Understanding Language Ext

by Stuart Mathews

Contents

Part I: Monads

Introduction

This is a tutorial that aims to demonstrate the fundamentals behind using LanguageExt in a practical fashion though step-by-step tutorials which introduce and then build up on concepts.

Furthermore, the tutorial shows how to use ideas such as pipelineing, delcarative style coding and walks through the fundamentals behind Select() SelectMany() and Bind() and Map() while covering implementation via Linq's Fluent and Linq Expression syntax.

The general tutorial is structured like this:

- Monads
- Map/Bind
- Fluent/Expression Linq
- Pipelining
- Declarative Style
- Monad Validation
- Function Composition
- Pure Functions
- Basics of Either<L,R>
- Operations with Lists Eithers<,>
- Basics of Option
- ThrowIfFailed()
- FailureToNone()

Bonus: Functional programming concepts

- Caching - Using pure functions to cache things, ensuring that you need not call expensive calls if they've been done once already.
- Changing object state over time
- Immutability

Note: To run any specific tutorial, right-click on the project in the solution explorer and 'Set as start-up project'

Acknowledgements

Language.Ext library was created by Paul Louthy. See https://github.com/louthy/language-ext

References:

- https://stackoverflow.com/questions/28139259/why-do-we-need-monads Why do we need monads?
- https://github.com/louthy/language-ext/wiki LanguageExt Wiki
- http://www.stuartmathews.com/index.php/component/tags/tag/functional-prograqmming My articles on Functional programming(old)

Scope

Monad Basics

- Tutorial01 - Introduction to the Box type (a Monad)
- Tutorial02 - Shows you how to use Map and Bind (also construction of a Mondad Type)
- Tutorial03 - Shows how Bind is used to create a pipeline of function calls
- Tutorial04 - Performing operations on a Box using Map() and Bind(), Select() and introducing SelectMany()
- Tutorial05 - More examples of performing operations on a Box
- Tutorial06 - This tutorial shows you how pipelining is used to call funtions using Linq Fluent and Expression syntax
- Tutorial07 - Shows you when to use Map() and Bind()
- Tutorial08 - Shows you how to transition from Imperative style coding to Declarative style coding with an example
- Tutorial09 - Shows you that pipelines include automatic validation
- Tutorial10 - Expands on Tutorial09 to show that transformation function always return a Monad
- Tutorial11 - Shows how monad built in validation, affords short-cuircuiting functionality.
- Tutorial12 - Composition of functions
- Tutorial13 - Pure Functions - immutable functions with now side effects ie mathematically correct

Language.Ext

Either<Left,Right> Basics

- Tutorial14 - Shows the basics of Either<L,R> using Bind()
- Tutorial15 - Using BiBind()
- Tutorial16 - Using BiExists()
- Tutorial17 - Using Fold() to change an initial state over time based on the contents of the Either
- Tutorial18 - Using Iter()
- Tutorial19 - Using BiMap()
- Tutorial20 - Using BindLeft()
- Tutorial21 - Using Match()

Operations on Lists of Either<left, Right>

- Tutorial22 - Using BiMapT and MapT
- Tutorial23 - Using BindT
- Tutorial24 - Using IterT
- Tutorial25 - Using Apply
- Tutorial26 - Using Partition
- Tutorial27 - Using Match

Option Basics

- Tutorial28 - Introduction to Option
- Tutorial29 - Basic use-case of Option
- Tutorial30 - Using Option in functions (passing in and returning)
- Tutorial31 - using IfSome() and IfNone()
- Tutorial32 - Creates an entire application of just functions via pipelineing which returning and receive Option
- Tutorial33 - Using ToEither<>
- Tutorial34 - Using BiMap() - see tutorial 19

Custom Specific

- Tutorial35 - Using custom extension method ThrowIfFailed() and introducing Either<IAmFailure, Option> as a standard return type for all functions
- Tutorial36 - Using custom extension method FailureToNone()
- Tutorial37 - Using pure functions to cache things, ensuring that you need not call expensive calls if they've been done once already. (Bonus: FP concepts)
- Tutorial38 - Changing state of an object over time (Fold) including concept of apply events over time to change an object (Bonus: FP concepts)
- Tutorial39 - Immutability - Designing your objects with immutability in mind: Smart constructors and Immutable data-types (Bonus: FP concepts)
- Tutorial40 - Try - Supressing Exceptions

Todo

- TutorialA - Partial Functions - Allowing multiple arguments to be 'baked' in and still appear as Math like functions (pure functions)
- TutorialB - Threading and parallelism benfits
- TutorialC - Guidelines for writing immutable code, starting with IO on the fringes (bycicle spoke design)
- TutorialD - Custom useful Monad Extensions

Basics

Introduction to the Box type (a Monad)

A Monad is a just type, much like any user-defined type that is created when designing a class. It has its own properties that represent its state, and methods that give the Monad behaviour and, in many cases, manipulates its state through a public interface it provides to the programmers to use.

A simple example of a Monad might be a Box type:

```
Box<int> myNumberBox1 = new Box<int>(1);
```

A *Box<T>,* can hold any type but in this case, it holds and makes provision for an integer type, i.e. *Box<int>*.

Look at the implementation of the *Box<T>*, notice it's just a C# Class that's been created:

```
/// <summary>
/// A box can hold a thing only
/// </summary>
/// <typeparam name="T">The type of the thing</typeparam>
public class Box<T>
{
    public Box(T newItem)
    {
        Item = newItem;
        IsEmpty = false;
    }

    public Box() { }

    private T _item;

    public T Item
    {
        get => _item;
        set
        {
            _item = value;
            IsEmpty = false;
        }
    }

    public bool IsEmpty = true;
}
```

So far this looks like a normal C# class for the Box type. You can create a *Box<T>* by specifying the type of contents it can hold, and internally to the class, it is stored as a member called *_item*. Additionally, you can set the contents of the *Box<T>* through its *Item* property. Thus, is it possible to extract and set the contents of the *Box<T>*:

```
myNumberBox1.Item = 99
```

Monads are types that allow you to transform the contents of the Monad, in this case the contents of the *Box<T>* which is currently set to 99. A series of defined functions need to exist on a Monad which will define the kinds of transformations that can occur – they are expected to exist. If a type has a *Map, Select* and a *Bind* function, it can *generally* be considered a Monad.

Transforming the contents of a Monad

The important thing to understand is that whatever the transformation will be, it needs to be provided by the programmer. Calling one of these functions, will then read the contents of the Monad and change the contents somehow – the programmer will provide a *Higher-Ordered Function* (HOF), which will define the transformation. The monadic functions, such as *Select/Map* or *Bind* acts an execution environment, passing its internal contents into the HOF, where additional checks can be done before or after the transformation is started. The HOF is known as the transformation function and is provided by the programmer using the Monad.

The *Select* function does exactly this, when called on the *Box<T>,* it passes the contents of the *Box<T>* monad into the user-defined transformation function and executes that function, and the results thereof i.e. the *transformation* become part of the result of the *Select* function call. Lets see how it does this:

```
public static class BoxMethods
{
    /// <summary>
    /// Transforms the contents of a Box, in a user defined way
    /// </summary>
    /// <typeparam name="TA">The type of the thing in the box to start with</typeparam>
    /// <typeparam name="TB">The result type that the transforming function to transform to</typeparam>
    /// <param name="box">The Box that the extension method will work on</param>
    /// <param name="map">User defined way to transform the contents of the box</param>
    /// <returns>The results of the transformation, put back into a box</returns>
    public static Box<TB> Select<TA, TB>(this Box<TA> box, Func<TA, TB> map)
    {
        // Validate/Check if box is valid(not empty) and if so, run the transformation function on it,
           otherwise don't
        if (box.IsEmpty)
        {
            // No, return the empty box
            return new Box<TB>();
        }

        // Extract the item from the Box and run the provided transform function ie run the map()
           Function ie map is the name of the transformation function the user provided.
        TB transformedItem = map(box.Item);

        return new Box<TB>(transformedItem);
    }
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial01_monad/Box.cs

TA is the type of the content of the Box i.e. *Box<TA>, TB* is the type that *TA* will be transformed into, *box* is the *Box<T>* type where conceptually *T* is replaced with *TA*. The last parameter *map* is the user-provided function aka *the transformation function.*

In this example, the transformation function is called *map*, which will be provided by the programmer defined as *Func<TA, TB>* which is any function accepting a type of *TA* (which also happens to be the type which Box holds), and which returns a *TB* type as its transformation result.

You'll notice how the *Box<T>*'s *Select* function wraps the supplied transformation function, *map()* by first checking if the *Box<T>* is empty, and if so returns an empty *Box<TB>,* otherwise executes the map function, producing a transformation of the content. The transformation is represented by the return type of *TB*, as the original type of the content of the box is *TA*.

An important distinction is that the transformation result is placed into a *new Box<T>* but specifically as a transformed representation, *Box<TB>* and the original *Box<T>*'s content is not modified during the *Select* function's operation. This is a preview the concept of immutability, and of how other Monad functions that

accept a user-defined transformation function - like *Bind,* will work.

Using the Select function on a Monad

Box<T>'s *Select* function is called by the programmer like this:

```
Box<int> result2 = myNumberBox1.Select(x => x + 1);
```

The *x => + 1* syntax is a definition of a lambda function that takes in one argument and the result of the expression after the *=>* is the result, and as such it is a suitable transformation function to pass into the *Select* function, provided it matches the *map* parameter's types as defined by *Func<TA, TB>.*

Incidentally, this notation of explicitly calling the *Select* function, as defined on the *Box<T>* type is called the *Linq Fluent Syntax.* An alternative is to use the *Linq Query Expression Syntax*:

```
var result = from number1 in myNumberBox1
             select number1 + 1;
```

This form, will fetch or *select* a value from the box by using the *Box<T>*'s *Select* function (this is done implicitly) and passing the expression immediately after the *select* statement i.e. *select number1 +1* as the mapping function for the *Select* function on the type, in this case a *Box<T>.*

As noted previously, the *'Linq Expression Syntax*' and requires *Box<T>* to have a *Select* function for it to work in this way.

Any type you define that has a *Select* function defined on it as an extension method can be used in both Linq Fluent and Expression syntax forms.

The resulting program illustrates the concepts discussed thus far:

```
static void Main(string[] args)
{
    // A Box, can hold any type but in this case it holds and makes provision for an int.
    // Look at the implementation of the Box, notice its just a C# Class I've created.
    Box<int> myNumberBox1 = new Box<int>(1);

    // I can look into the Box
    Console.WriteLine($"The contents of my NumberBox is initially is '{myNumberBox1.Item}'");

    // But this Box is different. It can be used in a Linq Expression.
    // This is because it has a special function defined for it called Select().
    // Have a look at Box class again, check out the Select() extension method at the bottom.
    // This single function allows the following Linq usages:

    // Fetch or Select() a value from the box by using the Box's Select() function implicitly
    var result = from number1 in myNumberBox1
        select number1 + 1; // This is called the 'Linq Expression Syntax' and requires Box<T> to
                            have a Select() function for it to work

    // This is called the Linq Fluent syntax - does the same thing as the above
    Box<int> result2 = myNumberBox1.Select(x => x + 1); // x=> x+1 is the transformation function
                                                           also known as a mapping function

    // Have a Look at Box class again, and specifically the Select() extension method again
       and try and understand what its doing.
    // Hint: Its doing two things 1) allowing you to pass a function to it that it will run
       for the item in the box 2) only doing 1) if
    // the contents of the box is valid (so it does some validation)

    Console.WriteLine($"'{result.Item}'");
    Console.WriteLine($"'{result2.Item}'");
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial01_monad/Program.cs

Many Monads in Language-Ext are utilized by taking advantage of C#'s *Query Expression Syntax* notation for accessing the *Select* function in this way. This is also largely due to being able to define an extension method for any type this way.

When working with Language-Ext in general, the already defined monad types such as *Option<T>*, *Either<L,R>* and *Try<T>* already have suitable *Select*, *Map* and *Bind* functions defined and as such its not needed to define them, and only need to pass in the transformation function, that is unless you want to create our own Monad like our *Box<T>* monad, and in which case, you now know how.

Why do we need to perform transformations? Traditional programming revolves around having functions. Almost every programming language allows you to define functions which take input and process it and then produce and output. This is what transformation is. So instead of defining explicit hard-coded functions in your source code, you can now pass around functions – transformation functions. Monads are datatypes that allow you to work on their contents but providing the transformation function, where the input of the transformation function will be the contents of the Monad. Monads to a little bit more, they can run checks to validate if that function should run or not, and how it will behave. In this was Monads provide monadic functions like Select, Map and Bind to house the incoming transformation function that the user will provide, and acts as an execution vehicle for running that function and passing to it, its own contents.

Next, we'll put the Select function aside, and discuss its relatives *Map* and *Bind*.

Understanding Map and Bind

Now we're going to look into the other monadic functions, map and bind. Like select, the both allow the programmer to provide a customer transformation function. The difference between the two is how that transformation is returned to the programmer.

As suggested previously, like Select, both Map and Bind serve as an execution environment for the transformation function provided by the user, and they follow a certain interesting pattern before performing the transformation function. This can be described using the acronym VETL or Validate, Extract, Transform and Lift/Leave. These phases all apply to the how the *Select*, *map* and *bind* functions work and dictate how they ultimately call the user-defined transformation function.

