Lec 11: Modules and Namespaces

CS220: Programming Principles

Sang Kil Cha

Modules and Namespaces

From a Single File to Multiple Files

So far, we have written a couple of functions within a single file. But what if we want to implement a real-world system? Can we put everything to a single file?

An example: Linux kernel has 230k LoC (Lines of Code).

What are the problems?

`fsproj` File

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="MyFile.fs" />
  </ItemGroup>

</Project>
```

Order of Items Matters

You can put as many items into `ItemGroup`, but you should be aware of their order. Each item can only see the other items above, but not below.

```
<ItemGroup>
  <Compile Include="A.fs" />
  <Compile Include="B.fs" />
</ItemGroup>
```

If type `Foo` is defined in `B.fs`, `A.fs` would not be able to access the type.

Adding Items to `fsproj`

- Directly modify the `fsproj` file.
- Or, use Visual Studio GUI.

`fs` File Requirement

- Create an empty file, named `MyType.fs`.
- Add a line to including `MyType.fs` to `fsproj` file.
- Compile (`dotnet build`) and see what happens.

```
error FS0222:  Files in libraries or multiple-file
applications must begin with a namespace or module
declaration ...
```

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Modules

A module is a group of code, such as types, functions, and values.

- Help organize code into logical groups.
- Prevent name collisions.
- Modules can be inside a namespace.

Declaring a Module

We can declare a module at the top of a file using the `module` keyword.

```
module MyModule

type MyType =
  | A
  | B

let add x y = x + y
```

Accessing Names in a Module

To access types and functions declared within a module, you should use a fully qualified name `MyModule.add`. However, if you have “opened” the module with the `open` keyword, you can directly access the names without specifying module names.

```
open MyModule

let x = add 1 2 // No need to use the full name
let y = MyModule.add 1 2 // although still possible
```

Multiple Modules in a File

We can have multiple modules within a single file.

```
module ModuleA =
  let add x y = x + y

module ModuleB =
  let add x y = - ((-x) - y)

  /// Nested module
  module ModuleC =
    let add x y = x + x + y - x
```

Nested Modules

Modules can be nested. We use dot (`.`) operator to access nested modules and their types and values.

For example, `ModuleB.ModuleC.add` refers to the `add` function defined in the `ModuleC` module, which is nested in the `ModuleB` module.

Namespaces

Namespaces are similar to modules, but they cannot directly contain values (and, thus, functions).

```
namespace MyNameSpace
module MyModule =
  let add x y = x + y // Indentation matters

let sub x y = x - y // Error. Why?
```

Implicit Declaration

Implicitly declare namespace.

```
// MyNameSpace is implicitly declared.
module MyNameSpace.MyModule

// Now we are in the MyModule module.
let add x y = x + y // No need to indent here.
```

Two Files in the Same Namespace

Unlike modules, multiple files can share the same namespace.

A.fs

```fsharp
namespace MyNameSpace

type OddOrEven =
  | Odd
  | Even
```

B.fs

```fsharp
module MyNameSpace.Checker

// No need to open MyNameSpace

let check n =
  if n % 2 = 0 then Even
  else Odd
```

Note: `A.fs` should appear earlier than `B.fs`.

Organizing Functions

Where Do I Put My Functions?

- Functions can be defined in another function.
- Functions can be grouped into ***modules***.
- There are other ways (using different paradigm), but we will learn them later as we would like to stick to the functional paradigm for now.

Nested Functions

```fsharp
let addThreeNumbers x y z =
  // A nested helper function
  let add a b = a + b
  // Actual logic of the function.
  x |> add y |> add z
```

Variable Scope

Nested functions can access its parent function’s parametes.

```
let addThreeNumbers x y z =
  // A nested helper function
  let addY n = n + y // y is within the scope of addY
  let addZ n = n + z // z is within the scope of addZ
  // Actual logic of the function.
  x |> addY |> addZ
```

Function Nesting is Great, but ...

Do not nest functions more than once!

Guess what this function do?

