INTRO TO FP (IN C#)

FUNCTIONAL PROGRAMMING IS ALL ABOUT FUNCTIONS

- PURE FUNCTIONS
- IMMUTABILITY
- LAMBDA FUNCTIONS (ANONYMOUS)
- HIGHER ORDER FUNCTIONS
- COMPOSITION
- CLOSURES (RETURNING FUNCTIONS FROM FUNCTIONS)
- CURRYING & PARTIAL APPLICATION
- PATTERN MATCHING
- RECURSION
- LAZY EVALUATION

ONE LIB TO RULE THEM ALL

language-ext

The namespace `LanguageExt` contains the core types, and `LanguageExt.Prelude` contains the functions that you bring into scope `using static LanguageExt.Prelude`.

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	`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 arithmetic 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 arithmetic 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	

ONE NAMESPACE TO RULE THEM ALL

```
using static LanguageExt.Prelude;
```

A note about naming

One of the areas that's likely to get seasoned C# heads worked up is my choice of naming style. The intent is to try and make something that 'feels' like a functional language rather than follows the 'rule book' on naming conventions (mostly set out by the BCL).

There is however a naming guide that will stand you in good stead whilst reading through this documentation:

- Type names are `PascalCase` in the normal way
- The types all have constructor functions rather than public constructors that you instantiate with `new`. They will always be `PascalCase`:

```
Option<int> x = Some(123);
Option<int> y = None;
List<int> items = List(1,2,3,4,5);
Map<int, string> dict = Map((1, "Hello"), (2, "World"));
```

- Any (non-type constructor) static function that can be used on its own by `using static LanguageExt.Prelude` are `camelCase`.

```
var x = map(opt, v => v * 2);
```

- Any extension methods, or anything 'fluent' are `PascalCase` in the normal way

```
var x = opt.Map(v => v * 2);
```

EASY IMMUTABLE RECORD TYPES

Difficulty in creating immutable record types

It's no secret that implementing immutable record types, with structural equality, structural ordering, and efficient hashing solutions is a real manual head-ache of implementing `Equals`, `GetHashCode`, deriving from `IEquatable<A>`, `IComparer<A>`, and implementing the operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. It is a constant maintenance headache of making sure they're kept up to date when new fields are added to the type - with no compilation errors if you forget to do it.

Record<A>

This can now be achieved simply by deriving your type from `Record<A>` where `A` is the type you want to have structural equality and ordering. i.e.

```
public class TestClass : Record<TestClass>
{
    public readonly int X;
    public readonly string Y;
    public readonly Guid Z;

    public TestClass(int x, string y, Guid z)
    {
        X = x;
        Y = y;
        Z = z;
    }
}
```

This gives you `Equals`, `IEquatable.Equals`, `IComparer.CompareTo`, `GetHashCode`, `operator==`, `operator!=`, `operator >`, `operator >=`, `operator <`, and `operator <=` implemented by default. It also gives you a default `ToString()` implementation and `ISerializable.GetObjectData()` with a deserialisation constructor.

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

    2 references | 1/1 passing | 0 changes | 0 authors, 0 changes
    public UserRecord(
        Guid id,
        string name,
        int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}
```

```
var spongeGuid = Guid.NewGuid();

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

Assert.False(spongeBob.Equals(spongeBob2));

var spongeBobRecord = new UserRecord(spongeGuid, "Spongebob", 40);
var spongeBobRecord2 = new UserRecord(spongeGuid, "Spongebob", 40);

Assert.True(spongeBobRecord.Equals(spongeBobRecord2));
```

NO MORE OUT PARAMETERS

```
// 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 );
```

PURE FUNCTIONS

PURE FUNCTIONS 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 THIS MEANS YOU CAN BE SURE THE SAME INPUTS WILL GIVE THE SAME OUTPUTS.

IMMUTABILITY

WE NEVER WANT TO MUTATE AN OBJECT IN FP, BUT A CREATE A NEW ONE.

THIS MAKES SURE THERE ARE NO SIDE EFFECTS CAUSED SOMEWHERE ELSE, THUS ENSURING A FUNCTION REMAINS PURE -> CONCURRENCY = SIMPLER

```
PersonOO dave = new PersonOO();
dave.Name = "crazy T";
dave.Age = 56;
```

NOT POSSIBLE IN F#

```
4 references | 0 changes | 0 authors, 0 changes
internal class Person
{
    2 references | 0 changes | 0 authors, 0 changes
    public string Name { get; } // C#6 Getter only auto-property
    2 references | 0 changes | 0 authors, 0 changes
    public int Age { get; }

    // Smart constructor enforcing a name and age are given
    2 references | 0 changes | 0 authors, 0 changes
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Updates here, enforcing a new object is returned every time
    // strings are nullable because they are reference types
    1 reference | 0/1 passing | 0 changes | 0 authors, 0 changes
    public Person With(string name = null, int? age = null)
    {
        // if null, return the current value
        // else set the newly passed in value
        // ?? null coalescing operator
        return new Person(name: name ?? Name, age: age ?? Age);
    }
}
```

```
var spongebob = new Person(name: "spongebob", age: 22);
var spongebob2 = spongebob.With(age: 43);
```

HIGHER ORDER FUNCTION

A FUNCTION THAT DOES AT LEAST ONE OF THE FOLLOWING:

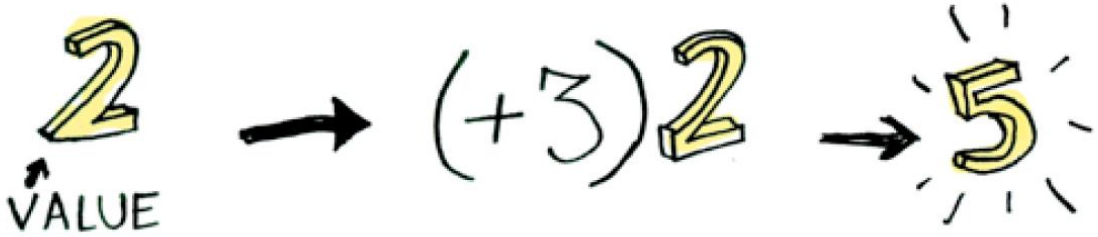
- TAKES ONE OR MORE FUNCTIONS AS ARGUMENTS
- RETURNS A FUNCTION AS ITS RESULT.

