Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2019-09-24 09:10:21+02:00

Contents

9 Organising Code in Libraries and Application Programs

In this chapter, we will focus on a number of ways to make the code available as *library* functions in F#. A library is a collection of types, values, and functions that an application program can use. A library does not perform calculations on its own.

· library

F# includes several programming structures to organize code in libraries: Modules, namespaces, and classes. In this chapter, we will describe modules and namespaces. Classes will be described in detail in Chapter 20.

9.1 Modules

An F# *module*, not to be confused with a Common Language Infrastructure module (see Appendix D), is a programming structure used to organize type declarations, values, functions, etc.

· module

Every implementation and script file in F# implicitly defines a module, and the module name is given by the filename. Consider the script file `Meta.fsx` shown in Listing 9.1.

Listing 9.1 Meta.fsx:
A script file defining the `apply` function.

```
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float
  = f x y
```

Here, we have implicitly defined a module with the name `Meta`. Another script file may now use this function, which is accessed using the "`.`" notation, i.e., `Meta.apply` will refer to this function in other programs. An application program could be as the one shown in Listing 9.3.

Listing 9.2 MetaApp.fsx:
Defining a script calling the module.

```
let add : Meta.floatFunction = fun x y -> x + y
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

In the exmple above, we have explicitly used the module's type definition for illustration purposes. A shorter and possibly simpler program would have been to define `add` as `let add x y = x + y`, since F#'s type system will infer the implied type. However, **explicit definitions of types is recommended for readability.** Hence, an alterna-

Advice

tive to the above example's use of anonymous functions is: `let add (x: float) (y: float) : float = x + y`. To compile the module and the application program, we write as demonstrated in Listing 9.3.

Listing 9.3: Compiling both the module and the application code. Note that file order matters when compiling several files.

```
$ fsharpc --nologo Meta.fsx MetaApp.fsx && mono MetaApp.exe
3.0 + 4.0 = 7.0
```

Since the F# compiler reads through the files once, the order of the filenames in the compile command is very important. Hence, the script containing the module and function definitions must be to the left of the script containing their use. Notice also that if not otherwise specified, the F# compiler produces an `.exe` file derived from the last filename in the list of filenames.

We may also explicitly define the module name using the *`module`* with the following syntax, · `module`

Listing 9.4: Outer module.

```
module <ident>
<script>
```

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is an expression. An example is given in Listing 9.20.

Listing 9.5 MetaExplicit.fsx:
Explicit definition of the outermost module.

```
module Meta
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float
  = f x y
```

Since we have created a new file, where the module `Meta` is explicitly defined, we can use the same application program. This is demonstrated in Listing 9.6.

Listing 9.6: Changing the module definition to explicit naming has no effect on the application nor the compile command.

```
$ fsharpc --nologo MetaExplicit.fsx MetaApp.fsx && mono
  MetaApp.exe
3.0 + 4.0 = 7.0
```

Since `MetaExplicit.fsx` explicitly defines the module name, `apply` is not available to an application program as `MetaExplicit.apply`. **It is recommended that module names are defined explicitly, since filenames may change due to external conditions.** Advice In other words, filenames are typically set from the perspective of the filesystem. The user may choose to change names to suit a filesystem structure, or different platforms may impose different file naming conventions. Thus, direct linking of filenames with the internal workings of a program is a needless complication of structure.

The definitions inside a module may be accessed directly from an application program, omitting the "."-notation, by use of the *open* keyword,

· open

Listing 9.7: Open module.

```
open <ident>
```

We can modify `MetaApp.fsx`, as shown in Listing 9.9.

Listing 9.8 MetaAppWOpen.fsx:
Avoiding the "."-notation by the open keyword.

```
open Meta
let add : floatFunction = fun x y -> x + y
let result = apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

In this case, the namespace of our previously defined module is included into the scope of the application functions, and its types, values, functions, etc. can be used directly, as shown in Listing 9.9.

Listing 9.9: How the application program opens the module has no effect on the module code nor compile command.

```
$ fsharpc --nologo MetaExplicit.fsx MetaAppWOpen.fsx && mono MetaAppWOpen.exe
3.0 + 4.0 = 7.0
```

The `open`-keyword should be used sparingly, since including a library's definitions into the application scope can cause surprising naming conflicts, because the user of a library typically has no knowledge of the inner workings of the library. E.g., the user may accidentally use code defined in the library, but with different type and functionality than intended, which the type system will use to deduce types in the application program, and therefore will either give syntax or runtime errors that are difficult to understand. This problem is known as *namespace pollution*, and for clarity, **it is recommended to use the `open`-keyword sparingly**. Note that for historical reasons, the phrase 'namespace pollution' is used to cover pollution both due to modules and namespaces.

· namespace pollution
Advice

Modules may also be nested, in which case the nested definitions must use the "="-sign and must be appropriately indented.

Listing 9.10: Nested modules.

```
module <ident> = <script>
```

In lightweight syntax, a newline may be entered before the script `<script>`, and the script must be indented. An example is shown in Listing 9.11.