Let's have a look at *Map* and *Bind* for the *Box<T>* Monad. As show previously with *Select*, these are just two extension methods on the Box<T> type. Also shown is the previously talked about, *Select* method which shows how it too features the same VETL convention.

```
public static class BoxMethods
{
    /// <summary>
    /// Validate, Extract, Transform and Lift (If Valid)
    /// </summary>
    public static Box<TB> Select<TA, TB>(this Box<TA> box, Func<TA, TB> map)
    {
        // Validate
        if (box.IsEmpty)
            return new Box<TB>();

        // Extract
        var extracted = box.Item;

        // Transform
        TB transformedItem = map(extracted);

        // Lift
        return new Box<TB>(transformedItem);
    }

    /// <summary>
    /// Validate, Extract, Transform and Lift (If Valid)
    /// Check/Validate then transform to T and lift into Box<T>
    /// </summary>
    public static Box<TB> Bind<TA, TB>(this Box<TA> box, Func<TA, Box<TB>> bind /*liftAndTransform*/)
    {
        // Validate
        if(box.IsEmpty)
            return new Box<TB>();

        //Extract
        TA extract = box.Item;

        // Transform and the user-defined function (notice that its up to the user defined function to 'lift'
        // any result of the transformation into a new Box)
        Box<TB> transformedAndLifted = bind(extract); //  should return its results of its transformation in a Box

        return transformedAndLifted;
    }

    /// <summary>
    /// Validate, Extract, Transform and automatic Lift (If Valid)
    /// </summary>
    public static Box<TB> Map<TA, TB>(this Box<TA> box, Func<TA, TB> select /*Transform*/)
    {
        // Validate
        if(box.IsEmpty)
            return new Box<TB>();

        // Extract
        TA extract = box.Item;

        // Transform
        TB transformed = select(extract); // user provided function does not need to 'lift' its result
                                          // into a Box like Bind() requires

        // Lift
        return new Box<TB>(transformed);
    }
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial02_transformations_1/Box.cs

First, there is no difference between *Map* and *Select*. They are identical functions, and function as previously described i.e. it *validates* if the *Box<T>* is empty, it *Extracts* the content of the Monad, our *Box<T>*'s *_Item* member in this case, runs the user-provided transformation function on that content and *Lifts* the

result of the transformation into a new Box object. In this way, it follows the above pattern of *VETL*.

Map and *Bind* do much the same, however they differ only in how they return the transformed result i.e. the Lift/leave phase. In all cases the transformation function is run, however the requirements of the transformation's functions arguments and return type have changed:

Bind requires the programmer to define and provide a transformation function that will take as input the item provided to it by the *Bind* function, but crucially it needs to transform it in such a way that the result is a new *Box<TB>* while *Map*, only requires the transformation function to return the transformation from TA to TB without needing to place it into a new *Box<TB>*.

Fundamentally, what this means is that both Map and Bind do the same transformation, but package up the result in different ways.

Let have a look at this in an example

```
static void Main(string[] args)
{
    // Now my Box contains another kind of thing, a list of integers.
       So effectively my Box is a box of numbers!
    Box<int[]> numbers1 = new Box<int[]>(new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    Box<int[]> numbers2 = new Box<int[]>(new[] { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 });

    // Now have a look at Box class again, it has changed a little with two new extension methods.
    // Notice how the it (the extension methods) follows an interesting trend of doing VETL or
    // Validate content, Extract Content, Transform Content and Lift the content

    // Validate, Extract, Transform and YouLift(If Valid)
    // See how the user define transformation passed to bind(), this currently just ignores the
    //  extracted contents in the Box and returns a new set of numbers as dictated by MyFunction()

    Box<int[]> transformedResult1 = numbers1.Bind(contents => MyFunction(contents));

    // Validate, Extract, Transform and automatic Lift (If Valid)
    // Same transformation result, but we our transformation function didn't
    // We have to return a Box (we used map to run our transformation which automatically will put
       the result of our transform function in a new Box)

    Box<int[]> transformedResult2 = numbers2.Map(contents => MyFunction2(contents));


    // The Box class is now considered a Monad, because it has these two additional functions
    // defined on it (map and bind). Note in both cases of Bind() and Map() we did a transformation
    // of the contents ie we provided a function that would work on the item in the Box(or Monad)

    // In both cases we didn't doing anything with the contents of the box, so we didn't really
    // transform the contents(we just used what Map() and Bind() extracted from the box)
    // We could have included the contents in our transformation and manipulated it...

    // The take away is that Map() and Bind() do the same thing but Bind() requires you to put
    // your transformation result back in the Box, while Map doesn't.
    // Both methods 'manage' your user defined transform function by running it only if it
    // deems it should (validation passes) and then depending on the specific function,
    // it will either lift the result of the transformation(map) or require that your
    // transformation function's signature explicitly says it will it it itself (bind)

    // This means, with a you don't have to return a Box (like you do when you with Bind), when
       transforming with the Map() function...it automatically does this for you

    Console.WriteLine($"The result of the Bind operation was {transformedResult1.Item}");
    Console.WriteLine($"The result of the Map operation was {transformedResult2.Item}");

    // But here is something sneaky
    // Look, we've been able to change the type of the Box from a int[] to a string!
    // This is by virtue of the fact that its what your transformation function returned,
    // and it transformed the return type also...and put it back into a box
    // Its still in a Box, but is a box of a string now instead of a box of numbers.... Map() and
    // Bind() can do this, and this is what makes these function really at transformations

    Box<string> transformedResult3 = numbers1.Map(contents => "I'm a string!");
}

private static int[] MyFunction2(int[] numbers)
{
    // Used as a map() transformation, so no need to lift into a Box or anything...
    return new int[] {3, 4, 5};
}

private static Box<int[]> MyFunction(int[] integerArray)
{
    // Notice that this function if its going to be used in Box's Bind() function (and is indeed
       compatible with it - see Bind function signature), needs to return a new Box<T> ie lift into
        a Box() again
    return new Box<int[]>(new int[] {1, 2});
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial02_transformations_1/Program.cs

Now the Boxes contains another kind of thing, a list of integers. So effectively the Boxes are boxes containing numbers. Two *Box<int[]>* variables are defined that contain a list of numbers. As the Box<*T*> type has *Map* and *Bind* defined for it, we can call them by providing a transformation functions i.e. *MyFunction()* and *MyFunction2()*.

Notice how *MyFunction* takes in a list of numbers, i.e *int[]* which is the *T* in the *Box<T>* definition when *Box<T>* is defined as *Box<int[]>*. This is the same for *MyFunction2*. The difference is what they *return*, and as a consequence, limits each one to being used in either *Map* i.e *MyFunction2* or *Bind* i.e *MyFunction*. They can't switched around because *Map* and *Bind* have different type requirements for the transformation functions.

Here the contents of the Boxes are fed into the transformation functions as lambda expressions:

```
Box<int[]> transformedResult1 = numbers1.Bind(contents => MyFunction(contents))
Box<int[]> transformedResult2 = numbers2.Map(contents => MyFunction2(contents))
```

Ultimately the return type of the enclosing Map and Bind functions, i.e. the vehicles for the user-defined function need to return a Monad or a *Box<int[]>* in this case. So, depending on if we're calling *Map* or *Bind*, internally these two functions need to know if the transformation function will lift the transformation result into the Monad type or not, hence why *Map* and *Bind* function *lift* or *leave* the result of transformed content respectively.

How *Map* or *Bind* does this is by expecting a certain return-type of the user-provided transformation function and either automatically *lifting* the result into a *Box<T>* where the transformation function does not do that, as is the case with the *Map* function, or *leaving* the user-provided transformation function to do so, as is the case with the *Bind* function. This is why they require different forms of the user-provided transformation function, but ultimately the transformation is the same for both map and bind. In many cases you can use them interchangeably with the exception that the supplied lambda function or transformation function meets the requirements of either returning a Monad or not.

The take away is that *Map*() and *Bind*() do the same thing but *Bind*() requires you to put your transformation result back in the Box, while *Map* doesn't.

Both methods 'manage' your user defined transform function by running it only if it deems it should (if validation passes) and then depending on the specific function, it will either lift the result of the transformation(*Map*) or require that your transformation function's signature explicitly says it will do it itself (B*ind*)

This means, with a you don't have to return a Box (like you do when you with *Bind*), when transforming with the *Map*() function...it automatically does this for you

But here is something sneaky but very useful to know: The user-provided transformation function by definition can change the return type of the content it is transforming:

```
Box<string> transformedResult3 = numbers1.Map(contents => "I'm a string!");
```

Look, we've been able to change the type of the Box from a *int[]* to a *string*!

This is by virtue of the fact that it's what your transformation function returned, and it thus transformed the return type also, and put it back into a box(because that's what map does). It's still in a *Box<T>* but is a box of a *string* now instead of a box of numbers....

Map() and *Bind*() can do this, and this is what makes these function really useful for chaining transformations together. His behaviour allows use to instead of designing hard coded functions like other programming languages like C or Pascal does, we can instead pass our functions into our Monads when designing logic that works with them, instead of passing out Monads into function, which is the old way of thinking.

We'll explore this next but before working further on out Monad Box, why do we need Monads? Why do we have to have Map() and Bind()..

Here is why: https://stackoverflow.com/questions/28139259/why-do-we-need-monads

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial02_transformations_1/Box.cs

Pipelining transformation workloads (3)

We will continue to define and explain our Monad type, *Box<T>* and showing how transformations are use in real time as well as previewing an implicit behaviour of Monads: short-circuiting.

Short-circuiting works when chaining or cascading multiple *Bind*() or *Map*() transformations together as part of a pipeline. Lets look at an example, which we'll go through shortly:

```
static void Main(string[] args)
{
    // Before working further on out Monad Box, read this :
    // Why do we actually need Monads? Why do we have to have Map() and Bind()
    // Here is why? https://stackoverflow.com/questions/28139259/why-do-we-need-monads

    Box<int> numberHolder = new Box<int>(25);
    Box<string> stringHolder = new Box<string>("Twenty Five");

    //Validate, Extract, Transform, lift
    Box<string> result1 = numberHolder.Map(i => " Transformed To a string automatically");

    // Transform the contents of the box by passing it down a series of Bind()s
    // so there are multiple associated VETL->VETL->VETL steps that represents the Binds()
    // effectively representing a pipeline of data going in and out of Bind(VETL) functions
    Box<string> result2 = stringHolder
        .Bind(s => new Box<int>(2)) //Validate Extract Transform lift
        .Bind(i => new Box<int>()) // Validate step only
        .Bind(i => new Box<string>("hello")); // Validate step only

    // Remember that the 'validate' step is implicit and is actually coded directly into the Bind()
    // or Map() functions. The validity of a box in this case, is if it is empty or not(look at the
    // Bind functions' code that explicitly checks this),
    // It is empty(invalid) it will not run the the user provided transformation function, otherwise
    // it will.

    // This is an example of 'short-circuiting' out-of the
    // entire set of transformations early. So if the First Bind short-circuits, the next Binds will
    // too and so they will not run

    Console.WriteLine($"The contents of result 2 is {result2.Item}");
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial03_pipelines/Program.cs

As can be seen in the example, we're creating a pipeline of *Bind* operations, each one doing its defined transformation as provided by the user, and each time the *Bind* returns a Monad, which is a valid type for the next *Bind* extension method to work on. These cascade into a series of transformations, but this has hidden and useful feature: short-circuiting.

Put differently, this code transform the contents of the box by passing it down a series of *Bind*() operations,

so there are multiple associated VETL->VETL->VETL steps that represents the *Binds*()

effectively representing a pipeline of data going in and out of *Bind*(VETL) functions

Remember that the 'validate' step is implicit and is actually hard-coded directly into the *Bind*() or *Map*() functions that we previously looked at.

The validity of a box in this case, is if it is empty or not (remember that at the Bind functions' code that explicitly checks this),

If it is empty(invalid) it will not run the user provided transformation function, otherwise it will.

This is an example of *'short-circuiting'* out of the pipeline of stacked transformations early on. So, if the first *Bind* short-circuits, i.e. the box is empty (the validation phase fails), the next *Bind* or *Map* will do the same and so they will not run the user-provided transformation function, as the validation phase failed.

When the validation phase fails, it returns a 'bad' result which is what other bind function will detect during their validation phase, in this case whenever an empty Box is encountered, skipping the transformation execute phase and returning the 'bad' or empty box to the next bind in the pipeline – which too will return early. In this way, the pipeline has short-circuited and none of the other remining transformation functions are run.

Many Language.Ext monads such as Option<T>, Either<L,R> and Try<T> work like this.

Again, you will notice, that during each transformation, it is possible to change the content's type in the monad being passed into the next bind, as previously discussed.

Next we'll discuss the SelectMany() function which like Map/Select and Bind is used to transform the contents of monads, and like the those, it has a new requirement on the user-provided transformation function.

Monadic functions: Complete example (4)

Now let's look at a complete example showing all the operations on a *Box<T>* using *Map*() and *Bind*() and introducing a new *SelectMany*() extension method on *Box<T>:*

```
static void Main(string[] args)
{
    Box<int[]> boxOfIntegers = new Box<int[]>(new []{ 3, 5, 7, 9, 11, 13, 15});
    var doubled1 = DoubleBox1(boxOfIntegers); // Use Bind
    var doubled2 = DoubleBox2(boxOfIntegers); // use Map
    var doubled3 = DoubleBox3(boxOfIntegers); // Use SelectMany via linq expression syntax
    var doubled4 = DoubleBox4(boxOfIntegers); // Use Select via linq expression syntax
}

/// <summary>
/// Transform(double) the contents of a box ie get the value extracted from box using Bind,
/// and then transform(though using Bind() will not automatically lift the transformed value,
/// so you need to do that).
/// You'll need to lift the transformed value into a Box again
/// </summary>
/// <param name="boxOfIntegers"></param>
/// <returns></returns>
private static Box<int[]> DoubleBox1(Box<int[]> boxOfIntegers)
{
    // If the transformed result is already lifted, which it is as DoubleNumbers() already returns
    // a Box, use Bind to achieve the transform without an explicitly lift into the Box being
    // applied
    return boxOfIntegers.Bind(numbers => DoubleNumbers(numbers));
}

static Box<int[]> DoubleNumbers(int[] extract) // transform Extracted, and Lift it
{
    // Remember a Select() run a function on the item in the box
    // Also note that the Select() function here is provided by the .NET runtime and it not
    // the Select() we wrote for the Box type, as extract is of type int[].
    // This version of select behaves similarly to what the Box's Select does - we just can't see
    // it (it runs the user provided transform function)
    return new Box<int[]>(extract.Select(x => x * 2).ToArray());
}

/// <summary>
/// Transform (by doubling) the contents of a box ie. the value extracted from it using Map,
/// which will happily and automatically lift the transformed value into a box so you don't have to.
/// </summary>
private static Box<int[]> DoubleBox2(Box<int[]> boxOfIntegers)
{
    // Use Map to automatically lift transformed result
    return boxOfIntegers.Map(numbers => DoubleNumbersNoLift(numbers));
}

/// <summary>
/// Extract the contents of the box and then transform it using SelectMany via the Linq Expression syntax
/// </summary>
private static Box<int[]> DoubleBox3(Box<int[]> boxOfIntegers)
{
    // We can use the SelectMany() extension method to Validate, Extract, and transform its
    // contents. Have a look at Box's SelectMany() implementation now and realize that its this that
    // is used to allow this 'double from from' Linq expression construct, that you see from time to
    // time in peoples code

    return
        from extract in boxOfIntegers
        let transformedAndLifted = DoubleNumbers(extract) // bind() part of SelectMany() ie
                                                          // transform extracted value
        from transformed in transformedAndLifted
                select transformed; // see internals of SelectMany function
                                    // --> project(extract, transformedAndLiftedResult) as this
                                   // select statement is this project() function in SelectMany imp
                                   // lementation

/* Note: we are not using 'extract' value in this project function (the final select), just the transformed value we could have used in
         during our transformation, because it in scope and is accessible to be included
*/
}

/// <summary>
/// Shows how our Select() is called in a linq expression using from
/// </summary>
/// <param name="boxOfIntegers"></param>
/// <returns></returns>
private static Box<int[]> DoubleBox4(Box<int[]> boxOfIntegers)
{
    Box<int[]> t  = from extract in boxOfIntegers
                    select DoubleNumbersNoLift(extract); // Remember Select() does the
                                                         // lift!

    return t;
}

// As this does not return a Monad, it can be used as a transformation function, that is
// passed to Map()
static int[] DoubleNumbersNoLift(int[] extract)
=> extract.Select(x => x * 2).ToArray();
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial04_methods_1/Program.cs

This example starts by showing the passing of Monads into functions. Each function *DoubleBox* n..4 will demonstrate a different variant to help you get familiar with their usages and occurances.

