Functional Programming made easy in C# with Language-ext

Yoan Thirion
Dec 19, 2019 · 7 min read

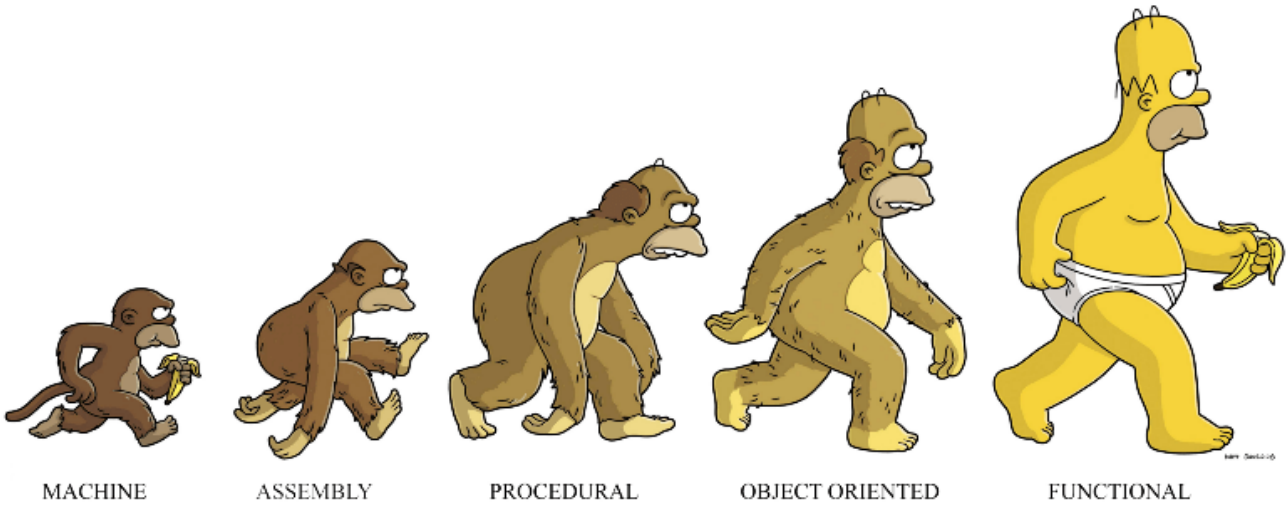
> *Read this article if you want to demystify Functional Programming and understand why and how to start using FP paradigms in C#.*

In the title I have written FP and C# but the question you probably wonder is :

What the hell should we want to use FP paradigms in an OO language like C# ?

Here are some arguments to answer this question (from my POV) :

- Write less code and more meaningful one : improved code clarity
- Avoid side effects by using immutability and pure functions : Easier concurrent programming
- Easier testing : it's really easy to test functions
- Easier debugging
- Funnier to write