Listing 9.11 nestedModules.fsx:
Modules may be nested.

```
module Utilities
let PI = 3.1415
module Meta =
  type floatFunction = float -> float -> float
  let apply (f : floatFunction) (x : float) (y : float) :
   float = f x y
module MathFcts =
  let add : Meta.floatFunction = fun x y -> x + y
```

In this case, `Meta` and `MathFcts` are defined at the same level and said to be siblings, while `Utilities` is defined at a higher level. In this relation, the former two are said to be the children of the latter. Note that the nesting respects the lexical scope rules, such that the constant `PI` is directly accessible in both modules `Meta` and `MathFcts`, as is the module `Meta` in `MathFcts`, but not `MathFcts` in `Meta`. The "`.`"-notation is reused to index deeper into the module hierarchy, as the example in Listing 9.12 shows.

Listing 9.12 nestedModulesApp.fsx:
Applications using nested modules require additional usage of the "`.`" notation to navigate the nesting tree.

```

let add : Utilities.Meta.floatFunction = fun x y -> x + y
let result = Utilities.Meta.apply Utilities.MathFcts.add 3.0
   Utilities.PI
printfn "3.0 + 4.0 = %A" result
```

Modules can be recursive using the *`rec`*-keyword, meaning that in our example we can make the outer module recursive, as demonstrated in Listing 9.13. · `rec`

Listing 9.13 nestedRecModules.fsx:
Mutual dependence on nested modules requires the `rec` keyword in the module definition.

```
module rec Utilities
  module Meta =
    type floatFunction = float -> float -> float
    let apply (f : floatFunction) (x : float) (y : float) :
   float = f x y
  module MathFcts =
    let add : Meta.floatFunction = fun x y -> x + y
```

The consequence is that the modules `Meta` and `MathFcts` are accessible in both modules, but compilation will now give a warning since soundness of the code will first be checked at runtime. In general, it is advised to **avoid programming constructions whose validity cannot be checked at compile-time.** Advice

9.2 Namespaces

An alternative way to structure code in modules is to use a *namespace*, which can only hold modules and type declarations and only works in compiled mode. Namespaces are defined as explicitly defined outer modules, using the `namespace` keyword in accordance with the following syntax.

· namespace

· `namespace`

Listing 9.14: Namespace.

```
namespace <ident>
<script>
```

An example is given in Listing 9.15.

Listing 9.15 namespace.fsx:
Defining a namespace is similar to explicitly named modules.

```
namespace Utilities
type floatFunction = float -> float -> float
module Meta =
  let apply (f : floatFunction) (x : float) (y : float) :
   float = f x y
```

Notice that when organizing code in a namespace, the first line of the file, other than comments and compiler directives, must be the one starting with `namespace`.

As for modules, the content of a namespace is accessed using the "`.`" notation, as demonstrated in Listing 9.16.

Listing 9.16 namespaceApp.fsx:
The "`.`"-notation lets the application program access functions and types in a namespace.

```
let add : Utilities.floatFunction = fun x y -> x + y
let result = Utilities.Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

Likewise, the compilation is performed in the same way as for modules, see Listing 9.17.

Listing 9.17: Compilation of files including namespace definitions uses the same procedure as modules.

```
$ fsharpc --nologo namespace.fsx namespaceApp.fsx && mono
   namespaceApp.exe
3.0 + 4.0 = 7.0
```

Hence, from an application point of view, it is not immediately possible to see that `Utilities` is defined as a namespace and not a module. However, in contrast to modules, namespaces may span several files. E.g., we may add a third file extending the `Utilities` namespace with the `MathFcts` module, as demonstrated in Listing 9.18.

Listing 9.18 namespaceExtension.fsx:
Namespaces may span several files. Here is shown an extra file which extends the `Utilities` namespace.

```
namespace Utilities
module MathFcts =
  let add : floatFunction = fun x y -> x + y
```

To compile, we now need to include all three files in the right order, see Listing 9.19.

Listing 9.19: Compilation of namespaces defined in several files requires careful consideration of order, since the compiler reads once and only once through the files in the order they are given.

```
$ fsharpc --nologo namespace.fsx namespaceExtension.fsx
   namespaceApp.fsx && mono namespaceApp.exe
3.0 + 4.0 = 7.0
```

The order matters, since `namespaceExtension.fsx` relies on the definition of `floatFunction` in the file `namespace.fsx`. You can use extensions to extend existing namespaces included with the F# compiler.

Namespaces may also be nested. In contrast to modules, nesting is defined using the "`.`" notation. That is, to create a child namespace `more` of `Utilities`, we must use initially write `namespace Utilities.more`. Indentation is ignored in the `namespace` line, thus left-most indentation is almost always used. Namespaces follow lexical scope rules, and identically to modules, namespaces containing mutually dependent children can be declared using the `rec` keyword, e.g., `namespace rec Utilities`.

9.3 Compiled Libraries

Libraries may be distributed in compiled form as `.dll` files. This saves the user from having to recompile a possibly large library every time library functions needs to be compiled with an application program. In order to produce a library file from `MetaExplicitModuleDefinition.fsx` and then compile an application program, we first use the compiler's `-a` option to produce the `.dll`. A demonstration is given in Listing 9.20.

Listing 9.20: A stand-alone `.dll` file is created and used with special compile commands.