DoubleBox1 calls *DoubleNumbers* which accepts a Box<int[]> Monad and transforms it using *Bind*, thus *DoubleNumbers* needs to return a *Box<T>* type.

DoubleBox2 instead using *Map* which uses a necessary function *DoubleNumbersNoLift* to show that that function does not return a *Box<T>* and as such the *Map* function can be used with it, which will lift it into a Monad automatically.

These are familiar based on what we've learnt so far and it puts this into practise, in a practical fashion.

DoubleBox3 however uses Box<T>'s *SelectMany* function implicitly through its use in the *Linq Query Expression Syntax*.

This is new, we will discuss this now. Lets look at how SelectMany() is defined, as we've already seen how Select/Map and Bind look.

```
/// <summary>
/// Validate, Extract, Transform, Project (Transform, Extract) and automatic Lift
/// </summary>
public static Box<TC> SelectMany<TA, TB, TC>(this Box<TA> box, Func<TA, Box<TB>> bind, Func<TA, TB, TC>
project)
{
    // Validate
    if(box.IsEmpty)
        return new Box<TC>();

    // Extract
    var extract = box.Item;

    // Transform and LiftTo
    Box<TB> liftedResult = bind(extract);

    if(liftedResult.IsEmpty)
        return new Box<TC>();

    // Project/Combine - This forms the select statement in the Linq Expression
    TC t2 = project(extract, liftedResult.Item);
    return new Box<TC>(t2);
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial04_methods_1/Box.cs

You'll notice that this again, looks remarkably like the other monadic function, Select/Map and Bind in as much as they follow the Validate, Extract, Transform, and Lift/Load process. However, with SelectMany there is an additional step: Once the transformation function is run, bind in this case, a 2nd new function can be provided by the user, namely project.

Project will take the newly transformed and lifted Monad, along with the original content and allow that to be transformed into a 3rd Type, TC.

Its not clear yet why this additional bootstrapping is required until you look at how the Linq Query Expression Syntax is used and how powerful it is: it allows this function's internals (as read above) to be exposed and manipulated dynamically in LINQ. Here is an example:

```
/// <summary>
/// Extract the contents of the box and then transform it using SelectMany via the Linq Expression
syntax
/// </summary>
private static Box<int[]> DoubleBox3(Box<int[]> boxOfIntegers)
{
    // We can use the SelectMany() extension method to Validate, Extract, and transform its
    // contents. Have a look at Box's SelectMany() implementation now
    // and realize that its this that is used to allow this 'double from from' Linq expression
    // construct, that you see from time to time in peoples code

    return
        from extract in boxOfIntegers
        let transformedAndLifted = DoubleNumbers(extract) // bind() part of SelectMany() ie
                                                          // transform extracted value
        from transformed in transformedAndLifted
                 select transformed; // see internals of SelectMany function
                                     // --> project(extract, transformedAndLiftedResult) as this
                                     // select statement is this project() function in
                                     // SelectMany implementation

// Note: we are not using 'extract' value in this project function (the final select), just the
// transformed value we could have used in during our transformation, because it in scope and is
// accessible to be included */
}

/// <summary>
/// Like DoubleBox3 but shows that you dont have to put the same type (int[]) of item back
/// in the box, you can put any type of item back in the box (string)
/// </summary>
private static Box<string> DoubleBox11(Box<int[]> boxOfIntegers)
{
    return
        from start in boxOfIntegers
        from startTransformed in DoTransformToAnyBox(start) // bind() part of SelectMany() ie
                                                            // transform extracted value
        select start + startTransformed; // Project(extract, transformedAndLifted)
                                         // part of SelectMany

    // local function
    Box<string> DoTransformToAnyBox(int[] input)
    =>  new Box<string>($"{input.Sum()}");

}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial04_methods_1/Program.cs

DoubleBox3 and *DoubleBox11* are equivalent with *DoubleBox3* being the function used in the previous example. *DoubleBox11* is used here to highlight how it actually works:

The first *from* statement calls *Select* on boxOfIntegers and exposes the *Extract* phase of the call, effectively extracting the contents of *boxOfIntegers* into *start*. The transformation function has not been run yet - the transformation function is only run in the 2nd *from*, see the invocation of the user-provided transform function, *DoTransformToAnyBox* which now using the item extracted from the first *from, start* as its input(as one would expect for a transformation function compatible with the bind function). This is effectively the user-provided transform function used for passing into Bind. This results in the transformed result, *startTransformed.*

So far, this has shown how the Linq Query Syntax can open up both the *Select*() and *Bind*() functions and expose it as a Linq Query Expression. The next part is is the select statement which is effectively the *project* function called implicitly. The *project* function has access to the start variable as well as the *startTransformed* variable (its in scope) and as such it can use it to perform a transformation expression using the two, as per the project function definition:

```
// Project/Combine
TC t2 = project(extract, liftedResult.Item);


return new Box<TC>(t2);
```

Note too, that the result of the *project* transformation function is put into a *Box<T>* and so just like *Bind* and *Map*, need to return a Monad.

This has shown how defining a *SelectMany*() function on the *Box<T>* monad, allows LINQ access to specifying the *bind* transformation function inline as well as the *projection* transformation function in one LINQ query expression. The benefit of doing transformation this way is clearly that you can define transformation functions inline and have access to previous expression results, as they remain in scope!

This becomes more useful when you see it in an example with multiple from statements, simulating chaining of the transformations:

```
Box<object> doubled3 =
                from extract in boxOfIntegers
                from transformed in DoubleNumbers(extract)
                from transformed2 in DoubleNumbers(transformed)
                from transformed3 in DoubleNumbers(transformed2)
                from transformed4 in DoubleNumbers(transformed3)
                    select Dosomethingwith(transformed, transformed2, transformed3, transformed4);
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial07_transformations_2/Program.cs

We will investigate examples like this later, but for now its useful to see how the LINQ query expression syntax can reach deep into *Select*(), *Bind*() and *SelectMany*() and provides an alternative to specifying transformation and chaining the results.

Next, we'll bring what we've learnt thus far into a cohesive example:

```
static void Main(string[] args)
{
    // A Box
    Box<int[]> boxOfIntegers = new Box<int[]>(new[] { 3, 5, 7, 9, 11, 13, 15 });
    Box<int[]> boxOfNewIntegers = new Box<int[]>(new[] { 3, 5, 88, 29, 155, 123, 1 });

    // Do something with or to the Box, uses user defined function specified in the form of
    // lambdas to the Map() and Bind() functions

    // Extract, Validate and transform using Bind()
    var doubled1 = boxOfIntegers
                    .Bind(extract => new Box<int[]>(extract.Select(x => x * 2).ToArray()));

    // Extract, Validate and transform using Map()
    var doubled2 = boxOfIntegers
                    .Map(numbers => numbers.Select(x => x * 2).ToArray());

    // Extract, Validate and transform using SelectMany()
    var doubled3 = from extract in boxOfIntegers
                   from transformed in DoubleNumbers(extract) // bind() part of SelectMany()
                                                              // ie transform extracted value
        select transformed; // project(extract, transformedAndLiftedResult) part of SelectMany

    var doubled4 = from extract in boxOfIntegers
        select DoubleNumbers(extract).Extract; // Use Select via linq expression syntax

    // Note we can use Map or Bind for transformation, but it becomes necessary to choose/use a
    // specific one depending on if or not the provided transformation function returns a box or not
    // (lifts or doesn't),  ie is transformed in a call to Bind() or Map()
    Box<int[]> doubleDouble1 = boxOfIntegers
        .Bind(numbers => DoubleNumbers(numbers)) // need to use a transformation function that will
                                                 // lift
        .Map(DoubleNumbers) // need to use a transformation that does not already lift
        .Bind(box => box.Bind( numbers => DoubleNumbers(numbers) )); // same as above bind() case

    // Using Linq query syntax
    var doubleDouble2 = from numbers in boxOfIntegers
        from redoubled in DoubleNumbers(numbers)  // transformation function needs to lift
        select redoubled;  // Box's Select() function will do the lift here into Box<int[]> so
                           // no need to in this line

    // Give me a box of Double Double of my Box
    var doubleDouble3 =
      from firstDoubleTransformation in DoubleMyBox(boxOfIntegers)
      from secondDoubleTransformation in DoubleNumbers(firstDoubleTransformation) //VET: bind part
                                                                                   // of SelectMany()
        select secondDoubleTransformation; // project(reDouble, firstDouble)
}

/// <summary>
/// Takes a Box of numbers and produces a box of doubled numbers
/// </summary>
private static Box<int[]> DoubleMyBox(Box<int[]> boxOfIntegers)
=> from extract in boxOfIntegers
            from doubledNumber in DoubleNumbers(extract)
                select doubledNumber;


// transform Extracted, and Lift it
static Box<int[]> DoubleNumbers(int[] extract)
=> new Box<int[]>(extract.Select(x => x * 2).ToArray());
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial05_methods_2/Program.cs

Notice how transformation functions passed to Bind return Monads explicitly, while those passed to Map do not. Also Notice we can use either Map or Bind for transformation within a pipeline, but it becomes necessary to choose/use a specific one depending on if or not the provided transformation function returns a box or not (lifts or doesn't), i.e. is transformed in a call to Bind() or Map().

Next, we'll spend more time concentrating on using the LINQ fluent vs Expression syntax styles when working with monads.

Using LINQ Fluent and Expression syntax (6)

The following shows the two approaches, which yield the same result but look syntactically different. Remember, the LINQ expression syntax allow access to previous transformation results, while the fluent notation, only provides you access to the last transformation:

```
static void Main(string[] args)
{
    Box<int[]> boxOfIntegers = new Box<int[]>(new[] { 3, 5, 7, 9, 11, 13, 15 });
    Box<int[]> boxOfNewIntegers = new Box<int[]>(new[] { 3, 5, 88, 29, 155, 123, 1 });

    // Do something with or to the Box

    Box<object> doubled1 = from extract in boxOfIntegers // extract items out
        from transformed in DoubleNumbers(extract) // bind() part of SelectMany() ie transform extracted value (and
below):
        from transformed2 in DoubleNumbers(transformed) // use/transform/doublenumbers() the last transformed result
        from transformed3 in DoubleNumbers(transformed2) // use/transform/doublenumbers() the last transformed result
        from transformed4 in DoubleNumbers(transformed3) // use/transform/doublenumbers() the lsat transformed result
            select DoSomethingWith(transformed, transformed2, transformed3, transformed4); // this calls
                                                                                           // project(extract,
                                                                                           // transformed) part of
                                                                                           // SelectMany which always
                                                                                           // does an automatic result
                                                                                           // of the projected result
                                                                                           // (as a box<>)

    Box<object> doubled2 = boxOfIntegers
        .Bind(extract => DoubleNumbers(extract))
        .Bind(transformed => DoubleNumbers(transformed))
        .Bind(transformed => DoubleNumbers(transformed))
        .Bind(transformed => DoubleNumbers(transformed)) // I have to use map next because DoSomethingWith() does not
                                                         // return a Box and the result of a Map(will always do that)
                                                         // or Bind must do make its transform function do that

        .Map(lastTransformedFromAbove => DoSomethingWith(lastTransformedFromAbove));

}

// Perfect/valid Map function to use with a Map, as map will automatically lift this and so this function does
// not have to lift its result
private static object DoSomethingWith(params int[][] varargs)
{
    // Note we dont have to return a Box<> because as we''ll be running within a SelectMany() expression - it
    // automatically lifts the result in this case a object type
    return new object();
    // Note that this function will be acting as the bind() transformation function within the SelectMany() function
    // defined for the Box class (see SelectMany())
}

// transform Extracted, and Lift it.
// Perfect bind function to be used in a Bind() because it lifts the result
static Box<int[]> DoubleNumbers(int[] extract)
{
    /* do something with the numbers we extracted from the box and then put them back in the box again
     because this function will be run in the bind() phase of the SelectMany() function (see the selectMany function
     implementation) and that function requires a signature of :  int[] => Box<int[]>
    */
    return new Box<int[]>(extract.Select(x => x * 2).ToArray());
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial06_Linq/Program.cs

Shown: Using the select many way, i.e. the LINQ expression syntax as shown below allows an extracted item from the box, then to be passed down a series of transforms by way of those transform functions being compatible with the bind() phase of the *SelectMany*() function.

Each can see the prior transformation and can act on it subsequently.

And as each transformation function is run as part of the *SelectMany*()'s implementation in Box, it will also be subject to the VETL phases, which means if the input is not valid, it will return an invalid value and subsequent transforms upon receiving that invalid input will also return an invalid input and in all those cases, the underlying transform is not run (short-circuiting). This is also an illustration of Lazy-evaluation,

where a function does not need to be run unless is necessary (i.e. valid in this case).

I need to use *Map* in the transformation pipeline of *doubled2* because *DoSomethingWith*() does not return a Box and the result of a *Map*(will always do that).

The *doSomethingWith* function, represents a valid *Map* function to use with a *Map*, as map will automatically lift this and so this function does not have to lift its result. Note we don't have to return a *Box<T>* because as we"ll be running within a *SelectMany*() expression - it automatically lifts the result, in this case an *object* type. Note that this function will be acting as the bind() transformation function within the *SelectMany*() function defined for the Box class (see *SelectMany*())

The *DoubleNumbers* function is a valid bind function to be used in a *Bind*() because it lifts the result.

This function does something with the numbers we extracted from the box and then put them back in the box again because this function will be run in the bind() phase of the *SelectMany*() function (see the selectMany function implementation) and that function requires a signature of : *int[] => Box<int[]>*

When to use *Map()* or *Bind()* (7)

As an extension of the previous example, this shows you how to structure and deal with situations when choosing what function will be used in the bind(), map() functions and how choices you make impact the subsequent invocations of those functions when pipe-lining or chaining these functions together