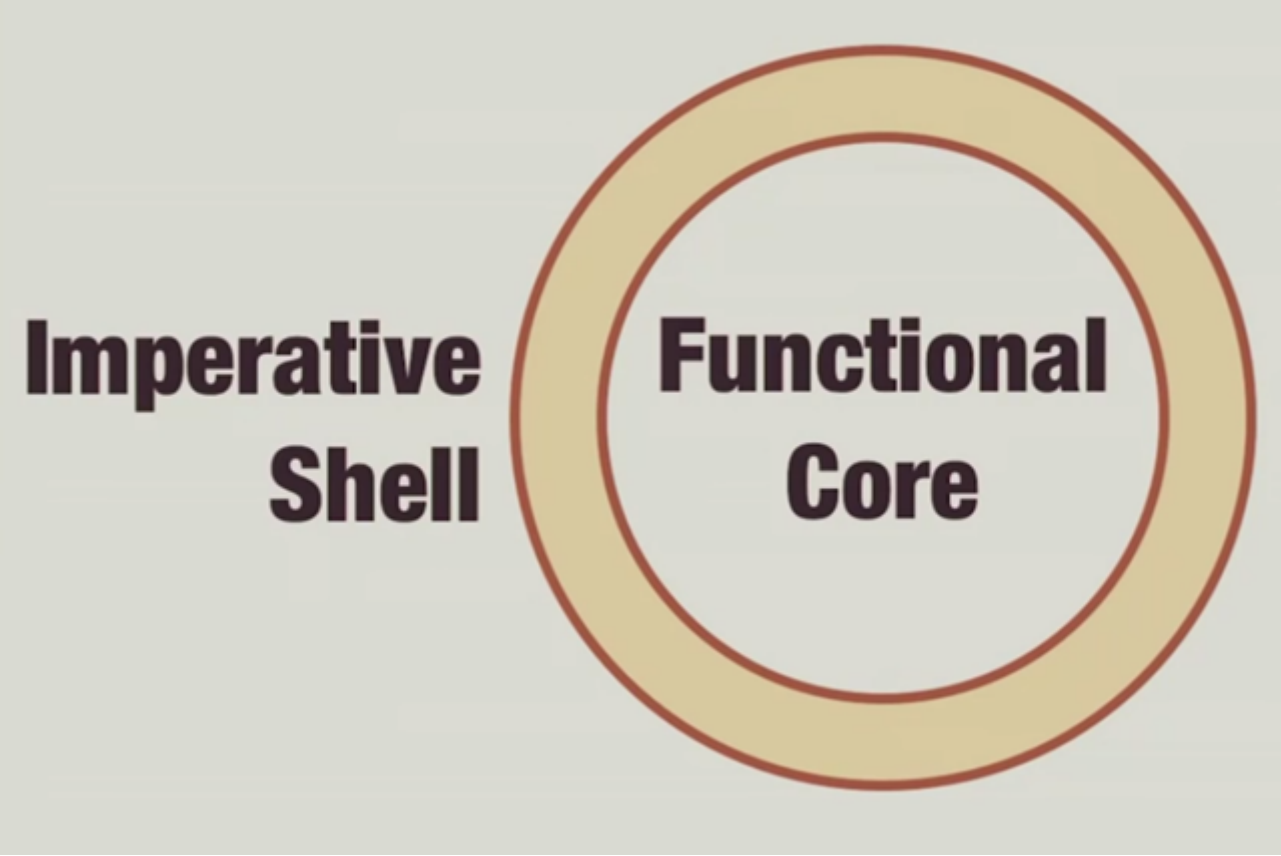


OK but why don't we use a FP language instead of C# ?

In companies it's really hard to change the habits. We don't all work in the Sillicon Valley and in my reality the latest versions in the production environments are : Java 8, .NET Framework 4, NodeWhat ?, …

In my world organizations are not yet ready to invest in pure FP languages like F#, Kotlin, Clojure or whatever. They are stuck in legacy code…

To go further on this topic I invite you to watch this video on ***Functional Core, Imperative Shell*** : https://discventionstech.wordpress.com/2017/06/30/functional-core-and-imperative-shell/



It's a pattern that could be really useful to every developers on earth I think.

Functional core (declarative) composed of pure functions (in / out) and easy to test without any mocks.

Imperative shell or reactive container the service logic that you can test with integration tests

Now I have proved the interest of FP in OO language let's go for a quick lesson.

. . .

Let's demystify Functional Programming (FP)

> *"In computer science, **functional programming** is a **programming** paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data." — wikipedia*

Basically FP is all about **functions and immutability :**

- Pure functions
- Lambda functions (anonymous)
- High order functions
- Composition
- Closures
- Currying & partial application
- Recursion

If you have already looked at it, you have probably read other words like functors, monads and monoïds that can be very scary but in reality concepts are not.

Let's start by explaining 3 concepts :

Immutability

We **never** want to **mutate an object in FP we want to avoid side effects so we prefer** creating a new one.

If we have mutable objects it could be possible for another function to mutate the object we were working on concurrently.

With immutability it ensures a function remains pure, it makes concurrency much more easy to handle and makes our dev life easier.

Pure Functions

A pure function don't refer to any global state. The same inputs will always get the same output.

```
static int Double(int i) => i * 2;
```

Combined with **immutable data** types mean you can be sure the same inputs will give the same outputs.