```
let f x =
  let a y =
    let b z =
      x * z
    let c z =
      let d z =
        y * z
      y * x
    c y
  x * a x // ???
```

Rule of Thumb

Avoid using nested functions unless necessary. For example, define your function in a nested manner, if the function is not likely to be used by other functions. However, there is another way to hide functions without function nesting: access control.

Access Control

Any values or types defined in a module can be exposed to other modules (i.e., files) or not. There are three access control specifiers: `public`, `internal`, and `private`.

- `public`: the entity can be accessed by all callers.
- `internal`: the entity can be accessed only from the same assembly[1].
- `private`: the entity can be accessed only from the enclosing module/file.

[1] An assembly is an executable binary that we can create by compiling a project (an `fsproj` file). So this means that the entity can be accessed by any files within the same project.

Example: Access Control

```
module MyNameSpace.A

// This function is accessible only within this module
let private add x y = x + y

let addThreeNumbers x y z =
  x |> add y |> add z
```

```
module MyNameSpace.B

let x = A.add 1 2 // Error.
```

Private Module

We can hide the entire module from other files. This is useful when we want to hide some helper functions that are not supposed to be used by other files.

A.fs

```
namespace MyNameSpace

module private A =
  let myadd x y = x + y

module B =
  let x = 42
  let y = 24
  let z = A.myadd x y // accessible only within the same file.
```

Signature Files

`.fsi` files can be used to access-control functions and types defined in a file. For any `.fs` file, you can create a corresponding `.fsi` file in order to define the ***interface*** for the `.fs` file.

A.fsi

```
module MyModule.A

val add: int -> int -> int
```

A.fs

```
module MyModule.A

let add x y = x + y

// Not accessible outside.
let sub x y = x - y
```

Why Signature Matters

One can easily understand the purpose of functions declared in a signature file without looking at the implementation. This is useful when you are developing a library and want to provide a documentation for the library.

A.fsi

```
/// Add two integers.
val add: int -> int -> int

/// Subtract two integers.
val sub: int -> int -> int
```

Documentation Comments in F#

Comments starting with three slashes (`///`) are documentation comments. We can use them to document our code and your IDE will show the documentation when you hover your mouse over a function/type[2].

[2] `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/xml-documentation`

Functional Design

Data vs. Behavior

Always separate behavior from types. In functional language, values are immutable, and they should be just values. All you need to do is to define functions that act on those data, and you group them into modules. You don't need to use data encapsulation (we will discuss this later) at all: just use transparent values.

Data vs. Behavior (cont'd)

- Data: `[1; 2; 3; 4]`
- Behavior: `List.map, List.filter, ...`

Attach Functions (Behavior) to Data

Consider we are writing a program for managing students for CS220. We represent students with a `Student` datatype as follows:

```
type Student = {
  FirstName: string
  LastName: string
  StudentID: int
}

module Student =
  let create first last id =
    { FirstName = first; LastName = last; StudentID = id }

  // we can add more behaviors here.
```

Mixing Types and Functions

A common pattern in F# is to define your types in a namespace, and create a module for each type to define functions associated with the types.

```
type OddNumber = OddNumber of int

module OddNumber =
  let create n = ...

type EvenNumber = EvenNumber of int

module EvenNumber =
  let create n = ...
```

In-Class Activity #11

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   ```
   - git clone https://github.com/KAIST-CS220/CS220-Main.git
   ```
2. Move in to the directory `CS220-Main/Activities`
   ```
   - cd CS220-Main
   - cd Activities
   ```

Problem

Fix the `convert` function to make `list1` and `list2` the same. Try to use higher-order functions defined the `List` module.

Common Modules and Namespaces

System

`Sysname` is a namespace that contains fundamental types and modules used by .NET programs. For example, `System.Char` is a module that includes many useful functions for manipulating characters.

System.Console

`System.Console` represents the standard input/output/error streams for console applications. One can use `Console.WriteLine` to print a string to the standard output stream.

```
open System
Console.WriteLine "Hello World"
printfn "Hello World" // same as above
```

printfn?

`printfn` is a function that prints a formatted string to the standard output stream[3].

```
printfn "%d %s" 1 "ABC"
printfn "%f" 3.14
```

[3]See: https://fsharpforfunandprofit.com/posts/printf/

Interpolated Strings

You can also embed F# expressions into a string using an interpolated string, which is a string that starts with $[4].

```
open System
let x = 42
Console.WriteLine $"Hello {x}"
Console.WriteLine $"Hello {x + x}"
Console.WriteLine $"Hello {{x}}" // escape
Console.WriteLine $"0x%08x{43962}" // formated
```

[4]https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/interpolated-strings

Cannot Open Some Modules?

You can open `System` as it is a namespace, but you cannot open `System.Console`! This is because `Console` is actually not a module but a class (we will learn later). The `open` expression is only applicable to F# modules. However, you can use `open type` to open a static class, too.

```
open type System.Console
WriteLine "Hello World"
```

Conclusion

Further Readings

- https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/entry-point
- https://fsharpforfunandprofit.com/posts/function-signatures/
- https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/access-control

Question?