Exceptions, Loops, Disposals, and Quotations

We have covered most, but not all, of the members that make up computation expressions. Fortunately, these are not defined as part of any interface, with rigid type signatures, as we would otherwise have already stumbled into several issues with our previous exercises. on the other hand, these next members are all typically implementable with the same, basic signature, regardless of computation expression.

While these members have not been core to our previous exercises, they can be used almost entirely on their own to solve some very useful problems. Expecto's `test` CE -- which we have been using in all our tests thus far -- is almost entirely composed of only these members.

In this exercise, we'll build our own `MyTestBuilder` to demonstrate how to write similar helpers. Feel free to also go back to the previous builders and add these members if you like.

1. Create a new file, `ExceptionsLoopsDispoal.fs`.
2. Add the file to your `.fsproj` with `<Compile Include="ExceptionsLoopsDisposal.fs" />` just below `Sequences.fs`.
3. Add the following lines to the `ExceptionsLoopsDisposal.fs` file:

```
module ExceptionsLoopsDisposal

open Expecto

type MyTestBuilder() = class end

let myTest = MyTestBuilder()

[<Tests>]
let tests =
    testList "my tests" [
        testCase "A simple test" (fun () ->
            let expected = 4
            Expect.equal expected (2+2) "2+2 = 4"
        )
    ]
```

Delaying Execution

We want to achieve the following syntax:

```
        myTest "A simple test" {
            let expected = 4
            Expect.equal expected (2+2) "2+2 = 4"
        }
```

Change your current test to reflect our desired syntax with `myTest`.

This will work, but the test will evaluate immediately, which we don't want. Based on the standard `testCase` from Expecto above, we want to delay execution until we are ready to run the test. We know how to do that already:

```
type MyTestBuilder() =
    member __.Delay(f) = f
    member __.Run(f) = testCase name f
```

This of course won't compile, as we don't have a `name` value available, and we don't want to hard-code it. Fortunately, we know how to create class types, and class constructors can take parameters for the class. We also see from our desired syntax that our CE needs to accept a string, i.e. the `name` parameter:

```
type MyTestBuilder(name) =
    member __.Delay(f) = f
    member __.Run(f) = testCase name f

let myTest name = MyTestBuilder(name)
```

Close but not quite, as the compiler is still unhappy:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(21,13): error FS0708: This
control construct may only be used if the computation expression builder
definesa 'Zero' method
```

`Zero` is required as we never return a value, and the computation needs some sort of return value. Tests always return `unit`, so this implementation is very simple:

```
type MyTestBuilder(name) =
    member __.Delay(f) = f
    member __.Run(f) = testCase name f
    member __.Zero() = ()
```

Run `dotnet test`, and you should see all tests pass.

Exceptions

Much like F# exception handling `try ... with` and `try ... finally`, computation expressions provide two members for dealing with exception handling: `TryWith` and `TryFinally`.

`TryWith`

Add the following test and attempt to build the program again:

```
        myTest "myTest supports try...with" {
            try
                raise(System.Exception("Failed!"))
            with
            | e -> Expect.equal e.Message "Failed!" "Expected an exception
with message \"Failed!\""
        }
```

You should see the following error:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(28,13): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'TryWith' method
```

The `TryWith` member exposes the `try ... with` syntax within the CE. It's signature is:

```
M<'T> * (exn -> M<'T>) -> M<'T>
```

> **NOTE:** I've never seen a variation from this signature, though I suspect there are ways to vary the `M`, just as with `Bind`, `Combine`, etc.

The basic implementation strategy is something along the lines of:

```
    member __.TryWith(tryBlock, withBlock) =
        // May need to return a lambda taking a parameter, e.g. `State`
        try tryBlock // may need to pass a parameter, e.g. `State` or
`Expecto`
        with e -> withBlock e
```

Other strategies may be required based on the wrapper or computation type, but this general strategy typically serves the purpose.

> **NOTE:** You can find some other variations in the sample CE in the Computation Expressions docs and the FSharpx.Extras `Continuation` module.

For our builder, the implementation follows the basic implementation strategy:

```
    member __.TryWith(tryBlock, withBlock) =
        try tryBlock()
        with e -> withBlock e
```

If you are wondering why you must call the `tryBlock`, the answer is in the expansion:

```
builder.TryWith(builder.Delay({| cexpr |}), (fun value -> match value with
| pattern_i -> expr_i | exn -> reraise exn)))
```

Our computation type is simply `unit`, as is easily identified by the type returned by `Zero`. However, `TryWith` will expand to call `Delay`, which will, based on our definition of delay, turn the incoming expression into a function of type `unit -> unit`. In order to evaluate the result, we must therefore call the provided function.