```
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);
}
```

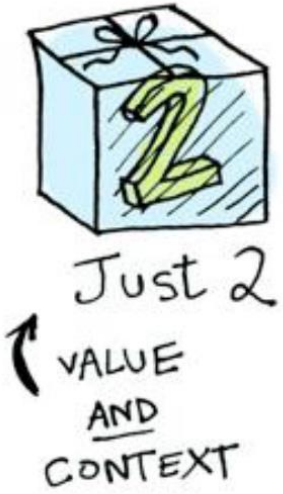
CONTEXT

Here's a simple value

And we know how to apply a function to this value:

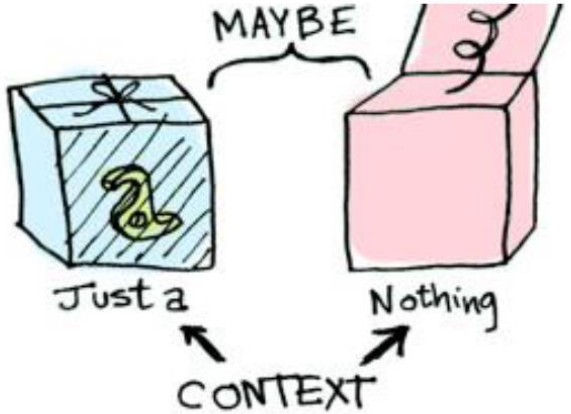


Lets extend this by saying that any value can be in a context.
For now you can think of a context as a box that you can put a value in



Now when you apply a function to this value, you'll get different results **depending on the context**.

This is the idea that Functors, Applicatives, Monads, Arrows etc are all based on. The Maybe data type defines two related contexts

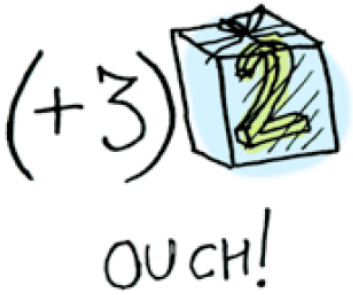



```
data Maybe a = Nothing | Just a
```

SOURCE: HTTP://ADIT.IO/POSTS/2013-04-17-FUNCTORS,_APPLICATIVES,_AND_MONADS_IN_PICTURES.HTML#JUST-WHAT-IS-A-FUNCTOR,-REALLY?

FUNCTORS

When a value is wrapped in a box, you can't apply a normal function to it



This is where **map** comes in! **map** knows how to apply functions to values that are wrapped in a box.

A FUNCTOR IS ANY TYPE THAT DEFINES HOW **MAP** WORKS.

Functors - 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.*

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.*

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

Option<int> none = Option<int>.None;
none.Map(x => x + 3); // None
```

Match: match down to primitive type

```
aValue.Match(x => x + 3, () => 0); //5
none.Match(x => x + 3, () => 0); //0
```

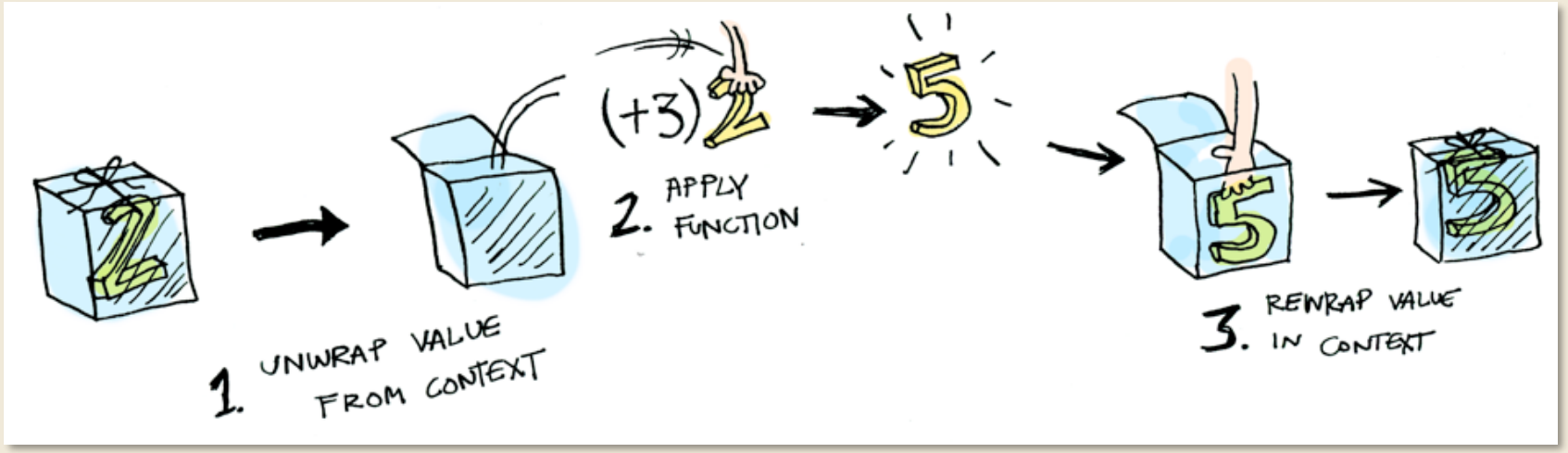
Map: We can match down to a primitive type,or can stay in the elevated types and do logic using map.

- 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

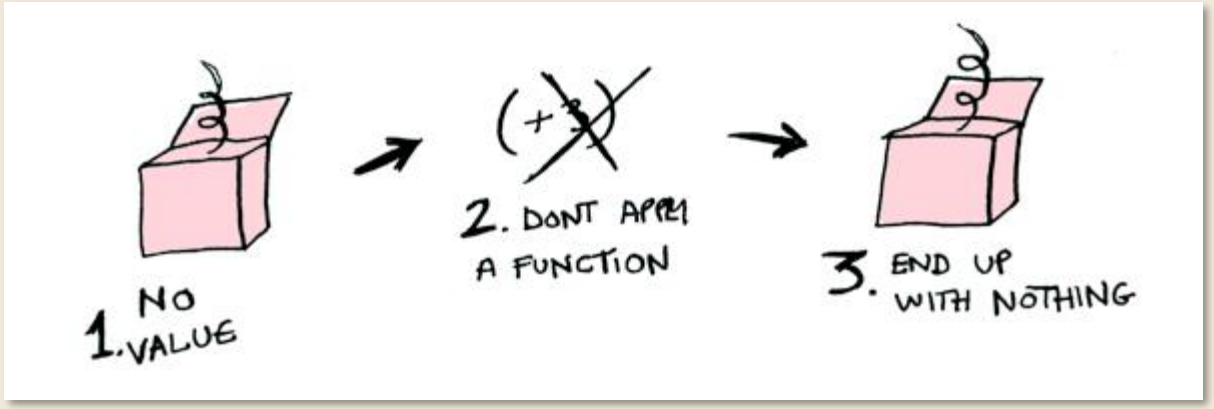
```
// Returns the Some case 'as is' -> 2 and 1 in the None case
int value = aValue.IfNone(1);
int noneValue = none.IfNone(42); // 42
```

FUNCTORS

HERE'S WHAT IS HAPPENING BEHIND THE SCENES WHEN WE WRITE : `Option<int>.Some(2).Map(x => x + 3);`