```
$ fsharpc --nologo -a MetaExplicit.fsx
```

This produces the file `MetaExplicit.dll`, which may be linked to an application by using the `-r` option during compilation, see Listing 9.21.

Listing 9.21: The library is linked to an application during compilation to produce runnable code.

```
$ fsharpc --nologo -r MetaExplicit.dll MetaApp.fsx && mono
  MetaApp.exe
3.0 + 4.0 = 7.0
```

A library can be the result of compiling a number of files into a single `.dll` file. `.dll`-files may be loaded dynamically in script files (`.fsx`-files) by using the ***`#r`* directive**, as illustrated in Listing 9.22. · `#r` directive

Listing 9.22 MetaHashApp.fsx:
The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.

```
#r "MetaExplicit.dll"
let add : Meta.floatFunction = fun x y -> x + y
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

We may now omit the explicit mentioning of the library when compiling, as shown in Listing 9.23.

Listing 9.23: When using the `#r` directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.

```
$ fsharpc --nologo MetaHashApp.fsx && mono MetaHashApp.exe
3.0 + 4.0 = 7.0
```

The `#r` directive is also used to include a library in interactive mode. However, for the code to be compiled, the use of the `#r` directive requires that the filesystem path to the library is coded inside the script. As for module names, direct linking of filenames with the internal workings of a program is a needless complication of structure, and **it is recommended not to rely on the use of the `#r` directive.** Advice

In the above listings we have compiled *script files* into libraries. However, F# has reserved the `.fs` filename suffix for library files, and such files are called *implementation files*. In contrast to script files, implementation files do not support the `#r` directive. When compiling a list of implementation and script files, all but the last file must explicitly define a module or a namespace. · script file · implementation file

Both script and implementation files may be augmented with *signature files*. A signature file contains no implementation, only type definitions. Signature files offer three distinct features: · signature file

1. Signature files can be used as part of the documentation of code, since type information is of paramount importance for an application programmer to use a library.

2. Signature files may be written before the implementation file. This allows for a higher-level programming design that focuses on *which* functions should be included and *how* they can be composed.

3. Signature files allow for access control. Most importantly, if a type definition is not available in the signature file, then it is not available to the application program. Such definitions are private and can only be used internally in the library code. More fine-grained control related to classes is available and will be discussed in Chapter 20.

Signature files can be generated automatically using the --sig:<filename> compiler directive. To demonstrate this feature, we will first move the definition of `add` to the implementation file, see Listing 9.28.

Listing 9.24 MetaWAdd.fs:
An implementation file including the `add` function.

```
module Meta
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
let add (x : float) (y : float) : float = x + y
```

A signature file may be automatically generated, as shown in Listing 9.25.

Listing 9.25: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo --sig:MetaWAdd.fsi MetaWAdd.fs

MetaWAdd.fs(4,48): warning FS0988: Main module of program is empty: nothing will happen when it is run
```

The warning can safely be ignored, since at this point it is not our intention to produce runnable code. The above listing has generated the signature file in Listing 9.26.

Listing 9.26 MetaWAdd.fsi:
An automatically generated signature file from `MetaWAdd.fs`.

```
module Meta
type floatFunction = float -> float -> float
val apply : f:floatFunction -> x:float -> y:float -> float
val add : x:float -> y:float -> float
```

We can generate a library using the automatically generated signature file by writing `fsharpc -a MetaWAdd.fsi MetaWAdd.fs`, which is identical to compiling the `.dll` file without the signature file. However, if we remove, e.g., the type definition for `add` in the signature file, then this function becomes private to the module and cannot be accessed outside. Hence, using the signature file in Listing 9.27, and recompiling the `.dll` with Listing 9.24 does not generates errors.

Listing 9.27 MetaWAddRemoved.fsi:
Removing the type defintion for `add` from `MetaWAdd.fsi`.

```
module Meta
type floatFunction = float -> float -> float
val apply : f:floatFunction -> x:float -> y:float -> float
```

Listing 9.28: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo -a MetaWAddRemoved.fsi MetaWAdd.fs
```

However, when using the newly created `MetaWAdd.dll` with an application that does not itself supply a definition of `add`, we get a syntax error, since `add` now is inaccessible to the application program. This is demonstrated in Listing 9.29 and 9.30.

Listing 9.29 MetaWOAddApp.fsx:
A version of Listing 9.3 without a definition of `add`.

```
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

Listing 9.30: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo -r MetaWAdd.dll MetaWOAddApp.fsx

MetaWOAddApp.fsx(1,25): error FS0039: The value or constructor
  'add' is not defined.
```

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

[6] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[8] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[9] George Pólya. *How to solve it.* Princeton University Press, 1945.

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index