TryFinally

The other half of exception handling is cleanup with `try ... finally`. Add the following test and attempt to build:

```
        myTest "myTest supports try...finally" {
            let mutable calledFinally = false
            try
                try
                    raise(System.Exception("Failed!"))
                // without this, the test will fail anyway,
                // as the exception will trigger a test failure.
                with _ -> ()
            finally
                calledFinally <- true
            Expect.isTrue calledFinally "Expected test to call finally
block"
        }
```

This fails, as expected, and indicates `TryFinally` is required:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(39,13): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'TryFinally' method
```

The `TryFinally` member exposes the `try ... finally` syntax within the CE. It's signature is:

```
M<'T> * (unit -> unit) -> M<'T>
```

The basic implementation strategy is something along the lines of:

```
    member __.TryFinally(tryBlock, finallyBlock) =
        // May need to return a lambda taking a parameter, e.g. `State`
        try tryBlock // may need to pass a parameter, e.g. `State` or
```

```
`Expecto`
        finally finallyBlock()
```

Variations on this strategy are similar to those noted above for `TryWith`. Like `TryWith`, `TryFinally` expands to call `Delay`:

```
builder.TryFinally(builder.Delay( {| cexpr |}), (fun () -> expr))
```

We need follow only the basic strategy to get our test to pass:

```
    member __.TryFinally(tryBlock, finallyBlock) =
        try tryBlock()
        finally finallyBlock()
```

Run `dotnet test` and:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(42,13): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'Combine' method
```

Well, almost. We've seen `Combine` before:

```
    member __.Combine(f1:'a, f2: unit -> unit) =
        f2()
        f1
```

This looks a little different than what we've done before. This `Combine` exists to support the compensating `finallyBlock`, so we need to run that and return whatever result was provided as the first parameter.

Running `dotnet test` *now* should result in a successful test run.

Disposal

Another common pattern in .NET coding is resource disposal. CEs have you covered, as well. In order to verify this works, we need to add a test dummy just above the `tests`:

```
[<AllowNullLiteral>]
type ObservableDisposable() =
    member val IsDisposed = false with get, set
    interface System.IDisposable with
```

```
        member this.Dispose() =
            this.IsDisposed <- true
```

Add the following test:

```
        myTest "myTest supports use" {
            let disp = new ObservableDisposable()
            do use d = disp
               ()
            Expect.isTrue disp.IsDisposed "Expected the instance to be
disposed"
        }
```

The build fails with:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(60,21): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'Using' method
```

`Using` is another straightforward member to implement. While it looks like the `use` keyword, you must define this similar to the `using` function:

```
// NOTE: the documentation is incorrect on which type must be disposable.
'T * ('T -> M<'U>) -> M<'U> when 'T :> IDisposable
```

For reference, the `using` function has the following signature:

```
'T -> ('T -> 'U) -> 'U when 'a :> IDisposable
```

Add the following implementation for `Using`:

```
    member __.Using(disposable:#System.IDisposable, f) =
        try
            f disposable
        finally
            match disposable with
            | null -> ()
            | disp -> disp.Dispose()
```

> **NOTE:** You can also implement this with the `TryFinally` member, which adds a `Delay` in its expansion, but that's typically not necessary.

Run the tests with `dotnet test`, and you should see they all pass.

Looping

We looked at `for` in the previous exercise but neglected to implement it based on its documented signature(s):

```
seq<'T> * ('T -> M<'U>) -> M<'U> or
```

```
seq<'T> * ('T -> M<'U>) -> seq<M<'U>>
```

The following, silly test should help us verify our implementation:

```
        myTest "myTest supports for" {
            for i in 1..10 do
                Expect.equal i i "i should equal itself within a for loop"
        }
```

The build will fail with the following error:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(76,13): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'For' method
```

The solution for our `MyTestBuilder` is simply to iterate and pass each value to the callback:

```
    member __.For(sequence, f) =
        for i in sequence do f i
```

This solution should pass your test.