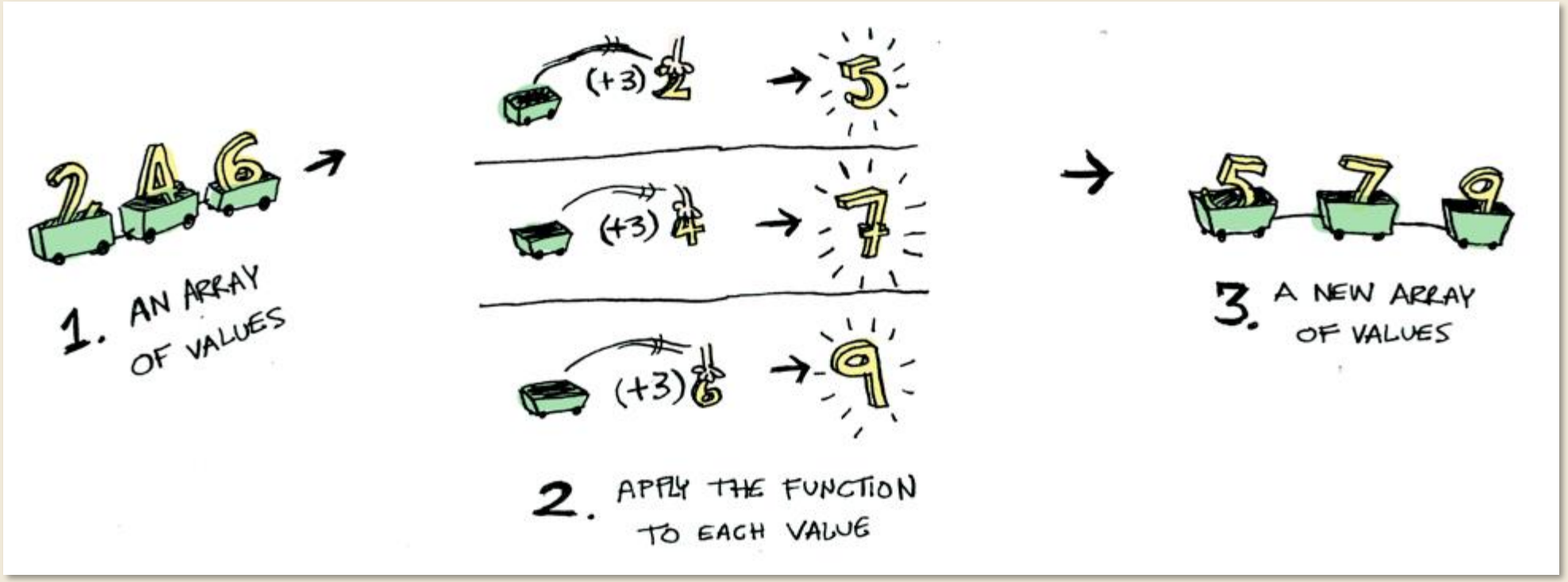


HERE'S WHAT IS HAPPENING BEHIND THE SCENES WHEN WE TRY TO MAP A FUNCTION ON AN EMPTY BOX



FUNCTORS - LISTS

WHAT HAPPENS WHEN YOU APPLY A FUNCTION TO A LIST ?



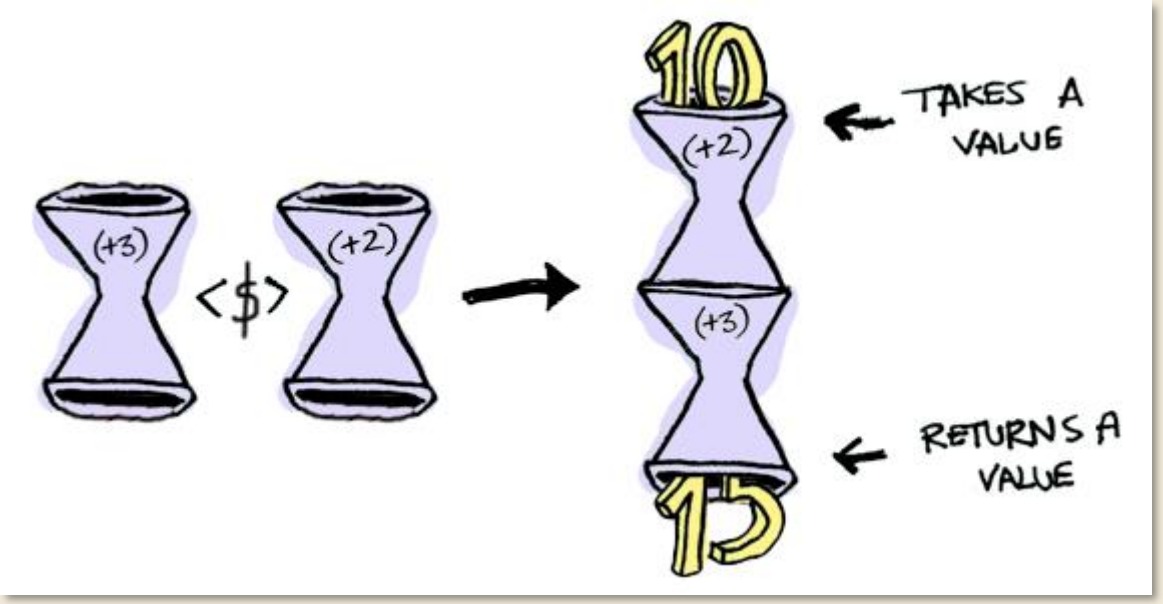
LISTS ARE FUNCTORS TOO

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

FUNCTORS - FUNCTIONS

WHAT HAPPENS WHEN YOU APPLY A FUNCTION TO ANOTHER FUNCTION?

WHEN YOU USE **MAP ON A FUNCTION**, YOU'RE JUST DOING **FUNCTION COMPOSITION**!




```
static Func<int, int> Add2 = x => x + 2;
static Func<int, int> Add3 = x => x + 3;

1 reference | 0 changes | 0 authors, 0 changes
static int Add5(int x) => Add2.Compose(Add3)(x);
```

```
List(2, 4, 6).Map(Add5);// 7,9,11
```