Higher order function

A function that does at least one of the following:

- Takes one or more functions as arguments
- Returns a function as its result

```
private string Print(
    Invoice invoice,
    Dictionary<string, Play> plays,
    Func<string, int, int, string> lineFormatter,
    Func<string, Statement, string> statementFormatter)
{
   return invoice.Performances
        .Map(performance => CreateStatement(plays, performance, lineFormatter))
        .Reduce((context, line) => context.Append(line))
        ?.FormatFor(invoice.Customer, statementFormatter);
}
```

A higher order function that takes 2 functions as arguments

Before demonstrating code I highly recommend you to read this great article about FP concepts (I won't do better for sure) :

http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html#just-what-is-a-functor,-really?

Now you know the basics behind FP we can start deep diving into Language-ext.

. . .

What is Language-ext ?

According to the creator of the lib **Paul Louth** : It's a library that uses and abuses the features of C# to provide a functional-programming 'base class library' that, if you squint, can look like extensions to the language itself.

His desire here is to make programming in C# much more reliable and to make the engineer's inertia flow in the direction of declarative and functional code rather than imperative.

It's basically "**one lib to rule them all**" : https://github.com/louthy/language-ext

Features

Location	Feature	Description
Core	`Arr<A>`	Immutable array
Core	`Lst<A>`	Immutable list
Core	`Map<K, V>`	Immutable map
Core	`Map<OrdK, K, V>`	Immutable map with Ord constraint on `K`
Core	`HashMap<K, V>`	Immutable hash-map
Core	`HashMap<EqK, K, V>`	Immutable hash-map with Eq constraint on `K`
Core	`Set<A>`	Immutable set
Core	`Set<OrdA, A>`	Immutable set with Ord constraint on `A`
Core	`HashSet<A>`	Immutable hash-set
Core	`HashSet<EqA, A>`	Immutable hash-set with Eq constraint on `A`
Core	`Que<A>`	Immutable queue
Core	`Stck<A>`	Immutable stack
Core	`Option<A>`	Option monad that can't be used with `null` values
Core	`OptionAsync<A>`	OptionAsync monad that can't be used with `null` values with all value realisation does asynchronously
Core	`OptionUnsafe<T>`	Option monad that can be used with `null` values
Core	`Either<L,R>`	Right/Left choice monad that won't accept `null` values
Core	`EitherUnsafe<L, R>`	Right/Left choice monad that can be used with `null` values
Core	`EitherAsync<L, R>`	EitherAsync monad that can't be used with `null` values with all value realisation done asynchronously
Core	`Try<A>`	Exception handling lazy monad
Core	`TryAsync<A>`	Asynchronous exception handling lazy monad
Core	`TryOption<A>`	Option monad with third state 'Fail' that catches exceptions
Core	`TryOptionAsync<A>`	Asynchronous Option monad with third state 'Fail' that catches exceptions
Core	`Record<A>`	Base type for creating record types with automatic structural equality, ordering, and hash code calculation.
Core	`Lens<A, B>`	Well behaved bidirectional transformations - i.e. the ability to easily generate new immutable values from existing ones, even when heavily nested.
Core	`Reader<E, A>`	Reader monad
Core	`Writer<MonoidW, W, T>`	Writer monad that logs to a `W` constrained to be a Monoid
Core	`State<S, A>`	State monad
Core	`Patch<EqA, A>`	Uses patch-theory to efficiently calculate the difference (`Patch.diff(list1, list2)`) between two collections of `A` and build a patch which can be applied (`Patch.apply(patch, list)`) to one to make the other (think git diff).
Parsec	`Parser<A>`	String parser monad and full parser combinators library
Parsec	`Parser<I, O>`	Parser monad that can work with any input stream type
Core	`NewType<SELF, A, PRED>`	Haskell `newtype` equivalent i.e: `class Hours : NewType<Hours, double> { public Hours(double value) : base(value) { } }` . The resulting type is: equatable, comparable, foldable, a functor, monadic, and iterable
Core	`NumType<SELF, NUM, A, PRED>`	Haskell `newtype` equivalent but for numeric types i.e: `class Hours : NumType<Hours, TDouble, double> { public Hours(double value) : base(value) { } }` . The resulting type is: equatable, comparable, foldable, a functor, a monoid, a semigroup, monadic, iterable, and can have basic artithmetic operations performed upon it.
Core	`FloatType<SELF, FLOATING, A, PRED>`	Haskell `newtype` equivalent but for real numeric types i.e: `class Hours : FloatType<Hours, TDouble, double> { public Hours(double value) : base(value) { } }` . The resulting type is: equatable, comparable, foldable, a functor, a monoid, a semigroup, monadic, iterable, and can have complex artithmetic operations performed upon it.
Core	`Nullable<T>` extensions	Extension methods for `Nullable<T>` that make it into a functor, applicative, foldable, iterable and a monad
Core	`Task<T>` extensions	Extension methods for `Task<T>` that make it into a functor, applicative, foldable, iterable and a monad
Core	`Validation<FAIL,SUCCESS>`	Validation applicative and monad for collecting multiple errors before aborting an operation
Core	`Validation<MonoidFail, FAIL, SUCCESS>`	Validation applicative and monad for collecting multiple errors before aborting an operation, uses the supplied monoid in the first generic argument to collect the failure values.
Core	Monad transformers	A higher kinded type (ish)
Core	Currying	Translate the evaluation of a function that takes multiple arguments into a sequence of functions, each with a single argument
Core	Partial application	the process of fixing a number of arguments to a function, producing another function of smaller arity
Core	Memoization	An optimization technique used primarily to speed up programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again
Core	Improved lambda type inference	`var add = fun( (int x, int y) => x + y)`
Core	`IQueryable<T>` extensions	
Core	`IObservable<T>` extensions	

