OptionBuilder

In this first tutorial, we'll create a computation expression to remove the arrowhead pattern from working with `option` types.

In order to understand what we want to accomplish, we should also be familiar with how to do the task without a computation expression.

Without a Computation Expression

1. Remove `Sample.fs`.
2. Create a new file, `OptionBuilder.fs`.
3. Add the file to your `.fsproj` with `<Compile Include="OptionBuilder.fs" />` above `Program.fs`.
4. Add the following lines of code to `OptionBuilder.fs`:

```
module Options

open Expecto

let opt1 = Some 1
let opt2 = Some 2
let opt3 = Some 3
let opt4 = Some 4
let sum4 w x y z = w + x + y + z

let nested =
    match opt1 with
    | Some w ->
        match opt2 with
        | Some x ->
            match opt3 with
            | Some y ->
                match opt4 with
                | Some z ->
                    let result = sum4 w x y z
                    printfn "Nested: %d" result
                    Some result
                | None -> None
            | None -> None
        | None -> None
    | None -> None

let composed =
    opt1
    |> Option.bind (fun w ->
        opt2
        |> Option.bind (fun x ->
            opt3
```

```
            |> Option.bind (fun y ->
                opt4
                |> Option.map (fun z ->
                    let result = sum4 w x y z
                    printfn "Composed: %d" result
                    result
                )
            )
        )
    )

[<Tests>]
let tests =
    testList "OptionBuilder" [
        test "nested = composed" {
            Expect.equal nested composed "Expected nested to equal
composed"
        }
    ]
```

5. Build and run the program with `dotnet test`.

> **NOTE:** if you run into problems, notably an error regarding `EntryPointAttribute`, try the following: Rename `Main.fs` to `Program.fs` and make the same change in the `fsproj`. You may also remove the `module` declaration in the `Program.fs` file, but that should not have any material impact.

Objective

We will build an `OptionBuilder` to flatten the code we wrote above in `nested` and `composed` into the following:

```
let actual = maybe {
    let! w = opt1
    let! x = opt2
    let! y = opt3
    let! z = opt4
    let result = sum4 w x y z
    printfn "Result: %d" result
    return result
}
```

Empty builder

1. Add a new type, `type OptionBuilder() = class end`
2. Create an instance of the builder: `let maybe = OptionBuilder()`
3. Add a test to validate the maybe builder exists. We'll assert that it should support `return`, which we'll implement next, and that it should return the value provided.

```
type OptionBuilder() = class end

let maybe = OptionBuilder()

[<Tests>]
let tests =
    testList "OptionBuilder" [
        // ...
        // previous tests
        // ...

        test "OptionBuilder returns value" {
            let expected = 1
            let actual = maybe { return expected }
            Expect.equal actual (Some expected) "Expected Some 1"
        }
    ]
```

4. Build and run the program with `dotnet test`. Your program should fail to compile with the following error:

```
/Users/ryan/Code/computation-expressions-
workshop/solution/OptionBuilder.fs(53,34): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Return' method
```

`Return` a Value

The compiler informs us that in order to use the `return` keyword, we must implement the `Return` method on our builder.

Replace:

```
type OptionBuilder() = class end
```

with

```
type OptionBuilder() =
    member __.Return(value) = Some value
```

Build and run the program with `dotnet test`. Your tests should pass.

Composing `'a option` Values

`let` bindings binds a value to a name. Computation expressions provide a `let!` binding that can bind a value according to rules specified by the computation expression. This facilitates several possibilities:

- making a decision as to whether to continue or halt a computation
- side effects, e.g. printing to the screen or making a network call
- transform a result into another form

To find out how to implement `let!`, add a test:

```
        test "OptionBuilder can bind option values" {
            let actual = maybe {
                let! w = opt1
                let! x = opt2
                let! y = opt3
                let! z = opt4
                let result = sum4 w x y z
                printfn "Result: %d" result
                return result
            }
            Expect.equal actual nested "Actual should sum to the same
value as nested."
        }
```

