Functional Programming with Language-Ext in C#

FP WHAT ?

From what you know about Functional Programming :

- *What are the benefits to use those idioms ?*

FUNCTIONAL PROGRAMMING IS ALL ABOUT FUNCTIONS

Pure functions (no side effect)

Lambda functions (anonymous)

Higher order functions

Composition

Closures
(returning functions from functions)

Currying & partial application

Pattern matching

Lazy evaluation

Immutability

Recursion

PURE FUNCTIONS

Pure functions don't refer to any global state.
The same inputs will always get the same output.

```
private static int Double(int x) => x * 2;
```

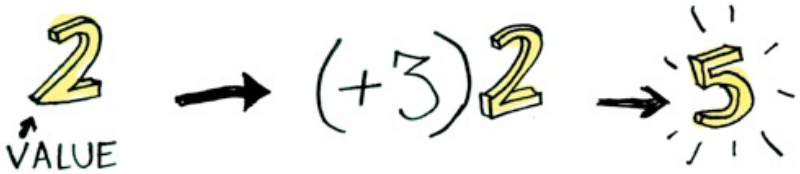
Combined with **immutable data types** this means you can be sure the same inputs will give the same outputs.

https://yoan-thirion.gitbook.io/knowledge-base/software-craftsmanship/code-katas/pure-functions

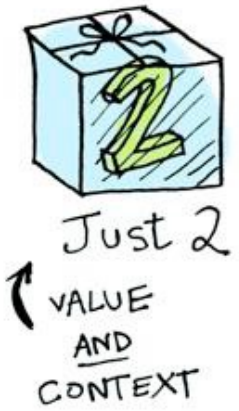
CONTEXT

Here's a simple value

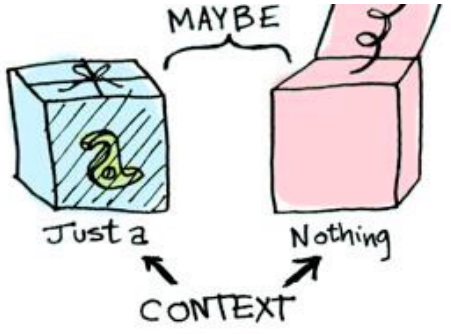
And we know how to apply a function to this value:



Let's extend this by saying that any value can be in a context. You can think of a **context as a box** that you can put a value in



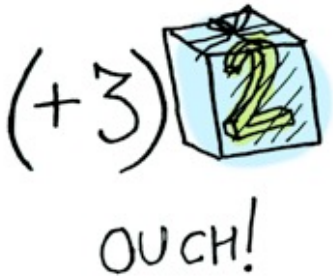
Now when you apply a function to this value, you'll get different results depending on the context.
This is the idea that Functors, Applicatives, Monads, Arrows etc are all based on.
The Maybe data type defines two related contexts



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.

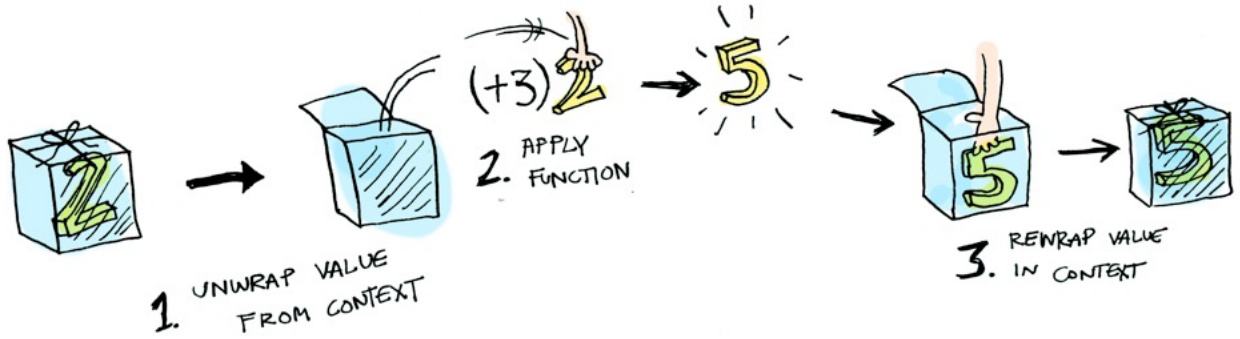

```
Some(2).Map(x => x + 3); // Some(5)
Option<int>.None.Map(x => x + 3); // None
```

A functor is any type that defines how **map** works.