```
Box<int[]> doubleDouble1 = boxOfIntegers
       .Bind(numbers => DoubleNumbers(numbers)) // Non automatically lifted result, so DoubleNumbers
                                                   will have to lift for it to be compatible with
                                                   Bind(), and it has to lift because subsequent
                                                   Map or Bind need to work from a lifted value
                                                   i.e a Box<>

    .Map(DoubleNumbers) // now we have DoubleNumbers lifting the transformed value and becasue
                           we used Map to transform it, it automatically lifts it again... so we
                           really should have used bind() here but no matter, we'll deal with it:

   .Bind(box => box.Bind( numbers => DoubleNumbers(numbers) )) // due to to the double lift, ie
                                                                  Box<Box<>> we first the extracted
                                                                  Item is a Box<Box<>> which is a
                                                                  Box<> so to extract something from
                                                                  that box that i need to use a map
                                                                  or bind, in this case I chose
                                                                  bind, which additionally will not
                                                                  lift and I can use Bind next:
    .Bind(numbers => DoubleNumbers(numbers))
    .Bind(numbers => DoubleNumbers(numbers));
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial07_transformations_2/Program.cs

This shows that you can use either *Map* or *Bind* (they do the same thing i.e. both transform their input) but map will lift automatically and *Bind*() needs its transform function to explicitly do that.

This choice of using *Map* or *Bind* will only impact on how the associated transformation functions require to produce either a lifted (*Box<T>)* or a non-lifted result.

Next, we'll look into defining the imperative and declarative style. Imperative refers the procedural view of having functions, and passing data into them, while declarative refers to having data and passing functions into them, *a-la* Monads!

Transitioning from Imperative to Declarative style (8)

Moving from traditional programming approaches which centre around defining imperative construction of logic within programs, we well look at what is required to convert from a Procedural way of thinking to declarative and as a consequence a Pipeline way of thinking.

As previously discussed, pipelining allows the movement of data from one Monad to the next, through a series of transformations that occurs through the pipelines. This simulates what procedural code does by calling function upon function to establish a set of logic for a program. We can do this using declarative style also:

```
static void Main(string[] args)
{
    // 2 Boxes!
    Box<int[]> boxOfIntegers = new Box<int[]>(new[] { 3, 5, 7, 9, 11, 13, 15 });
    Box<int[]> boxOfNewIntegers = new Box<int[]>(new[] { 3, 5, 88, 29, 155, 123, 1 });

    int[] portfolioIds = new int[]{ 77,88,99 };
    DateTime HoldingFrom = new DateTime(2019, 07, 25);

    // Procedural

    // 1. Go get the portfolio names for the following portfolio ids:
    Portfolio[] portfolios = GetPortfoliosByIds(portfolioIds);

    // 2. Get the holdings for these portfolios which are dated later than xyz
    Portfolio[] portfoliosWithHoldings = PopulatePortfolioHoldings(portfolios, HoldingFrom);

    // 3. do something with these populated portfolios
    var result = DoSomething(portfoliosWithHoldings);

    // Pipeline way of thinking

    // Linq Fluent syntax way
    Box<Portfolio[]> result2 = GetPortfoliosByIds1(portfolioIds) // note same input, portfolioIds can be
                                                                 // used but this is returning a box now
        .Bind(ports => PopulatePortfolioHoldings1(ports, HoldingFrom))
        .Map(ports => DoSomething(ports).ToArray());// notice you can use the same name of the
                                                    // transformed item as the last(ports)


    // Linq Expression syntax way
    // Notice you cannot use the name of last transform again when using linq query expression syntax
    Box<Portfolio[]> result3 = from ports in GetPortfoliosByIds1(portfolioIds)
                               from ports1 in PopulatePortfolioHoldings1(ports, HoldingFrom)
                                select DoSomething(ports1);
}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial08_declarative_style/Program.cs

```
/// <summary>
/// do something
/// </summary>
private static Portfolio[] DoSomething(Portfolio[] portfoliosWithHoldings)
=> portfoliosWithHoldings.Select(p =>
    {
        p.Name = p.Name.ToUpper();
        return p;
    }).ToArray();

private static Portfolio[] PopulatePortfolioHoldings(Portfolio[] portfolios, DateTime holdingFrom)
{
    foreach( var portfolio in portfolios)
        portfolio.Holdings = GetPortfolioHoldingsFrom(portfolio, holdingFrom);

    return portfolios;
}

/// <summary>
/// Returns a Box
/// </summary>
private static Box<Portfolio[]> PopulatePortfolioHoldings1(Portfolio[] portfolios, DateTime holdingFrom)
{
    var listOfPortfolios = portfolios.Select(p=>
    {
        p.Holdings = GetPortfolioHoldingsFrom(p, holdingFrom);
        return p;
    }).ToArray();
    return new Box<Portfolio[]>(listOfPortfolios);
}


private static DateTime[] GetPortfolioHoldingsFrom(Portfolio portfolio, DateTime holdingFrom)
{
    Dictionary<int, List<DateTime>> portfolioHoldings = new Dictionary<int, List<DateTime>>
    {
        { 1, new List<DateTime> { new DateTime(2019,07,23),new DateTime(2019,07,24), new DateTime(2019,07,25), new DateTime(2019,07,26)  } },
    { 2, new List<DateTime> { new DateTime(2019,07,23),new DateTime(2019,07,24), new DateTime(2019,07,25), new DateTime(2019,07,26)  } },
    { 77, new List<DateTime> { new DateTime(2019,07,23),new DateTime(2019,07,24), new DateTime(2019,07,25), new DateTime(2019,07,26)  } },
    { 88, new List<DateTime> { new DateTime(2019,07,23),new DateTime(2019,07,24), new DateTime(2019,07,25), new DateTime(2019,07,26)  } },
    { 99, new List<DateTime> { new DateTime(2019,07,23),new DateTime(2019,07,24), new DateTime(2019,07,25), new DateTime(2019,07,26)  } },
    };

    return portfolioHoldings
        .Where(x=>x.Key == portfolio.Id)
        .SelectMany(x=> x.Value)
        .Where(x=>x >= holdingFrom).ToArray();
}

public static Portfolio[] GetPortfoliosByIds(int[] portfolioIds)
{
    Dictionary<int, Portfolio> portfolios = new Dictionary<int, Portfolio>
    {
        { 1, new Portfolio("Portfolio1",1) },
        { 2, new Portfolio("Portfolio2", 2) },
        { 77, new Portfolio("Portfolio77", 77) },
        { 88, new Portfolio("Portfolio88", 88) },
        { 99, new Portfolio("Portfolio99", 99) },
    };

    return portfolios.Where(x=>portfolioIds.Contains(x.Key))
        .Select(x=>x.Value)
        .ToArray();
}

public static Box<Portfolio[]> GetPortfoliosByIds1(int[] portfolioIds)
{
    Dictionary<int, Portfolio> portfolios = new Dictionary<int, Portfolio>
    {
        { 1, new Portfolio("Portfolio1",1) },
        { 2, new Portfolio("Portfolio2", 2) },
        { 77, new Portfolio("Portfolio77", 77) },
        { 88, new Portfolio("Portfolio88", 88) },
        { 99, new Portfolio("Portfolio99", 99) },
    };
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial08_declarative_style/Program.cs

As can be seen from the above example, using either LINQ fluent or query syntax as the base of defining a declarative transformations using Monads to express logic.

Monad validation (9)

When using a Monad like *Box<T>* in a *SelectMany* statement like LINQ expression style used above, remember the implementation of the Monads *SelectMany*(), i.e for each bind() phase of the *SelectMany*(), that phase requires a function such PopulatePortfolioHoldings1() and any following ones to transform the extracted item and then put it back into a Box, so thats why each function must return a Box<> in the pipeline.

Also note how the logical way of planning the steps can be replicated in both the procedural and pipeline ways (you don't have to think that differently)

But wait, I can make a pipeline too without chaining Bind() or Map() statements too? Why can't I just do this:

```
var result1 = DoSoething(PopulatePortfolioHoldings(GetPortfoliosByIds(portfolioIds), HoldingFrom));
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial09_automatic_validation/Program.cs

The problem with this is you don't get automatic validation i.e. (V)ET it's not short-circuit-able.

You could do it but it would mean every function would need to have internal validation and check for invalid data whereas in a Monad like Box, that validation is built into the Select and SelectMany() implementation as thus us automatic each usage of those function on the monad!

This is another key aspect of using monads.

Returning Monads: transformations always return a Monad (10)

Let's have a bit of a recap:

Map and *Bind* both extract and validate the item within the box (that it's not empty) i.e. do VETL and then proceeds to run the transform function on it if it's not empty, otherwise returns empty box.

Both are equivalent in as much as they perform VETL but differ in what form they require their transform function to either lift or not lift the transformation (the function prototype must match what a *Map* or *Bind*() function requires)

In the LINQ Query Expression Syntax method, Box's *SelectMany*() is used to transform successive transformations, and you have access to each of the transformed results, as well as much earlier transformations. The final select statement is run via the *Box<T>'s Select*() function (technically this is the *projection*() function) and therefor it will automatically be lifted and you don't need to do it.

The Fluent mechanism also uses *Box<T>*'s Map and Bind functions.

Each fluent style *Map* and *Bind* has access to the last transformation before it, and unlike the LINQ expression syntax cannot see before the last transformation (as that is the only input it gets).

Transformations from a call to *Bind* and *Map* must result in a *Box<T>* either explicitly via *Bind*() or automatically via *Map*()

Your logical planning or thinking of logical programming tasks in your design can equally be represented procedurally and using pipelining.

Another key aspect when using monads, particularly when they are involved in successive chained calls such as within a pipeline, is that each returns a monad that the other uses as input.

Extending the last example by removing the transformation function and exposing them as inline lambda expressions, you can see how each must conform the function prototypes for transformation functions within the Map() or Bind() functions that exist on Box<T>:

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial10_returning_monads/Program.cs

Built-in validation and short-circuiting in Monads (11)

We now turn to a specific example of short-circuiting behaviour of monads with special emphasis in seeing it in action when using LINQ query expression syntax:

```
static void Main(string[] args)
{
    var aBoxOfNumbers1 = new Box<int[]>( new int[] {1,2,3,4,5 });
    var aBoxOfNumbers2 = new Box<int[]>(); // empty box
    var aBoxOfNumbers3 = new Box<int[]>( new int[] {6, 7, 8, 9, 10 });

    // The way Box's validation step works in its Bind/Map functions says thet an empty box is
    // invalid. Furthermore we cannot process any boxes if even one box is empty:

    var result = from number1 in aBoxOfNumbers1
            from number2 in aBoxOfNumbers2 // this result causes the next Bind() to check see that
                                           // its an invalid input and itself returns empty box and
                                           // this repeats until the result is deemed empty
            from number3 in aBoxOfNumbers3 // this does run, but it just bails out at the Validation
                                           // phase in the bind()'s VETL stage
            select number3;


    Console.WriteLine($"The result is: {result}");

}
```

```
static void Main(string[] args)
{
    var aBoxOfNumbers1 = new Box<int[]>( new int[] {1,2,3,4,5 });
    var aBoxOfNumbers2 = new Box<int[]>(); // empty box
    var aBoxOfNumbers3 = new Box<int[]>( new int[] {6, 7, 8, 9, 10 });

    // The way Box's validation step works in its Bind/Map functions says thet an empty box is
    // invalid. Furthermore we cannot process any boxes if even one box is empty:

    var result = from number1 in aBoxOfNumbers1
            from number2 in aBoxOfNumbers2 // this result causes the next Bind() to check see that
                                           // its an invalid input and itself returns empty box and
                                           // this repeats until the result is deemed empty
            from number3 in aBoxOfNumbers3 // this does run, but it just bails out at the Validation
                                           // phase in the bind()'s VETL stage
            select number3;


    Console.WriteLine($"The result is: {result}");

}
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial11_short_circuiting/Program.cs

An empty box, i.e aBoxOfNumbers2 will cause the entire pipeline to return an Empty Box<T> instead of the ultimate result of transforming all the boxes contents. This is by design and later when this is not

desired, you can use LanguageExt's *match*() function to determine how to deal with invalid data and how the pipeline should proceed. For now we'll leave this ideas until later, when we cover *match*.

A diversion: composition of functions (12)

Now we turn our attention to function composition, see: https://github.com/louthy/language-ext/wiki/Thinking-Functionally:-Function-composition for more details.

