumentThree
        argumentFour
then
    d
```

It is, however, better style to refactor long conditions to a let binding or separate function:

```
// ✔️ OK
let performAction =
    complexExpression a b && env.IsDevelopment()
    || someFunctionToCall
        aVeryLongParameterNameOne
        aVeryLongParameterNameTwo
        aVeryLongParameterNameThree

if performAction then
    e1
else
    e2
```

Formatting union case expressions

Applying discriminated union cases follows the same rules as function and method applications. That is, because the name is capitalized, code formatters will remove a space before a tuple:

```
// ✔️ OK
let opt = Some("A", 1)

// OK, but code formatters will remove the space
let opt = Some ("A", 1)
```

Like function applications, constructions that split across multiple lines should use indentation:

```
// ✔️ OK
let tree1 =
    BinaryNode(
        BinaryNode (BinaryValue 1, BinaryValue 2),
        BinaryNode (BinaryValue 3, BinaryValue 4)
    )
```

Formatting list and array expressions

Write `x :: l` with spaces around the `::` operator (`::` is an infix operator, hence surrounded by spaces).

List and arrays declared on a single line should have a space after the opening bracket and before the closing bracket:

```
// ✔️ OK
let xs = [ 1; 2; 3 ]

// ✔️ OK
let ys = [| 1; 2; 3; |]
```

Always use at least one space between two distinct brace-like operators. For example, leave a space between a `[` and a `{`.

```
// ✔️ OK
[ { Ingredient = "Green beans"; Quantity = 250 }
  { Ingredient = "Pine nuts"; Quantity = 250 }
  { Ingredient = "Feta cheese"; Quantity = 250 }
  { Ingredient = "Olive oil"; Quantity = 10 }
  { Ingredient = "Lemon"; Quantity = 1 } ]

// ❌ Not OK
[{ Ingredient = "Green beans"; Quantity = 250 }
 { Ingredient = "Pine nuts"; Quantity = 250 }
 { Ingredient = "Feta cheese"; Quantity = 250 }
 { Ingredient = "Olive oil"; Quantity = 10 }
 { Ingredient = "Lemon"; Quantity = 1 }]
```

The same guideline applies for lists or arrays of tuples.

Lists and arrays that split across multiple lines follow a similar rule as records do:

```
// ✔️ OK
let pascalsTriangle =
    [| [| 1 |]
       [| 1; 1 |]
       [| 1; 2; 1 |]
       [| 1; 3; 3; 1 |]
       [| 1; 4; 6; 4; 1 |]
       [| 1; 5; 10; 10; 5; 1 |]
       [| 1; 6; 15; 20; 15; 6; 1 |]
       [| 1; 7; 21; 35; 35; 21; 7; 1 |]
       [| 1; 8; 28; 56; 70; 56; 28; 8; 1 |] |]
```

And as with records, declaring the opening and closing brackets on their own line will make moving code around and piping into functions easier.

When generating arrays and lists programmatically, prefer `->` over `do ... yield` when a value is always generated:

```
// ✔️ OK
let squares = [ for x in 1..10 -> x * x ]

// ❌ Not preferred, use "->" when a value is always generated
let squares' = [ for x in 1..10 do yield x * x ]
```

Older versions of F# required specifying `yield` in situations where data may be generated conditionally, or there may be consecutive expressions to be evaluated. Prefer omitting these `yield` keywords unless you must compile with an older F# language version:

```
// ✔️ OK
let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]

// ❌ Not preferred - omit yield instead
let daysOfWeek' includeWeekend =
    [
        yield "Monday"
        yield "Tuesday"
        yield "Wednesday"
        yield "Thursday"
        yield "Friday"
        if includeWeekend then
            yield "Saturday"
            yield "Sunday"
    ]
```

In some cases, `do...yield` may aid in readability. These cases, though subjective, should be taken into consideration.

Formatting record expressions

Short records can be written in one line:

```
// ✔️ OK
let point = { X = 1.0; Y = 0.0 }
```

Records that are longer should use new lines for labels:

```
// ✔️ OK
let rainbow =
    { Boss = "Jeffrey"
      Lackeys = ["Zippy"; "George"; "Bungle"] }
```

Long record field expressions should use a new line and have one indent from the opening `{`:

```
{ A = a
  B =
    someFunctionCall
        arg1
        arg2
        // ...
        argX
  C = c }
```

Placing the `{` and `}` on new lines with contents indented is possible, however code formatters may reformat this by default:

```
// ✔️ OK
let rainbow =
    { Boss1 = "Jeffrey"
      Boss2 = "Jeffrey"
      Boss3 = "Jeffrey"
      Lackeys = ["Zippy"; "George"; "Bungle"] }

// ❌ Not preferred, code formatters will reformat to the above by default
let rainbow =
    {
        Boss1 = "Jeffrey"
        Boss2 = "Jeffrey"
        Boss3 = "Jeffrey"
        Lackeys = ["Zippy"; "George"; "Bungle"]
    }
```

The same rules apply for list and array elements.

Formatting copy-and-update record expressions

A copy-and-update record expression is still a record, so similar guidelines apply.

Short expressions can fit on one line:

```
// ✔️ OK
let point2 = { point with X = 1; Y = 2 }
```

Longer expressions should use new lines:

```
// ✔️ OK
let rainbow2 =
    { rainbow with
        Boss = "Jeffrey"
        Lackeys = [ "Zippy"; "George"; "Bungle" ] }
```

You may want to dedicate separate lines for the braces and indent one scope to the right with the expression, however code formatters may reformat it. In some special cases, such as wrapping a value with an optional without parentheses, you may need to keep a brace on one line:

```
// ✔️ OK
let newState =
    { state with
          Foo = Some { F1 = 0; F2 = "" } }

// ❌ Not OK, code formatters will reformat to the above by default
let newState =
    {
        state with
            Foo =
                Some {
                    F1 = 0
                    F2 = ""
                }
    }
```

Formatting pattern matching

Use a `|` for each clause of a match with no indentation. If the expression is short, you can consider using a single line if each subexpression is also simple.

```
// ✔️ OK
match l with
| { him = x; her = "Posh" } :: tail -> x
| _ :: tail -> findDavid tail
| [] -> failwith "Couldn't find David"

// ❌ Not OK, code formatters will reformat to the above by default
match l with
    | { him = x; her = "Posh" } :: tail -> x
    | _ :: tail -> findDavid tail
    | [] -> failwith "Couldn't find David"
```

If the expression on the right of the pattern matching arrow is too large, move it to the following line, indented one step from the `match`/`|`.

```
// ✔️ OK
match lam with
| Var v -> 1
| Abs(x, body) ->
    1 + sizeLambda body
| App(lam1, lam2) ->
    sizeLambda lam1 + sizeLambda lam2
```

Similar to large if conditions, if a match expression is multiline or exceeds the default tolerance of the single-line, the match expression should use one indentation and a new line. The `match` and `with` keyword should align when encapsulating the long match expression.

```
// ✔️ OK, but better to refactor, see below
match
    complexExpression a b && env.IsDevelopment()
    || someFunctionToCall
        aVeryLongParameterNameOne
        aVeryLongParameterNameTwo
        aVeryLongParameterNameThree
with
| X y -> y
| _ -> 0
```

It is, however, better style to refactor long match expressions to a let binding or separate function:

```
// ✔️ OK
let performAction =
    complexExpression a b && env.IsDevelopment()
    || someFunctionToCall
        aVeryLongParameterNameOne
        aVeryLongParameterNameTwo
        aVeryLongParameterNameThree

match performAction with
| X y -> y
| _ -> 0
```

Aligning the arrows of a pattern match should be avoided.

```
// ✔️ OK
match lam with
| Var v -> v.Length
| Abstraction _ -> 2

// ❌ Not OK, code formatters will reformat to the above by default
match lam with
| Var v         -> v.Length
| Abstraction _ -> 2
```

Pattern matching introduced by using the keyword `function` should indent one level from the start of the previous line:

```
// ✔️ OK
lambdaList
|> List.map (function
    | Abs(x, body) -> 1 + sizeLambda 0 body
    | App(lam1, lam2) -> sizeLambda (sizeLambda 0 lam1) lam2
    | Var v -> 1)
```

The use of `function` in functions defined by `let` or `let rec` should in general be avoided in favor of a `match`. If used, the pattern rules should align with the keyword `function`:

```
// ✔️ OK
let rec sizeLambda acc =
    function
    | Abs(x, body) -> sizeLambda (succ acc) body
    | App(lam1, lam2) -> sizeLambda (sizeLambda acc lam1) lam2
    | Var v -> succ acc
```

Formatting try/with expressions

Pattern matching on the exception type should be indented at the same level as `with`.

```
// ✔️ OK
try
    if System.DateTime.Now.Second % 3 = 0 then
        raise (new System.Exception())
    else
        raise (new System.ApplicationException())
with
| :? System.ApplicationException ->
    printfn "A second that was not a multiple of 3"
| _ ->
    printfn "A second that was a multiple of 3"
```

Add a `|` for each clause, except when there is only a single clause:

```
// ✔️ OK
try
    persistState currentState
with ex ->
    printfn "Something went wrong: %A" ex

// ✔️ OK
try
    persistState currentState
with :? System.ApplicationException as ex ->
    printfn "Something went wrong: %A" ex

// ❌ Not OK, see above for preferred formatting
try
    persistState currentState
with
| ex ->
    printfn "Something went wrong: %A" ex

// ❌ Not OK, see above for preferred formatting
try
    persistState currentState
with
| :? System.ApplicationException as ex ->
    printfn "Something went wrong: %A" ex
```

Formatting named arguments

Named arguments should have spaces surrounding the `=`:

```
// ✔️ OK
let makeStreamReader x = new System.IO.StreamReader(path = x)

// ❌ Not OK, spaces are necessary around '=' for named arguments
let makeStreamReader x = new System.IO.StreamReader(path=x)
```

When pattern matching using discriminated unions, named patterns are formatted similarly, for example.

```
type Data =
    | TwoParts of part1: string * part2: string
    | OnePart of part1: string

// ✔️ OK
let examineData x =
    match data with
    | OnePartData(part1 = p1) -> p1
    | TwoPartData(part1 = p1; part2 = p2) -> p1 + p2

// ❌ Not OK, spaces are necessary around '=' for named pattern access
let examineData x =
    match data with
    | OnePartData(part1=p1) -> p1
    | TwoPartData(part1=p1; part2=p2) -> p1 + p2
```

Formatting mutation expressions

Mutation expressions `location <- expr` are normally formatted on one line. If multi-line formatting is required, place the right-hand-side expression on a new line.

```
// ✔️ OK
ctx.Response.Headers[HeaderNames.ContentType] <-
    Constants.jsonApiMediaType |> StringValues

ctx.Response.Headers[HeaderNames.ContentLength] <-
    bytes.Length |> string |> StringValues

// ❌ Not OK, code formatters will reformat to the above by default
ctx.Response.Headers[HeaderNames.ContentType] <- Constants.jsonApiMediaType
                                                 |> StringValues
ctx.Response.Headers[HeaderNames.ContentLength] <- bytes.Length
                                                   |> string
                                                   |> StringValues
```

Formatting object expressions

Object expression members should be aligned with `member` being indented by one level.

```
// ✔️ OK
let comparer =
    { new IComparer<string> with
          member x.Compare(s1, s2) =
              let rev (s: String) = new String (Array.rev (s.ToCharArray()))
              let reversed = rev s1
              reversed.CompareTo (rev s2) }
```

Formatting index/slice expressions

Index expressions shouldn't contain any spaces around the opening and closing brackets.

```
// ✔️ OK
let v = expr[idx]
let y = myList[0..1]

// ❌ Not OK
let v = expr[ idx ]
let y = myList[ 0 .. 1 ]
```

This also applies for the older `expr.[idx]` syntax.

```
// ✔️ OK
let v = expr.[idx]
let y = myList.[0..1]

// ❌ Not OK
let v = expr.[ idx ]
let y = myList.[ 0 .. 1 ]
```

Formatting quoted expressions

The delimiter symbols (`<@` , `@>` , `<@@` , `@@>`) should be placed on separate lines if the quoted expression is a multi-line expression.

```
// ✔️ OK
<@
    let f x = x + 10
    f 20
@>

// ❌ Not OK
<@ let f x = x + 10
   f 20
@>
```

In single-line expressions the delimiter symbols should be placed on the same line as the expression itself.

```
// ✔️ OK
<@ 1 + 1 @>

// ❌ Not OK
<@
    1 + 1
@>
```

Formatting declarations

This section discusses formatting declarations of different kinds.

Add blank lines between declarations

Separate top-level function and class definitions with a single blank line. For example:

```
// ✔️ OK
let thing1 = 1+1

let thing2 = 1+2

let thing3 = 1+3

type ThisThat = This | That

// ❌ Not OK
let thing1 = 1+1
let thing2 = 1+2
let thing3 = 1+3
type ThisThat = This | That
```

If a construct has XML doc comments, add a blank line before the comment.

```
// ✔️ OK

/// This is a function
let thisFunction() =
    1 + 1

/// This is another function, note the blank line before this line
let thisFunction() =
    1 + 1
```

Formatting let and member declarations

When formatting `let` and `member` declarations, the right-hand side of a binding either goes on one line, or (if it's too long) goes on a new line indented one level.

For example, the following are compliant:

```
// ✔️ OK
let a =
    """
foobar, long string
"""

// ✔️ OK
type File =
    member this.SaveAsync(path: string) : Async<unit> =
        async {
            // IO operation
            return ()
        }

// ✔️ OK
let c =
    { Name = "Bilbo"
      Age = 111
      Region = "The Shire" }

// ✔️ OK
let d =
    while f do
        printfn "%A" x
```

The following are non-compliant:

```
// ❌ Not OK, code formatters will reformat to the above by default
let a = """
foobar, long string
"""

type File =
    member this.SaveAsync(path: string) : Async<unit> = async {
        // IO operation
        return ()
    }

let c = {
    Name = "Bilbo"
    Age = 111
    Region = "The Shire"
}

let d = while f do
    printfn "%A" x
```

Separate members with a single blank line and document and add a documentation comment:

```
// ✔️ OK

/// This is a thing
type ThisThing(value: int) =

    /// Gets the value
    member _.Value = value

    /// Returns twice the value
    member _.TwiceValue() = value*2
```

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (for example, a set of dummy implementations). Use blank lines in functions, sparingly, to indicate logical sections.

Formatting function and member arguments

When defining a function, use white space around each argument.

```
// ✔️ OK
let myFun (a: decimal) (b: int) c = a + b + c

// ❌ Not OK, code formatters will reformat to the above by default
let myFunBad (a:decimal)(b:int)c = a + b + c
```

If you have a long function definition, place the parameters on new lines and indent them to match the indentation level of the subsequent parameter.