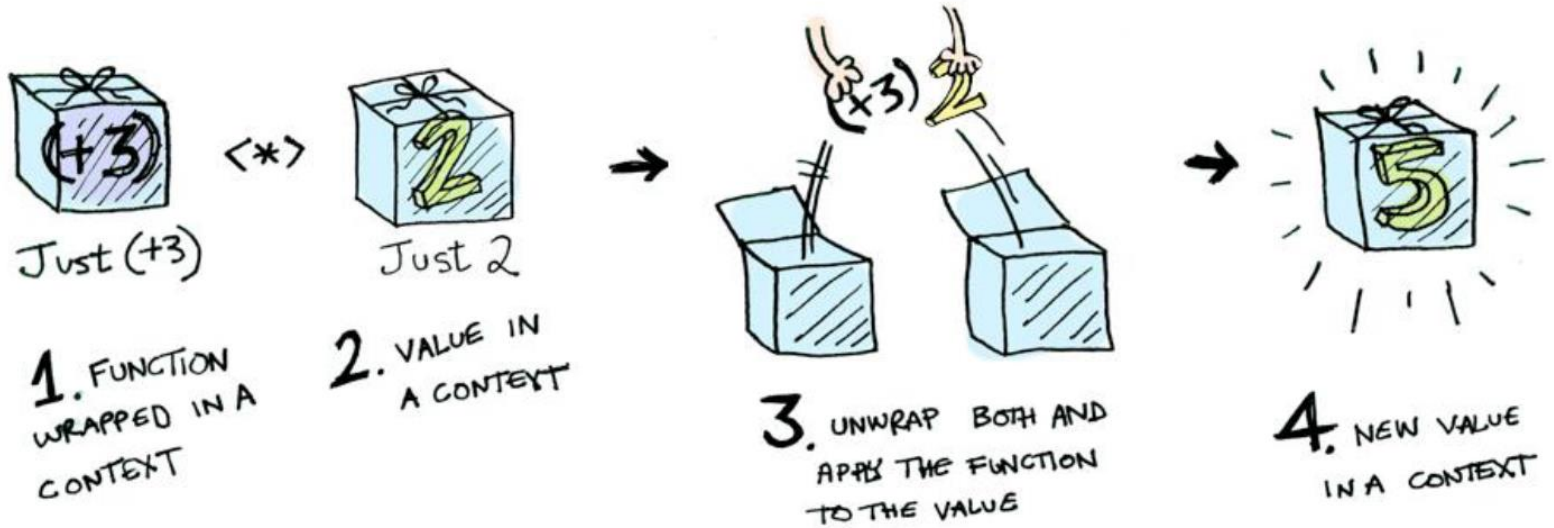
FUNCTORS - FUNCTIONS

WE CAN CHAIN MAP

```
List(2, 4, 6)
      .Map(x => Add5(x)) // 7,9,11
      .Map(x => Add3(x)) // 10,12,14
      .Map(x => Add2(x)) // 12,14,16
      .Map(x => Add5(x)); // 17,19,21
```

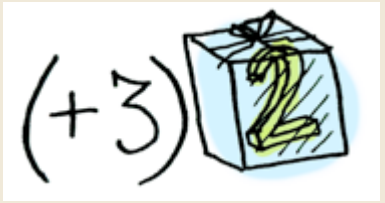
APPLICATIVES

Applicatives are like **functors**, except that not only the value is being wrapped in a context, but the **function to apply** to it also !

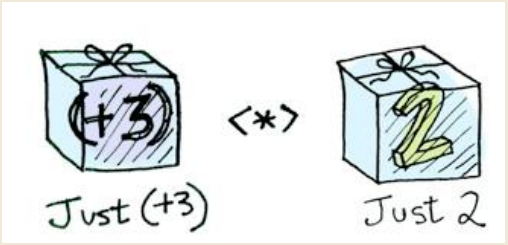


MONADS

FUNCTORS APPLY A FUNCTION TO A WRAPPED VALUE

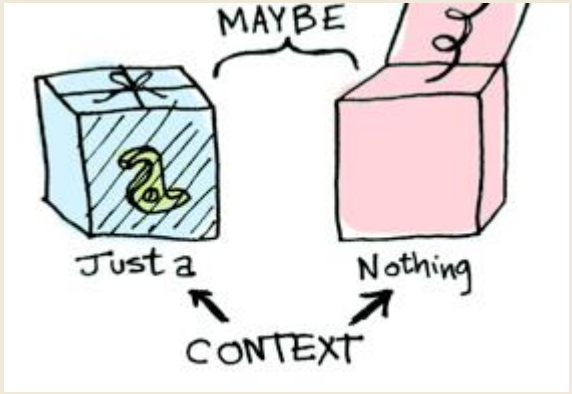


APPLICATIVES APPLY A WRAPPED FUNCTION TO A WRAPPED VALUE



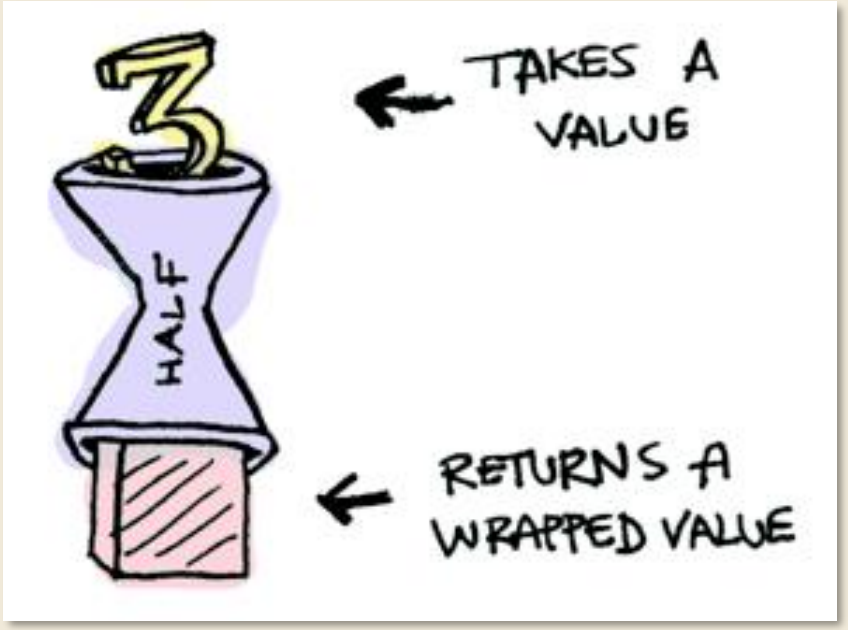
MONADS APPLY A FUNCTION THAT **RETURNS A WRAPPED VALUE TO A WRAPPED VALUE**.
MONADS HAVE A FUNCTION >>= (PRONOUNCED "BIND") TO DO THIS.