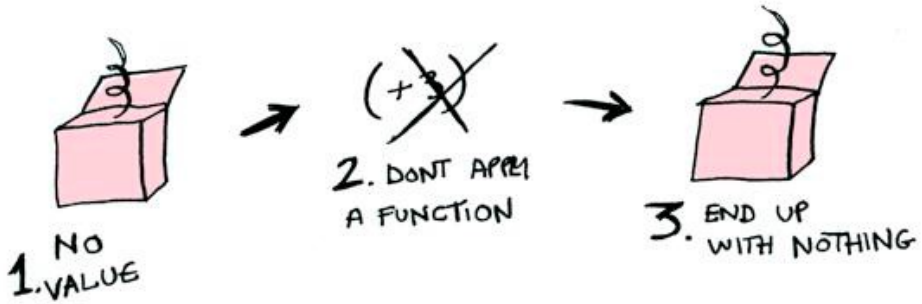
FUNCTORS

Here's what is happening behind the scenes when we write

```
Some(2).Map(x => x + 3); // Some(5)
```

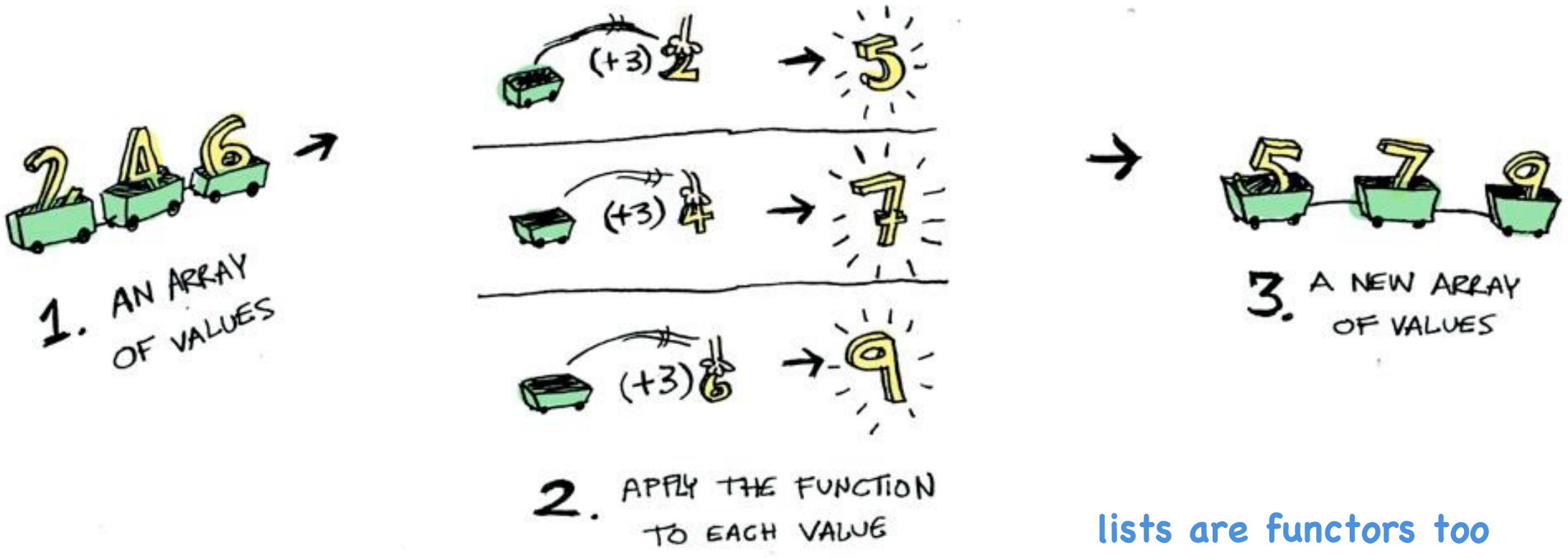


Here's what is happening behind the scenes when we try to map a function on an empty box



FUNCTORS

What happens when you apply a function to a list?