```
// ✔️ OK
module M =
    let longFunctionWithLotsOfParameters
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        =
        // ... the body of the method follows

    let longFunctionWithLotsOfParametersAndReturnType
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        : ReturnType =
        // ... the body of the method follows

    let longFunctionWithLongTupleParameter
        (
            aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
        ) =
        // ... the body of the method follows

    let longFunctionWithLongTupleParameterAndReturnType
        (
            aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
        ) : ReturnType =
        // ... the body of the method follows
```

This also applies to members, constructors, and parameters using tuples:

```fsharp
// ✔️ OK
type TypeWithLongMethod() =
    member _.LongMethodWithLotsOfParameters
        (
            aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
            aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
        ) =
        // ... the body of the method

// ✔️ OK
type TypeWithLongConstructor
    (
        aVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse,
        aSecondVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse,
        aThirdVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse
    ) =
    // ... the body of the class follows
```

If the parameters are curried, place the `=` character along with any return type on a new line:

```fsharp
// ✔️ OK
type TypeWithLongCurriedMethods() =
    member _.LongMethodWithLotsOfCurriedParamsAndReturnType
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        : ReturnType =
        // ... the body of the method

    member _.LongMethodWithLotsOfCurriedParams
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        =
        // ... the body of the method
```

This is a way to avoid too long lines (in case return type might have long name) and have less line-damage when adding parameters.

Formatting operator declarations

Optionally use white space to surround an operator definition:

```fsharp
// ✔️ OK
let ( !> ) x f = f x

// ✔️ OK
let (!>) x f = f x
```

For any custom operator that starts with `*` and that has more than one character, you need to add a white space to the beginning of the definition to avoid a compiler ambiguity. Because of this, we recommend that you simply surround the definitions of all operators with a single white-space character.

Formatting record declarations

For record declarations, indent `{` in type definition by four spaces, start the field list on the same line and align any members with the `{` token:

```
// ✔️ OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
    member x.ZipAndCity = $"{x.Zip} {x.City}"
```

Don't place the `{` at the end of the type declaration line, and don't use `with`/`end` for members, which are redundant.

```
// ❌ Not OK, code formatters will reformat to the above by default
type PostalAddress = {
    Address: string
    City: string
    Zip: string
  }
  with
    member x.ZipAndCity = $"{x.Zip} {x.City}"
  end
```

When XML documentation is added for record fields, it becomes normal to indent and add whitespace:

```
// ✔️ OK
type PostalAddress =
    {
        /// The address
        Address: string

        /// The city
        City: string

        /// The zip code
        Zip: string
    }

    /// Format the zip code and the city
    member x.ZipAndCity = $"{x.Zip} {x.City}"
```

Placing the opening token on a new line and the closing token on a new line is preferable if you're declaring interface implementations or members on the record:

```
// ✔️ OK
// Declaring additional members on PostalAddress
type PostalAddress =
    {
        /// The address
        Address: string

        /// The city
        City: string

        /// The zip code
        Zip: string
    }

    member x.ZipAndCity = $"{x.Zip} {x.City}"

type MyRecord =
    {
        /// The record field
        SomeField: int
    }
    interface IMyInterface
```

Formatting discriminated union declarations

For discriminated union declarations, indent `|` in type definition by four spaces:

```
// ✔️ OK
type Volume =
    | Liter of float
    | FluidOunce of float
    | ImperialPint of float

// ❌ Not OK
type Volume =
| Liter of float
| USPint of float
| ImperialPint of float
```

When there is a single short union, you can omit the leading `|`.

```
// ✔️ OK
type Address = Address of string
```

For a longer or multi-line union, keep the `|` and place each union field on a new line, with the separating `*` at the end of each line.

```
// ✔️ OK
[<NoEquality; NoComparison>]
type SynBinding =
    | SynBinding of
        accessibility: SynAccess option *
        kind: SynBindingKind *
        mustInline: bool *
        isMutable: bool *
        attributes: SynAttributes *
        xmlDoc: PreXmlDoc *
        valData: SynValData *
        headPat: SynPat *
        returnInfo: SynBindingReturnInfo option *
        expr: SynExpr *
        range: range *
        seqPoint: DebugPointAtBinding
```

When documentation comments are added, use an empty line before each `///` comment.

```
// ✔️ OK

/// The volume
type Volume =

    /// The volume in liters
    | Liter of float

    /// The volume in fluid ounces
    | FluidOunce of float

    /// The volume in imperial pints
    | ImperialPint of float
```

Formatting literal declarations

F# literals using the `Literal` attribute should place the attribute on its own line and use PascalCase naming:

```
// ✔️ OK

[<Literal>]
let Path = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let MyUrl = "www.mywebsitethatiamworkingwith.com"
```

Avoid placing the attribute on the same line as the value.

Formatting module declarations

Code in a local module must be indented relative to the module, but code in a top-level module should not be indented. Namespace elements do not have to be indented.

```
// ✔️ OK - A is a top-level module.
module A

let function1 a b = a - b * b
```

```
// ✔️ OK - A1 and A2 are local modules.
module A1 =
    let function1 a b = a * a + b * b

module A2 =
    let function2 a b = a * a - b * b
```

Formatting do declarations

In type declarations, module declarations and computation expressions, the use of `do` or `do!` is sometimes required for side-effecting operations. When these span multiple lines, use indentation and a new line to keep the indentation consistent with `let` / `let!`. Here's an example using `do` in a class:

```
// ✔️ OK
type Foo () =
    let foo =
        fooBarBaz
        |> loremIpsumDolorSitAmet
        |> theQuickBrownFoxJumpedOverTheLazyDog

    do
        fooBarBaz
        |> loremIpsumDolorSitAmet
        |> theQuickBrownFoxJumpedOverTheLazyDog

// ❌ Not OK - notice the "do" expression is indented one space less than the `let` expression
type Foo () =
    let foo =
        fooBarBaz
        |> loremIpsumDolorSitAmet
        |> theQuickBrownFoxJumpedOverTheLazyDog
    do fooBarBaz
       |> loremIpsumDolorSitAmet
       |> theQuickBrownFoxJumpedOverTheLazyDog
```

Here's an example with `do!` using two spaces of indentation (because with `do!` there is coincidentally no difference between the approaches when using four spaces of indentation):

```
// ✔️ OK
async {
  let! foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog

  do!
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
}

// ❌ Not OK - notice the "do!" expression is indented two spaces more than the `let!` expression
async {
  let! foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
  do! fooBarBaz
      |> loremIpsumDolorSitAmet
      |> theQuickBrownFoxJumpedOverTheLazyDog
}
```

Formatting computation expression operations

When creating custom operations for computation expressions, it is recommended to use camelCase naming:

```
// ✔️ OK
type MathBuilder () =
    member _.Yield _ = 0

    [<CustomOperation("addOne")>]
    member _.AddOne (state: int) =
        state + 1

    [<CustomOperation("subtractOne")>]
    member _.SubtractOne (state: int) =
        state - 1

    [<CustomOperation("divideBy")>]
    member _.DivideBy (state: int, divisor: int) =
        state / divisor

    [<CustomOperation("multiplyBy")>]
    member _.MultiplyBy (state: int, factor: int) =
        state * factor

let math = MathBuilder()

let myNumber =
    math {
        addOne
        addOne
        addOne
        subtractOne
        divideBy 2
        multiplyBy 10
    }
```

The domain that's being modeled should ultimately drive the naming convention. If it's idiomatic to use a different convention, that convention should be used instead.

Formatting types and type annotations

This section discusses formatting types and type annotations. This includes formatting signature files with the `.fsi` extension.

For types, prefer prefix syntax for generics (`Foo<T>`), with some specific exceptions

F# allows both postfix style of writing generic types (for example, `int list`) and the prefix style (for example, `list<int>`). Postfix style can only be used with a single type argument. Always prefer the .NET style, except for five specific types:

1. For F# Lists, use the postfix form: `int list` rather than `list<int>`.
2. For F# Options, use the postfix form: `int option` rather than `option<int>`.
3. For F# Value Options, use the postfix form: `int voption` rather than `voption<int>`.
4. For F# arrays, use the syntactic name `int[]` rather than `int array` or `array<int>`.
5. For Reference Cells, use `int ref` rather than `ref<int>` or `Ref<int>`.

For all other types, use the prefix form.

Formatting function types

When defining the signature of a function, use white space around the `->` symbol:

```
// ✔️ OK
type MyFun = int -> int -> string

// ❌ Not OK
type MyFunBad = int->int->string
```

Formatting value and argument type annotations

When defining values or arguments with type annotations, use white space after the `:` symbol, but not before:

```
// ✔️ OK
let complexFunction (a: int) (b: int) c = a + b + c

let simpleValue: int = 0 // Type annotation for let-bound value

type C() =
    member _.Property: int = 1

// ❌ Not OK
let complexFunctionPoorlyAnnotated (a :int) (b :int) (c:int) = a + b + c
let simpleValuePoorlyAnnotated1:int = 1
let simpleValuePoorlyAnnotated2 :int = 2
```

Formatting return type annotations

In function or member return type annotations, use white space before and after the `:` symbol:

```
// ✔️ OK
let myFun (a: decimal) b c : decimal = a + b + c

type C() =
    member _.SomeMethod(x: int) : int = 1

// ❌ Not OK
let myFunBad (a: decimal) b c:decimal = a + b + c

let anotherFunBad (arg: int): unit = ()

type C() =
    member _.SomeMethodBad(x: int): int = 1
```

Formatting types in signatures

When writing full function types in signatures, it's sometimes necessary to split the arguments over multiple lines. The return type is always indented.

For a tupled function, the arguments are separated by `*`, placed at the end of each line.

For example, consider a function with the following implementation:

```
let SampleTupledFunction(arg1, arg2, arg3, arg4) = ...
```

In the corresponding signature file (`.fsi` extension) the function can be formatted as follows when multi-line formatting is required:

```
// ✔️ OK
val SampleTupledFunction:
    arg1: string *
    arg2: string *
    arg3: int *
    arg4: int ->
        int list
```

Likewise consider a curried function:

```
let SampleCurriedFunction arg1 arg2 arg3 arg4 = ...
```

In the corresponding signature file, the `->` are placed at the end of each line:

```
// ✔️ OK
val SampleCurriedFunction:
    arg1: string ->
    arg2: string ->
    arg3: int ->
    arg4: int ->
        int list
```

Likewise, consider a function that takes a mix of curried and tupled arguments:

```
// Typical call syntax:
let SampleMixedFunction
        (arg1, arg2)
        (arg3, arg4, arg5)
        (arg6, arg7)
        (arg8, arg9, arg10) = ..
```

In the corresponding signature file, the types preceded by a tuple are indented

```
// ✔️ OK
val SampleMixedFunction:
    arg1: string *
    arg2: string ->
        arg3: string *
        arg4: string *
        arg5: TType ->
            arg6: TType *
            arg7: TType ->
                arg8: TType *
                arg9: TType *
                arg10: TType ->
                    TType list
```

The same rules apply for members in type signatures:

```
type SampleTypeName =
    member ResolveDependencies:
        arg1: string *
        arg2: string ->
            string
```

Formatting explicit generic type arguments and constraints

The guidelines below apply to function definitions, member definitions, type definitions, and function applications.

Keep generic type arguments and constraints on a single line if it’s not too long:

```
// ✔️ OK
let f<'T1, 'T2 when 'T1: equality and 'T2: comparison> param =
    // function body
```

If both generic type arguments/constraints and function parameters don’t fit, but the type parameters/constraints alone do, place the parameters on new lines:

```
// ✔️ OK
let f<'T1, 'T2 when 'T1 : equality and 'T2 : comparison>
    param
    =
    // function body
```

If the type parameters or constraints are too long, break and align them as shown below. Keep the list of type parameters on the same line as the function, regardless of its length. For constraints, place `when` on the first line, and keep each constraint on a single line regardless of its length. Place `>` at the end of the last line. Indent the constraints by one level.

```
// ✔️ OK
let inline f< ^T1, ^T2
    when ^T1 : (static member Foo1: unit -> ^T2)
    and ^T2 : (member Foo2: unit -> int)
    and ^T2 : (member Foo3: string -> ^T1 option)>
    arg1
    arg2
    =
    // function body
```

If the type parameters/constraints are broken up, but there are no normal function parameters, place the `=` on a new line regardless:

```
// ✔️ OK
let inline f<^T1, ^T2
    when ^T1 : (static member Foo1: unit -> ^T2)
    and ^T2 : (member Foo2: unit -> int)
    and ^T2 : (member Foo3: string -> ^T1 option)>
    =
    // function body
```

The same rules apply for function applications:

```
// ✔️ OK
myObj
|> Json.serialize<
    {| child: {| displayName: string; kind: string |}
       newParent: {| id: string; displayName: string |}
       requiresApproval: bool |}>

// ✔️ OK
Json.serialize<
    {| child: {| displayName: string; kind: string |}
       newParent: {| id: string; displayName: string |}
       requiresApproval: bool |}>
    myObj
```

Formatting attributes

Attributes are placed above a construct:

```
// ✔️ OK
[<SomeAttribute>]
type MyClass() = ...

// ✔️ OK
[<RequireQualifiedAccess>]
module M =
    let f x = x

// ✔️ OK
[<Struct>]
type MyRecord =
    { Label1: int
      Label2: string }
```

They should go after any XML documentation:

```
// ✔️ OK

/// Module with some things in it.
[<RequireQualifiedAccess>]
module M =
    let f x = x
```

Formatting attributes on parameters

Attributes can also be placed on parameters. In this case, place then on the same line as the parameter and before the name:

```
// ✔️ OK - defines a class that takes an optional value as input defaulting to false.
type C() =
    member _.M([<Optional; DefaultParameterValue(false)>] doSomething: bool)
```

Formatting multiple attributes

When multiple attributes are applied to a construct that's not a parameter, place each attribute on a separate line:

```
// ✔️ OK

[<Struct>]
[<IsByRefLike>]
type MyRecord =
    { Label1: int
      Label2: string }
```

When applied to a parameter, place attributes on the same line and separate them with a `;` separator.

Acknowledgments

These guidelines are based on A comprehensive guide to F# Formatting Conventions by Anh-Dung Phan.

F# coding conventions

8/5/2022 • 26 minutes to read • Edit Online

The following conventions are formulated from experience working with large F# codebases. The Five principles of good F# code are the foundation of each recommendation. They are related to the F# component design guidelines, but are applicable for any F# code, not just components such as libraries.

Organizing code

F# features two primary ways to organize code: modules and namespaces. These are similar, but do have the following differences:

- Namespaces are compiled as .NET namespaces. Modules are compiled as static classes.
- Namespaces are always top level. Modules can be top-level and nested within other modules.
- Namespaces can span multiple files. Modules cannot.
- Modules can be decorated with `[<RequireQualifiedAccess>]` and `[<AutoOpen>]`.

The following guidelines will help you use these to organize your code.

Prefer namespaces at the top level

For any publicly consumable code, namespaces are preferential to modules at the top level. Because they are compiled as .NET namespaces, they are consumable from C# with no issue.

```
// Good!
namespace MyCode

type MyClass() =
    ...
```

Using a top-level module may not appear different when called only from F#, but for C# consumers, callers may be surprised by having to qualify `MyClass` with the `MyCode` module.

```
// Bad!
module MyCode

type MyClass() =
    ...
```

Carefully apply `[<AutoOpen>]`

The `[<AutoOpen>]` construct can pollute the scope of what is available to callers, and the answer to where something comes from is "magic". This is not a good thing. An exception to this rule is the F# Core Library itself (though this fact is also a bit controversial).

However, it is a convenience if you have helper functionality for a public API that you wish to organize separately from that public API.

```fsharp
module MyAPI =
    [<AutoOpen>]
    module private Helpers =
        let helper1 x y z =
            ...

    let myFunction1 x =
        let y = ...
        let z = ...

        helper1 x y z
```

This lets you cleanly separate implementation details from the public API of a function without having to fully qualify a helper each time you call it.

Additionally, exposing extension methods and expression builders at the namespace level can be neatly expressed with `[<AutoOpen>]`.

Use `[<RequireQualifiedAccess>]` whenever names could conflict or you feel it helps with readability

Adding the `[<RequireQualifiedAccess>]` attribute to a module indicates that the module may not be opened and that references to the elements of the module require explicit qualified access. For example, the `Microsoft.FSharp.Collections.List` module has this attribute.

This is useful when functions and values in the module have names that are likely to conflict with names in other modules. Requiring qualified access can greatly increase long-term maintainability and the ability of a library to evolve.