Scenario:

You have a program that has existing functions. You change one of those functions to now return a Monad. You need to ensure that you program still works i.e. existing functions can use your Monad returning function, even though it does not expect a Monad!

So, we'll compose a new function that will take the monad, and adapt it to the interface of the original function.

```
static void Main(string[] args)
{
    var encrypted = EncryptWord("Stuart");
    Console.WriteLine($"The encrypted word is '{encrypted}'");
}

public static string EncryptWord(string word)
{
    // First, reverse the characters
    var reversedBox = ReverseCharacters(word);

    // But SwapFirstAndLastChar() doesn't expect our newly monadic version of ReverseCharacters!
    // It still expects a string, not a monad.
    // So, the following now will not work...
    //var swapped = SwapFirstAndLastChar(reversed);
    // because reversed now returns a monad and SwapFirstAndLastChar expects it to be a string.

    // So to ensure that we can still use the existing Code ie SwapFirstAndLastChar, we'll need
    // to 'compose' an adapter function, which will basically convert the now incompatible input
    // from ReverseCharacters ie a Box<> monad into a string which it can use as was expected in
    // the commented out call above.

    var swapAdapter = Compose<Box<string>, string, string>(inputWord => SwapFirstAndLastChar(inputWord), converter-
Function: box => box.Extract as string);

    // So now We can 'replace' calls to SwapFirstAndLastChar() with calls to its 'replaced' adapter
    // function, which is composed of the original function (so you dont really replace it,
    // its still there)
    // Pass in the now monadic result of the modified ReverseCharacters function to the adapter
    // we just composed and it will call our original SwapFirstAndLastChar() for use
    // after it turns the moandic input into the orignal form that SwapFirstAndLastChar expected.

    // Takes in the box and extracts the string out of it so it can be passed to the original
    // function that needs a string
    string swapped = swapAdapter(reversedBox);
    return swapped;
}

/// <summary>
/// Makes a new function ie composes one, from two other functions
/// </summary>
/// <typeparam name="TA">Input type of the first functions input</typeparam>
/// <typeparam name="TB">The Transformed output type of the first function</typeparam>
/// <typeparam name="TC">The transformed output type of the result of the second function
///  using the first's output</typeparam>
/// <param name="converterFunction">the function that will run on the output of the first
/// function</param>
/// <param name="originalFunction">the first function</param>
/// <returns>A new function ie a delegate</returns>
public static Func<TA, TC> Compose<TA, TB, TC>(Func<TB, TC> originalFunction, Func<TA, TB> converterFunction)
    => input => originalFunction(converterFunction(input));

/// <summary>
/// This function used to return a string, but we've decided that it should return a Monad.
/// Now in order to use this function in places that previously expected it to be a string,
/// we'll need to create an adapter for it. And we'll do this be composing a new function which takes
/// this function's output, now a monad, and returns it as a string...see above.
/// </summary>
/// <param name="word"></param>
/// <remarks>But how do/will the existing functions that expect the previous un-monadic form
/// of the function? We'll create an adapter by composing one</remarks>
/// <returns></returns>
public static Box<string> ReverseCharacters(string word)
{
    var charArray = word.ToCharArray();
    Array.Reverse(charArray);
    return new Box<string>(new string(charArray));
}

/// <summary>
/// This is the existing function that we don't want to change and it requires a string, still
/// </summary>
/// <param name="word"></param>
/// <returns></returns>
public static string SwapFirstAndLastChar(string word)
{
    var words = word.ToCharArray();
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial12_function_compositionial12/Program.cs

A diversion: Pure Functions (13)

A pure function's return value is a product of its arguments i.e. only its arguments are used to determine the return value.

The expectation is that if it does this, then the same input to the function will yield the same output too.

To ensure this, you need to further restrict the function to not use/depend on anything that might jeopardise this, i.e. fetch/use a source that today might be one thing and tomorrow might be something else.

For instance, if you get a value from the DB today, tomorrow it might be removed and then the guarantee you made that the function will return the same value breaks.

So, you can't use Input/output or anything that is a source of changeable circumstances.

Side note: Pure functions are immediately parallelizable and can be used for Memoization (storing the result and input of the function requires that the function never needs to run again)

```
static void Main(string[] args)
{

   Box<int[]> aBoxOfNumbers = new Box<int[]>(new int[]{ 1, 2, 3, 4, 5, 6, 7});

   var result = aBoxOfNumbers.Map(numbers => GenerateFibonacciSeriesFrom(numbers));
   var result2 = aBoxOfNumbers.Map(ImpureGenerateFibonacciSeries);

    // fibos generated from pure function - will ALWAYS return same result
    Console.WriteLine($"fibos are {string.Join(',', result.Extract)}");

    // fibos generated from impure function - wont always return the same result
    Console.WriteLine($"{string.Join(',', result2.Extract)} is not to be this always.");

    // Pure function only uses its input to generate its result and doesn't depend/get/fetch data
    // from anywhere else that might impact the result.
    // It certainly does not depend on things like I/O and db calls that might not bring the same
    // result on subsequent calls of the function and it does not throw exceptions(later tutorial
    // show how to remove exceptions from your code using Either<IFailure, Result>).
    int[] GenerateFibonacciSeriesFrom(int[] numbers)
   {
       var fibs = new List<int>(numbers.Length);
       for (var i = 0; i < numbers.Length + 1; i++)
       {
           if (i > 1)
               fibs.Insert(i, i - 1 + i);
           else
               fibs.Insert(i, i);
       }

       return fibs.ToArray();
   }

    int[] ImpureGenerateFibonacciSeries(int[] numbers)
    {
        var fibs = new List<int>(numbers.Length);
        for (var i = 0; i < numbers.Length + 1; i++)
        {
            fibs.Insert(i, i > 1 ? AddFn(i - 1, i) : i);
        }

        // Impure add function because it is not guarantee to always do a number1+number2 addition
        // consistently, as it depends on the day of the week which can change the result
        // even though the input numbers are the same on calls to it.
        int AddFn(int number1, int number2)
        {
            if(DateTime.Today.DayOfWeek == DayOfWeek.Thursday) // Dependency on this condition
                                                               // breaks guarantee that the same
                                                               // input provided in number1 and
                                                               // number2 will ALWAYS yield the
                                                               // same result.
                return number1 + number2;
            else
                return (number1 + number2 + 1);
        }

        // Eg. Input is 1,2,3
        // On Monday:    0,1,4,6,8,110,12,14
        // On Tuesday:   0,1,4,6,8,110,12,14
        // On Wednesday: 0,1,4,6,8,110,12,14
        // On Thursday:  0,1,3,5,7,9,11,13   <--- Breaks that promise that for input 1,2,3
        // On Friday:    0,1,4,6,8,110,12,14
        // On Saturday:  0,1,4,6,8,110,12,14
        // On Sunday:    0,1,4,6,8,110,12,14

        return fibs.ToArray();
```

https://github.com/stumathews/UnderstandingLanguageExt/blob/master/Tutorial13_pure_functions/Program.cs

The last three sections don't fit well with the narrative of the proceeding sections. Should better integrate them with relatable use cases.There might be some better examples in Mazer.

Part II: Language-Ext

The proceeding section will have suitably prepared you for your fist encounter with LanguageExt monads, how they work and what you can come to expect from them. Essentially, they are not very different to the Box<T> monad.

We'll cover Either, Option and Try Monads and everything you've learnt thus far about Box<T> applies to them also, including calling there Map, Bind and using them in LINQ expressions in a declarative way.

The Either<L,R> Monad

Introducing the Either monad (14)

This tutorial shows you what a what an Either<> type is and how to use it generally

```
static void Main(string[] args)
{
    string word = "five";

    // Note: We're creating an either with the left type as int and the right type as string.
    // You can assign either type to an either
    Either<int, string> amount = 5;
    amount = word;
    var emptyEither = new Either<int, string>();

    // Note you can do the same transformations you did on a Box<T> on an Either because it too
    // is a Monad and it too has a Bind(), Map(), Select() and SelectMany() extension method.
    // This transformation occurs on the right hand value, provided that the either contains the
    // right hand value and not the left (this is the inbuilt validation or short-circuiting Monad
    // idea in action)

    var resultA = amount.Bind(str => TransformRight(str)); // Extract EitherData from either once
                                                           // validated ie that its not a right
                                                           // value, run transform function.

    amount = 25;

    // notice that it's biased to its right value and only cares about running a
    // transform on a right type. The validation within Either's Bind() will check for a Right
    // Type and then run the provided transform, if its left it will return what it has (25) but
    // no transform will occur
    var resultB = amount.Bind(integer => TransformRight(integer)); // Wont run transformation
                                                                   // because the validation
                                                                   // will fail because its
                                                                   // a left type.

    Console.WriteLine($"The value of resulta is '{resultA}' and the result of result b is '{re-
sultB}' and an empty either looks like this '{emptyEither}' ");

    // notice how Either's bind function will extract the right part, if validation succeeds
    //  and runs the transformation as expected.
    Either<int, string> TransformRight(string extractedRight)
    {
        Either<int, string> result = extractedRight.ToUpper(); // Like all Bind functions we need
                                                               // to lift the result into the
                                                               // monad ie Either type
        return result;
    }

}
```

Operations on Either<L,R>

Using BiBind (15)

This tutorial shows you what a what an Either<>'s BiBind() functionality

```
static void Main(string[] args)
{

    int number = 5;
    string word = "five";

    Either<int, string> amount = 5;
    amount = word;

    // Instead of only being able to run a transform on the right hand side only as
    // bind() does, you can use BiBind() to prepare transform functions for whatever
    // side, left or right type is assigned to it!
    // The result is the result of whichever bind run, depending which state the
    // either is in (left=has integer)(right=has string)
    var resultOfTransform =
               amount.BiBind(rightString => TransformExtractedRight(rightString),
                             leftInteger => TransformExtractedLeft(leftInteger));

    Console.WriteLine($"the result of the transform is {resultOfTransform}");

    amount = number;

    resultOfTransform = amount.BiBind(rightString => TransformExtractedRight(rightString),
                                      leftInteger => TransformExtractedLeft(leftInteger));
    Console.WriteLine($"the result of the transform is {resultOfTransform}");


    Either<int, string> TransformExtractedRight(string rightString)
    {
        Either<int, string> ret = rightString + " is a word, i think";
        return ret;
    }

    Either<int, string> TransformExtractedLeft(int leftInteger)
    {
        Either<int, string> ret = leftInteger + 1;
        return ret;
    }
}
```

Using BiExists

This tutorial shows you what a what an Either<>'s BiExists()

```
static void Main(string[] args)
{

    int number = 5;
    string word = "five";

    Either<int, string> amount = 5;
    amount = word;

    // like BiBind() allows you to provide both transform functions and the correct transform will
    // run depending on is it is a right or left type contained within it - you can do something
    // similar here.
    // BiExists allows you to test/use/inspect the content and return true/false based on it.
    // BiExists can be viewed as a the 'existance' of a provided validation check being successful,
    // specific to each type left or right
    bool isEitherGreaterThanNothing = amount.BiExists(stringRight => stringRight.Length > 0 ? true: false,
                                                      integerleft => integerleft > 0 ? true : false);

    Console.WriteLine($"The result value is {isEitherGreaterThanNothing}");
}
```

Using Fold() to change an initial state over time based on the contents of the Either

This tutorial shows you what fold() does

```
static void Main(string[] args)
{
    Either<int, string> intOrString = "start";

    // A state (InitialResult) changes over time and it changes using results of the previous
    // change.... It uses an new item extracted from the array in changing the state each time.
    // The state changes the number of elements in the either - there will only be one - Left or
    // right type.
    // the state will change once based on the one value in the either.
    // For a List which has multiple items in it, the state will change that many times
    var result = intOrString.Fold("InitialState", (previousResult, extract) => changeState(extract, previousResult));

    // The result is the last state change
    Console.WriteLine($"The result value is {result}");

    string changeState(EitherData<int, string> extracted, string previousResult)
    {
        var content = extracted.State == EitherStatus.IsLeft ?  $"{extracted.Left}" : $"{extracted.Right}" ;
        var newResult = $"{previousResult} and {content}";
        return newResult;
    }
}
```

Using Iter

Iter: run an arbitary function on the Either<> if its value is right type or chose BiIter() to specify a function to run on both types

```
static void Main(string[] args)
{
    Either<int, string> intOrString = 45; // put it in left state by default (integer)

    bool didFunctionRunForRight = false;
    bool didFunctionRunForLeft = false;

    // extracts the right content and if it is right, it run this non-transforming ie void returning
    // function on it. The function wont  run it its as left type
    intOrString.Iter(rightString => RunAFunctionOnRightContents(rightString)); // So only runs the
                                                                                // function if its
                                                                                // right type

    // both actions are void returning (Unit represents a typed void result)
    Unit ret = intOrString.BiIter(rightString => RunAFunctionOnRightContents(rightString),
                                  leftInteger => didFunctionRunForLeft = true);

    Console.WriteLine($"Did function run on right contents? {didFunctionRunForRight}");
    Console.WriteLine($"Did function run on left contents? {didFunctionRunForLeft}");

    void RunAFunctionOnRightContents(string str)
    {
        Console.WriteLine($"Hello {str}");
        didFunctionRunForRight = true;

    }
}
```

Using BiMap (19)

Using BiMap() to make provision for a transform function for both the left and right types of the either. The transform is automatically lifted.

```
 static void Main(string[] args)
 {
     Either<int, string> intOrString = 45;

     // This will extract the item and allow you to run a transformation like a Map, which
     // automatically lifts the result of the transform. it will allow you to specify this
     // transform function for both types that the either can hold and the correct one will run
     // depending on the actual type that the either holds at that moment in time.
     var result = intOrString.BiMap(rightString => rightString.ToUpper(),
                                    leftInteger => leftInteger + 1024);

     Console.WriteLine($"The result is {result}");

     intOrString = "Stuart";
     result = intOrString.BiMap(rightString => rightString.ToUpper(),
                                leftInteger => leftInteger + 1024);

     Console.WriteLine($"The result is {result}");
 }
```

Using BindLeft (20)

Shows the basics of Either<L,R>, using BindLeft() to make provision for a transform function for the left types of the either (which is unusual for the default Bind() function).

The transform is NOT automatically lifted(this is a bind() after all).