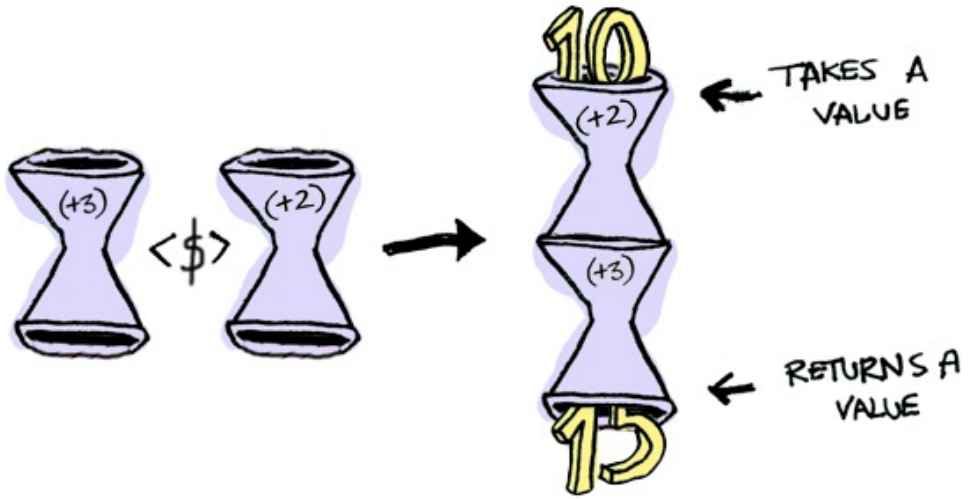


lists are functors too

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

FUNCTORS

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



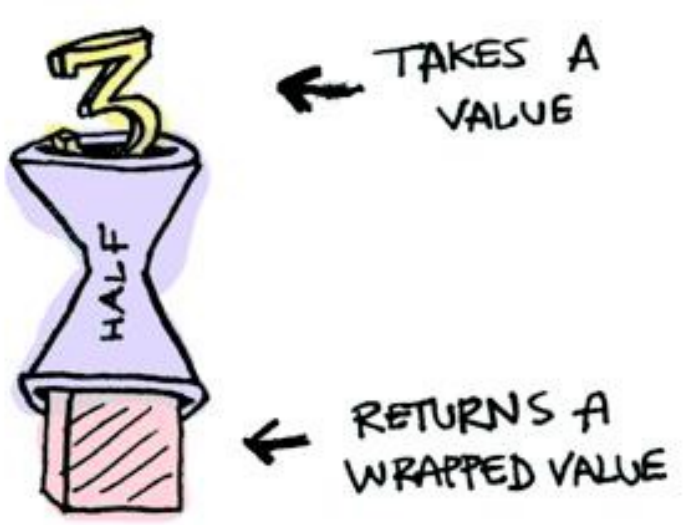
functions are also functors

In C# with **language-ext** itis called **Compose**

```
Func<int, int> add2 = x => x + 2;
Func<int, int> add3 = x => x + 3;
Func<int, int> add5 = add2.Compose(add3);
add5(10); // 15
```

MONADS

Monads apply a function that **takes a value and returns a wrapped value.**




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

Implicit cast operator

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

MONADS

What if we feed it with a wrapped value?

This is where **bind** also called **flatMap** or **chain** comes in!