```fsharp
[<RequireQualifiedAccess>]
module StringTokenization =
    let parse s = ...

...

let s = getAString()
let parsed = StringTokenization.parse s // Must qualify to use 'parse'
```

Sort `open` statements topologically

In F#, the order of declarations matters, including with `open` statements. This is unlike C#, where the effect of `using` and `using static` is independent of the ordering of those statements in a file.

In F#, elements opened into a scope can shadow others already present. This means that reordering `open` statements could alter the meaning of code. As a result, any arbitrary sorting of all `open` statements (for example, alphanumerically) is not recommended, lest you generate different behavior that you might expect.

Instead, we recommend that you sort them topologically; that is, order your `open` statements in the order in which *layers* of your system are defined. Doing alphanumeric sorting within different topological layers may also be considered.

As an example, here is the topological sorting for the F# compiler service public API file:

```
namespace Microsoft.FSharp.Compiler.SourceCodeServices

open System
open System.Collections.Generic
open System.Collections.Concurrent
open System.Diagnostics
open System.IO
open System.Reflection
open System.Text

open FSharp.Compiler
open FSharp.Compiler.AbstractIL
open FSharp.Compiler.AbstractIL.Diagnostics
open FSharp.Compiler.AbstractIL.IL
open FSharp.Compiler.AbstractIL.ILBinaryReader
open FSharp.Compiler.AbstractIL.Internal
open FSharp.Compiler.AbstractIL.Internal.Library

open FSharp.Compiler.AccessibilityLogic
open FSharp.Compiler.Ast
open FSharp.Compiler.CompileOps
open FSharp.Compiler.CompileOptions
open FSharp.Compiler.Driver

open Internal.Utilities
open Internal.Utilities.Collections
```

A line break separates topological layers, with each layer being sorted alphanumerically afterwards. This cleanly organizes code without accidentally shadowing values.

Use classes to contain values that have side effects

There are many times when initializing a value can have side effects, such as instantiating a context to a database or other remote resource. It is tempting to initialize such things in a module and use it in subsequent functions:

```
// This is bad!
module MyApi =
    let dep1 = File.ReadAllText "/Users/<name>/connectionstring.txt"
    let dep2 = Environment.GetEnvironmentVariable "DEP_2"

    let private r = Random()
    let dep3() = r.Next() // Problematic if multiple threads use this

    let function1 arg = doStuffWith dep1 dep2 dep3 arg
    let function2 arg = doSutffWith dep1 dep2 dep3 arg
```

This is frequently a bad idea for a few reasons:

First, application configuration is pushed into the codebase with `dep1` and `dep2`. This is difficult to maintain in larger codebases.

Second, statically initialized data should not include values that are not thread safe if your component will itself use multiple threads. This is clearly violated by `dep3`.

Finally, module initialization compiles into a static constructor for the entire compilation unit. If any error occurs in let-bound value initialization in that module, it manifests as a `TypeInitializationException` that is then cached for the entire lifetime of the application. This can be difficult to diagnose. There is usually an inner exception that you can attempt to reason about, but if there is not, then there is no telling what the root cause is.

Instead, just use a simple class to hold dependencies:

```
type MyParametricApi(dep1, dep2, dep3) =
    member _.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1
    member _.Function2 arg2 = doStuffWith dep1 dep2 dep3 arg2
```

This enables the following:

1. Pushing any dependent state outside of the API itself.
2. Configuration can now be done outside of the API.
3. Errors in initialization for dependent values are not likely to manifest as a `TypeInitializationException`.
4. The API is now easier to test.

Error management

Error management in large systems is a complex and nuanced endeavor, and there are no silver bullets in ensuring your systems are fault-tolerant and behave well. The following guidelines should offer guidance in navigating this difficult space.

Represent error cases and illegal state in types intrinsic to your domain

With Discriminated Unions, F# gives you the ability to represent faulty program state in your type system. For example:

```
type MoneyWithdrawalResult =
    | Success of amount:decimal
    | InsufficientFunds of balance:decimal
    | CardExpired of DateTime
    | UndisclosedFailure
```

In this case, there are three known ways that withdrawing money from a bank account can fail. Each error case is represented in the type, and can thus be dealt with safely throughout the program.

```
let handleWithdrawal amount =
    let w = withdrawMoney amount
    match w with
    | Success am -> printfn $"Successfully withdrew %f{am}"
    | InsufficientFunds balance -> printfn $"Failed: balance is %f{balance}"
    | CardExpired expiredDate -> printfn $"Failed: card expired on {expiredDate}"
    | UndisclosedFailure -> printfn "Failed: unknown"
```

In general, if you can model the different ways that something can **fail** in your domain, then error handling code is no longer treated as something you must deal with in addition to regular program flow. It is simply a part of normal program flow, and not considered **exceptional**. There are two primary benefits to this:

1. It is easier to maintain as your domain changes over time.
2. Error cases are easier to unit test.

Use exceptions when errors cannot be represented with types

Not all errors can be represented in a problem domain. These kinds of faults are *exceptional* in nature, hence the ability to raise and catch exceptions in F#.

First, it is recommended that you read the Exception Design Guidelines. These are also applicable to F#.

The main constructs available in F# for the purposes of raising exceptions should be considered in the following order of preference:

FUNCTION	SYNTAX	PURPOSE
`nullArg`	`nullArg "argumentName"`	Raises a `System.ArgumentNullException` with the specified argument name.
`invalidArg`	`invalidArg "argumentName" "message"`	Raises a `System.ArgumentException` with a specified argument name and message.
`invalidOp`	`invalidOp "message"`	Raises a `System.InvalidOperationException` with the specified message.
`raise`	`raise (ExceptionType("message"))`	General-purpose mechanism for throwing exceptions.
`failwith`	`failwith "message"`	Raises a `System.Exception` with the specified message.
`failwithf`	`failwithf "format string" argForFormatString`	Raises a `System.Exception` with a message determined by the format string and its inputs.

Use `nullArg`, `invalidArg`, and `invalidOp` as the mechanism to throw `ArgumentNullException`, `ArgumentException`, and `InvalidOperationException` when appropriate.

The `failwith` and `failwithf` functions should generally be avoided because they raise the base `Exception` type, not a specific exception. As per the Exception Design Guidelines, you want to raise more specific exceptions when you can.

Use exception-handling syntax

F# supports exception patterns via the `try...with` syntax:

```
try
    tryGetFileContents()
with
| :? System.IO.FileNotFoundException as e -> // Do something with it here
| :? System.Security.SecurityException as e -> // Do something with it here
```

Reconciling functionality to perform in the face of an exception with pattern matching can be a bit tricky if you wish to keep the code clean. One such way to handle this is to use active patterns as a means to group functionality surrounding an error case with an exception itself. For example, you may be consuming an API that, when it throws an exception, encloses valuable information in the exception metadata. Unwrapping a useful value in the body of the captured exception inside the Active Pattern and returning that value can be helpful in some situations.

Do not use monadic error handling to replace exceptions

Exceptions are often seen as taboo in functional programming. Indeed, exceptions violate purity, so it's safe to consider them not-quite functional. However, this ignores the reality of where code must run, and that runtime errors can occur. In general, write code on the assumption that most things aren't pure or total, to minimize unpleasant surprises.

It is important to consider the following core strengths/aspects of Exceptions with respect to their relevance and appropriateness in the .NET runtime and cross-language ecosystem as a whole:

- They contain detailed diagnostic information, which is helpful when debugging an issue.
- They are well understood by the runtime and other .NET languages.
- They can reduce significant boilerplate when compared with code that goes out of its way to *avoid* exceptions by implementing some subset of their semantics on an ad-hoc basis.

This third point is critical. For nontrivial complex operations, failing to use exceptions can result in dealing with structures like this:

```
Result<Result<MyType, string>, string list>
```

Which can easily lead to fragile code like pattern matching on "stringly typed" errors:

```
let result = doStuff()
match result with
| Ok r -> ...
| Error e ->
    if e.Contains "Error string 1" then ...
    elif e.Contains "Error string 2" then ...
    else ... // Who knows?
```

Additionally, it can be tempting to swallow any exception in the desire for a "simple" function that returns a "nicer" type:

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with _ -> None
```

Unfortunately, `tryReadAllText` can throw numerous exceptions based on the myriad of things that can happen on a file system, and this code discards away any information about what might actually be going wrong in your environment. If you replace this code with a result type, then you're back to "stringly typed" error message parsing:

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Ok
    with e -> Error e.Message

let r = tryReadAllText "path-to-file"
match r with
| Ok text -> ...
| Error e ->
    if e.Contains "uh oh, here we go again..." then ...
    else ...
```

And placing the exception object itself in the `Error` constructor just forces you to properly deal with the exception type at the call site rather than in the function. Doing this effectively creates checked exceptions, which are notoriously unfun to deal with as a caller of an API.

A good alternative to the above examples is to catch *specific* exceptions and return a meaningful value in the context of that exception. If you modify the `tryReadAllText` function as follows, `None` has more meaning:

```
let tryReadAllTextIfPresent (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with :? FileNotFoundException -> None
```

Instead of functioning as a catch-all, this function will now properly handle the case when a file was not found and assign that meaning to a return. This return value can map to that error case, while not discarding any contextual information or forcing callers to deal with a case that may not be relevant at that point in the code.

Types such as `Result<'Success, 'Error>` are appropriate for basic operations where they aren't nested, and F# optional types are perfect for representing when something could either return *something* or *nothing*. They are not a replacement for exceptions, though, and should not be used in an attempt to replace exceptions. Rather, they should be applied judiciously to address specific aspects of exception and error management policy in targeted ways.

Partial application and point-free programming

F# supports partial application, and thus, various ways to program in a point-free style. This can be beneficial for code reuse within a module or the implementation of something, but it is not something to expose publicly. In general, point-free programming is not a virtue in and of itself, and can add a significant cognitive barrier for people who are not immersed in the style.

Do not use partial application and currying in public APIs

With little exception, the use of partial application in public APIs can be confusing for consumers. Usually, `let`-bound values in F# code are **values**, not **function values**. Mixing together values and function values can result in saving a few lines of code in exchange for quite a bit of cognitive overhead, especially if combined with operators such as `>>` to compose functions.

Consider the tooling implications for point-free programming

Curried functions do not label their arguments. This has tooling implications. Consider the following two functions:

```
let func name age =
    printfn $"My name is {name} and I am %d{age} years old!"

let funcWithApplication =
    printfn "My name is %s and I am %d years old!"
```

Both are valid functions, but `funcWithApplication` is a curried function. When you hover over their types in an editor, you see this:

```
val func : name:string -> age:int -> unit

val funcWithApplication : (string -> int -> unit)
```

At the call site, tooltips in tooling such as Visual Studio will give you the type signature, but since there are no names defined, it won't display names. Names are critical to good API design because they help callers better understanding the meaning behind the API. Using point-free code in the public API can make it harder for callers to understand.

If you encounter point-free code like `funcWithApplication` that is publicly consumable, it is recommended to do a full η-expansion so that tooling can pick up on meaningful names for arguments.

Furthermore, debugging point-free code can be challenging, if not impossible. Debugging tools rely on values bound to names (for example, `let` bindings) so that you can inspect intermediate values midway through execution. When your code has no values to inspect, there is nothing to debug. In the future, debugging tools may evolve to synthesize these values based on previously executed paths, but it's not a good idea to hedge your bets on *potential* debugging functionality.

Consider partial application as a technique to reduce internal boilerplate

In contrast to the previous point, partial application is a wonderful tool for reducing boilerplate inside of an application or the deeper internals of an API. It can be helpful for unit testing the implementation of more complicated APIs, where boilerplate is often a pain to deal with. For example, the following code shows how you can accomplish what most mocking frameworks give you without taking an external dependency on such a framework and having to learn a related bespoke API.

For example, consider the following solution topography:

```
MySolution.sln
|_/ImplementationLogic.fsproj
|_/ImplementationLogic.Tests.fsproj
|_/API.fsproj
```

`ImplementationLogic.fsproj` might expose code such as:

```
module Transactions =
    let doTransaction txnContext txnType balance =
        ...

type Transactor(ctx, currentBalance) =
    member _.ExecuteTransaction(txnType) =
        Transactions.doTransaction ctx txtType currentBalance
        ...
```

Unit testing `Transactions.doTransaction` in `ImplementationLogic.Tests.fsproj` is easy:

```
namespace TransactionsTestingUtil

open Transactions

module TransactionsTestable =
    let getTestableTransactionRoutine mockContext = Transactions.doTransaction mockContext
```

Partially applying `doTransaction` with a mocking context object lets you call the function in all of your unit tests without needing to construct a mocked context each time:

```
module TransactionTests

open Xunit
open TransactionTypes
open TransactionsTestingUtil
open TransactionsTestingUtil.TransactionsTestable

let testableContext =
    { new ITransactionContext with
        member _.TheFirstMember() = ...
        member _.TheSecondMember() = ... }

let transactionRoutine = getTestableTransactionRoutine testableContext

[<Fact>]
let ``Test withdrawal transaction with 0.0 for balance``() =
    let expected = ...
    let actual = transactionRoutine TransactionType.Withdraw 0.0
    Assert.Equal(expected, actual)
```

Don't apply this technique universally to your entire codebase, but it is a good way to reduce boilerplate for complicated internals and unit testing those internals.

Access control

F# has multiple options for Access control, inherited from what is available in the .NET runtime. These are not just usable for types - you can use them for functions, too.

- Prefer non-`public` types and members until you need them to be publicly consumable. This also minimizes what consumers couple to.
- Strive to keep all helper functionality `private`.
- Consider the use of `[<AutoOpen>]` on a private module of helper functions if they become numerous.

Type inference and generics

Type inference can save you from typing a lot of boilerplate. And automatic generalization in the F# compiler can help you write more generic code with almost no extra effort on your part. However, these features are not universally good.

- Consider labeling argument names with explicit types in public APIs and do not rely on type inference for this.

  The reason for this is that **you** should be in control of the shape of your API, not the compiler. Although the compiler can do a fine job at inferring types for you, it is possible to have the shape of your API change if the internals it relies on have changed types. This may be what you want, but it will almost certainly result in a breaking API change that downstream consumers will then have to deal with. Instead, if you explicitly control the shape of your public API, then you can control these breaking changes. In DDD terms, this can be thought of as an Anti-corruption layer.

- Consider giving a meaningful name to your generic arguments.

  Unless you are writing truly generic code that is not specific to a particular domain, a meaningful name can help other programmers understanding the domain they're working in. For example, a type parameter named `'Document` in the context of interacting with a document database makes it clearer that generic document types can be accepted by the function or member you are working with.

- Consider naming generic type parameters with PascalCase.

  This is the general way to do things in .NET, so it's recommended that you use PascalCase rather than snake_case or camelCase.

Finally, automatic generalization is not always a boon for people who are new to F# or a large codebase. There is cognitive overhead in using components that are generic. Furthermore, if automatically generalized functions are not used with different input types (let alone if they are intended to be used as such), then there is no real benefit to them being generic then. Always consider if the code you are writing will actually benefit from being generic.

Performance

Consider structs for small types with high allocation rates

Using structs (also called Value Types) can often result in higher performance for some code because it typically avoids allocating objects. However, structs are not always a "go faster" button: if the size of the data in a struct exceeds 16 bytes, copying the data can often result in more CPU time spend than using a reference type.

To determine if you should use a struct, consider the following conditions:

- If the size of your data is 16 bytes or smaller.
- If you're likely to have many instances of these types resident in memory in a running program.

If the first condition applies, you should generally use a struct. If both apply, you should almost always use a struct. There may be some cases where the previous conditions apply, but using a struct is no better or worse than using a reference type, but they are likely to be rare. It's important to always measure when making changes like this, though, and not operate on assumption or intuition.

Consider struct tuples when grouping small value types with high allocation rates

Consider the following two functions:

```
let rec runWithTuple t offset times =
    let offsetValues x y z offset =
        (x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let (x, y, z) = t
        let r = offsetValues x y z offset
        runWithTuple r offset (times - 1)

let rec runWithStructTuple t offset times =
    let offsetValues x y z offset =
        struct(x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let struct(x, y, z) = t
        let r = offsetValues x y z offset
        runWithStructTuple r offset (times - 1)
```

When you benchmark these functions with a statistical benchmarking tool like BenchmarkDotNet, you'll find that the `runWithStructTuple` function that uses struct tuples runs 40% faster and allocates no memory.

However, these results won't always be the case in your own code. If you mark a function as `inline`, code that uses reference tuples may get some additional optimizations, or code that would allocate could simply be optimized away. You should always measure results whenever performance is concerned, and never operate based on assumption or intuition.

Consider struct records when the type is small and has high allocation rates

The rule of thumb described earlier also holds for F# record types. Consider the following data types and functions that process them:

```
type Point = { X: float; Y: float; Z: float }

[<Struct>]
type SPoint = { X: float; Y: float; Z: float }

let rec processPoint (p: Point) offset times =
    let inline offsetValues (p: Point) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processPoint r offset (times - 1)

let rec processStructPoint (p: SPoint) offset times =
    let inline offsetValues (p: SPoint) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processStructPoint r offset (times - 1)
```

This is similar to the previous tuple code, but this time the example uses records and an inlined inner function.

When you benchmark these functions with a statistical benchmarking tool like BenchmarkDotNet, you'll find that `processStructPoint` runs nearly 60% faster and allocates nothing on the managed heap.

Consider struct discriminated unions when the data type is small with high allocation rates

The previous observations about performance with struct tuples and records also holds for F# Discriminated Unions. Consider the following code:

```
    type Name = Name of string

    [<Struct>]
    type SName = SName of string

    let reverseName (Name s) =
        s.ToCharArray()
        |> Array.rev
        |> System.String
        |> Name

    let structReverseName (SName s) =
        s.ToCharArray()
        |> Array.rev
        |> System.String
        |> SName
```

It's common to define single-case Discriminated Unions like this for domain modeling. When you benchmark these functions with a statistical benchmarking tool like BenchmarkDotNet, you'll find that `structReverseName` runs about 25% faster than `reverseName` for small strings. For large strings, both perform about the same. So, in this case, it's always preferable to use a struct. As previously mentioned, always measure and do not operate on assumptions or intuition.

Although the previous example showed that a struct Discriminated Union yielded better performance, it is common to have larger Discriminated Unions when modeling a domain. Larger data types like that may not perform as well if they are structs depending on the operations on them, since more copying could be involved.

Functional programming and mutation

F# values are immutable by default, which allows you to avoid certain classes of bugs (especially those involving concurrency and parallelism). However, in certain cases, in order to achieve optimal (or even reasonable) efficiency of execution time or memory allocations, a span of work may best be implemented by using in-place mutation of state. This is possible in an opt-in basis with F# with the `mutable` keyword.

Use of `mutable` in F# may feel at odds with functional purity. This is understandable, but functional purity everywhere can be at odds with performance goals. A compromise is to encapsulate mutation such that callers need not care about what happens when they call a function. This allows you to write a functional interface over a mutation-based implementation for performance-critical code.

Wrap mutable code in immutable interfaces

With referential transparency as a goal, it is critical to write code that does not expose the mutable underbelly of performance-critical functions. For example, the following code implements the `Array.contains` function in the F# core library:

```
[<CompiledName("Contains")>]
let inline contains value (array:'T[]) =
    checkNonNull "array" array
    let mutable state = false
    let mutable i = 0
    while not state && i < array.Length do
        state <- value = array[i]
        i <- i + 1
    state
```

Calling this function multiple times does not change the underlying array, nor does it require you to maintain any mutable state in consuming it. It is referentially transparent, even though almost every line of code within it uses mutation.

Consider encapsulating mutable data in classes

The previous example used a single function to encapsulate operations using mutable data. This is not always sufficient for more complex sets of data. Consider the following sets of functions:

```
open System.Collections.Generic

let addToClosureTable (key, value) (t: Dictionary<_,_>) =
    if not (t.ContainsKey(key)) then
        t.Add(key, value)
    else
        t[key] <- value

let closureTableCount (t: Dictionary<_,_>) = t.Count

let closureTableContains (key, value) (t: Dictionary<_, HashSet<_>>) =
    match t.TryGetValue(key) with
    | (true, v) -> v.Equals(value)
    | (false, _) -> false
```

This code is performant, but it exposes the mutation-based data structure that callers are responsible for maintaining. This can be wrapped inside of a class with no underlying members that can change:

```
open System.Collections.Generic

/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =
    let t = Dictionary<Item0, HashSet<TerminalIndex>>()

    member _.Add(key, value) =
        if not (t.ContainsKey(key)) then
            t.Add(key, value)
        else
            t[key] <- value

    member _.Count = t.Count

    member _.Contains(key, value) =
        match t.TryGetValue(key) with
        | (true, v) -> v.Equals(value)
        | (false, _) -> false
```

`Closure1Table` encapsulates the underlying mutation-based data structure, thereby not forcing callers to maintain the underlying data structure. Classes are a powerful way to encapsulate data and routines that are mutation-based without exposing the details to callers.

Prefer `let mutable` to reference cells

Reference cells are a way to represent the reference to a value rather than the value itself. Although they can be used for performance-critical code, they are not recommended. Consider the following example:

```
let kernels =
    let acc = ref Set.empty

    processWorkList startKernels (fun kernel ->
        if not ((!acc).Contains(kernel)) then
            acc := (!acc).Add(kernel)
        ...)

    !acc |> Seq.toList
```

The use of a reference cell now "pollutes" all subsequent code with having to dereference and re-reference the underlying data. Instead, consider `let mutable`:

```
let kernels =
    let mutable acc = Set.empty

    processWorkList startKernels (fun kernel ->
        if not (acc.Contains(kernel)) then
            acc <- acc.Add(kernel)
        ...)

    acc |> Seq.toList
```

Aside from the single point of mutation in the middle of the lambda expression, all other code that touches `acc` can do so in a manner that is no different to the usage of a normal `let`-bound immutable value. This will make it easier to change over time.

Object programming

F# has full support for objects and object-oriented (OO) concepts. Although many OO concepts are powerful and useful, not all of them are ideal to use. The following lists offer guidance on categories of OO features at a high level.

Consider using these features in many situations:

- Dot notation (`x.Length`)
- Instance members
- Implicit constructors
- Static members
- Indexer notation (`arr[x]`), by defining an `Item` property
- Slicing notation (`arr[x..y]` , `arr[x..]` , `arr[..y]`), by defining `GetSlice` members
- Named and Optional arguments
- Interfaces and interface implementations

Don't reach for these features first, but do judiciously apply them when they are convenient to solve a problem:

- Method overloading
- Encapsulated mutable data
- Operators on types
- Auto properties
- Implementing `IDisposable` and `IEnumerable`
- Type extensions
- Events
- Structs
- Delegates
- Enums

Generally avoid these features unless you must use them:

- Inheritance-based type hierarchies and implementation inheritance
- Nulls and `Unchecked.defaultof<_>`

Prefer composition over inheritance

Composition over inheritance is a long-standing idiom that good F# code can adhere to. The fundamental principle is that you should not expose a base class and force callers to inherit from that base class to get functionality.

Use object expressions to implement interfaces if you don't need a class

Object Expressions allow you to implement interfaces on the fly, binding the implemented interface to a value without needing to do so inside of a class. This is convenient, especially if you *only* need to implement the interface and have no need for a full class.

For example, here is the code that is run in Ionide to provide a code fix action if you've added a symbol that you don't have an `open` statement for:

```
    let private createProvider () =
        { new CodeActionProvider with
            member this.provideCodeActions(doc, range, context, ct) =
                let diagnostics = context.diagnostics
                let diagnostic = diagnostics |> Seq.tryFind (fun d -> d.message.Contains "Unused open
statement")
                let res =
                    match diagnostic with
                    | None -> [||]
                    | Some d ->
                        let line = doc.lineAt d.range.start.line
                        let cmd = createEmpty<Command>
                        cmd.title <- "Remove unused open"
                        cmd.command <- "fsharp.unusedOpenFix"
                        cmd.arguments <- Some ([| doc |> unbox; line.range |> unbox; |] |> ResizeArray)
                        [|cmd |]
                res
                |> ResizeArray
                |> U2.Case1
        }
```

Because there is no need for a class when interacting with the Visual Studio Code API, Object Expressions are an ideal tool for this. They are also valuable for unit testing, when you want to stub out an interface with test routines in an improvised manner.

Consider Type Abbreviations to shorten signatures

Type Abbreviations are a convenient way to assign a label to another type, such as a function signature or a more complex type. For example, the following alias assigns a label to what's needed to define a computation with CNTK, a deep learning library:

```
open CNTK

// DeviceDescriptor, Variable, and Function all come from CNTK
type Computation = DeviceDescriptor -> Variable -> Function
```

The `Computation` name is a convenient way to denote any function that matches the signature it is aliasing. Using Type Abbreviations like this is convenient and allows for more succinct code.

Avoid using Type Abbreviations to represent your domain

Although Type Abbreviations are convenient for giving a name to function signatures, they can be confusing when abbreviating other types. Consider this abbreviation:

```
// Does not actually abstract integers.
type BufferSize = int
```

This can be confusing in multiple ways:

- `BufferSize` is not an abstraction; it is just another name for an integer.
- If `BufferSize` is exposed in a public API, it can easily be misinterpreted to mean more than just `int`. Generally, domain types have multiple attributes to them and are not primitive types like `int`. This abbreviation violates that assumption.
- The casing of `BufferSize` (PascalCase) implies that this type holds more data.
- This alias does not offer increased clarity compared with providing a named argument to a function.
- The abbreviation will not manifest in compiled IL; it is just an integer and this alias is a compile-time construct.

```
module Networking =
    ...
    let send data (bufferSize: int) = ...
```

In summary, the pitfall with Type Abbreviations is that they are **not** abstractions over the types they are abbreviating. In the previous example, `BufferSize` is just an `int` under the covers, with no extra data, nor any benefits from the type system besides what `int` already has.

An alternative approach to using type abbreviations to represent a domain is to use single-case discriminated unions. The previous sample can be modeled as follows:

```
type BufferSize = BufferSize of int
```

If you write code that operates in terms of `BufferSize` and its underlying value, you need to construct one rather than pass in any arbitrary integer:

```
module Networking =
    ...
    let send data (BufferSize size) =
    ...
```

This reduces the likelihood of mistakenly passing an arbitrary integer into the `send` function, because the caller must construct a `BufferSize` type to wrap a value before calling the function.

F# component design guidelines

8/5/2022 • 27 minutes to read • Edit Online

This document is a set of component design guidelines for F# programming, based on the F# Component Design Guidelines, v14, Microsoft Research, and a version that was originally curated and maintained by the F# Software Foundation.

This document assumes you are familiar with F# programming. Many thanks to the F# community for their contributions and helpful feedback on various versions of this guide.

Overview

This document looks at some of the issues related to F# component design and coding. A component can mean any of the following:

- A layer in your F# project that has external consumers within that project.
- A library intended for consumption by F# code across assembly boundaries.
- A library intended for consumption by any .NET language across assembly boundaries.
- A library intended for distribution via a package repository, such as NuGet.

Techniques described in this article follow the Five principles of good F# code, and thus utilize both functional and object programming as appropriate.

Regardless of the methodology, the component and library designer faces a number of practical and prosaic issues when trying to craft an API that is most easily usable by developers. Conscientious application of the .NET Library Design Guidelines will steer you towards creating a consistent set of APIs that are pleasant to consume.

General guidelines

There are a few universal guidelines that apply to F# libraries, regardless of the intended audience for the library.

Learn the .NET Library Design Guidelines

Regardless of the kind of F# coding you are doing, it is valuable to have a working knowledge of the .NET Library Design Guidelines. Most other F# and .NET programmers will be familiar with these guidelines, and expect .NET code to conform to them.

The .NET Library Design Guidelines provide general guidance regarding naming, designing classes and interfaces, member design (properties, methods, events, etc.) and more, and are a useful first point of reference for a variety of design guidance.

Add XML documentation comments to your code

XML documentation on public APIs ensures that users can get great Intellisense and Quickinfo when using these types and members, and enable building documentation files for the library. See the XML Documentation about various xml tags that can be used for additional markup within xmldoc comments.

```
/// A class for representing (x,y) coordinates
type Point =

    /// Computes the distance between this point and another
    member DistanceTo: otherPoint:Point -> float
```

You can use either the short form XML comments (`/// comment`), or standard XML comments (`///<summary>comment</summary>`).

Consider using explicit signature files (.fsi) for stable library and component APIs

Using explicit signatures files in an F# library provides a succinct summary of public API, which helps to ensure that you know the full public surface of your library, and provides a clean separation between public documentation and internal implementation details. Signature files add friction to changing the public API, by requiring changes to be made in both the implementation and signature files. As a result, signature files should typically only be introduced when an API has become solidified and is no longer expected to change significantly.

Always follow best practices for using strings in .NET

Follow Best Practices for Using Strings in .NET guidance. In particular, always explicitly state *cultural intent* in the conversion and comparison of strings (where applicable).

Guidelines for F#-facing libraries

This section presents recommendations for developing public F#-facing libraries; that is, libraries exposing public APIs that are intended to be consumed by F# developers. There are a variety of library-design recommendations applicable specifically to F#. In the absence of the specific recommendations that follow, the .NET Library Design Guidelines are the fallback guidance.

Naming conventions

Use .NET naming and capitalization conventions

The following table follows .NET naming and capitalization conventions. There are small additions to also include F# constructs.

CONSTRUCT	CASE	PART	EXAMPLES	NOTES
Concrete types	PascalCase	Noun/ adjective	List, Double, Complex	Concrete types are structs, classes, enumerations, delegates, records, and unions. Though type names are traditionally lowercase in OCaml, F# has adopted the .NET naming scheme for types.
DLLs	PascalCase		Fabrikam.Core.dll	
Union tags	PascalCase	Noun	Some, Add, Success	Do not use a prefix in public APIs. Optionally use a prefix when internal, such as `type Teams = TAlpha | TBeta | TDelta.`
Event	PascalCase	Verb	ValueChanged / ValueChanging	
Exceptions	PascalCase		WebException	Name should end with "Exception".

CONSTRUCT	CASE	PART	EXAMPLES	NOTES
Field	PascalCase	Noun	CurrentName	
Interface types	PascalCase	Noun/ adjective	IDisposable	Name should start with "I".
Method	PascalCase	Verb	ToString	
Namespace	PascalCase		Microsoft.FSharp.Core	Generally use `<Organization>.<Technology>[.<Subnamespace>]`, though drop the organization if the technology is independent of organization.
Parameters	camelCase	Noun	typeName, transform, range	
let values (internal)	camelCase or PascalCase	Noun/ verb	getValue, myTable	
let values (external)	camelCase or PascalCase	Noun/verb	List.map, Dates.Today	let-bound values are often public when following traditional functional design patterns. However, generally use PascalCase when the identifier can be used from other .NET languages.
Property	PascalCase	Noun/ adjective	IsEndOfFile, BackColor	Boolean properties generally use Is and Can and should be affirmative, as in IsEndOfFile, not IsNotEndOfFile.

Avoid abbreviations

The .NET guidelines discourage the use of abbreviations (for example, "use `OnButtonClick` rather than `OnBtnClick`"). Common abbreviations, such as `Async` for "Asynchronous", are tolerated. This guideline is sometimes ignored for functional programming; for example, `List.iter` uses an abbreviation for "iterate". For this reason, using abbreviations tends to be tolerated to a greater degree in F#-to-F# programming, but should still generally be avoided in public component design.

Avoid casing name collisions

The .NET guidelines say that casing alone cannot be used to disambiguate name collisions, since some client languages (for example, Visual Basic) are case-insensitive.

Use acronyms where appropriate

Acronyms such as XML are not abbreviations and are widely used in .NET libraries in uncapitalized form (Xml). Only well-known, widely recognized acronyms should be used.

Use PascalCase for generic parameter names

Do use PascalCase for generic parameter names in public APIs, including for F#-facing libraries. In particular, use names like `T`, `U`, `T1`, `T2` for arbitrary generic parameters, and when specific names make sense, then for F#-facing libraries use names like `Key`, `Value`, `Arg` (but not for example, `TKey`).

Use either PascalCase or camelCase for public functions and values in F# modules

camelCase is used for public functions that are designed to be used unqualified (for example, `invalidArg`), and for the "standard collection functions" (for example, List.map). In both these cases, the function names act much like keywords in the language.

Object, Type, and Module design

Use namespaces or modules to contain your types and modules

Each F# file in a component should begin with either a namespace declaration or a module declaration.