Build and run the program with `dotnet test`. Your program should fail to compile with the following error:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/OptionBuilder.fs(61,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Bind' method
```

The compiler informs us that in order to use the `let!` keyword, we must implement the `Bind` method on our builder.

The F# Language Specification indicates that the `Bind` member should have the following signature (specialized for our immediate use case):

```
member Bind : 'a option * ('a -> 'b option) -> 'b option
```

As you either know or expect, this matches very closely with the signature of `Option.bind`:

```
module Option =
    val bind : ('a -> 'b option) -> 'a option -> 'b option
```

We can therefore implement `Bind` as follows:

```
    member Bind(m, f) = Option.bind f m // notice the parameter
orientation
```

We could also implement `Bind` like this:

```
    member Bind(m:'a option, f:'a -> 'b option) =
        match m with
        | Some x -> f x
        | None -> None
```

Build and run the program with `dotnet test`. Your tests should pass.

Expansion

It's worthwhile to pause here to understand what's happening. When the F# compiler encounters an instance of an object followed by `{}`, in this case `maybe`, it will attempt to expand it based on the computation expression rules. In our example above, the expansion ends up looking very similar to the `composed` value we expressed above:

```
let actual =
    maybe.Bind(opt1, fun w ->
        maybe.Bind(opt2, fun x ->
            maybe.Bind(opt3, fun y ->
                maybe.Bind(opt4, fun z ->
                    let result = sum4 w x y z
                    printfn "Result: %d" result
                    maybe.Return(result)
                )
            )
        )
    )
```

You can add a test to show that you could write this explicitly yourself:

```
        test "OptionBuilder instance can be used directly" {
            let actual =
                maybe.Bind(opt1, fun w ->
                    maybe.Bind(opt2, fun x ->
                        maybe.Bind(opt3, fun y ->
                            maybe.Bind(opt4, fun z ->
                                let result = sum4 w x y z
                                printfn "Result: %d" result
                                maybe.Return(result)
                            )
                        )
```

```
                    )
                )
            Expect.equal actual composed "Actual should sum to the same
value as nested."
        }
```

Aside from reading the specification, it can be useful to add traces or `printfn` statements into your CE builder definition while developing it. Let's do that now:

```
type OptionBuilder() =
    member __.Return(value) =
        printfn "maybe.Return(%A)" value
        Some value
    member __.Bind(m, f) =
        printfn "maybe.Bind(%A, %A)" m f
        Option.bind f m
```

Running the computation again should produce output that shows the calls made, and their order:

```
maybe.Bind(Some 1, <fun:actual@281-27>)
maybe.Bind(Some 2, <fun:actual@282-28>)
maybe.Bind(Some 3, <fun:actual@283-29>)
maybe.Bind(Some 4, <fun:actual@284-30>)
maybe.Return(10)
val actual : int option = Some 10
```

> **Observation:** the `maybe` instance is an object instance. While F# is a functional-first language, it also supports the .NET object model. Computation Expressions leverage this object model, and this can lead to some interesting possibilities we'll investigate later in the workshop.

Executing without Returning

F# is not a pure functional language, meaning you can perform side-effects like writing to the file system without returning a value indicating something like that happened. We looked briefly at `Async` earlier and will look at another CE that can make this explicit, but you may not want or need that.

Let's look at a case where you want to write a file to the file system if a path was provided and the path directory exists:

```
        test "OptionBuilder can exit without returning a value" {
            let fullyQualified path =
                let fileInfo = System.IO.FileInfo(path)
                let fileName = fileInfo.Name
                let pathDir = fileInfo.Directory.FullName.TrimEnd('~')
                if System.IO.Directory.Exists(pathDir) then
                    Some (System.IO.Path.Combine(pathDir, fileName))
```

```
                else None

            let maybePath = Some "~/test.txt"

            let actual =
                maybe {
                    let! path = maybePath
                    let! fullPath = fullyQualified path
                    System.IO.File.WriteAllText(fullPath, "Test
succeeded")
                }