1. BIND UNWRAPS THE VALUE

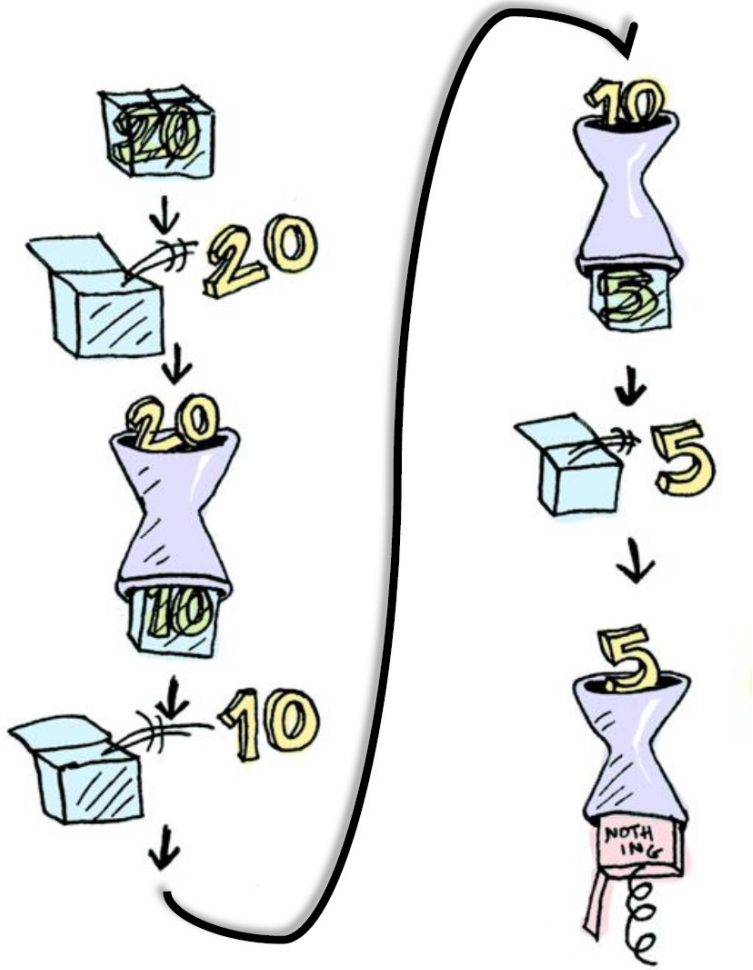

```
Some(3).Bind(Half); // None
Some(4).Bind(Half); // Some(2)
```

If you pass in **Nothing** it's even simpler

MONADS

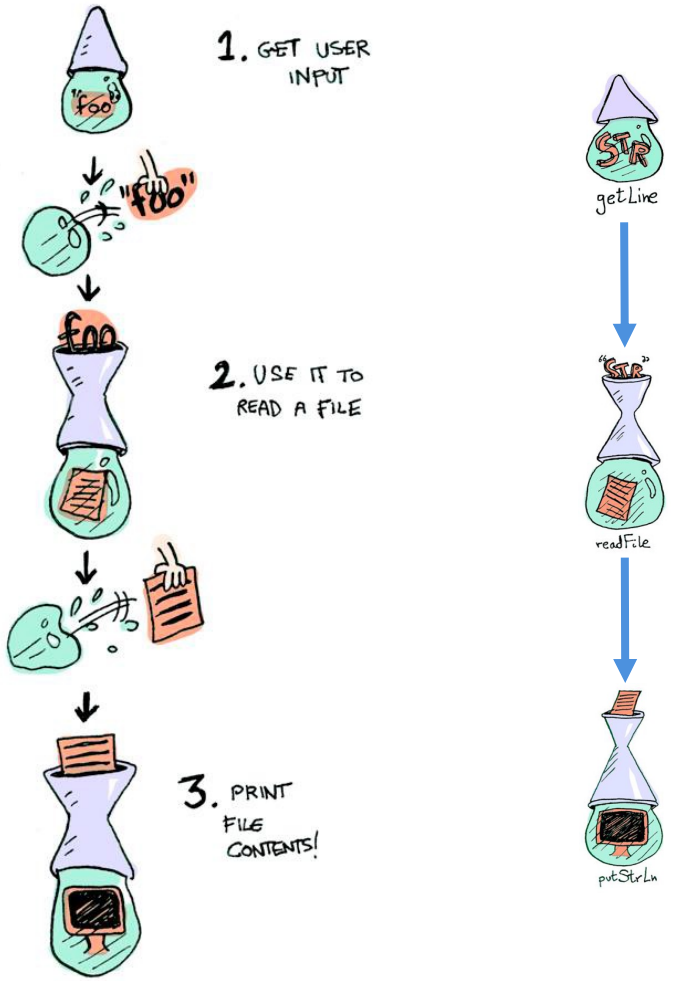
We can **chain calls** to bind

```
Some(20)
    .Bind(Half) // Some(10)
    .Bind(Half) // Some(5)
    .Bind(Half); // None
```