The other form of looping, other than recursion, is the `while`. Here's another loop test for this syntax:

```
        myTest "myTest supports while" {
            let mutable i = 1
            while i <= 10 do
                Expect.equal i i "i should equal itself within a for loop"
                i <- i + 1
        }
```

Again, we get a build failure message:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(84,13): error FS0708: This
control construct may only be used if the computation expression builder
defines a 'While' method
```

`While` has the following signature:

```
(unit -> bool) * M<'T> -> M<'T>
```

The first parameter is a predicate function. For our current builder, the solution is once again simple:

```
    member __.While(pred, f) =
        while pred() do f()
```

Unfortunately, things are rarely this easy when dealing with looping constructs. Note above that the syntax takes a computation, e.g. `'a option`, `State<'a, 's>`, etc., and you will quickly see that the solutions above for `For` and `While` are sufficient only for simple builders that have the type `unit`.

Were we to implement these for one of the other types we've seen previously, we would need to `Bind` the values together in order to use them meaningfully. Let's try another approach for `While`:

```
    member __.While(pred, body) =
        if pred() then
            __.Bind(body, (fun () -> __.While(pred,body)))
        else __.Zero()
```

This is a recursive function (and really is begging to be made a helper outside of the builder). It also requires we implement `Bind`, which we have avoided:

```
    member __.Bind(m:unit -> unit, f) =
        m()
        f()
```

Once again we see an odd signature for a member. This one takes a delayed computation and the binder function `f`, executes it and then calls `f`, as there really is nothing to bind.

Running `dotnet test` shows our tests still pass.

However, we are not quite done. Our `While` will work for other CEs, but our `For` will not. In order to define a `For` that binds properly, we need to implement it in terms of `Bind`, as well. We should also probably dispose of the provided sequence, if necessary.

> **NOTE:** it's *possible* to implement `For` in other ways, as shown previously. We'll look at yet another implementation in the next section.

```
    member __.For(items:seq<_>, f) =
        __.Using(items.GetEnumerator(), (fun enum ->
            __.While((fun () -> enum.MoveNext()),
                     __.Delay(fun () -> f enum.Current))))
```

This is a lot to take in all at once.

1. Explicitly `Using` the `IEnumerator<_>` from the `seq<_>`, which is an `IDisposable`
2. `For` then uses `While` to iterate using `IEnumerator<_>.MoveNext()`
3. The body of the `While` uses a `Delay`ed function that calls `f` with the `IEnumerator<_>.Current` value

This change also requires we change our "myTest supports for" test to use an explicit collection type implementing `seq<_>`:

```
        myTest "myTest supports for" {
            for i in [1..10] do
                Expect.equal i i "i should equal itself within a for loop"
        }
```

With these changes, our tests still pass when running `dotnet test`.

These changes weren't required for our `MyTestBuilder`, but they do provide more general strategies for implementing these members for other types. However, you ***should*** use the minimum implementation necessary, both to keep things simple and for performance.

Quotations

Our last member is fairly uncommon, or so I've found. It's documented in the language specification but not in the language reference. I'm not aware of any way to trigger a build failure that informs you of the member to implement. Similar to `Run`, `Quote` will be added if it is implemented and ignored otherwise. The relevant section of the language specification can be found at the bottom of page 67 and top of page 68:

> If the inferred type of `b` has one or more of the `Run`, `Delay`, or `Quote` methods when `builder-expr` is checked, the translation involves those methods. For example, when all three methods exist, the same expression translates to:
>
> ```
> let b = builder-expr in b.Run (<@ b.Delay(fun () -> {| cexpr |}C) >@)
> ```

> If a `Run` method does not exist on the inferred type of `b`, the call to `Run` is omitted. Likewise, if no `Delay` method exists on the type of `b`, that call and the inner lambda are omitted, so the expression translates to the following:
>
> ```
> let b = builder-expr in b.Run (<@ {| cexpr |}C >@)
> ```
>
> Similarly, if a `Quote` method exists on the inferred type of `b`, at-signs `<@ @>` are placed around `{| cexpr |}C` or `b.Delay(fun () -> {| cexpr |}C)` if a `Delay` method also exists.

Let's see what will happen if we add it to our builder:

```
    member __.Quote(q:Quotations.Expr<_>) = q
```

If you try to run the build now, you'll see it turn red with errors such as:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ExceptionsLoopsDisposal.fs(60,9): error FS0193: Type
constraint mismatch. The type  'Quotations.Expr<('a -> unit)>'    is not
compatible with type    'unit-> unit'
```

You could resolve this issue by evaluating the quotation within the `Run` member. As that's slightly beyond the scope of this exercise, make a mental note to think on this feature later.

Review

We have completed our review of the available computation expression members by re-implementing the `TestBuilder` from `Expecto`, or something like it. The new members we covered in this section were:

- `TryWith`
- `TryFinally`
- `Using`
- `While`
- `Quote`

We also saw new signatures for `For` and `Bind` and investigated different approaches to implementing `For` and `While` depending on the type used by the builder.

By now, you should have enough examples to see that the computation expression feature is ***very*** flexible and can accommodate an impressive number of variations. In the next section, we'll exercise this flexibility even more by extending existing CEs in new ways.