MAYBE IS A MONAD:



MONADS - EXAMPLE

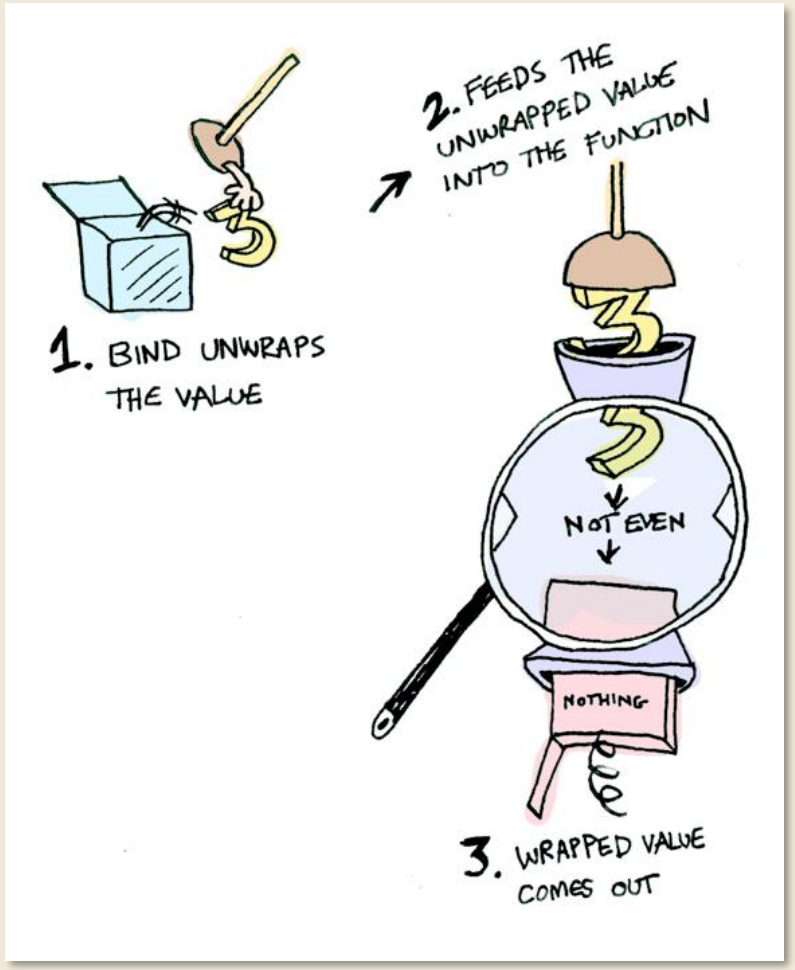
SUPPOSE **HALF** IS A FUNCTION THAT ONLY **WORKS ON EVEN NUMBERS**



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

MONADS - EXAMPLE

WHAT IF WE FEED IT A WRAPPED VALUE?



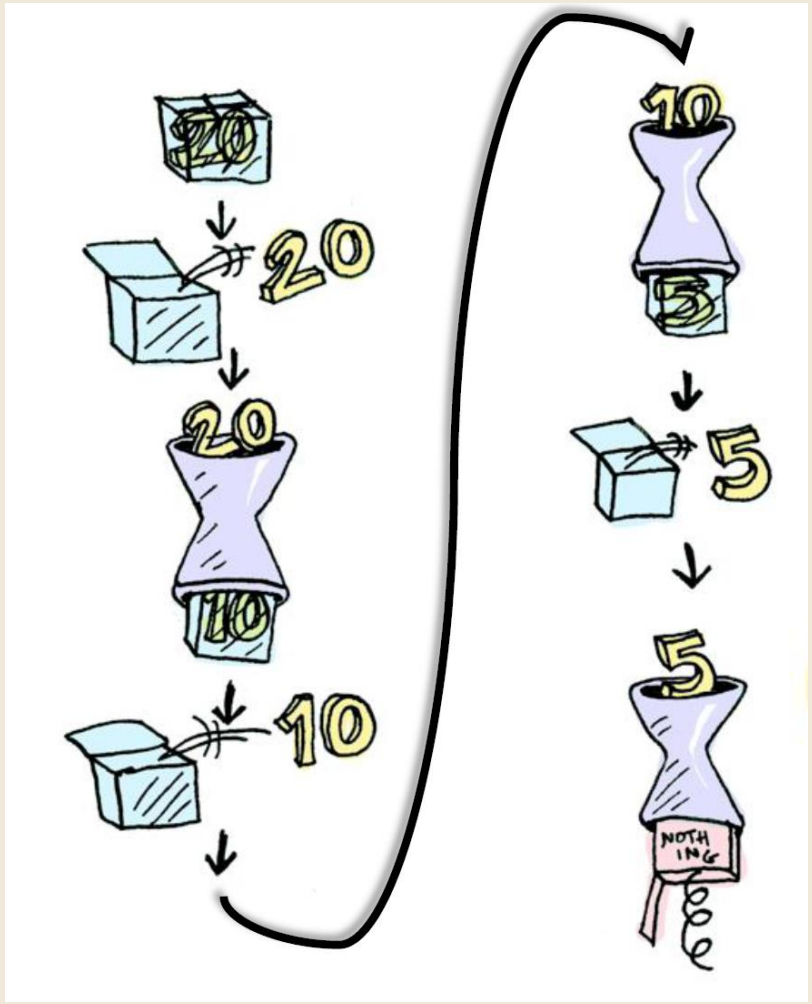
THIS IS WHERE **BIND** COMES IN!

```
Option<double>.Some(3).Bind(x => Half(x));// None
Option<double>.Some(4).Bind(x => Half(x));// Some(2)
```

IF YOU PASS IN **NONE** IT'S EVEN SIMPLER

MONADS - EXAMPLE

WE CAN CHAIN CALLS TO BIND

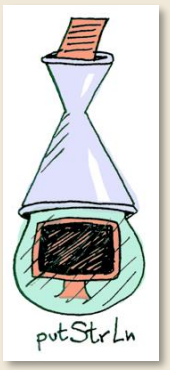



```
Option<double>.Some(20)
      .Bind(x => Half(x))// Some(10)
      .Bind(x => Half(x))// Some(5)
      .Bind(x => Half(x));// None
```

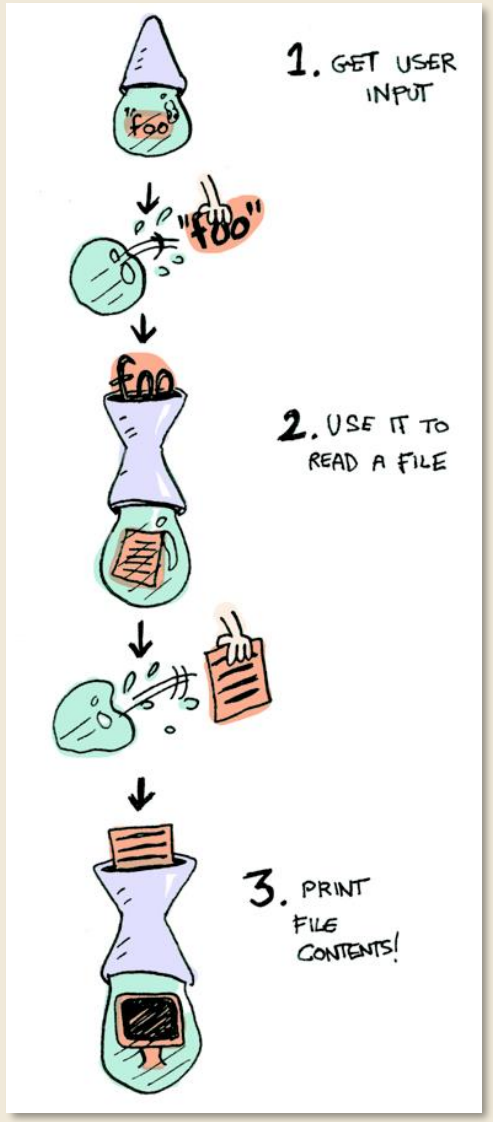
MONADS — ANOTHER EXAMPLE

USER TYPES A PATH, WE LOAD THE FILE CONTENT AND DISPLAY IT

```
1 reference | 0 changes | 0 authors, 0 changes
Option<string> GetLine()
{
    Console.Write("File:");
    return Console.ReadLine();
}

1 reference | 0 changes | 0 authors, 0 changes
Option<string> ReadFile(string filePath) => File.ReadAllText(filePath);

1 reference | 0 changes | 0 authors, 0 changes
Option<bool> PrintStrln(string line)
{
    Console.WriteLine(line);
    return true;
}
```




```
GetLine()
    .Bind(ReadFile)
    .Bind(PrintStrln);
```