```
namespace Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
     ...

module CommonOperations =
    ...
```

or

```
module Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...
```

The differences between using modules and namespaces to organize code at the top level are as follows:

- Namespaces can span multiple files
- Namespaces cannot contain F# functions unless they are within an inner module
- The code for any given module must be contained within a single file
- Top-level modules can contain F# functions without the need for an inner module

The choice between a top-level namespace or module affects the compiled form of the code, and thus will affect the view from other .NET languages should your API eventually be consumed outside of F# code.

Use methods and properties for operations intrinsic to object types

When working with objects, it is best to ensure that consumable functionality is implemented as methods and properties on that type.

```
type HardwareDevice() =

    member this.ID = ...

    member this.SupportedProtocols = ...

type HashTable<'Key,'Value>(comparer: IEqualityComparer<'Key>) =

    member this.Add(key, value) = ...

    member this.ContainsKey(key) = ...

    member this.ContainsValue(value) = ...
```

The bulk of functionality for a given member need not necessarily be implemented in that member, but the consumable piece of that functionality should be.

Use classes to encapsulate mutable state

In F#, this only needs to be done where that state is not already encapsulated by another language construct, such as a closure, sequence expression, or asynchronous computation.

```
type Counter() =
    // let-bound values are private in classes.
    let mutable count = 0

    member this.Next() =
        count <- count + 1
        count
```

Use interfaces to group related operations

Use interface types to represent a set of operations. This is preferred to other options, such as tuples of functions or records of functions.

```
type Serializer =
    abstract Serialize<'T> : preserveRefEq: bool -> value: 'T -> string
    abstract Deserialize<'T> : preserveRefEq: bool -> pickle: string -> 'T
```

In preference to:

```
type Serializer<'T> = {
    Serialize: bool -> 'T -> string
    Deserialize: bool -> string -> 'T
}
```

Interfaces are first-class concepts in .NET, which you can use to achieve what Functors would normally give you. Additionally, they can be used to encode existential types into your program, which records of functions cannot.

Use a module to group functions that act on collections

When you define a collection type, consider providing a standard set of operations like `CollectionType.map` and `CollectionType.iter`) for new collection types.

```
module CollectionType =
    let map f c =
        ...
    let iter f c =
        ...
```

If you include such a module, follow the standard naming conventions for functions found in FSharp.Core.

Use a module to group functions for common, canonical functions, especially in math and DSL libraries

For example, `Microsoft.FSharp.Core.Operators` is an automatically opened collection of top-level functions (like `abs` and `sin`) provided by FSharp.Core.dll.

Likewise, a statistics library might include a module with functions `erf` and `erfc`, where this module is designed to be explicitly or automatically opened.

Consider using RequireQualifiedAccess and carefully apply AutoOpen attributes

Adding the `[<RequireQualifiedAccess>]` attribute to a module indicates that the module may not be opened and that references to the elements of the module require explicit qualified access. For example, the `Microsoft.FSharp.Collections.List` module has this attribute.

This is useful when functions and values in the module have names that are likely to conflict with names in other modules. Requiring qualified access can greatly increase the long-term maintainability and evolvability of a library.

Adding the `[<AutoOpen>]` attribute to a module means the module will be opened when the containing namespace is opened. The `[<AutoOpen>]` attribute may also be applied to an assembly to indicate a module that is automatically opened when the assembly is referenced.

For example, a statistics library **MathsHeaven.Statistics** might contain a `module MathsHeaven.Statistics.Operators` containing functions `erf` and `erfc`. It is reasonable to mark this module as `[<AutoOpen>]`. This means `open MathsHeaven.Statistics` will also open this module and bring the names `erf` and `erfc` into scope. Another good use of `[<AutoOpen>]` is for modules containing extension methods.

Overuse of `[<AutoOpen>]` leads to polluted namespaces, and the attribute should be used with care. For specific libraries in specific domains, judicious use of `[<AutoOpen>]` can lead to better usability.

Consider defining operator members on classes where using well-known operators is appropriate

Sometimes classes are used to model mathematical constructs such as Vectors. When the domain being modeled has well-known operators, defining them as members intrinsic to the class is helpful.

```
type Vector(x: float) =

    member v.X = x

    static member (*) (vector: Vector, scalar: float) = Vector(vector.X * scalar)

    static member (+) (vector1: Vector, vector2: Vector) = Vector(vector1.X + vector2.X)

let v = Vector(5.0)

let u = v * 10.0
```

This guidance corresponds to general .NET guidance for these types. However, it can be additionally important in F# coding as this allows these types to be used in conjunction with F# functions and methods with member constraints, such as List.sumBy.

Consider using CompiledName to provide a .NET-friendly name for other .NET language consumers

Sometimes you may wish to name something in one style for F# consumers (such as a static member in lower case so that it appears as if it were a module-bound function), but have a different style for the name when it is compiled into an assembly. You can use the `[<CompiledName>]` attribute to provide a different style for non F# code consuming the assembly.

```
type Vector(x:float, y:float) =

    member v.X = x
    member v.Y = y

    [<CompiledName("Create")>]
    static member create x y = Vector (x, y)

let v = Vector.create 5.0 3.0
```

By using `[<CompiledName>]`, you can use .NET naming conventions for non F# consumers of the assembly.

Use method overloading for member functions, if doing so provides a simpler API

Method overloading is a powerful tool for simplifying an API that may need to perform similar functionality, but with different options or arguments.

```
type Logger() =

    member this.Log(message) =
        ...
    member this.Log(message, retryPolicy) =
        ...
```

In F#, it is more common to overload on number of arguments rather than types of arguments.

Hide the representations of record and union types if the design of these types is likely to evolve

Avoid revealing concrete representations of objects. For example, the concrete representation of DateTime values is not revealed by the external, public API of the .NET library design. At run time, the Common Language Runtime knows the committed implementation that will be used throughout execution. However, compiled code doesn't itself pick up dependencies on the concrete representation.

Avoid the use of implementation inheritance for extensibility

In F#, implementation inheritance is rarely used. Furthermore, inheritance hierarchies are often complex and difficult to change when new requirements arrive. Inheritance implementation still exists in F# for compatibility and rare cases where it is the best solution to a problem, but alternative techniques should be sought in your F# programs when designing for polymorphism, such as interface implementation.

Function and member signatures

Use tuples for return values when returning a small number of multiple unrelated values

Here is a good example of using a tuple in a return type:

```
val divrem: BigInteger -> BigInteger -> BigInteger * BigInteger
```

For return types containing many components, or where the components are related to a single identifiable entity, consider using a named type instead of a tuple.

Use `Async<T>` for async programming at F# API boundaries

If there is a corresponding synchronous operation named `Operation` that returns a `T`, then the async operation should be named `AsyncOperation` if it returns `Async<T>` or `OperationAsync` if it returns `Task<T>`. For commonly used .NET types that expose Begin/End methods, consider using `Async.FromBeginEnd` to write extension methods as a façade to provide the F# async programming model to those .NET APIs.

```fsharp
type SomeType =
    member this.Compute(x:int): int =
        ...
    member this.AsyncCompute(x:int): Async<int> =
        ...

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        ...
```

Exceptions

See Error Management to learn about appropriate use of exceptions, results, and options.

Extension Members

Carefully apply F# extension members in F#-to-F# components

F# extension members should generally only be used for operations that are in the closure of intrinsic operations associated with a type in the majority of its modes of use. One common use is to provide APIs that are more idiomatic to F# for various .NET types:

```fsharp
type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

type System.Collections.Generic.IDictionary<'Key,'Value> with
    member this.TryGet key =
        let ok, v = this.TryGetValue key
        if ok then Some v else None
```

Union Types

Use discriminated unions instead of class hierarchies for tree-structured data

Tree-like structures are recursively defined. This is awkward with inheritance, but elegant with Discriminated Unions.

```fsharp
type BST<'T> =
    | Empty
    | Node of 'T * BST<'T> * BST<'T>
```

Representing tree-like data with Discriminated Unions also allows you to benefit from exhaustiveness in pattern matching.

Use `[<RequireQualifiedAccess>]` on union types whose case names are not sufficiently unique

You may find yourself in a domain where the same name is the best name for different things, such as Discriminated Union cases. You can use `[<RequireQualifiedAccess>]` to disambiguate case names in order to avoid triggering confusing errors due to shadowing dependent on the ordering of `open` statements

Hide the representations of discriminated unions for binary compatible APIs if the design of these types is likely to evolve

Unions types rely on F# pattern-matching forms for a succinct programming model. As mentioned previously, you should avoid revealing concrete data representations if the design of these types is likely to evolve.

For example, the representation of a discriminated union can be hidden using a private or internal declaration, or by using a signature file.

```
type Union =
    private
    | CaseA of int
    | CaseB of string
```

If you reveal discriminated unions indiscriminately, you may find it hard to version your library without breaking user code. Instead, consider revealing one or more active patterns to permit pattern matching over values of your type.

Active patterns provide an alternate way to provide F# consumers with pattern matching while avoiding exposing F# Union Types directly.

Inline Functions and Member Constraints

Define generic numeric algorithms using inline functions with implied member constraints and statically resolved generic types

Arithmetic member constraints and F# comparison constraints are a standard for F# programming. For example, consider the following code:

```
let inline highestCommonFactor a b =
    let rec loop a b =
        if a = LanguagePrimitives.GenericZero<_> then b
        elif a < b then loop a (b - a)
        else loop (a - b) b
    loop a b
```

The type of this function is as follows:

```
val inline highestCommonFactor : ^T -> ^T -> ^T
                when ^T : (static member Zero : ^T)
                and ^T : (static member ( - ) : ^T * ^T -> ^T)
                and ^T : equality
                and ^T : comparison
```

This is a suitable function for a public API in a mathematical library.

Avoid using member constraints to simulate type classes and duck typing

It is possible to simulate "duck typing" using F# member constraints. However, members that make use of this should not in general be used in F#-to-F# library designs. This is because library designs based on unfamiliar or non-standard implicit constraints tend to cause user code to become inflexible and tied to one particular framework pattern.

Additionally, there is a good chance that heavy use of member constraints in this manner can result in very long compile times.

Operator Definitions

Avoid defining custom symbolic operators

Custom operators are essential in some situations and are highly useful notational devices within a large body of implementation code. For new users of a library, named functions are often easier to use. In addition, custom symbolic operators can be hard to document, and users find it more difficult to look up help on operators, due to existing limitations in IDE and search engines.

As a result, it is best to publish your functionality as named functions and members, and additionally expose operators for this functionality only if the notational benefits outweigh the documentation and cognitive cost of having them.

Units of Measure

Carefully use units of measure for added type safety in F# code

Additional typing information for units of measure is erased when viewed by other .NET languages. Be aware that .NET components, tools, and reflection will see types-sans-units. For example, C# consumers will see `float` rather than `float<kg>`.

Type Abbreviations

Carefully use type abbreviations to simplify F# code

.NET components, tools, and reflection will not see abbreviated names for types. Significant usage of type abbreviations can also make a domain appear more complex than it actually is, which could confuse consumers.

Avoid type abbreviations for public types whose members and properties should be intrinsically different to those available on the type being abbreviated

In this case, the type being abbreviated reveals too much about the representation of the actual type being defined. Instead, consider wrapping the abbreviation in a class type or a single-case discriminated union (or, when performance is essential, consider using a struct type to wrap the abbreviation).

For example, it is tempting to define a multi-map as a special case of an F# map, for example:

```
type MultiMap<'Key,'Value> = Map<'Key,'Value list>
```

However, the logical dot-notation operations on this type are not the same as the operations on a Map – for example, it is reasonable that the lookup operator `map[key]` return the empty list if the key is not in the dictionary, rather than raising an exception.

Guidelines for libraries for Use from other .NET Languages

When designing libraries for use from other .NET languages, it is important to adhere to the .NET Library Design Guidelines. In this document, these libraries are labeled as vanilla .NET libraries, as opposed to F#-facing libraries that use F# constructs without restriction. Designing vanilla .NET libraries means providing familiar and idiomatic APIs consistent with the rest of the .NET Framework by minimizing the use of F#-specific constructs in the public API. The rules are explained in the following sections.

Namespace and Type design (for libraries for use from other .NET Languages)

Apply the .NET naming conventions to the public API of your components

Pay special attention to the use of abbreviated names and the .NET capitalization guidelines.

```
type pCoord = ...
    member this.theta = ...

type PolarCoordinate = ...
    member this.Theta = ...
```

Use namespaces, types, and members as the primary organizational structure for your components

All files containing public functionality should begin with a `namespace` declaration, and the only public-facing entities in namespaces should be types. Do not use F# modules.

Use non-public modules to hold implementation code, utility types, and utility functions.

Static types should be preferred over modules, as they allow for future evolution of the API to use overloading and other .NET API design concepts that may not be used within F# modules.

For example, in place of the following public API:

```
module Fabrikam

module Utilities =
    let Name = "Bob"
    let Add2 x y = x + y
    let Add3 x y z = x + y + z
```

Consider instead:

```
namespace Fabrikam

[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
```

Use F# record types in vanilla .NET APIs if the design of the types won't evolve

F# record types compile to a simple .NET class. These are suitable for some simple, stable types in APIs. Consider using the `[<NoEquality>]` and `[<NoComparison>]` attributes to suppress the automatic generation of interfaces. Also avoid using mutable record fields in vanilla .NET APIs as these expose a public field. Always consider whether a class would provide a more flexible option for future evolution of the API.

For example, the following F# code exposes the public API to a C# consumer:

F#:

```
[<NoEquality; NoComparison>]
type MyRecord =
    { FirstThing: int
        SecondThing: string }
```

C#:

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

Hide the representation of F# union types in vanilla .NET APIs

F# union types are not commonly used across component boundaries, even for F#-to-F# coding. They are an excellent implementation device when used internally within components and libraries.

When designing a vanilla .NET API, consider hiding the representation of a union type by using either a private declaration or a signature file.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True
```

You may also augment types that use a union representation internally with members to provide a desired .NET-facing API.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

    /// A public member for use from C#
    member x.Evaluate =
        match x with
        | And(a,b) -> a.Evaluate && b.Evaluate
        | Not a -> not a.Evaluate
        | True -> true

    /// A public member for use from C#
    static member CreateAnd(a,b) = And(a,b)
```

Design GUI and other components using the design patterns of the framework

There are many different frameworks available within .NET, such as WinForms, WPF, and ASP.NET. Naming and design conventions for each should be used if you are designing components for use in these frameworks. For example, for WPF programming, adopt WPF design patterns for the classes you are designing. For models in user interface programming, use design patterns such as events and notification-based collections such as those found in System.Collections.ObjectModel.

Object and Member design (for libraries for use from other .NET Languages)

Use the CLIEvent attribute to expose .NET events

Construct a `DelegateEvent` with a specific .NET delegate type that takes an object and `EventArgs` (rather than an `Event`, which just uses the `FSharpHandler` type by default) so that the events are published in the familiar way to other .NET languages.