```
static void Main(string[] args)
{
    Either<int, string> intOrString = "Stuart";

    // Transform the left hand side, remember bind will not automatically life the result of that
    // transformation, so you transformation function will need to do that
    // We call BindLeft because by default Either is right biased so default Bind() only transforms
    // the right side if there is one(there there isn't as we've assigned a right value of string
    // 'Stuart'
    var result = intOrString.BindLeft(left => TransformLeft(left));

    // Note the transformation did not occur because either contained a right type ie string
    Console.WriteLine($"result is {result}");

    intOrString = 55;
    // Note the transformation should now occur because either contained a left type and we've
    // defined a transformation for that on this either
    result = intOrString.BindLeft(left => TransformLeft(left));

    Console.WriteLine($"result is {result}");

    Either<int, string> TransformLeft(int left)
    {
        Either<int, string> transformedResult = left + 22;
        return transformedResult;
    }
}
```

Using Match (21)

Using Match to extract the contents of an Either<> but not put it back into and either types (as map() and Bind() would do)

```
static void Main(string[] args)
{
    Either<int, string> intOrString = "Stuart";

    // Match will run a function for which ever type is contained within the either.
    // Transform functions for both types of content are specified.  The function will run depending
    // on the underlying type in the either. The return result of each function must be the same
    // type (both string or both int) such that you can assign the result of the Match to a
    // types variable. Only one function will run, as only one of the two types can be in the either
    // at any one moment in time.
    string result = intOrString.Match(rightString => $"Right value is {rightString}",
                                       leftInteger => $"left value is {leftInteger}");

    Console.WriteLine($"Result is {result}");

    intOrString = 32;
    result = intOrString.Match(rightString => $"Right value is {rightString}",
                               leftInteger => $"left value is {leftInteger}");

    Console.WriteLine($"Result is {result}");
}
```

Operating on Lists of Either<left, Right>

Using BiMapT and MapT

This tutorial shows you how you can transform a List of Eithers, effectively doing a Map on each either in the list, and this Bi variety allows you to specify how make provision to map/transform both types

When a Bind/Map function is called on a list of monads, it is BindT or BiMapT, otherwise operating on a single monad, use Bind() or Map() alone

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "Bruce";
    Either<int, string> intOrString5 = 66;

   IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
   {
       intOrString1, intOrString2, intOrString3, intOrString4, intOrString5
   };

    // Extracts the rights only, ignores the lefts(they disapear from the result) - not lifted back into either
    // - comparable to a Match() in this way
    IEnumerable<string> rights = listOfEithers.Rights();
    IEnumerable<int> lefts = listOfEithers.Lefts();

    // Runs a map transform function(automatic lift) on every either in the lift
    // Remember a map and a Bind can change the contained type of the returned either but it must be an either
    // the T in BiMapT really inndicates that the BiMapT works on a list of monads, in this case a list of Eithers
    IEnumerable<Either<char, string>> result = listOfEithers.BiMapT(
                                rightString => $"TurnAllRightStringToThis ({rightString})",
                                leftInteger => 'A' /*Turn all left values in the eithers to 'A'*/).ToList();

    // ok so now we've transformed the Eithers in different ways depending on their type, lets look at them

    // we could look at our transformations done on each using this:
    var lefts1 = result.Lefts();
    var rights1 = result.Rights();

    // or we could get a string representation of either type of the either and print that to show us how that
    // either was transformed
    foreach (var either in result)
    {
        var stringResult = either.Match(rightString => rightString,
                                        leftInteger => $"{leftInteger}");
        Console.WriteLine(stringResult);
    }

    // Transform each either that has in in the right state, ie has a value of the right type
    // Remember to bind is to lift by yourself
    var result1 = result.MapT(rightString => TransformMe(rightString));

    // if you prefer to transform the left type for each either in the list, you can:
    var result2 = result.MapLeftT(c => char.ToUpper(c));

    // You can also transform the eithers in the list using a bind(not automatic lift)

    var result3 = result.BindT(s =>
    {
        Either<char, string> either = 'c'; // we need to lift a bind transformation function as always
        return either;
    });



    Either<char, string> TransformMe(string rightString)
     {
         return $"the value is {rightString}";

     }

     Console.WriteLine($"mapT result is:");
     foreach (var either in result1)
     {
         var str = either.Match(Right: (rightString) => $"String is '{rightString}'",
                                Left: (leftChar) => $"Char is '{leftChar}'");
         Console.WriteLine(str);
     }
}
```

Using BindT

This tutorial shows you how you can transform a List of Eithers, effectively doing a Bind on each either in the list and a Bi variety allows you to specify how make provision to map/transform both types

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "Bruce";
    Either<int, string> intOrString5 = 66;

    IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
    {
        intOrString1, intOrString2, intOrString3, intOrString4, intOrString5
    };

    // transform the right values (if they are there) for each either in the list
    // As this is a bind, you need to lift the result in to a Either
    var transformedList = listOfEithers.BindT(rightString => TransformRight(rightString));

    Either<int, string> TransformRight(string rightString)
    {
        Either<int, string> t = $"My name is '{rightString}'";
        return t;
    }

    var newRights = transformedList.Rights(); /* note we dont care care about the lefts,
    if we did we might you match to see what both left and right values would be if they are set on
    the eithers we are looking at - see Tutorial 22 */
    foreach (var str in newRights)
    {
        Console.WriteLine(str);
    }
}
```

Using IterT

This tutorial shows you how you can call a function on each right value for the list of monads in the list, using IterT() (Either is a monad)

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "Bruce";
    Either<int, string> intOrString5 = 66;

    IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
    { intOrString1, intOrString2, intOrString3, intOrString4, intOrString5 };

    // Extract right values from the eithers in the list and run this function to get them.
    // note if there is no right value for the either being inspected, this function is not run
    // ie skips 66
    listOfEithers.IterT(rightString => Console.WriteLine($"{rightString}"));
}
```

Using Apply

Using Apply both on a simple Either<> and a List of Eithers to demonstrates its simplicity

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "Bruce";
    Either<int, string> intOrString5 = 66;

    // basically, pass yourself ie your current value to the function provided. Ie apply allows you
    // to transform yourself (makes a copy of the result, not an in-place modification)
    var resultA =  intOrString5.Apply(me => UseThis(me));

    // the function can transform the type of the either
    var resultB = intOrString5.Apply(me => UseThisAndChangeType(me));

    Console.WriteLine($"ResultA = {resultA}, ResultB = {resultB}");

    IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
    { intOrString1, intOrString2, intOrString3, intOrString4, intOrString5 };

    // So in the same way, give your self ie your content (which is an List of Eithers) to the
    // provided function
    var result = listOfEithers.Apply(enumerable => UseThisListOfEithers(enumerable));

    Console.WriteLine($"The result of is '{result}'");

    Either<int, string> UseThis(Either<int, string> useThis)
    {
        var str = useThis.Match(rightString => rightString,
                                leftInteger => "was left");
        Either<int, string> t = str;
        return t;
    }

    Either<int, char> UseThisAndChangeType(Either<int, string> useThis)
    {
        var str = useThis.Match(rightString => 'T',
                                leftInteger => 'F');
        Either<int, char> t = str;
        return t;
    }

    IEnumerable<Either<int, string>> UseThisListOfEithers(IEnumerable<Either<int, string>> useMe)
    {
        // Transform the things in whatever state (type ie left or right) they are in
        return useMe.BiMapT(rightString => rightString,
                            leftInteger => leftInteger * 2);
    }
}
```

Using Partition

Using Partition to easily get both the lefts() and the Rights() in one call - as a tuple of (lefts,rights)

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "Bruce";
    Either<int, string> intOrString5 = 66;

    IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
    { intOrString1, intOrString2, intOrString3, intOrString4, intOrString5 };

    // instead of calling Lefts() and Rights(), you can get them all in one go as a tuple
    var (lefts, rights) = listOfEithers.Partition();

    foreach(var left in lefts)
        Console.WriteLine($"Left: {left}");

    foreach (var right in rights)
        Console.WriteLine($"Right: {right}");
}
```

Using Match

Shows the basics of transforming a list of Eithers using match to understand whats in them (both left and right values) and

then transform them based on their values into a single type that represents either in one way (a string)

Along with Map() and Bind() this extracts the value from the Either and provides transformation functions for both Left and Right sides of the Either

```
static void Main(string[] args)
{
    Either<int, string> intOrString1 = "Stuart";
    Either<int, string> intOrString2 = "Jenny";
    Either<int, string> intOrString3 = "Bruce";
    Either<int, string> intOrString4 = "zxcmbasdjkfkejrfg";
    Either<int, string> intOrString5 = 66;
    Either<int, string> intOrString6 = 234;

    IEnumerable<Either<int, string>> listOfEithers = new Either<int, string>[]
    { intOrString1, intOrString2, intOrString3, intOrString4, intOrString5, intOrString6, };

    // Go through each of the Eithers and depending on their types and their content, transform them
    // and we can represent values from either type, string or int as one uniform type, in this case
    // we can represent a left and a right as a string and
    // thus return the set of all these representation as a list of strings
    // Note that in order to assign the result of a match, both sides' transformation function needs
    // to return the same type, particularly the type of the receiving variable

    var result = listOfEithers.Match(Right:  MakeGenderAwareString,
                                      Left: MakeOneHundredAwareString);

    foreach (var transform in result)
    {
        Console.WriteLine(transform);
    }
}

private static string MakeOneHundredAwareString(int leftInteger)
{
    return leftInteger > 100 ? $"{leftInteger} is > 100" : $"{leftInteger} < 100";
}

private static string MakeGenderAwareString(string rightString)
{
    string[] boysNames = new[] {"Stuart", "Bruce"};
    string[] girlsNames = new[] {"Jenny"};

    if (boysNames.Contains(rightString)) return $"{rightString} is a Boys name";
    if (girlsNames.Contains(rightString)) return $"{rightString} is a Girls name";
    return $"i dont know if {rightString} not registered with me as a boys name or a girls name";
}
```

The Option<T> Monad

Introduction to Option<T> (28)

Option<T> type effectively removes the need to use NULL in your code. Nulls can produce unexpected behavior and as such have no place in pure functions where unexpected behavior would

render them otherwise impure

```
static void Main(string[] args)
{
    // An optional type can hold an integer or a none
    Option<int> optionalInteger = 34; // note we can just assign it straight away
    optionalInteger = Option<int>.Some(34); // save as above. Shown just for demonstration purposes
    optionalInteger = Option<int>.None;
    // optionalInteger = null; Options effectively eliminate nulls in your code.

    var resultA = DivideBy(25, 5);
    var resultB = DivideBy1(optionalInteger, 5);

    Console.WriteLine($"The result A is '{resultA}' and B is '{resultB}'");
}

static int DivideBy(int thisNumber, int dividedByThatNumber)
{
    return thisNumber / dividedByThatNumber;
}

static Option<int> DivideBy1(Option<int> thisNumber, int dividedByThatNumber)
{
    return thisNumber.Map( i => i / dividedByThatNumber);
}
```

Basic use-case of Option<T> (29)

```
static void Main(string[] args)
{
    int         resultA = DivideBy(25, 5);
    Option<int> resultB = DivideBy1(25, 0);

    Console.WriteLine($"The result A is '{resultA}' and B is '{resultB}'");

    /*
     * Discussion:
     * DivideBy1 will return an Option<T> and it knows what an invalid result is - its encapsulated
     *  within the option<T>
     * DivideBy will return an int, but the caller needs to know that 0 is an invalid result.
     */

    var result1 = Add5ToIt(resultA);
    var result2 = Add5ToIt(resultB);

    Console.WriteLine($"The result A is '{result1}' and B is '{result2}'");
}


/// <summary>
/// Normal function, not using optional parameters
/// </summary>
/// <param name="thisNumber"></param>
/// <param name="dividedByThatNumber"></param>
/// <returns>integer</returns>
static int DivideBy(int thisNumber, int dividedByThatNumber)
{
    if (dividedByThatNumber == 0)
        return 0;
    return thisNumber / dividedByThatNumber;
}

/// <summary>
/// Function returns Monad, one construct that represents both failure and success.
/// </summary>
/// <returns>Option of an integer</returns>
static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber)
{
    if (dividedByThatNumber == 0)
    {
        Option<int> t = Option<int>.None;
        return t;
    }

    return thisNumber / dividedByThatNumber;
}

static int Add5ToIt(int input)
{
    // Whoops, I've forgotten to check if input is valid.
    return input + 5;
    // And even if i did, I've have to know what an invalid input is - its 0 in this case.
}

static Option<int> Add5ToIt(Option<int> input)
{
   // I can assume that its valid, because Map will run a transformation function on the valid input
   Option<int> t = input.Map(validInput => validInput + 5); // if its invalid the Validation phase
                                                            // of the map() function will return a
                                                             // None ie an invalid input and so the
```

Using Option<T> in functions (passing in and returning)

Contrived example of passing around Option<T> arguments

```
static void Main(string[] args)
{

    var resultB = DivideBy1(225, 5);
    var result2 = Add5ToIt(resultB);

    // Now we can continue with the knowledge that result2 as 5 added to it or not
    // (and everyone else will do that too:)

    var result = PerformPensionCalculations(result2);

    if (result.IsNone)
        theBadMessage();

    if (result.IsSome)
        theGoodMessage();

    // of we could use a BiIter without any conditionals above (if statements)

    result.BiIter(i => theGoodMessage(),
                  ()=> theBadMessage());

    void theBadMessage()
     => Console.WriteLine($"Could not determine your pension, because invalid input was used");

    void theGoodMessage()
    {
        var pension = result.Match(Some: i => i, None: 0);
        Console.WriteLine($"Your pension is '{pension}'");
    }

}

/// <summary>
/// Example of passing in a option monad
/// </summary>
static Option<int> PerformPensionCalculations(Option<int> input)
{
    // Extract, Validate and transform it using Map (reember is a Monad)
    return input.Map(validInput => CalculateYourPension(validInput));
    // we can call a normal function (non monad returning or accepting) inside a Map or Bind
}


static int CalculateYourPension(int input)
{
    return (input * 3) / 26;
}

static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber)
{
    if (dividedByThatNumber == 0)
    {
        Option<int> t = Option<int>.None;
        return t;
    }

    return thisNumber / dividedByThatNumber;
}

static Option<int> Add5ToIt(Option<int> input)
{
 // View a Bind/Map as 'Try to add 5' if the input is valid ie not a None
   return input.Bind(validInput =>
   {
       Option<int> option = validInput + 5;
```

using IfSome() and IfNone()