            Expect.equal actual (Some ()) "Actual should be Some unit"
        }
```

Once again, the project fails to compile:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/OptionBuilder.fs(120,21): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Zero' method
```

`Zero` provides us with several conveniences and is one of the first examples of why you cannot always write a canonical computation expression for a given type. `Zero` allows you to return from the computation expression without explicitly returning a value. The most typical implementation for our `OptionBuilder` would be to return `Some ()`, as this indicates "success" with no value:

```
    member __.Zero() =
        printfn "maybe.Zero()"
        Some ()
```

Running `dotnet test` with this definition will now complete successfully, and you should see the following printed out:

```
maybe.Bind(Some "~/test.txt", <fun:actual@455-41>)
maybe.Bind(Some "/Users/ryan/Code/computation-expressions-
workshop/test.txt", <fun:actual@456-42>)
maybe.Zero()
```

`Zero` will also allow an `if ... then` without an `else`:

```
        test "OptionBuilder supports if then without an else" {
            let maybePath = Some "~/test.txt"
```

```
            let actual =
                maybe {
                    let! path = maybePath
                    let pathDir = System.IO.Path.GetDirectoryName(path)
                    if not(System.IO.Directory.Exists(pathDir)) then
                        return "Select a valid path."
                }

            Expect.equal actual (Some "Select a valid path.") "Actual
should return Some(\"Select a valid path.\")"
        }
```

Unfortunately, the compiler doesn't like what we've done here:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/OptionBuilder.fs(465,21): error FS0001: Type mismatch.
Expecting a
    'string option'
but given a
    'unit option'
The type 'string' does not match the type 'unit'
```

You can probably see the issue: `Zero` currently returns `Some ()`, but our `return` expression returns a `string`. A better solution for this case is to have `Zero` return `None`:

```
    member __.Zero() = None
```

Running `dotnet test` now compiles and runs with one test failure b/c our previous test expected to return `Some ()`. Change that to `None`, and your tests will pass.

> **Observation:** computation expressions are not always one-size-fits-all. Our tests pass, but you can no longer use the result from the previous test in a continuing chain of computations, as it will return `None`, which will currently cause our `maybe` computations to propogate the `None`. You may find you need multiple `OptionBuilder` CEs for different use cases.

Returning a Computation

Before moving on to our next exercise, let's consider how you might escape a `maybe` computation early. Let's say that based on a condition, you want to immediately get a `None` and avoid any further computation. How might you do this? In order to return an `'a option` from our computation, we need to use `return!`. This is similar to `return` in the same way `let` and `let!` are similar. `return!` handles `'a option` just like `let!` handles an `'a option`, where as their `!`-less counterparts deal with `'a` values.

```
        test "OptionBuilder allows for early escape with return!" {
            let actual =
                maybe {
```

```
                    if true then
                        return! None
                    else
                        let! w = opt1
                        let! x = opt2
                        let! y = opt3
                        let! z = opt4
                        let result = sum4 w x y z
                        printfn "Result: %d" result // print if a result
was computed.
                        return result
                }
            Expect.equal actual None "Should return None immediately"
        }
```

Add the test above and run `dotnet test`. The build will fail with the following:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/OptionBuilder.fs(146,25): error FS0708: This control
construct may only be used if the computation expression builder defines a
'ReturnFrom' method
```

`ReturnFrom` is a very simple method to implement:

```
    member __.ReturnFrom(m) = m
```

or to be a bit more precise and add our tracing print statements while we are building this up:

```
    member __.ReturnFrom(m:'a option) =
        printfn "maybe.ReturnFrom(%A)" m
        m
```

Adding this method and running `dotnet test` should result in a successful test run.

Review

We implemented the following builder methods for `OptionBuilder`:

- `Return`
- `Bind`
- `Zero`
- `ReturnFrom`

We also observed that a given method, e.g. `Zero`, does not necessarily have a canonical implementation.

In the next exercise, we'll look at how we might be able to overcome our choice of `None` as a result for `Zero` to continue, rather than cancel, execution of a computation involving `'a option`.