MONADS — ANOTHER EXAMPLE

WHAT ABOUT EXCEPTIONS ?

```
GetLine()
    .Bind(ReadFile)
    .Bind(PrintStrln);
```

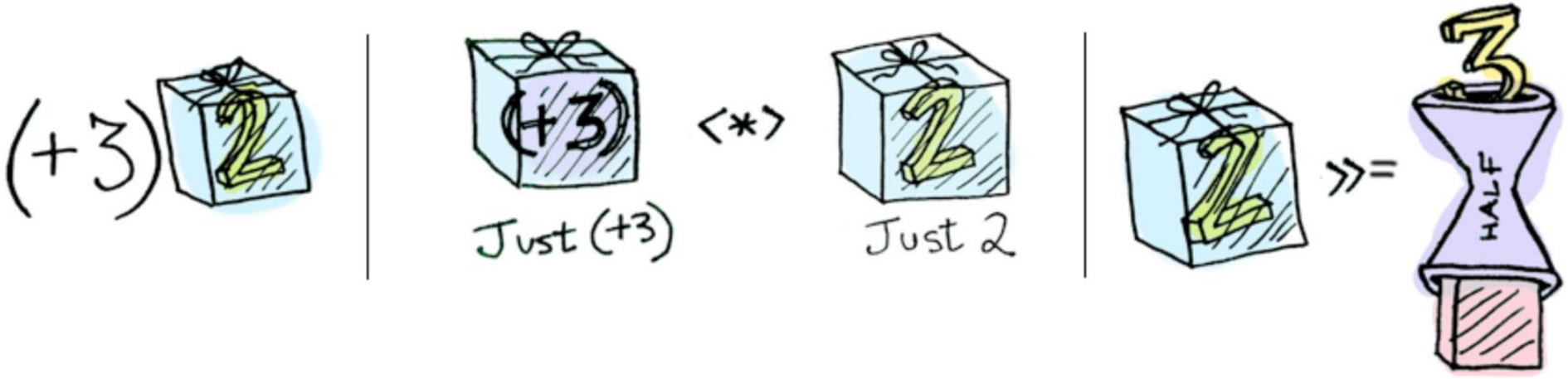
IF EXCEPTION => NONE

```
0 references | 0 changes | 0 authors, 0 changes
static void Main(string[] args)
{
    GetLine()
        .Bind(ReadFile)
        .Bind(PrintStrln)
        .Match(success => Console.WriteLine("SUCCESS"),
               failure => Console.WriteLine("FAILURE"));
}

1 reference | 0 changes | 0 authors, 0 changes
static Try<string> GetLine()
{
    Console.Write("File:");
    return Try(() => Console.ReadLine());
}

1 reference | 0 changes | 0 authors, 0 changes
static Try<string> ReadFile(string filePath) => Try(() => File.ReadAllText(filePath));

1 reference | 0 changes | 0 authors, 0 changes
static Try<bool> PrintStrln(string line)
{
    Console.WriteLine(line);
    return Try(true);
}
```

WE CAN USE TRY

USE LAMBDAS

What is the difference between the three?



Functor | Applicative | Monad

functors: you apply a function to a wrapped value using `fmap` or `<$>` MAP

applicatives: you apply a wrapped function to a wrapped value using `<*>` or `liftA`

monads: you apply a function that returns a wrapped value, to a wrapped value using `>>=` or `liftM` BIND

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);

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);

Multiply(3, 4); // 12
TwoTimes(9); // 18
```

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; }
}

GetHtml("unknown url"); // Left InvalidOperationException
GetHtml("https://www.google.com"); // Right <!doctype html...

var result = GetHtml("https://www.google.com");

result.Match(
    Left: ex => Console.WriteLine("an exception occured" + ex),
    Right: r => Console.WriteLine(r)
);
```

LANGUAGE-EXT / 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
```

FOLD VS REDUCE

- ***FOLD*** TAKES AN EXPLICIT INITIAL VALUE FOR THE ACCUMULATOR
- ***REDUCE*** USES THE FIRST ELEMENT OF THE INPUT LIST AS THE INITIAL ACCUMULATOR VALUE

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

```
var foldResult = List(1, 2, 3, 4, 5)
    .Map(x => x * 10)
    .Fold(0, (x, s) => s + x); // 150

var reduceResult = List(1, 2, 3, 4, 5)
     .Map(x => x * 10)
     .Reduce((x, s) => s + x); // 150
```

- ***REDUCE*** : THE ACCUMULATOR AND THEREFORE RESULT TYPE MUST MATCH THE LIST ELEMENT TYPE.
- ***FOLD :*** THE ACCUMULATOR AND RESULT TYPE CAN DIFFER AS THE ACCUMULATOR IS PROVIDED SEPARATELY.

```
var foldResult = List(1, 2, 3, 4, 5)
    .Map(x => x * 10)
    .Fold(0m, (x, s) => s + x); // 150m

var reduceResult = List(1, 2, 3, 4, 5)
     .Map(x => x * 10)
     .Reduce((x, s) => Convert.ToDecimal(s + x)); // Does not compile
```

And in our real life ?

REAL LIFE EXAMPLE

```
class PersonService
{
    private readonly PersonRepository personRepository;
    private readonly EmailService emailService;
    private readonly Logger logger;

    0 references | 0 changes | 0 authors, 0 changes
    public void SendEmail(long personId)
    {
        var email = personRepository.GetPersonEmail(personId);

        if(email != null)
        {
            emailService.SendWelcome(email);
            logger.LogSuccess($"Email sent for {personId}");
        }
        else
        {
            logger.LogFailure($"Email not sent for {personId}");
        }
    }
}
```

```
public string GetPersonEmail(long id)
{
    try
    {
        var person = GetById(id);

        if(person == null)
        {
            return null;
        }
        else
        {
            return person.Email;
        }
    }
    catch(Exception ex)
    {
        logger.LogFailure($"Error while retrieving : {id} {ex.Message}");
        return null;
    }
}
```

REAL LIFE EXAMPLE - HANDLING EXCEPTIONS WITH TRY EMPTY VALUES WITH OPTION

```
class PersonService
{
    private readonly PersonRepository personRepository;
    private readonly EmailService emailService;
    private readonly Logger logger;

    0 references | 0 changes | 0 authors, 0 changes
    public void SendEmail(long personId)
    {
        personRepository.GetPersonEmail(personId)
            .Match(email =>
            {
                emailService.SendWelcome(email);
                logger.LogSuccess($"Email sent for {personId}");
            },
            () => logger.LogFailure($"Email not sent for {personId}"),
            exception => logger.LogFailure($"Error for {personId} {exception}"));
    }
}
```

```
public TryOption<string> GetPersonEmail(long id)
{
    return Try(() => GetById(id))
            .Map(person => person.Email)
            .ToTryOption();
}
```

REAL LIFE EXAMPLE - CHAINING OPERATIONS: BAD ERROR HANDLING, REDUNDANT CHECKS, ...