```
type MyBadType() =
    let myEv = new Event<int>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

type MyEventArgs(x: int) =
    inherit System.EventArgs()
    member this.X = x

    /// A type in a component designed for use from other .NET languages
type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

Expose asynchronous operations as methods that return .NET tasks

Tasks are used in .NET to represent active asynchronous computations. Tasks are in general less compositional than F# `Async<T>` objects, since they represent "already executing" tasks and can't be composed together in ways that perform parallel composition, or which hide the propagation of cancellation signals and other contextual parameters.

However, despite this, methods that return Tasks are the standard representation of asynchronous programming on .NET.

```
/// A type in a component designed for use from other .NET languages
type MyType() =

    let compute (x: int): Async<int> = async { ... }

    member this.ComputeAsync(x) = compute x |> Async.StartAsTask
```

You will frequently also want to accept an explicit cancellation token:

```
/// A type in a component designed for use from other .NET languages
type MyType() =
    let compute(x: int): Async<int> = async { ... }
    member this.ComputeAsTask(x, cancellationToken) = Async.StartAsTask(compute x, cancellationToken)
```

Use .NET delegate types instead of F# function types

Here "F# function types" mean "arrow" types like `int -> int`.

Instead of this:

```
member this.Transform(f: int->int) =
    ...
```

Do this:

```
member this.Transform(f: Func<int,int>) =
    ...
```

The F# function type appears as `class FSharpFunc<T,U>` to other .NET languages, and is less suitable for language features and tooling that understands delegate types. When authoring a higher-order method targeting .NET Framework 3.5 or higher, the `System.Func` and `System.Action` delegates are the right APIs to publish to enable .NET developers to consume these APIs in a low-friction manner. (When targeting .NET Framework 2.0, the system-defined delegate types are more limited; consider using predefined delegate types such as `System.Converter<T,U>` or defining a specific delegate type.)

On the flip side, .NET delegates are not natural for F#-facing libraries (see the next Section on F#-facing libraries). As a result, a common implementation strategy when developing higher-order methods for vanilla .NET libraries is to author all the implementation using F# function types, and then create the public API using delegates as a thin façade atop the actual F# implementation.

Use the TryGetValue pattern instead of returning F# option values, and prefer method overloading to taking F# option values as arguments

Common patterns of use for the F# option type in APIs are better implemented in vanilla .NET APIs using standard .NET design techniques. Instead of returning an F# option value, consider using the bool return type plus an out parameter as in the "TryGetValue" pattern. And instead of taking F# option values as parameters, consider using method overloading or optional arguments.

```
member this.ReturnOption() = Some 3

member this.ReturnBoolAndOut(outVal: byref<int>) =
    outVal <- 3
    true

member this.ParamOption(x: int, y: int option) =
    match y with
    | Some y2 -> x + y2
    | None -> x

member this.ParamOverload(x: int) = x

member this.ParamOverload(x: int, y: int) = x + y
```

Use the .NET collection interface types IEnumerable<T> and IDictionary<Key,Value> for parameters and return values

Avoid the use of concrete collection types such as .NET arrays `T[]`, F# types `list<T>`, `Map<Key,Value>` and `Set<T>`, and .NET concrete collection types such as `Dictionary<Key,Value>`. The .NET Library Design Guidelines have good advice regarding when to use various collection types like `IEnumerable<T>`. Some use of arrays (`T[]`) is acceptable in some circumstances, on performance grounds. Note especially that `seq<T>` is just the F# alias for `IEnumerable<T>`, and thus seq is often an appropriate type for a vanilla .NET API.

Instead of F# lists:

```
member this.PrintNames(names: string list) =
    ...
```

Use F# sequences:

```
member this.PrintNames(names: seq<string>) =
    ...
```

Use the unit type as the only input type of a method to define a zero-argument method, or as the only return type to define a void-returning method

Avoid other uses of the unit type. These are good:

```
✔ member this.NoArguments() = 3

✔ member this.ReturnVoid(x: int) = ()
```

This is bad:

```
member this.WrongUnit( x: unit, z: int) = ((), ())
```

Check for null values on vanilla .NET API boundaries

F# implementation code tends to have fewer null values, due to immutable design patterns and restrictions on use of null literals for F# types. Other .NET languages often use null as a value much more frequently. Because of this, F# code that is exposing a vanilla .NET API should check parameters for null at the API boundary, and prevent these values from flowing deeper into the F# implementation code. The `isNull` function or pattern matching on the `null` pattern can be used.

```
let checkNonNull argName (arg: obj) =
    match arg with
    | null -> nullArg argName
    | _ -> ()

let checkNonNull` argName (arg: obj) =
    if isNull arg then nullArg argName
    else ()
```

Avoid using tuples as return values

Instead, prefer returning a named type holding the aggregate data, or using out parameters to return multiple values. Although tuples and struct tuples exist in .NET (including C# language support for struct tuples), they will most often not provide the ideal and expected API for .NET developers.

Avoid the use of currying of parameters

Instead, use .NET calling conventions `Method(arg1,arg2,…,argN)`.

```
member this.TupledArguments(str, num) = String.replicate num str
```

Tip: If you're designing libraries for use from any .NET language, then there's no substitute for actually doing some experimental C# and Visual Basic programming to ensure that your libraries "feel right" from these languages. You can also use tools such as .NET Reflector and the Visual Studio Object Browser to ensure that libraries and their documentation appear as expected to developers.

Appendix

End-to-end example of designing F# code for use by other .NET languages

Consider the following class:

```
open System

type Point1(angle,radius) =
    new() = Point1(angle=0.0, radius=0.0)
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
```

The inferred F# type of this class is as follows:

```
type Point1 =
    new : unit -> Point1
    new : angle:double * radius:double -> Point1
    static member Circle : n:int -> Point1 list
    member Stretch : l:double -> Point1
    member Warp : f:(double -> double) -> Point1
    member Angle : double
    member Radius : double
```

Let's take a look at how this F# type appears to a programmer using another .NET language. For example, the approximate C# "signature" is as follows:

```
// C# signature for the unadjusted Point1 class
public class Point1
{
    public Point1();

    public Point1(double angle, double radius);

    public static Microsoft.FSharp.Collections.List<Point1> Circle(int count);

    public Point1 Stretch(double factor);

    public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

There are some important points to notice about how F# represents constructs here. For example:

- Metadata such as argument names has been preserved.
- F# methods that take two arguments become C# methods that take two arguments.
- Functions and lists become references to corresponding types in the F# library.

The following code shows how to adjust this code to take these things into account.

```
namespace SuperDuperFSharpLibrary.Types

type RadialPoint(angle:double, radius:double) =

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,_>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
                RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }
```

The inferred F# type of the code is as follows:

```
type RadialPoint =
    new : unit -> RadialPoint
    new : angle:double * radius:double -> RadialPoint
    static member Circle : count:int -> seq<RadialPoint>
    member Stretch : factor:double -> RadialPoint
    member Warp : transform:System.Func<double,double> -> RadialPoint
    member Angle : double
    member Radius : double
```

The C# signature is now as follows:

```
public class RadialPoint
{
    public RadialPoint();

    public RadialPoint(double angle, double radius);

    public static System.Collections.Generic.IEnumerable<RadialPoint> Circle(int count);

    public RadialPoint Stretch(double factor);

    public RadialPoint Warp(System.Func<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

The fixes made to prepare this type for use as part of a vanilla .NET library are as follows:

- Adjusted several names: `Point1`, `n`, `l`, and `f` became `RadialPoint`, `count`, `factor`, and `transform`, respectively.
- Used a return type of `seq<RadialPoint>` instead of `RadialPoint list` by changing a list construction using `[ ... ]` to a sequence construction using `IEnumerable<RadialPoint>`.
- Used the .NET delegate type `System.Func` instead of an F# function type.

This makes it far nicer to consume in C# code.

Machine Learning with F#

8/5/2022 • 2 minutes to read • Edit Online

F# excels at data science and machine learning. This article gives links to some significant resources related to this mode of use of F#.

For information about other options that are available for machine learning and data science, see the F# Software Foundation's Guide to Data Science with F#.

ML.NET

ML.NET is an open source and cross-platform machine learning framework built for .NET developers. With ML.NET, you can create custom ML models using C# or F# without having to leave the .NET ecosystem. ML.NET lets you reuse all the knowledge, skills, code, and libraries you already have as a .NET developer so that you can easily integrate machine learning into your web, mobile, desktop, games, and IoT apps.

Deep Learning with TorchSharp

TorchSharp is an open source set of bindings for the Pytorch engine usable for deep-learning from F#. Examples in F# are available in TorchSharpExamples.

FsLab

FsLab is an F# community incubation space for data science with F#.

See also

- F# Notebooks
- A Guide to Data Access with F#
- A Guide to Data Science with F#

F# for Web Development

8/5/2022 • 2 minutes to read • Edit Online

F# excels at building efficient, scalable, and robust web solutions. This article gives links to some significant resources related to web programming with F#. Some frameworks for web programming with F# are listed below.

Other web development options are documented in the F# Software Foundation's Guide to Web Programming with F#.

ASP.NET Core

ASP.NET Core is a modern, cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. It runs on .NET Core and supports F# out of the box. If you install the .NET SDK, there are F# templates available via the `dotnet new` command.

Giraffe

Giraffe is a community-driven F# library for building rich web applications with superb performance. It has been specifically designed with ASP.NET Core in mind and can be added into ASP.NET Core pipelines.

Saturn

Saturn is a community-driven F# web development framework that implements the server-side MVC pattern. Many of its components and concepts will seem familiar to anyone with experience in other web frameworks like Ruby on Rails or Python’s Django. It’s built on top of Giraffe and ASP.NET Core - a modern, cross-platform, high-performance development platform for building cloud-ready web applications.

Fable

Fable is a compiler that brings F# into the JavaScript ecosystem. It generates modern JavaScript output, interoperates with JavaScript packages, and supports multiple development models including React.

SAFE Stack

SAFE Stack is a community-driven technology stack for functional-first web applications using Azure. SAFE Stack allows you to quickly develop compelling web applications that use industry-standard technologies whilst using F# to ensure an enjoyable development experience. SAFE includes Giraffe, Saturn, and other components.

WebSharper

WebSharper is a community-driven, full-stack, functional reactive web programming technology for .NET, allowing you to develop microservices, client-server web applications, reactive SPAs, and more in F#.

Falco

Falco is a community-driven toolkit for building *fast*, functional-first, and fault-tolerant web applications using F#. It's built upon the high-performance components of ASP.NET Core and is optimized for building HTTP applications quickly. Falco has a built-in view engine and seamlessly integrates with existing .NET Core middleware and frameworks.

See also

- F# for JavaScript
- A Guide to Web Programming with F#

Using Apache Spark with F# on Azure

8/5/2022 • 2 minutes to read • Edit Online

Apache Spark for Azure HDInsight is an open source processing framework that runs large-scale data analytics applications. Azure Databricks is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Azure makes Apache Spark easy and cost effective to deploy. Develop your Spark application in F# using .NET for Apache Spark, a set of .NET bindings for Apache Spark.

- .NET for Apache Spark F# samples
- Install .NET Interactive Jupyter notebooks in Azure HDInsight
- Submit Apache Spark jobs to Azure HDInsight
- Submit Apache Spark jobs to Azure Databricks

Other resources

- Apache Spark for Azure HDInsight
- Full documentation on all Azure services

Deploying and Managing Azure Resources with F#

8/5/2022 • 2 minutes to read • Edit Online

F# may be used to configure, deploy, and manage Azure resources. This articles provides links for the F#-specific technology Farmer.

Other options for Azure resource deployment and management are documented in Azure Resource Manager. This includes the Bicep language.

Farmer

For F# programmers, Farmer is a free, open-source, community-driven technology providing an easy-to-learn library for rapidly authoring and deploying entire Azure architectures. Farmer generates Azure Resource Manager (ARM) templates that define the infrastructure and configuration for your project.

Farmer provides simple code snippets that allow you to rapidly construct complex topologies and idempotent deployments. Farmer is cross-platform and completely backwards compatible with ARM templates. Farmer generates standard ARM templates so you can continue to use existing deployment processes.

See also

- Azure Resource Manager
- Bicep
- Farmer

Get started with Azure Blob Storage using F#

8/5/2022 • 8 minutes to read • Edit Online

Azure Blob Storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This article shows you how to perform common tasks using Blob storage. The samples are written using F# using the Azure Storage Client Library for .NET. The tasks covered include how to upload, list, download, and delete blobs.

For a conceptual overview of blob storage, see the .NET guide for blob storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You also need your storage access key for this account.

Create an F# script and start F# interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `blobs.fsx`, in your F# development environment.

How to execute scripts

F# Interactive, `dotnet fsi`, can be launched interactively, or it can be launched from the command line to run a script. The command-line syntax is

```
> dotnet fsi [options] [ script-file [arguments] ]
```

Add packages in a script

Next, use `#r` `nuget:package name` to install the `Azure.Storage.Blobs` package and `open` namespaces.Such as

```
> #r "nuget: Azure.Storage.Blobs"
open Azure.Storage.Blobs
open Azure.Storage.Blobs.Models
open Azure.Storage.Blobs.Specialized
```

Add namespace declarations

Add the following `open` statements to the top of the `blobs.fsx` file:

```
open System
open System.IO
open Azure.Storage.Blobs // Namespace for Blob storage types
open Azure.Storage.Blobs.Models
open Azure.Storage.Blobs.Specialized
open System.Text
```

Get your connection string

You need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

Create some local dummy data

Before you begin, create some dummy local data in the directory of our script. Later you upload this data.

```
// Create a dummy file to upload
let localFile = "./myfile.txt"
File.WriteAllText(localFile, "some data")
```

Create the blob service client

The `BlobContainerClient` type enables you to create containers and retrieve blobs stored in Blob storage. Here's one way to create the container client:

```
let container = BlobContainerClient(storageConnString, "myContainer")
```

Now you are ready to write code that reads data from and writes data to Blob storage.

Create a container

This example shows how to create a container if it does not already exist:

```
container.CreateIfNotExists()
```

By default, the new container is private, meaning that you must specify your storage access key to download blobs from this container. If you want to make the files within the container available to everyone, you can set the container to be public using the following code:

```
let permissions = PublicAccessType.Blob
container.SetAccessPolicy(permissions)
```

Anyone on the Internet can see blobs in a public container, but you can modify or delete them only if you have the appropriate account access key or a shared access signature.

Upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. In most cases, a block blob is the recommended type to use.

To upload a file to a block blob, get a container client and use it to get a block blob reference. Once you have a blob reference, you can upload any stream of data to it by calling the `Upload` method. This operation overwrites the contents of the blob, creating a new block blob if none exists.

```
// Retrieve reference to a blob named "myblob.txt".
let blockBlob = container.GetBlobClient("myblob.txt")

// Create or overwrite the "myblob.txt" blob with contents from the local file.
use fileStream = new FileStream(localFile, FileMode.Open, FileAccess.Read, FileShare.Read)
do blockBlob.Upload(fileStream)
```

List the blobs in a container

To list the blobs in a container, first get a container reference. You can then use the container's `GetBlobs` method to retrieve the blobs and/or directories within it. To access the rich set of properties and methods for a returned `BlobItem`.

```
for item in container.GetBlobsByHierarchy() do
    printfn $"Blob name: {item.Blob.Name}"
```

For example, consider the following set of block blobs in a container named `photos`:

photo1.jpg
2015/architecture/description.txt
2015/architecture/photo3.jpg
2015/architecture/photo4.jpg
2016/architecture/photo5.jpg
2016/architecture/photo6.jpg
2016/architecture/description.txt
2016/photo7.jpg\

When you call `GetBlobsByHierarchy` on a container (as in the above sample), a hierarchical listing is returned.

```
Directory: https://<accountname>.blob.core.windows.net/photos/2015/
Directory: https://<accountname>.blob.core.windows.net/photos/2016/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Download blobs

To download blobs, first retrieve a blob reference and then call the `DownloadTo` method. The following example uses the `DownloadTo` method to transfer the blob contents to a stream object that you can then persist to a local file.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDownload = container.GetBlobClient("myblob.txt")

// Save blob contents to a file.
do
    use fileStream = File.OpenWrite("path/download.txt")
    blobToDownload.DownloadTo(fileStream)
```

You can also use the `DownloadContent` method to download the contents of a blob as a text string.

```
let text = blobToDownload.DownloadContent().Value.Content.ToString()
```

Delete blobs

To delete a blob, first get a blob reference and then call the `Delete` method on it.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDelete = container.GetBlobClient("myblob.txt")

// Delete the blob.
blobToDelete.Delete()
```

List blobs in pages asynchronously

If you are listing a large number of blobs, or you want to control the number of results you return in one listing operation, you can list blobs in pages of results. This example shows how to return results in pages.

This example shows a hierarchical listing, by using the `GetBlobsByHierarchy` method of the `BlobClient` .

```
let ListBlobsSegmentedInHierarchicalListing(container:BlobContainerClient) =
        // List blobs to the console window, with paging.
        printfn "List blobs in pages:"

        // Call GetBlobsByHierarchy to return an async collection
        // of blobs in this container. AsPages() method enumerate the values
        //a Page<T> at a time. This may make multiple service requests.

        for page in container.GetBlobsByHierarchy().AsPages() do
            for blobHierarchyItem in page.Values do
                printf $"The BlobItem is : {blobHierarchyItem.Blob.Name} "

        printfn ""
```

We can now use this hierarchical listing routine as follows. First, upload some dummy data (using the local file created earlier in this tutorial).

```
for i in 1 .. 100 do
    let blob  = container.GetBlobClient($"myblob{i}.txt")
    use fileStream = System.IO.File.OpenRead(localFile)
    blob.Upload(localFile)
```

Now, call the routine.

```
ListBlobsSegmentedInHierarchicalListing container
```

Writing to an append blob

An append blob is optimized for append operations, such as logging. Like a block blob, an append blob is composed of blocks, but when you add a new block to an append blob, it is always appended to the end of the blob. You cannot update or delete an existing block in an append blob. The block IDs for an append blob are not exposed as they are for a block blob.

Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include a maximum of 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

The following example creates a new append blob and appends some data to it, simulating a simple logging operation.

```
// Get a reference to a container.
let appendContainer = BlobContainerClient(storageConnString, "my-append-blobs")

// Create the container if it does not already exist.
appendContainer.CreateIfNotExists() |> ignore

// Get a reference to an append blob.
let appendBlob = appendContainer.GetAppendBlobClient("append-blob.log")

// Create the append blob. Note that if the blob already exists, the
// CreateOrReplace() method will overwrite it. You can check whether the
// blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateIfNotExists()

let numBlocks = 10

// Generate an array of random bytes.
let rnd = Random()
let bytesArray = Array.zeroCreate<byte>(numBlocks)
rnd.NextBytes(bytesArray)

// Simulate a logging operation by writing text data and byte data to the
// end of the append blob.
for i in 0 .. numBlocks - 1 do
    let msg = sprintf $"Timestamp: {DateTime.UtcNow} \tLog Entry: {bytesArray.[i]}\n"
    let array = Encoding.ASCII.GetBytes(msg);
    use stream = new MemoryStream(array)
    appendBlob.AppendBlock(stream)

// Read the append blob to the console window.
let downloadedText = appendBlob.DownloadContent().ToString()
printfn $"{downloadedText}"
```

See Understanding Block Blobs, Page Blobs, and Append Blobs for more information about the differences between the three types of blobs.

Concurrent access

To support concurrent access to a blob from multiple clients or multiple process instances, you can use **ETags** or **leases**.

- **Etag** - provides a way to detect that the blob or container has been modified by another process
- **Lease** - provides a way to obtain exclusive, renewable, write, or delete access to a blob for a period of time

For more information, see Managing Concurrency in Microsoft Azure Storage.

Naming containers

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mydata` is the name of the container in these sample blob URIs:

- `https://storagesample.blob.core.windows.net/mydata/blob1.txt`
- `https://storagesample.blob.core.windows.net/mydata/photos/myphoto.jpg`

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive

dashes are not permitted in container names.

3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

The name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

Managing security for blobs

By default, Azure Storage keeps your data secure by limiting access to the account owner, who is in possession of the account access keys. When you need to share blob data in your storage account, it is important to do so without compromising the security of your account access keys. Additionally, you can encrypt blob data to ensure that it is secure going over the wire and in Azure Storage.

Controlling access to blob data

By default, the blob data in your storage account is accessible only to storage account owner. Authenticating requests against Blob storage requires the account access key by default. However, you might want to make certain blob data available to other users.

Encrypting blob data

Azure Storage supports encrypting blob data both at the client and on the server.

See also

- Azure Storage APIs for .NET
- Azure Storage Services REST API Reference
- Get started with AzCopy
- Configure Azure Storage connection strings
- Quickstart: Use .NET to create a blob in object storage

Get started with Azure File Storage using F#

8/5/2022 • 6 minutes to read • Edit Online

Azure File Storage is a service that offers file shares in the cloud using the standard Server Message Block (SMB) Protocol. Both SMB 2.1 and SMB 3.0 are supported. With Azure File Storage, you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. Applications running in Azure virtual machines or cloud services or from on-premises clients can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

For a conceptual overview of file storage, see the .NET guide for file storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You'll also need your storage access key for this account.

Create an F# script and start F# interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `files.fsx`, in your F# development environment.

How to execute scripts

F# Interactive, `dotnet fsi`, can be launched interactively, or it can be launched from the command line to run a script. The command-line syntax is

```
> dotnet fsi [options] [ script-file [arguments] ]
```

Add packages in a script

Use `#r` `nuget:package name` to install the `Azure.Storage.Blobs` and `Azure.Storage.Common` and `Azure.Storage.Files` packages and `open` namespaces. Such as

```
> #r "nuget: Azure.Storage.Blobs"
> #r "nuget: Azure.Storage.Common"
> #r "nuget: Azure.Storage.Files"
open Azure.Storage.Blobs
open Azure.Storage.Sas
open Azure.Storage.Files
open Azure.Storage.Files.Shares
open Azure.Storage.Files.Shares.Models
```

Add namespace declarations

Add the following `open` statements to the top of the `files.fsx` file:

```
open System
open System.IO
open Azure
open Azure.Storage // Namespace for StorageSharedKeyCredential
open Azure.Storage.Blobs // Namespace for BlobContainerClient
open Azure.Storage.Sas // Namespace for ShareSasBuilder
open Azure.Storage.Files.Shares // Namespace for File storage types
open Azure.Storage.Files.Shares.Models // Namespace for ShareServiceProperties
```

Get your connection string

You'll need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you'll enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

Create the file service client

The `ShareClient` type enables you to programmatically use files stored in File storage. Here's one way to create the service client:

```
let share = ShareClient(storageConnString, "shareName")
```

Now you are ready to write code that reads data from and writes data to File storage.

Create a file share

This example shows how to create a file share if it does not already exist:

```
share.CreateIfNotExistsAsync()
```

Create a directory

Here, you get the directory. You create if it doesn't already exist.

```
// Get a reference to the directory
let directory = share.GetDirectoryClient("directoryName")

// Create the directory if it doesn't already exist
directory.CreateIfNotExistsAsync()
```

Upload a file to the sample directory

This example shows how to upload a file to the sample directory.

```
let file = directory.GetFileClient("fileName")

let writeToFile localFilePath =
    use stream = File.OpenRead(localFilePath)
    file.Create(stream.Length)
    file.UploadRange(
        HttpRange(0L, stream.Length),
        stream)

writeToFile "localFilePath"
```

Download a file to a local file

Here you download the file just created, appending the contents to a local file.

```
let download = file.Download()

let copyTo saveDownloadPath =
    use downStream = File.OpenWrite(saveDownloadPath)
    download.Value.Content.CopyTo(downStream)

copyTo "Save_Download_Path"
```

Set the maximum size for a file share

The example below shows how to check the current usage for a share and how to set the quota for the share.

```
// stats.Usage is current usage in GB
let ONE_GIBIBYTE = 10_737_420_000L // Number of bytes in 1 gibibyte
let stats = share.GetStatistics().Value
let currentGiB = int (stats.ShareUsageInBytes / ONE_GIBIBYTE)

// Set the quota to 10 GB plus current usage
share.SetQuotaAsync(currentGiB + 10)

// Remove the quota
share.SetQuotaAsync(0)
```

Generate a shared access signature for a file or file share

You can generate a shared access signature (SAS) for a file share or for an individual file. You can also create a shared access policy on a file share to manage shared access signatures. Creating a shared access permissions is recommended, as it provides a means of revoking the SAS if it should be compromised.

Here, you create a shared access permissions on a share, and then set that permissions to provide the constraints for a SAS on a file in the share.

```
let accountName = "..." // Input your storage account name
let accountKey = "..." // Input your storage account key

// Create a 24-hour read/write policy.
let expiration = DateTimeOffset.UtcNow.AddHours(24.)
let fileSAS = ShareSasBuilder(
      ShareName = "shareName",
      FilePath = "filePath",
      Resource = "f",
      ExpiresOn = expiration)

// Set the permissions for the SAS
let permissions = ShareFileSasPermissions.All
fileSAS.SetPermissions(permissions)

// Create a SharedKeyCredential that we can use to sign the SAS token
let credential = StorageSharedKeyCredential(accountName, accountKey)

// Build a SAS URI
let fileSasUri =
UriBuilder($"https://{accountName}.file.core.windows.net/{fileSAS.ShareName}/{fileSAS.FilePath}")
fileSasUri.Query = fileSAS.ToSasQueryParameters(credential).ToString()
```

For more information about creating and using shared access signatures, see Using Shared Access Signatures (SAS) and Create and use a SAS with Blob storage.

Copy files

You can copy a file to another file or to a blob, or a blob to a file. If you are copying a blob to a file, or a file to a blob, you *must* use a shared access signature (SAS) to authenticate the source object, even if you are copying within the same storage account.

Copy a file to another file

Here, you copy a file to another file in the same share. Because this copy operation copies between files in the same storage account, you can use Shared Key authentication to perform the copy.

```
let sourceFile = ShareFileClient(storageConnString, "shareName", "sourceFilePath")
let destFile = ShareFileClient(storageConnString, "shareName", "destFilePath")
destFile.StartCopyAsync(sourceFile.Uri)
```

Copy a file to a blob

Here, you create a file and copy it to a blob within the same storage account. You create a SAS for the source file, which the service uses to authenticate access to the source file during the copy operation.

```
// Create a new file SAS
let fileSASCopyToBlob = ShareSasBuilder(
    ShareName = "shareName",
    FilePath = "sourceFilePath",
    Resource = "f",
    ExpiresOn = DateTimeOffset.UtcNow.AddHours(24.))
let permissionsCopyToBlob = ShareFileSasPermissions.Read
fileSASCopyToBlob.SetPermissions(permissionsCopyToBlob)
let fileSasUriCopyToBlob =
UriBuilder($"https://{accountName}.file.core.windows.net/{fileSASCopyToBlob.ShareName}/{fileSASCopyToBlob.Fi
lePath}")

// Get a reference to the file.
let sourceFileCopyToBlob = ShareFileClient(fileSasUriCopyToBlob.Uri)

// Get a reference to the blob to which the file will be copied.
let containerCopyToBlob = BlobContainerClient(storageConnString, "containerName");
containerCopyToBlob.CreateIfNotExists()
let destBlob = containerCopyToBlob.GetBlobClient("blobName")
destBlob.StartCopyFromUriAsync(sourceFileCopyToBlob.Uri)
```

You can copy a blob to a file in the same way. If the source object is a blob, then create a SAS to authenticate access to that blob during the copy operation.

Troubleshooting File storage using metrics

Azure Storage Analytics supports metrics for File storage. With metrics data, you can trace requests and diagnose issues.

You can enable metrics for File storage from the Azure portal, or you can do it from F# like this:

```
// Instatiate a ShareServiceClient
let shareService = ShareServiceClient(storageConnString);

// Set metrics properties for File service
let props = ShareServiceProperties()

props.HourMetrics = ShareMetrics(
    Enabled = true,
    IncludeApis = true,
    Version = "1.0",
    RetentionPolicy = ShareRetentionPolicy(Enabled = true,Days = 14))

props.MinuteMetrics = ShareMetrics(
    Enabled = true,
    IncludeApis = true,
    Version = "1.0",
    RetentionPolicy = ShareRetentionPolicy(Enabled = true,Days = 7))

shareService.SetPropertiesAsync(props)
```

Next steps

For more information about Azure File Storage, see these links.

Conceptual articles and videos

- Azure Files Storage: a frictionless cloud SMB file system for Windows and Linux
- How to use Azure File Storage with Linux

Tooling support for File storage

- Using Azure PowerShell with Azure Storage
- How to use AzCopy with Microsoft Azure Storage
- Create, download, and list blobs with Azure CLI

Reference

- Storage Client Library for .NET reference
- File Service REST API reference

Blog posts

- Azure File Storage is now generally available
- Inside Azure File Storage
- Introducing Azure File Service
- Persisting connections to Azure Files

Get started with Azure Queue Storage using F#

8/5/2022 • 5 minutes to read • Edit Online

Azure Queue Storage provides cloud messaging between application components. In designing applications for scale, application components are often decoupled, so that they can scale independently. Queue storage delivers asynchronous messaging for communication between application components, whether they are running in the cloud, on the desktop, on an on-premises server, or on a mobile device. Queue storage also supports managing asynchronous tasks and building process work flows.

About this tutorial

This tutorial shows how to write F# code for some common tasks using Azure Queue Storage. Tasks covered include creating and deleting queues and adding, reading, and deleting queue messages.

For a conceptual overview of queue storage, see the .NET guide for queue storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You'll also need your storage access key for this account.

Create an F# script and start F# interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `queues.fsx`, in your F# development environment.

How to execute scripts

F# Interactive, `dotnet fsi`, can be launched interactively, or it can be launched from the command line to run a script. The command-line syntax is

```
> dotnet fsi [options] [ script-file [arguments] ]
```

Add packages in a script

Next, use `#r` `nuget:package name` to install the `Azure.Storage.Queues` package and `open` namespaces.Such as

```
> #r "nuget: Azure.Storage.Queues"
open Azure.Storage.Queues
```

Add namespace declarations

Add the following `open` statements to the top of the `queues.fsx` file:

```
open Azure.Storage.Queues // Namespace for Queue storage types
open System
open System.Text
```

Get your connection string

You'll need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you'll enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

Create the queue service client

The `QueueClient` class enables you to retrieve queues stored in Queue storage. Here's one way to create the client:

```
let queueClient = QueueClient(storageConnString, "myqueue")
```

Now you are ready to write code that reads data from and writes data to Queue storage.

Create a queue

This example shows how to create a queue if it doesn't already exist:

```
queueClient.CreateIfNotExists()
```

Insert a message into a queue

To insert a message into an existing queue, first create a new Message. Next, call the `SendMessage` method. A Message can be created from either a string (in UTF-8 format) or a `byte` array, like this:

```
queueClient.SendMessage("Hello, World") // Insert a String message into a queue
queueClient.SendMessage(BinaryData.FromBytes(Encoding.UTF8.GetBytes("Hello, World"))) // Insert a BinaryData
message into a queue
```

Peek at the next message

You can peek at the message in the front of a queue, without removing it from the queue, by calling the `PeekMessage` method.