Demonstrates the usage of IfNone and IfSome which runs a user defined function provided the option is None or Some respectively

```
static void Main(string[] args)
{

    var resultB = DivideBy1(225, 5);

    var result2 = Add5ToIt(resultB);

    // Now we can continue with the knowledge that result2 as 5 added to it or not
    // (and everyone else can do that too...)

    var result = PerformPensionCalculations(result2);

    Unit noValue = result.IfSome(validInput => Console.WriteLine("Valid input is {validInput}"));
    int defaultPensionIfInvalidInput = result.IfNone(() => -1); // IfNone turns your result into a
                                                                // valid value, effectively a Some()
                                                                // but its actual value, int
}

static Option<int> PerformPensionCalculations(Option<int> input)
{
    // Extract, Validate and transform it using Map
    return input.Map(validInput => CalculateYourPension(validInput));
    // We can call a normal function (non monad returning or accepting) inside a Map or Bind
}


static int CalculateYourPension(int input)
{
    return (input * 3) / 26;
}

static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber)
{
    if (dividedByThatNumber == 0)
    {
        Option<int> t = Option<int>.None;
        return t;
    }

    return thisNumber / dividedByThatNumber;
}

static Option<int> Add5ToIt(Option<int> input)
{
    // View a Bind/Map as 'Try to add 5' if the input is valid ie not a None
    return input.Bind(validInput =>
    {
        Option<int> option = validInput + 5;
        return option;
    });
}
```

Pipelining with Options<T> (32)

Rosetta code! Procedural -> Fluent -> Query Syntax

```
static void Main(string[] args)
{
    int startingAmount = 225;

    // Procedural way
    var step1 = DivideBy(startingAmount, 5);
    var step2 = Add5ToIt(step1);
    var result = PerformPensionCalculations(step2);

    // Fluent way
    Option<int> result1 = DivideBy1(startingAmount, 5)
                              .Bind(input => Add5ToIt1(input))
                              .Bind(input => PerformPensionCalculations1(input));

    // Expression way
    Option<int> result2 = from input1 in DivideBy1(startingAmount, 5)
                          from input2 in Add5ToIt1(input1)
                          from input3 in PerformPensionCalculations1(input2)
                            select input3;

    // Oh Wow, we've really simplified the code, its smaller, all the steps are functions
    // and we are happy. Passing and sending Monads have helped us create step-by-step
    // logical means to execute code entirely from functions!
}

static Option<int> PerformPensionCalculations1(Option<int> input)
    => input.Map(CalculateYourPension);

static int PerformPensionCalculations(int input)
    => CalculateYourPension(input);

static int CalculateYourPension(int input)
    => (input * 3) / 26;

static int DivideBy(int thisNumber, int dividedByThatNumber)
    => dividedByThatNumber == 0 ? 0 : thisNumber / dividedByThatNumber;

static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber) =>
    dividedByThatNumber != 0
        ? thisNumber / dividedByThatNumber
        : Option<int>.None;

static Option<int> Add5ToIt1(Option<int> input)
    => input.Map(validInput => validInput + 5);

static int Add5ToIt(int input)
    => input + 5;
```

Using ToEither<>

ToEither extension method to convert a value to a right sided Either<L,R>

```
static void Main(string[] args)
{
    //Procedural way
    int startingAmount = 225;
    var step1 = DivideBy(startingAmount, 5);
    var step2 = Add5ToIt(step1);
    var result = PerformPensionCalculations(step2);


    // Fluent way
    Option<int> result1 = DivideBy1(startingAmount, 5)
                        .Bind(input => Add5ToIt1(input))
                        .Bind(input => PerformPensionCalculations1(input));

    // The some value of the Option<T> ie the integer will be the right value of the either,
    // and then you need to choose a left value:
    Either<int, int> either = result1.ToEither(() => 23);

}

static Option<int> PerformPensionCalculations1(Option<int> input)
    => input.Map(CalculateYourPension);

static int PerformPensionCalculations(int input)
    => CalculateYourPension(input);


static int CalculateYourPension(int input)
    => (input * 3) / 26;

static int DivideBy(int thisNumber, int dividedByThatNumber)
    => dividedByThatNumber == 0 ? 0 : thisNumber / dividedByThatNumber;

static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber) =>
    dividedByThatNumber != 0
        ? thisNumber / dividedByThatNumber
        : Option<int>.None;

static Option<int> Add5ToIt1(Option<int> input)
    => input.Map(validInput => validInput + 5);

static int Add5ToIt(int input)
    => input + 5;
```

Using BiMap() (34)

Bimap

```
static void Main(string[] args)
{
    //Procedural way

    var step1 = DivideBy(225, 5);
    var step2 = Add5ToIt(step1);
    var result = PerformPensionCalculations(step2);


    // Fluent way
    Option<int> result1 = DivideBy1(225, 5)
                        .Bind(input => Add5ToIt1(input))
                        .Bind(input => PerformPensionCalculations1(input));

    // Transform both the invalid and valid versions of the options to a single string
    Option<string> results = result1.BiMap(Some: validInteger => "Valid",
                                           None: () => "Invalid");

    Console.WriteLine($"The result is '{results}'");

}

static Option<int> PerformPensionCalculations1(Option<int> input)
    => input.Map(CalculateYourPension);

static int PerformPensionCalculations(int input)
    => CalculateYourPension(input);


static int CalculateYourPension(int input)
    => (input * 3) / 26;

static int DivideBy(int thisNumber, int dividedByThatNumber)
    => dividedByThatNumber == 0 ? 0 : thisNumber / dividedByThatNumber;

static Option<int> DivideBy1(int thisNumber, int dividedByThatNumber) =>
    dividedByThatNumber != 0
        ? thisNumber / dividedByThatNumber
        : Option<int>.None;

static Option<int> Add5ToIt1(Option<int> input)
    => input.Map(validInput => validInput + 5);

static int Add5ToIt(int input)
    => input + 5;
```

The Try<T> Monad

Supressing Exceptions

This tutorial demonstrates the use of the Try<> Monad.

```
static void Main(string[] args)
{
    // This is an function that could throw an exception. We dont want that because that would jump
    // straight out of our function
    // and cause our function not to complete fully. We want robust, dependable functions that
    // always return no matter what.
    int SomeExternalCode(int numerator, int denominator)
    {
        if(denominator == 0)
            throw new DivideByZeroException();
        return numerator / denominator;
    }

    // What we can do now is wrap this unsafe function, into a Try<> which will capture any
    // exceptions that are thrown (they dont leave the function)
    Try<int> try1 = new Try<int>(() =>
    {
        var result = SomeExternalCode(25, 5);
        return result;
    });

    Try<int> try2 = new Try<int>(() =>
    {
        var result = SomeExternalCode(25, 0);
        return result;
    });

    // Now we have encapsulated any exceptions into the Try<> type, we can now check if it had
    // any failures otherwise use the result
    var result1 = try1.Match(
        unit => unit.ToEither<IAmFailure, int>(), // was ok, no exceptions thrown
        exception => new ExternalLibraryFailure(exception)); // We got an exception, now how should
                                                             // we deal with it - lets turn it into
                                                             // a IAmFailure type

    var result2 = try2.Match(
        unit => unit.ToEither<IAmFailure, int>(), // was ok, no exceptions thrown
        exception => new ExternalLibraryFailure(exception)); // We got an exception, now how should
                                                             // we deal with it - lets turn it into
                                                             // a IAmFailure type

    // Print the results
    Console.WriteLine($"Result1 was: {result1} and Result2 was {result2}");

    // And if you required all/both to succeed you could use short-circuiting behavior in a pipeline
    var overallResult =
        from res1 in result1
        from res2 in result2
        select res1 / res2;

    Console.WriteLine($"The combined result is {overallResult}");
}
```

Part III: Everything else

Bonus

ThrowIfFailed() and introducing Either<IAmFailure, Option<T>>

ThrowIfFailed and a way to make functions return a standard return type of an Either of Right(T) or a failure(Left).

T can be any type your function deals with, as you'd use in any normal function you create, only you make it an Option.

You also bundle with your return type a failure if there is one, by returning an all encompassing return value of Either<IAmFailure, Option<T>>

Program.cs

```
static void Main(string[] args)
{
   // Note that a common Either of form Either<IAmFailure, Option<T>> is used here to
   // 1) Communicate if there anything or any reason while processing the Option<T> that is
   // erroneous - that will then fail-fast and return a IAmFailure ie Either in Left state
   // 2) If there wasn't while inspecting the Option<T> return that or a transformation of that
   // ie an Option<T>
    var result1 = DivideBy(225, 5)
                        .Bind(input => Add5ToIt(input).ThrowIfFailed()) // throw if failed either,
                                                                        // otherwise extract and
                                                                        // return the right value
                                // Inspect for either for a failure and throw if it is,
                                // otherwise return the value as-is
                    .Bind(input => PerformPensionCalculations(input).ThrowIfFailed());


    Console.WriteLine($"The result is '{result1}'");
}

static Either<IAmFailure, Option<int>> PerformPensionCalculations(Option<int> input)
{
    /* Remember, We can return a IAmFailure or a Option<int> */

    // Generate a failure while looking at the option, by extracting the value and determining
    //  if its valid or not
    var isGreaterThan100 = input.Match(Some: number => number > 100 ? true : false, None: false);
    if (!isGreaterThan100)
        return new GenericFailure("Must be greater than 100");

    // Do some validation and fail fast, otherwise return the value
    return input.Map(CalculateYourPension); //return Option<T>
}

static int CalculateYourPension(int input)
    => (input * 3) / 26;

static Option<int> DivideBy(int thisNumber, int dividedByThatNumber) =>
    dividedByThatNumber != 0
        ? thisNumber / dividedByThatNumber
        : Option<int>.None;

static Either<IAmFailure, Option<int>> Add5ToIt(Option<int> input)
{
    // arbitary test
    var isValid = input.Match(Some: number => number > 12, None: false);

    // We can return a IAmFailure or a Option<int>
    if(!isValid)
        return new GenericFailure("must be greater than 12");
    return input.Map(validInput => validInput + 5); //Return Option<T>
}
```

CustomExtensions:

```
// Note a 'Successful' either is in the Right state while a 'unsucessful' either is in Left state
public static class EitherExtensions
{
    /// <summary>
    /// Throws an exception if the either is in left state ie failed
    /// </summary>
    public static TRight ThrowIfFailed<TLeft, TRight>(this Either<TLeft, TRight> either) where TLeft : IAmFailure
        => either.IfLeft(failure => throw new UnexpectedFailureException(failure));

    //Given Either in Failure state (IAmFailure) to None, if the check on the failure fails - ie checking if
    // IAmFailure is suitable
    public static Either<IAmFailure, Option<T>>
      FailureToNone<T>(this Either<IAmFailure, Option<T>> item, Func<IAmFailure, bool> predicate)
        => item.Match(rightValue => rightValue,
                      failure => !predicate(failure)
                                 ? Prelude.Left<IAmFailure, Option<T>>(failure)
                                 : Option<T>.None, null);

    // Given Either with an optional Right state. If right state is none, make a failure using provided failure,
    // otherwise use the valid value as the valid/right value of the Either
    public static Either<IAmFailure, T> NoneToFailure<T>(this Either<IAmFailure, Option<T>> item,
                                                         Func<IAmFailure> failure)
        => item.Bind(e => e.Match(value => value.ToSuccess<T>(),
                                  () => Prelude.Left<IAmFailure, T>(failure())));

    /// <summary>
    /// Given An Either with an optional Right value. If option value is valid/some, make a Either Failure using
    ///  provided failure, otherwise a Successful Either
    /// </summary>
    public static Either<IAmFailure, Option<T>>
     SomeToFailure<T>(this Either<IAmFailure, Option<T>> item, Func<IAmFailure> failure)
        => item.Bind(either => either.Match(_ => Prelude.Left<IAmFailure, Option<T>>(failure()),
                                            () => either.ToSuccess()));

    public static Func<IEnumerable<Option<T>>, Either<IAmFailure, IEnumerable<T>>>
    NoneToFailure<T>(Func<IAmFailure> failure)
        => items => items.Sequence<T>().Match(item => item.ToSuccess<IEnumerable<T>>(),
                                              () => Prelude.Left<IAmFailure, IEnumerable<T>>(failure()));

    /// <summary>
    /// Make a Either<IAmFailure, T> in left(failure) state if the list is empty (use failure producing function
    /// provided), otherwise return the either in right state using the first item in the list as the light value in
    /// the either
    /// </summary>
    public static Either<IAmFailure, T> HeadOrFailure<T>(this IEnumerable<T> list, Func<IAmFailure> failure)
        => list.Match(() => Prelude.Left<IAmFailure, T>(failure()), s => Prelude.Right<IAmFailure, T>(s.First()));

    /// <summary>
    /// Make Either<IAmFailure, T> in right state
    /// </summary>
    public static Either<IAmFailure, T> ToSuccess<T>(this T value)
        => Prelude.Right<IAmFailure, T>(value);

    // Make an IAmFailure to a Either<IAmFailure, T> ie convert a failure to a either that represents that failure
    // ie an either in the left state
    public static Either<IAmFailure, T> ToEither<T>(this IAmFailure failure)
        => Prelude.Left<IAmFailure, T>(failure);

}

public class UnexpectedFailureException : Exception
{
    public UnexpectedFailureException(IAmFailure failure)
    {

    }
}
```

IamFailure:

```
public interface IAmFailure
{
    string Reason { get; set; }
}

public class GenericFailure : IAmFailure
{
    public GenericFailure(string reason)
    {
        Reason = reason;
    }

    public string Reason { get; set; }
}
```