Language-ext vs LinQ

Yes FP paradigms in C# exist since the introduction of LinQ. Here is a mapping between some language-ext functionalities and their equivalence in LinQ :

```
Sum                 // For Option<int> it's the wrapped value.
Count               // For Option<T> is always 1 for Some and 0 for None.
Bind                // Part of the definition of anything monadic - SelectMany in LINQ
Exists              // Any in LINQ - true if any element fits a predicate
Filter              // Where in LINQ
Fold                // Aggregate in LINQ
ForAll              // All in LINQ - true if all element(s) fits a predicate
Iter                // Passes the wrapped value(s) to an Action delegate
Map                 // Part of the definition of any 'functor'.  Select in LINQ
Lift / LiftUnsafe   // Different meaning to Haskell, this returns the wrapped value.  Dangerous, should be used
```

```
Select
SeletMany
Where
```

Easy immutable record types

Implementing immutable data structures is a nightmare in OO languages (Equals, GetHashCode, IEquatable<A>, IComparer<A>, implementing operators: ==, !=, <, <=, >, >=).

That's why there is a **Record** class in language-ext :

```
public class User
{
    public readonly Guid Id;
    public readonly string Name;
    public readonly int Age;

    public User(Guid id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}

public class UserRecord : Record<UserRecord>
{
    public readonly Guid Id;
    public readonly string Name;
    public readonly int Age;

    public UserRecord(Guid id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}

[Fact]
public void record_types()
{
    var spongeGuid = Guid.NewGuid();

    var spongeBob = new User(spongeGuid, "Spongebob", 40);
    var spongeBob2 = new User(spongeGuid, "Spongebob", 40);
```

```
36
37      Assert.False(spongeBob.Equals(spongeBob2));
38
39      // Use immutable records available in Language-ext
40      var spongeBobRecord = new UserRecord(spongeGuid, "Spongebob", 40);
41      var spongeBobRecord2 = new UserRecord(spongeGuid, "Spongebob", 40);
42
```

Yes in C# we have out parameters on TryParse for example now it's finished :

```
// Attempts to parse the value, uses 0 if it can't
int res = parseInt("123").IfNone(0);

// Attempts to parse the value, uses 0 if it can't
int res = ifNone(parseInt("123"), 0);

// Attempts to parse the value, doubles it if can, returns 0 otherwise
int res = parseInt("123").Match(
              Some: x => x * 2,
             None: () => 0
          );

// Attempts to parse the value, doubles it if can, returns 0 otherwise
int res = match( parseInt("123"),
                 Some: x => x * 2,
                 None: () => 0 );
```

Option

Many functional languages disallow null values, as null-references can introduce hard to find bugs. Option is a type safe alternative to null values. (it also exists in F#).