```
public string Register(long personId)
{
    try
    {
        var person = personRepository.GetById(personId);
        if(person == null)
        {
            return null;
        }

        var account = twitterService.Register(person.Email, person.Name);
        if(account == null)
        {
            return null;
        }

        var token = twitterService.Authenticate(person.Email, person.Password);
        if(token == null)
        {
            return null;
        }

        var tweet = twitterService.Tweet(token, "Hello les cocos");

        personRepository.Update(personId, account.Id);
        logger.LogSuccess($"User {personId} registered");

        return tweet.Url;
    }
    catch(Exception ex)
    {
        logger.LogFailure($"Unable to register user : {personId} {ex.Message}");
        return null;
    }
}
```

REAL LIFE EXAMPLE - CHAINING OPERATIONS: BAD ERROR HANDLING, REDUNDANT CHECKS, ...

```
14 references | 0 changes | 0 authors, 0 changes
class Context
{
    2 references | 0 changes | 0 authors, 0 changes
    public long Id { get; }
    3 references | 0 changes | 0 authors, 0 changes
    public string Email { get; }
    2 references | 0 changes | 0 authors, 0 changes
    public string Name { get; }
    2 references | 0 changes | 0 authors, 0 changes
    public string Password { get; }
    2 references | 0 changes | 0 authors, 0 changes
    public string AccountId { get; set; }
    2 references | 0 changes | 0 authors, 0 changes
    public string Token { get; set; }
    2 references | 0 changes | 0 authors, 0 changes
    public string Url { get; set; }

    1 reference | 0 changes | 0 authors, 0 changes
    public Context(Person person)
    {
        Id = person.Id;
        Email = person.Email;
        Name = person.Name;
        Password = person.Password;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    internal Context SetAccount(Account account)
    {
        AccountId = account.Id;
        return this;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    internal Context SetToken(string token)
    {
        Token = token;
        return this;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    internal Context SetTweet(Tweet tweet)
    {
        Url = tweet.Url;
        return this;
    }
}
```

```
1 reference | 0 changes | 0 authors, 0 changes
private Try<Context> CreateContext(long personId)
{
    return Try(() => personRepository.GetById(personId))
            .Map(person => new Context(person));
}

1 reference | 0 changes | 0 authors, 0 changes
private Try<Context> RegisterTwitter(Context context)
{
    return Try(() => twitterService.Register(context.Email, context.Name))
            .Map(account => context.SetAccount(account));
}

1 reference | 0 changes | 0 authors, 0 changes
private Try<Context> Authenticate(Context context)
{
    return Try(() => twitterService.Authenticate(context.Email, context.Password))
            .Map(token => context.SetToken(token));
}

1 reference | 0 changes | 0 authors, 0 changes
private Try<Context> Tweet(Context context)
{
    return Try(() => twitterService.Tweet(context.Token, "Hello les cocos"))
            .Map(tweet => context.SetTweet(tweet));
}

1 reference | 0 changes | 0 authors, 0 changes
private Try<Context> UpdateParty(Context context)
{
    return Try(() =>
    {
        personRepository.Update(context.Id, context.AccountId);
        return context;
    });
}
```

```
public Option<string> Register(long personId)
{
    return CreateContext(personId)
            .Bind(RegisterTwitter)
            .Bind(Authenticate)
            .Bind(Tweet)
            .Bind(UpdateParty)
            .Match(context =>
            {
                logger.LogSuccess($"User {personId} registered");
                return context.Url;
            },
            exception =>
            {
                logger.LogFailure($"Unable to register user : {personId} {exception.Message}");
                return null;
            });
}
```

ASYNC ?

```
class TwitterService
{
    1 reference | 0 changes | 0 authors, 0 changes
    public async Task<Account> Register(string email, string name)
    {
        return await Task.FromResult(new Account { Id = "9" });
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public async Task<string> Authenticate(string email, string password)
    {
        return await Task.FromResult(Guid.NewGuid().ToString());
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public async Task<Tweet> Tweet(string token, string message)
    {
        return await Task.FromResult(new Tweet { Url = "anUrl" });
    }
}
```

```
1 reference | 0 changes | 0 authors, 0 changes
private TryAsync<Context> CreateContext(long personId)
{
    return TryAsync(() => personRepository.GetById(personId))
            .Map(person => new Context(person));
}

1 reference | 0 changes | 0 authors, 0 changes
private TryAsync<Context> RegisterTwitter(Context context)
{
    return TryAsync(() => twitterService.Register(context.Email, context.Name))
            .Map(account => context.SetAccount(account));
}

1 reference | 0 changes | 0 authors, 0 changes
private TryAsync<Context> Authenticate(Context context)
{
    return TryAsync(() => twitterService.Authenticate(context.Email, context.Password))
            .Map(token => context.SetToken(token));
}

1 reference | 0 changes | 0 authors, 0 changes
private TryAsync<Context> Tweet(Context context)
{
    return TryAsync(() => twitterService.Tweet(context.Token, "Hello les cocos"))
            .Map(tweet => context.SetTweet(tweet));
}

1 reference | 0 changes | 0 authors, 0 changes
private TryAsync<Context> UpdateParty(Context context)
{
    return TryAsync(async () =>
    {
        await personRepository.Update(context.Id, context.AccountId);
        return context;
    });
}
```

```
1 reference | ✓ 1/1 passing | 0 changes | 0 authors, 0 changes
public async Task<string> Register(long personId)
{
    string result = string.Empty;
    await CreateContext(personId)
            .Bind(RegisterTwitter)
            .Bind(Authenticate)
            .Bind(Tweet)
            .Bind(UpdateParty)
            .Match(
            context =>
            {
                logger.LogSuccess($"User {personId} registered");
                result = context.Url;
            },
            exception =>
            {
                logger.LogFailure($"Unable to register user : {personId} {exception.Message}");
            });

    return result;
}
```

WHAT ABOUT C# 8 ?

NULLABLE REFERENCE TYPES

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

```
0 references | 0 changes | 0 authors, 0 changes
internal class Data
{
    0 references | 0 changes | 0 authors, 0 changes
    public int Id { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string SomeNonNullValue { get; set; }
    0 references | 0 changes | 0 authors,
    public string? SomeNullableVa
}
```

```
string Data.SomeNonNullValue { get; set; }

Non-nullable property 'SomeNonNullValue' is uninitialized. Consider declaring the property as nullable.