Using custom extension method FailureToNone()

We can convert a 'failed' standard wrapped function to a None. This is helpful if you want to turn a failure into a 'valid' standard function either but with None Right value

```
static void Main(string[] args)
{
    var result1 =
        from divideResult in DivideBy1(225, 66)
        from addResult in Add5ToIt1(divideResult).FailureToNone(failure => true)
        from calcResult in PerformPensionCalculations1(addResult).FailureToNone(failure => true)
        select calcResult;

    Console.WriteLine($"The result is '{result1.ThrowIfFailed()}'"); // this wont throw even though
                                                                     // we failed, becasue we
                                                                     // converted failures to None
                                                                     // above!
}

static Either<IAmFailure, Option<int>> PerformPensionCalculations1(Option<int> input)
{
    var isGreaterThan100 = input.Match(Some: number => number > 1,
                                       None: false); //match is like flattening an either in
                                                     // one common result type, in this case: bool
    return !isGreaterThan100
        ? (Either<IAmFailure, Option<int>>) new GenericFailure("Must be greater than 100")
        : input.Map(CalculateYourPension);
}

static int PerformPensionCalculations(int input)
    => CalculateYourPension(input);

static int CalculateYourPension(int input)
    => (input * 3) / 26;

static int DivideBy(int thisNumber, int dividedByThatNumber)
    => dividedByThatNumber == 0 ? 0 : thisNumber / dividedByThatNumber;

static Either<IAmFailure, Option<int>> DivideBy1(int thisNumber, int dividedByThatNumber) =>
    dividedByThatNumber == 0
        ? new GenericFailure("Can't divide by 0")
        : Option<int>.Some(thisNumber / dividedByThatNumber).ToSuccess();

static Either<IAmFailure, Option<int>> Add5ToIt1(Option<int> input)
{
    var isValid = input.Match(Some: number => number > 12, None: false);
    return !isValid
        ? (Either<IAmFailure, Option<int>>) new GenericFailure("must be greater than 12")
        : input.Map(validInput => validInput + 5);
}

static int Add5ToIt(int input)
    => input + 5;
```

Memoization

This tutorial exposes how functional programming, particularly caching results from pure functions, aids memoization, as they always return the same output for same input.

```
static void Main(string[] args)
{
    // This is our input strings
    var phrase1 = "1. When you need answers for programming";
    var phrase2 = "2. with C# 5.0, this proactical and tightly";
    var phrase3 = "3. Focused book tells you exactly what you need";
    var phrase4 = "4. to know-without long introductions or bloated samples.";
    var phrase5 = "5. Easy to browse, it's ideal as a quick";
    var phrase6 = "6. reference or a guide to get you";
    var phrase7 = "7. rapidly up to speed if you already know Java, C++,";
    var phrase8 = "8. or an earlier version of C#";
    var phrase9 = "9. Dynamic Binding and C# 5.0's new";
    var phrase10 = "10.asynchonous functions";

    var phrases = new string[] { phrase1, phrase2, phrase3, phrase4, phrase5, phrase6, phrase7,
                                 phrase8, phrase9, phrase10 };

    var encryptedResultsCache = new Dictionary<string, Option<string>>();
    var decryptedResultsCache = new Dictionary<string, Option<string>>();

    // Encrypt all the input phrases, and decrypt them
    for(int i = 1; i < phrases.Length-1; i++)
    {
        var phrase = phrases[i];

        var maybeEncrypted = SimpleEncrypt(phrase);

        // This transformation only occurs if the encrypted result is a Some(encryptedResult) and not a None
        // i.e invalid result
        SimpleEncrypt(phrase).Iter(encryptedString =>
        {
            encryptedResultsCache.TryAdd(phrase, maybeEncrypted);
            decryptedResultsCache.TryAdd(encryptedString, SimpleDecrypt(encryptedString));
        });
    }


    var randomCacheEntry = encryptedResultsCache.ElementAt(new Random().Next(encryptedResultsCache.Count));
    var getDecrypted = from encrypted in randomCacheEntry.Value
                       from decrypted in decryptedResultsCache[encrypted] // No need to call SimpleDecrypt(encrypted)
                       select decrypted;
    Console.WriteLine($"The encrypted form for '{randomCacheEntry.Key}' is '{getDecrypted}' is");
}

// Pure function, nothing that will jepordaise the consistency of same returned value given the same input.
static Option<string> SimpleEncrypt(string phrase)
{
    var result = new Option<string>();

    if(string.IsNullOrWhiteSpace(phrase))
        return Option<string>.None;

    char[] characters = phrase.ToArray();
    for(int i = 0;i<characters.Length;i++)
    {
        if(characters[i] == 'r')
            return Option<string>.None;
        characters[i]++;
    }

    result = new string(characters);
    return result;
}

static Option<string> SimpleDecrypt(Option<string> phrase)
{
    return phrase.Bind( phr =>
    {
        var result = new Option<string>();
        if(string.IsNullOrWhiteSpace(phr))
        return Option<string>.None;

        char[] characters = phr.ToArray();
        for(int i = 0;i<characters.Length;i++)
        {
            characters[i]--;
        }
```

Now because we've cached decrypted results for phrases, when we see that encrypted phrase we can use the cached decrypted result for that encrypted phrase to get without having to run the SimpleDecrypt() function again.

This is the same with all caching mechanism, which make the obvious seem transparent, however with a pure function, you know for certain that there is no chance that our cached decrypted result could be different from running a SimpleDecrypt() on the encrypted string we have - so we have double certainty that we dont have to run the SimpleDecrypt() function.

If SimpleEncrypt() or SimpleDecrypt() could sometimes return different outputs for the same input, then we'd have to cal SimpleDecrypt() to return what the decrypted result is that/this time.

Note using Monads, Select, Bind() within your functions you're making it unlikely that you functions will

never throw exceptions becasue you're catering for both the expected and unexpected data by virtue of using Monads (which ensure that you need to ie. they contain both failure and success logic such as Some/None or Either left or right emedded into themselves so you can and indeed have to cater for them)

In catering for them by extracting their values using Match() or transforming via Select()/Bind()/Map().

Now use cache to decrypt known inputs against outputs produced by pure function SimpleEncrypt():

Apply events over time to change an object

This tutorial shows how you can use the Fold() function in languageExt to change the state of an object over time

```
static void Main(string[] args)
{
    List<int> years = new List<int>
    {
        1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000,
        2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014,
        2015, 2016, 2017, 2018, 2019
    };

    List<Event<Person>> events = new List<Event<Person>>
    {
        new ChangeNameEvent("Stuart"),
        new ChangeExpertiseEvent("Programming"),
        new ChangeNameEvent("Stuart Mathews"),
        new ChangeExpertiseEvent("Running"),
        new ChangeAgeEvent(33),
    };

    Person person = new Person();

    // A state (InitialResult) changes over time and it changes using results of the previous
    // change. It uses an item from the array in changing the state each time.
    // The state changes the number of elements in the IEnumerable
    // For a Lst which has multiple items in it, the state will change that many times

    // NB: years represent the state changes that will occur
    var changedPerson = years.Fold(/*initial state*/ person,
                                   (previousResult, year) => ChangeState(year, previousResult));

    // Apply some events to the person over time
    changedPerson = events.Fold(person, (previousResult, evt) => evt.ApplyEventTo(previousResult));

    // View the changed person
    Console.WriteLine($"{changedPerson}");

    // local function
    Person ChangeState(int year, Person previousResult)
    {
        Person updatedPerson = new Person(previousResult);
               updatedPerson.History.Add($"\nIn {year}, person was {updatedPerson.Age} years old");
               updatedPerson.Age++;
        return updatedPerson;
    }
}

public abstract class Event<T>
{
    public string EventName {get;set;}
    public string EventDescrition {get;set;}
    public abstract T ApplyEventTo(T thing);
}
```

```
 public class Person
 {
     public int Age {get;set;} = 0;
     public string Name {get;set;} = "NoNameSet";
     public string Expertise {get;set;} = "NoExpertiseSet";
     public string Role {get;set;} = "NoRoleSet";
     public Person()
         => Age = 0;

     public Person(Person p) : this()
     {
         Age = p.Age;
         History = p.History;
         Name = p.Name;
         Expertise = p.Expertise;
         Role = p.Role;
     }

     public List<string> History = new List<string>();
     public override string ToString()
         => $"Person's name is {Name} and is {Age} years old now. Expertise is '{Expertise}',
             while Role is '{Role}'\nPrevious state chnges were: {string.Join(',', History)}";
 }

public class ChangeAgeEvent : Event<Person>
{
    public ChangeAgeEvent(int newAge)
    {
        NewAge = newAge;
        EventDescrition = "\nContains inforamtion about changing a persons age";
        EventName = "ChangeAgeEvent";
    }

    public int NewAge { get; }

    public override Person ApplyEventTo(Person person)
    {
        Person updated = new Person(person);
            updated.Age = NewAge;
        updated.History.Add($"\nChanged age to " + NewAge);
        return updated;
    }
}
```

```
public class ChangeRoleEvent : Event<Person>
{
    public ChangeRoleEvent(string newRole)
    {
        NewRole = newRole;
    }

    public string NewRole { get; }

    public override Person ApplyEventTo(Person person)
    {
        var updated = new Person(person);
            updated.Role = NewRole;
            updated.History.Add($"\nChanged Role to {NewRole}");
        return updated;
    }
}
```

```
public class ChangeNameEvent:  Event<Person>
{
    public ChangeNameEvent(string name)
    {
        NewName = name;
        EventName = "ChangedNameEvent";
        EventDescrition = "Changes a Persons name";
    }

    public string NewName { get; }

    public override Person ApplyEventTo(Person person)
    {
        Person updated = new Person(person);
               updated.Name = NewName;
        updated.History.Add($"\nChanged name to {NewName}");
        return updated;
    }
}
```

```
public class ChangeExpertiseEvent : Event<Person>
{
    public string NewExpertise {get;set;}
    public ChangeExpertiseEvent(string newExpertise)
    {
        NewExpertise = newExpertise;
    }
    public override Person ApplyEventTo(Person person)
    {
        var updated = new Person(person);
            updated.Expertise = NewExpertise;
            updated.History.Add($"\nChanged Expertise to {NewExpertise}");
        return updated;
    }
}
```

Smart constructors and Immutable data-types

This tutorial demonstrates creating immutable objects using Smart Constructors, and using them as you would use any other OOP objects.

However, this object is immutable in as much as its specifically designed not to have its state changed by operations

The operations that it does have, create a new object with the modification and leaves the original object untouched.

```
static void Main(string[] args)
{
    // Note we cannot create a person through a conventional constructor
    // we use a smart constructor ie Of() to perform validation on construction and return an
    // Option<Person> depending the construction
    // of the person is valid - no exceptions are thrown when they are not - which is usually
    // what happens with normal constructors.

    var originalMe = Person.Of("Steward", "Mathews");


    // Perform a state change on the person, however we enforce creation of a new copy of person,
    // pre-initialized with previous data.
    // We therefore don't modify state - particularly of the person that we are about to modify –
    // this is not a member function that changes te object as is traditionally done
    // Remember we're binidng here as Rename returns a Monad already if it didn't we'd use Map()
    var changedMe = originalMe.Bind(person => person.Rename("Steward Rob Charles", "Mathews"));

    // We can do this declaratively also
    var stateChangedMe1 = from person in originalMe
                          from renamedperson in person.Rename("Steward Rob Charles", "Mathews")
                            select renamedperson;

    // We can also chain the modifications using the extension method

    var lastChange = changedMe
        .Rename("Stuart", "Mathews")
        .Rename("Stuart Robert Charles", "Mathews");

    Console.WriteLine($"First I was {originalMe}, then ultimately I was {lastChange}");

}
```

```
// Extension methods allow us to chain non-instance method calls,
public static class PersonExtensions
{
    /// <summary>
    /// Functional because
    /// A) it does not call any non-pure functions or use any other data than the
    /// arguments(with are immutable) passed to it.
    /// B) functions should be declared separately from the data they act upon, like it is
    /// C) Creates a new Object
    /// </summary>
    /// <param name="person"></param>
    /// <param name="firstName"></param>
    /// <returns></returns>
    public static Option<Person>
         Rename(this Option<Person> person, string firstName, string lastName)
    {
        return person.Bind(per => per.Rename(firstName, lastName));
    }
}
```

```
/// <summary>
/// This is an immutable object because:
/// a) Has private setters and that means its state cannot be changed after creation
/// b) All properties are immutable - either strings or native types or
/// System.Collections.Immutable types
/// c) Any change that needs to take place must result in a newly create object of this type
/// </summary>
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    /// <summary>
    /// Notice this is a private constructor so a Person cannot be created normally...
    /// there must be another way in!
    /// It must be created indirectly via .Of() or .New() which then can use the
    /// private constructor below
    /// </summary>
    /// <param name="firstName">the persons first name</param>
    /// <param name="lastName">The persons last name</param>
    private Person(string firstName, string lastName)
    {
        if (!IsValid(firstName, lastName))
        {
            throw new ArgumentException("Invalid input");
        }
        FirstName = firstName;
        LastName = lastName;
    }
    public override string ToString()
        => $"{FirstName} {LastName}";

    /// <summary>
    /// This pretends to be a constructor by being the only method allowed to call the
    /// constructor. The actual and real constructor is only called if input is valid to this
    /// function. As a result it gaurentees no exceptions can ever be thrown during creation of
    /// the object through this method.
    /// </summary>
    /// <param name="firstName">Persons first name</param>
    /// <param name="lastName">Persons last name</param>
    /// <returns></returns>
    public static Option<Person> Of(string firstName, string lastName)
        => IsValid(firstName, lastName)
            ? Option <Person>.Some(new Person(firstName, lastName))
            : Option <Person>.None;

    private static bool IsValid(string firstName, string lastName)
        => !string.IsNullOrEmpty(firstName) && !string.IsNullOrEmpty(lastName);

    // Instance method allows this function to be chained.
    // With Helper to make object creation easier as properties of the object change
    public Option<Person> Rename(string firstName = null, string lastName = null)
        => Of(firstName, lastName);
}
```