> *Avoid nulls by using an Option*

An Option<T> can be in one of two states :
some => the presence of a value
none => lack of a value

```
1   Option<int> aValue = 2;
2   aValue.Map(x => x + 3); // Some(5)
```

```
3
4   Option<int> none = None;
5   none.Map(x => x + 3); // None
6
7   //Left -> Some, Right -> None
8   aValue.Match(x => x + 3, () => 0); // 5
9   none.Match(x => x + 3, () => 0); // 0
10
11  // Returns the Some case 'as is' -> 2 and 1 in the None case
12  int value = aValue.IfNone(1);
13  int noneValue = none.IfNone(42); // 42
```

Match : match down to primitive type

Map: We can match down to a primitive type,or can stay in the elevated types and do logic using map.

- Lambda inside map won't be invoked if Option is in None state
- Option is a replacement for if statements ie if obj == null
- Working in elevated context to do logic

Lists are functors

We can map them :

```
1   new int[] { 2, 4, 6 }.Map(x => x + 3); // 5,7,9
2   new List<int> { 2, 4, 6 }.Map(x => x + 3); // 5,7,9
3   //Prefer use List (Immutable list)
4   List(2, 4, 6).Map(x => x + 3); // 5,7,9
```

list_functors.cs hosted with ♡ by GitHub view raw

Function composition

What happens when you apply a function to another function? When you use **map on a function**, you're just doing **function composition :**

```
1     static Func<int, int> Add2 = x => x + 2;
2     static Func<int, int> Add3 = x => x + 3;
3
4     static int Add5(int x) => Add2.Compose(Add3)(x);
```

func_composition.cs hosted with ♡ by GitHub view raw

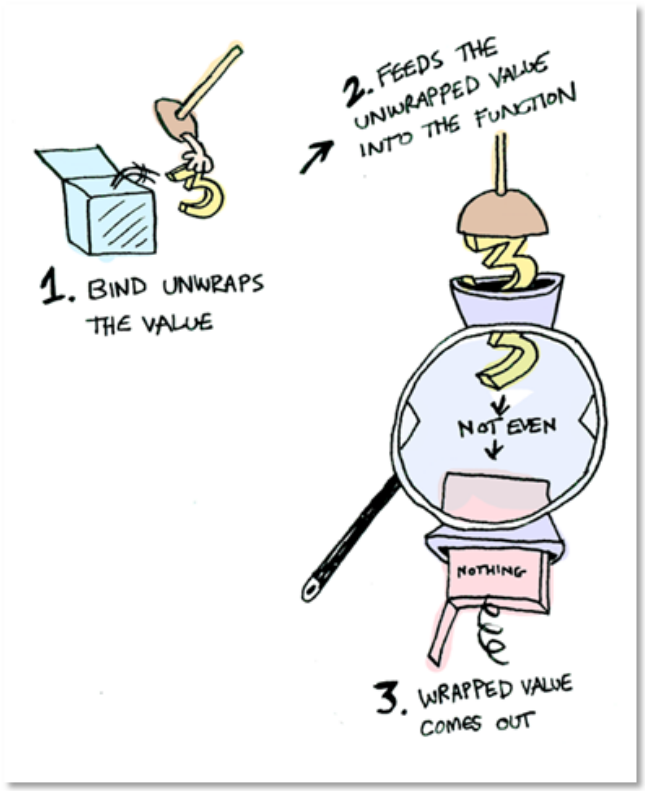
Bind

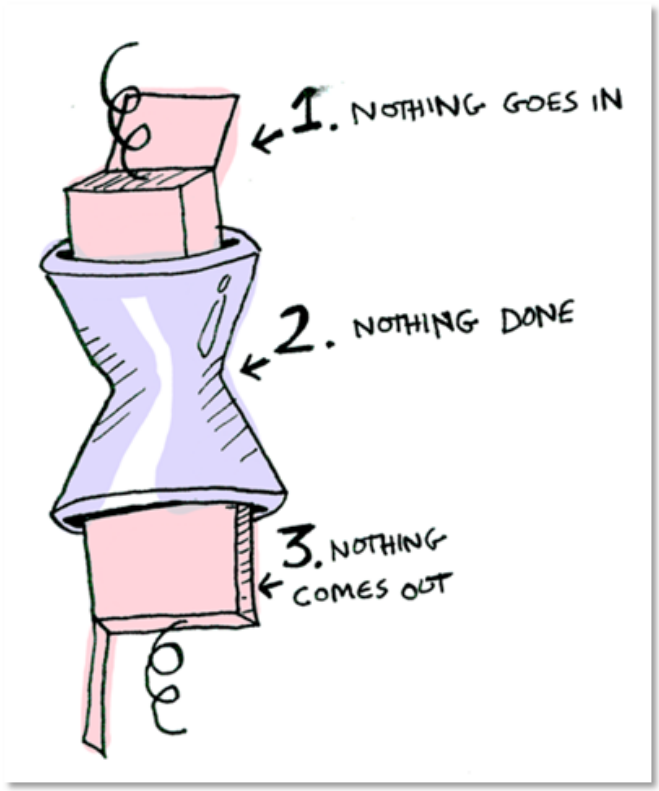
Monads apply a function that **returns a wrapped value to a wrapped value**. Monads have a function “bind” to do this.