Show potential fixes (Alt+Enter or Ctrl+;)
```

NULL COALESCING ASSIGNMENTS

```
public void SampleInCSharp7(string value)
{
    if (value == null)
    {
        value = "SpongeBob";
    }
    // Do something
}
```

```
public void SampleInCSharp8(string value)
{
    value ??= "SpongeBob";
    // Do something
}
```

READONLY MEMBERS -> PURE FUNCTIONS

PATTERN MATCHING

```
switch (entry.Name) {
  case "Bruce Wayne":
  case "Matt Eland":
     return Heroes.Batman;
  case "The Thing":
     if (entry.Source == "John Carpenter") {
       return Heroes.AntiHero;
     }
     return Heroes.ComicThing;
  case "Bruce Banner":
     return Heroes.TheHulk;
  // Many heroes omitted...
  default:
     return Heroes.NotAHero;
}
```

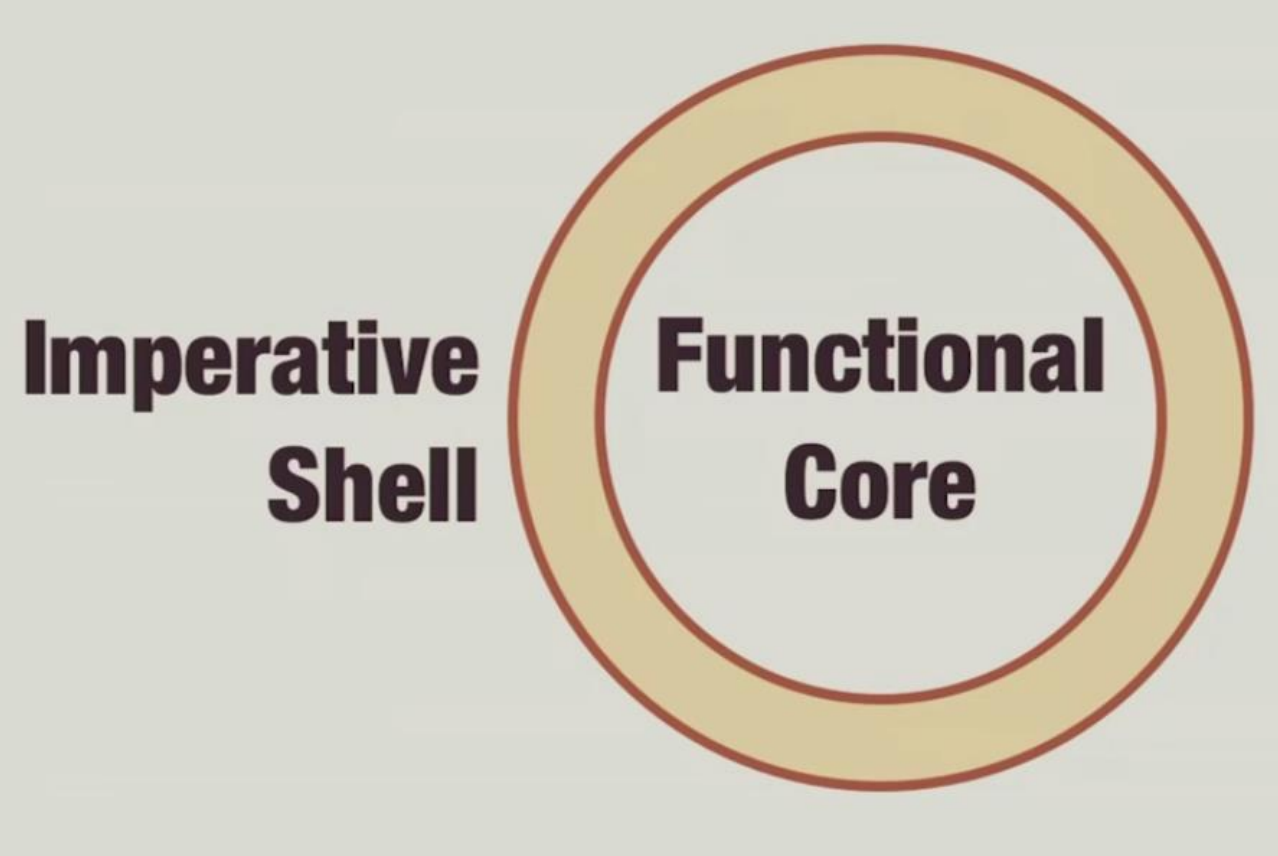
```
return entry.Name switch {
    "Bruce Wayne" => Heroes.Batman,
    "Matt Eland" => Heroes.Batman,
    "The Thing" when entry.Source == "John Carpenter" => Heroes.AntiHero,
    "The Thing" => Heroes.ComicThing,
    "Bruce Banner" => Heroes.TheHulk,
    _ => Heroes.NotAHero
};
```

PATTERN MATCHING

```
1   VehicleBase myVehicle = GetVehicle();
2   return myVehicle switch {
3       Tank { Movement = TankType.Treads } => 100000M,
4       Tank => 75000M,
5       RocketShip => 99999999M,
6       Car { Color = Colors.Red } => 21999M,
7       Car => 20000M,
8       _ => 1000M
9   };
```

WHAT ABOUT ARCHITECTURE ?

FUNCTIONAL CORE, IMPERATIVE SHELL



FUNCTIONAL CORE

- PURE FUNCTIONS : IN / OUT
- EASY TO TEST WITHOUT ANY MOCKS
  - PROPERTY BASED TESTING

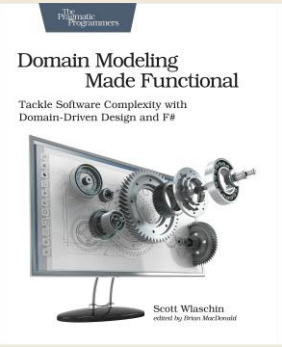
IMPERATIVE SHELL OR REACTIVE

- SERVICE LOGIC
- INTEGRATION TESTS

FIT BIEN AVEC ACTOR MODEL

Resources

- 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
- C# 8 : https://medium.com/swlh/how-c-8-helps-software-quality-cfa81a18907f
- Vidéo "Functional Core, Imperative Shell" : https://discventionstech.wordpress.com/2017/06/30/functional-core-and-imperative-shell/
- Book Domain Modeling Made Functional :



YOAN THIRION

SOFTWARE CRAFTSMAN,
AGILE ENTHUSIAST, TEAM PLAYER