```
let peekedMessage = queueClient.PeekMessage()
let messageContents = peekedMessage.Value.Body.ToString()
```

Get the next message for processing

You can retrieve the message at the front of a queue for processing by calling the `ReceiveMessage` method.

```
let updateMessage = queueClient.ReceiveMessage().Value
```

You later indicate successful processing of the message by using `DeleteMessage`.

Change the contents of a queued message

You can change the contents of a retrieved message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than some number of times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```
queueClient.UpdateMessage(
    updateMessage.MessageId,
    updateMessage.PopReceipt,
    "Updated contents.",
    TimeSpan.FromSeconds(60.0))
```

De-queue the next message

Your code de-queues a message from a queue in two steps. When you call `ReceiveMessage`, you get the next message in a queue. A message returned from `ReceiveMessage` becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call `DeleteMessage`. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls `DeleteMessage` right after the message has been processed. All of the Queue methods we've shown so far have `Async` alternatives.

```
let deleteMessage = queueClient.ReceiveMessage().Value
queueClient.DeleteMessage(deleteMessage.MessageId, deleteMessage.PopReceipt)
```

Use Async workflows with common Queue storage APIs

This example shows how to use an async workflow with common Queue storage APIs.

```
async {
    let! exists = queueClient.CreateIfNotExistsAsync() |> Async.AwaitTask

    let! delAsyncMessage = queueClient.ReceiveMessageAsync() |> Async.AwaitTask

    // ... process the message here ...

    // Now indicate successful processing:
    queueClient.DeleteMessageAsync(delAsyncMessage.Value.MessageId, delAsyncMessage.Value.PopReceipt) |>
Async.AwaitTask
}
```

Additional options for de-queuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses `ReceiveMessages` to get 20 messages in one call and then processes each message. It also sets the invisibility timeout to five minutes for each message. The 5 minutes starts for all messages at the same time, so after 5 minutes have passed since the call to `ReceiveMessages`, any messages that have not been deleted will become visible again.

```
for dequeueMessage in queueClient.ReceiveMessages(20, Nullable(TimeSpan.FromMinutes(5.))).Value do
        // Process the message here.
        queueClient.DeleteMessage(dequeueMessage.MessageId, dequeueMessage.PopReceipt)
```

Get the queue length

You can get an estimate of the number of messages in a queue. The `GetProperties` method asks the Queue service to retrieve the queue attributes, including the message count. The `ApproximateMessagesCount` property returns the last value retrieved by the `GetProperties` method.

```
let properties = queueClient.GetProperties().Value
let count = properties.ApproximateMessagesCount
```

Delete a queue

To delete a queue and all the messages contained in it, call the `Delete` method on the queue object.

```
queueClient.DeleteIfExists()
```

Note

If you're migrating from the old libraries, they Base64-encoded messages by default, but the new libraries don't because it's more performant. For information on how to set up encoding, see MessageEncoding.

See also

- Azure Storage APIs for .NET
- Configure Azure Storage connection strings
- Azure Storage Services REST API Reference

Get started with Azure Table Storage and the Azure Cosmos DB Table api using F#

8/5/2022 • 7 minutes to read • Edit Online

Azure Table Storage is a service that stores structured NoSQL data in the cloud. Table storage is a key/attribute store with a schemaless design. Because Table storage is schemaless, it's easy to adapt your data as the needs of your application evolve. Access to data is fast and cost-effective for all kinds of applications. Table storage is typically significantly lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets, such as user data for web applications, address books, device information, and any other type of metadata that your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

Azure Cosmos DB provides the Table API for applications that are written for Azure Table Storage and that require premium capabilities such as:

- Turnkey global distribution.
- Dedicated throughput worldwide.
- Single-digit millisecond latencies at the 99th percentile.
- Guaranteed high availability.
- Automatic secondary indexing.

Applications written for Azure Table Storage can migrate to Azure Cosmos DB by using the Table API with no code changes and take advantage of premium capabilities. The Table API has client SDKs available for .NET, Java, Python, and Node.js.

For more information, see Introduction to Azure Cosmos DB Table API.

About this tutorial

This tutorial shows how to write F# code to do some common tasks using Azure Table Storage or the Azure Cosmos DB Table API, including creating and deleting a table and inserting, updating, deleting, and querying table data.

Prerequisites

To use this guide, you must first create an Azure storage account or Azure Cosmos DB account.

Create an F# script and start F# interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example, `tables.fsx`, in your F# development environment.

How to execute scripts

F# Interactive, `dotnet fsi`, can be launched interactively, or it can be launched from the command line to run a script. The command-line syntax is

```
> dotnet fsi [options] [ script-file [arguments] ]
```

Add packages in a script

Next, use `#r` `nuget:package name` to install the `Azure.Data.Tables` package and `open` namespaces. Such as

```
> #r "nuget: Azure.Data.Tables"
open Azure.Data.Tables
```

Add namespace declarations

Add the following `open` statements to the top of the `tables.fsx` file:

```
open System
open Azure
open Azure.Data.Tables // Namespace for Table storage types
```

Get your Azure Storage connection string

If you're connecting to Azure Storage Table service, you'll need your connection string for this tutorial. You can copy your connection string from the Azure portal. For more information about connection strings, see Configure Storage Connection Strings.

Get your Azure Cosmos DB connection string

If you're connecting to Azure Cosmos DB, you'll need your connection string for this tutorial. You can copy your connection string from the Azure portal. In the Azure portal, in your Cosmos DB account, go to **Settings** > **Connection String**, and select the **Copy** button to copy your Primary Connection String.

For the tutorial, enter your connection string in your script, like the following example:

```
let storageConnString = "UseDevelopmentStorage=true" // fill this in from your storage account
```

Create the table service client

The `TableServiceClient` class enables you to retrieve tables and entities in Table storage. Here's one way to create the service client:

```
let tableClient = TableServiceClient storageConnString
```

Now you are ready to write code that reads data from and writes data to Table storage.

Create a table

This example shows how to create a table if it does not already exist:

```
// Retrieve a reference to the table.
let table = tableClient.GetTableClient "people"

// Create the table if it doesn't exist.
table.CreateIfNotExists () |> ignore
```

Add an entity to a table

An entity has to have a type that implements `ITableEntity`. You can extend `ITableEntity` in any way you like, but your type *must* have a parameter-less constructor. Only properties that have both `get` and `set` are stored in your Azure Table.

An entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys, but using diverse partition keys allows for greater scalability of parallel operations.

Here's an example of a `Customer` that uses the `lastName` as the partition key and the `firstName` as the row key.

```
type Customer (firstName, lastName, email: string, phone: string) =
    interface ITableEntity with
        member val ETag = ETag "" with get, set
        member val PartitionKey = "" with get, set
        member val RowKey = "" with get, set
        member val Timestamp = Nullable() with get, set

    new() = Customer(null, null, null, null)
    member val Email = email with get, set
    member val PhoneNumber = phone with get, set
    member val PartitionKey = lastName with get, set
    member val RowKey = firstName with get, set
```

Now add `Customer` to the table. To do so, we can use the AddEntity() method.

```
let customer = Customer ("Walter", "Harp", "Walter@contoso.com", "425-555-0101")
table.AddEntity customer
```

Insert a batch of entities

You can insert a batch of entities into a table using a single write operation. Batch operations allow you to combine operations into a single execution, but they have some restrictions:

- You can perform updates, deletes, and inserts in the same batch operation.
- A batch operation can include up to 100 entities.
- All entities in a batch operation must have the same partition key.
- While it is possible to perform a query in a batch operation, it must be the only operation in the batch.

Here's some code that combines two inserts into a batch operation:

```
let customers =
    [
        Customer("Jeff", "Smith", "Jeff@contoso.com", "425-555-0102")
        Customer("Ben", "Smith", "Ben@contoso.com", "425-555-0103")
    ]

// Add the entities to be added to the batch and submit it in a transaction.
customers
|> List.map (fun customer -> TableTransactionAction (TableTransactionActionType.Add, customer))
|> table.SubmitTransaction
```

Retrieve all entities in a partition

To query a table for all entities in a partition, use a `Query<T>` object. Here, you filter for entities where "Smith" is the partition key.

```
table.Query<Customer> "PartitionKey eq 'Smith'"
```

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by combining the partition key filter with a row key filter. Here, you use two filters to get all entities in the "Smith" partition where the row key (first name) starts with a letter earlier than "M" in the alphabet.

```
table.Query<Customer> "PartitionKey eq 'Smith' and RowKey lt 'J'"
```

Retrieve a single entity

To retrieve a single, specific entity, use `GetEntityAsync` to specify the customer "Ben Smith". Instead of a collection, you get back a `Customer`. Specifying both the partition key and the row key in a query is the fastest way to retrieve a single entity from the Table service.

```
let singleResult = table.GetEntity<Customer>("Smith", "Ben").Value
```

You now print the results:

```
// Evaluate this value to print it out into the F# Interactive console
singleResult
```

Update an entity

To update an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service using a `TableUpdateMode.Replace` operation. This causes the entity to be fully replaced on the server, unless the entity on the server has changed since it was retrieved, in which case the operation fails. This failure is to prevent your application from inadvertently overwriting changes from other sources.

```
singleResult.PhoneNumber <- "425-555-0103"
try
    table.UpdateEntity (singleResult, ETag "", TableUpdateMode.Replace) |> ignore
    printfn "Update succeeded"
with
| :? RequestFailedException as e ->
    printfn $"Update failed: {e.Status} - {e.ErrorCode}"
```

Upsert an entity

Sometimes, you don't know whether an entity exists in the table. And if it does, the current values stored in it are no longer needed. You can use `UpsertEntity` method to create the entity or replace it if it exists, regardless of its state.

```
singleResult.PhoneNumber <- "425-555-0104"
table.UpsertEntity (singleResult, TableUpdateMode.Replace)
```

Query a subset of entity properties

A table query can retrieve just a few properties from an entity instead of all of them. This technique, called projection, can improve query performance, especially for large entities. Here, you return only email addresses using `Query<T>` and `Select`. Projection is not supported on the local storage emulator, so this code runs only when you're using an account on the Table service.

```
query {
    for customer in table.Query<Customer> () do
    select customer.Email
}
```

Retrieve entities in pages asynchronously

If you are reading a large number of entities, and you want to process them as they are retrieved rather than waiting for them all to return, you can use a segmented query. Here, you return results in pages by using an async workflow so that execution is not blocked while you're waiting for a large set of results to return.

```
let pagesResults = table.Query<Customer> ()

for page in pagesResults.AsPages () do
    printfn "This is a new page!"
    for customer in page.Values do
        printfn $"customer: {customer.RowKey} {customer.PartitionKey}"
```

Delete an entity

You can delete an entity after you have retrieved it. As with updating an entity, this fails if the entity has changed since you retrieved it.

```
table.DeleteEntity ("Smith", "Ben")
```

Delete a table

You can delete a table from a storage account. A table that has been deleted will be unavailable to be re-created for some time following the deletion.

```
table.Delete ()
```

See also

- Introduction to Azure Cosmos DB Table API
- Storage Client Library for .NET reference
- Configuring Connection Strings

Using other Azure services with F#

8/5/2022 • 2 minutes to read • Edit Online

In the following sections, you will find resources on how to use a range of other Azure services with F#.

> **NOTE**
> If a particular Azure service isn't in this documentation set, please consult either the Azure Functions or .NET documentation for that service. Some Azure services are language-independent and require no language-specific documentation and are not listed here.

Using Azure Virtual Machines with F#

Azure supports a wide range of virtual machine (VM) configurations, see Linux and Azure Virtual Machines.

To install F# on a virtual machine for execution, compilation and/or scripting see Using F# on Linux and Using F# on Windows.

Using Azure Cosmos DB with F#

Azure Cosmos DB is a NoSQL service for highly available, globally distributed apps.

Azure Cosmos DB can be used with F# in two ways:

1. Through the creation of F# Azure Functions which react to or cause changes to Azure Cosmos DB collections. See Azure Cosmos DB bindings for Azure Functions, or
2. By using the Azure Cosmos DB .NET SDK for SQL API. The related samples are in C#.

Using Azure Event Hubs with F#

Azure Event Hubs provide cloud-scale telemetry ingestion from websites, apps, and devices.

Azure Event Hubs can be used with F# in two ways:

1. Through the creation of F# Azure Functions which are triggered by events. See Azure Function triggers for Event Hubs, or
2. By using the .NET SDK for Azure. Note these examples are in C#.

Using Azure Functions with F#

Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Your functions are connected to events in Azure storage and other cloud-hosted resources. Data flows into your F# functions via function arguments. You can use your development language of choice, trusting Azure to scale as needed.

Azure Functions provide efficient, reactive, scalable execution of F# code. See the Azure Functions F# Developer Reference for reference documentation on how to use F# with Azure Functions.

Using Azure App Service with F#

Azure App Service is a cloud platform to build powerful web and mobile apps that connect to data anywhere, in the cloud or on-premises.

- F# Azure Web API example
- Hosting F# in a web application on Azure

Using Azure Notification Hubs with F#

Azure Notification Hubs are multiplatform, scaled-out push infrastructure that enable you to send mobile push notifications from any backend (in the cloud or on-premises) to any mobile platform.

Azure Notification Hubs can be used with F# in two ways:

1. Through the creation of F# Azure Functions which send results to a notification hub. See Azure Function output triggers for Notification Hubs, or
2. By using the .NET SDK for Azure. Note these examples are in C#.

Implementing WebHooks on Azure with F#

A Webhook is a callback triggered via a web request. Webhooks are used by sites such as GitHub to signal events.

Webhooks can be implemented in F# and hosted on Azure via an Azure Function in F# with a Webhook Binding.

Using Webjobs with F#

Webjobs are programs you can run in your App Service web app in three ways: on demand, continuously, or on a schedule.

Example F# Webjob

Implementing Timers on Azure with F#

Timer triggers call functions based on a schedule, one time or recurring.

Timers can be implemented in F# and hosted on Azure via an Azure Function in F# with a Timer Trigger.

Other resources

- Full documentation on all Azure services