Suppose **half** is a function that only **works on even numbers.**

```
static Option<double> Half(double x)
    => x % 2 == 0 ? x / 2 : Option<double>.None;
```

What if we feed it a wrapped value? This is where **bind** comes in!






```
1  static Option<double> Half(double x)
2      => x % 2 == 0 ? x / 2 : Option<double>.None;
3
4  [Fact]
5  public void bind_monad()
6  {
7      Option<double>.Some(3).Bind(x => Half(x));// None
8      Option<double>.Some(4).Bind(x => Half(x));// Some(2)
9  }
10
11 [Fact]
12 public void chain_bind_monad()
```

```
{
    Option<double>.Some(20)
        .Bind(x => Half(x))// Some(10)
        .Bind(x => Half(x))// Some(5)
        .Bind(x => Half(x));// None
}
```

Try

Exception handling made easier the use of try. No more needs of ugly try/catch that pollutes the flow of your code you can describe your pipeline in a really readable way.

```
public void file_monad_example()
{
    GetLine()
        .Bind(ReadFile)
        .Bind(PrintStrln)
        .Match(success => Console.WriteLine("SUCCESS"),
                failure => Console.WriteLine("FAILURE"));
}

static Try<string> GetLine()
{
    Console.Write("File:");
    return Try(() => Console.ReadLine());
}

static Try<string> ReadFile(string filePath) => Try(() => File.ReadAllText(filePath));

static Try<bool> PrintStrln(string line)
{
    Console.WriteLine(line);
    return Try(true);
}
```

Here if something fails, you have the exception in the failure match part. The better part is that **Try** has a brother called **TryAsync** that is demonstrated in the real life example.

Memoization

Memoization is some kind of caching, if you memoize a function, it will be only executed once for a specific input.

```
static Func<string, string> GenerateGuidForUser = user => user + ":" + Guid.NewGuid();
```

```
static Func<string, string> GenerateGuidForUserMemoized = memo(GenerateGuidForUser);

[Fact]
public void memoization_example()
{
    GenerateGuidForUserMemoized("spongebob");// spongebob:e431b439-3397-4016-8d2e-e4629e51bf62
    GenerateGuidForUserMemoized("buzz");// buzz:50c4ee49-7d74-472c-acc8-fd0f593fccfe
    GenerateGuidForUserMemoized("spongebob");// spongebob:e431b439-3397-4016-8d2e-e4629e51bf62
}
```

Partial application

Partial application allows you to **create new function from an existing one by setting some arguments.**

```
static Func<int, int, int> Multiply = (a, b) => a * b;
static Func<int, int> TwoTimes = par(Multiply, 2);

[Fact]
public void partial_app_example()
{
    Multiply(3, 4); // 12
    TwoTimes(9); // 18
}
```

partial.cs hosted with ♡ by GitHub view raw

Either

Either represents a value of two types, it is either **a left or a right** by convention **left** is the **failure case**, and **right** the **success case.**

```
public static Either<Exception, string> GetHtml(string url)
{
    var httpClient = new HttpClient(new HttpClientHandler());
    try
    {
        var httpResponseMessage = httpClient.GetAsync(url).Result;
        return httpResponseMessage.Content.ReadAsStringAsync().Result;
    }
    catch (Exception ex) { return ex; }
}

[Fact]
public void either_example()
{
```

```
15
16      GetHtml("unknown url"); // Left InvalidOperationException
17      GetHtml("https://www.google.com"); // Right <!doctype html...
18
19      var result = GetHtml("https://www.google.com");
20
21      result.Match(
22              Left: ex => Console.WriteLine("an exception occured" + ex),
23              Right: r => Console.WriteLine(r)
24          );
25  }
```

Fold vs Reduce (Aggregate in LinQ)

- **Fold** takes an explicit initial value for the accumulator. The accumulator and result type can differ as the accumulator is provided separately.
- **Reduce** uses the first element of the input list as the initial accumulator value (just like the Aggregate function). The accumulator and therefore result type must match the list element type.

```
List.fold : ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State
List.reduce : ('T -> 'T -> 'T) -> 'T list -> 'T
```

```
1   [Fact]
2   public void fold_vs_reduce()
3   {
4       //fold takes an explicit initial value for the accumulator
5       //Can choose the result type
6       var foldResult = List(1, 2, 3, 4, 5)
7           .Map(x => x * 10)
8           .Fold(0m, (x, s) => s + x); // 150m
9
10      //reduce uses the first element of the input list as the initial accumulator value
11      //Result type will be the one of the list
12      var reduceResult = List(1, 2, 3, 4, 5)
13          .Map(x => x * 10)
14          .Reduce((x, s) => s + x); // 150
15  }
```

Real life example

This example demonstrates the kind of usage you could have of language-ext in application services : **chaining operations / pipelining, improve error handling, no more redundant checks, …**

```
private readonly PersonRepository personRepository;
private readonly ILogger logger;
private readonly TwitterService twitterService;

public PersonService(ILogger logger)
{
    this.logger = logger;
    personRepository = new PersonRepository();
    twitterService = new TwitterService();
}

private TryAsync<Context> CreateContext(long personId)
{
    return TryAsync(() => personRepository.GetById(personId))
            .Map(person => new Context(person));
}

private TryAsync<Context> RegisterTwitter(Context context)
{
    return TryAsync(() => twitterService.Register(context.Email, context.Name))
            .Map(account => context.SetAccount(account));
}

private TryAsync<Context> Authenticate(Context context)
{
    return TryAsync(() => twitterService.Authenticate(context.Email, context.Password))
            .Map(token => context.SetToken(token));
}

private TryAsync<Context> Tweet(Context context)
{
    return TryAsync(() => twitterService.Tweet(context.Token, "Hello les cocos"))
            .Map(tweet => context.SetTweet(tweet));
}

private TryAsync<Context> UpdateParty(Context context)
{
    return TryAsync(async () =>
    {
        await personRepository.Update(context.Id, context.AccountId);
        return context;
    });
```

```
43  }
44
45  public async Task<string> Register(long personId)
46  {
47      string result = string.Empty;
48      await CreateContext(personId)
49              .Bind(RegisterTwitter)
50              .Bind(Authenticate)
51              .Bind(Tweet)
52              .Bind(UpdateParty)
53              .Match(
54              context =>
55              {
56                  logger.LogSuccess($"User {personId} registered");
57                  result = context.Url;
58              },
59              exception => logger.LogFailure($"Unable to register user : {personId} {exception.Me
60
```

> *much more easier.*
>
> *You can see it at your first step to go into the FP world, before one day be able to use a language like Clojure or F#.*

Resources

All the source code is available in my github repository : https://github.com/ythirion/fp-in-csharp-sandbox. I have written this article based on slides I have created for FP workshops : https://github.com/ythirion/fp-in-csharp-sandbox/blob/master/fp-in-cs.pdf

If you want to go further :

- *FP in pictures* : http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html#just-what-is-a-functor,-really?
- *Gitter on language-ext* : https://gitter.im/louthy/language-ext
- *Doc and examples* : https://github.com/louthy/language-ext/issues

- *What's new in C# 8* : https://medium.com/swlh/how-c-8-helps-software-quality-cfa81a18907f
- *"functional core, imperative shell" video* : https://discventionstech.wordpress.com/2017/06/30/functional-core-and-imperative-shell/
- ***Domain modeling made functional book from Scott Wlaschin*** : https://www.amazon.com/Domain-Modeling-Made-Functional-Domain-Driven/dp/1680502549/ref=sr_1_1?crid=FXE4W7BCGETO&keywords=domain+modeling+f%23&qid=1576686914&sprefix=design+thin%2Caps%2C330&sr=8-1

Functional Programming Csharp Language Ext