MONADS BY EXAMPLE

Another example: user types a path
then we want to load the file content
and display it




```
private static Try<string> GetUserInput()
{
    Console.WriteLine("File :");
    return Try(Console.ReadLine)!;
}

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

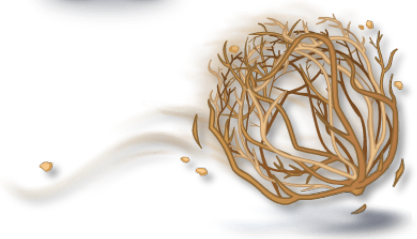
```
GetUserInput()
    .Bind(ReadFile)
    .Match(Console.WriteLine, // Success
        ex => Console.WriteLine($"FAILURE : {ex.StackTrace}")); // Failure
```

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

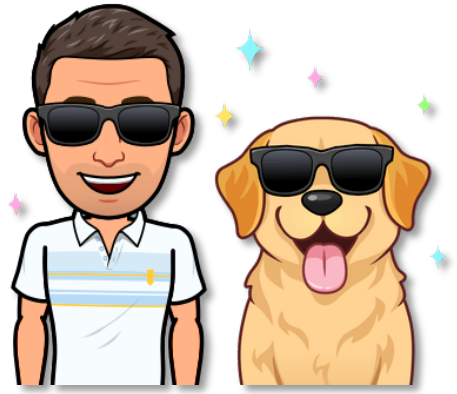
```
private string Print(
    Invoice invoice,
    Dictionary<string, Play> plays,
    Func<string, int, int, string> lineFormatter,
    Func<string, Statement, string> statementFormatter)
{
   return invoice.Performances
        .Map(performance => CreateStatement(plays, performance, lineFormatter))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



LANGUAGE-EXT

This library 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**.

The 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**.



https://github.com/louthy/language-ext

@YOI88

COLLECTIONS

Immutable collections

Location	Feature	Description
`Core`	`Arr<A>`	Immutable array
`Core`	`Seq<A>`	Immutable list with lazy behavior - a better `IEnumerable`. Very, very fast!
`Core`	`Lst<A>`	Immutable list - use `Seq` over `Lst` unless you need `InsertAt`
`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

```
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,
Select
SelectMany
Where
```

https://github.com/louthy/language-ext#immutable-collections

MONADS

Optional and alternative value monads

Location	Feature	Description
`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`	`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.

https://github.com/louthy/language-ext#optional-and-alternative-value-monads

PARTIAL APPLICATION

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

```
Func<int, int, int> multiply = (x, y) => x * y;
Func<int, int> twoTimes = par(multiply, 2);

multiply(3, 4); // 12
twoTimes(9); // 18
```

CURRYING

Currying is the same as converting a **function that takes n** arguments into **n functions taking a single argument each**.

F(x, y, z) = z(x(y))

Curried: *F(x, y, z) = F(y) { F(z) { F(x) } }*

To get the full application: *F(x)(y)(z)*

```
Func<int, int, int, int> compute = (x, y, z) => x + (y * z);
var curriedCompute = curry(compute);

compute(1, 2, 3); // 1 + (2 * 3) = 7
curriedCompute(1)(2)(3); // 7
```

Memoization

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

```
Func<string, string> generateGuidForUser = user => $"{user}:{Guid.NewGuid()}";
Func<string, string> generateGuidForUserMemoized = memo(generateGuidForUser);

generateGuidForUserMemoized("dusty"); // dusty:0b26fb2d-d371-447c-8d2d-e3e4e388d1fe
generateGuidForUserMemoized("hopson"); // hopson:10217302-2512-4777-967c-2588a74f4118
generateGuidForUserMemoized("dusty"); // dusty:0b26fb2d-d371-447c-8d2d-e3e4e388d1fe
```

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 and NPE by using Options

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
```

OPTION

Option is a monadic container with additions

Represents an optional value : None / Some(value)

```
var robots = new[] {"Tars", "Kipp", "Case"};
Func<string> randomRobot = () =>
{
    var shouldFail = random.Next(10) > 5;

    return shouldFail ?
        null :
        robots[random.Next((3))];
};
Func<string, string> upperCase = str => str.ToUpperInvariant();

Optional(randomRobot()).Map(upperCase); // Some(CASE)
Optional(randomRobot()).Map(upperCase); // None
Optional(randomRobot()).Map(upperCase); // Some(TARS)
Optional(randomRobot()).Map(upperCase); // SOME(KIPP)
Optional(randomRobot()).Map(upperCase); // None
```

Map, Bind, Filter, Do, ...

TRY

Try is a monadic container which represents a computation that **may either throw an exception or successfully completes**

```
var robots = new[] {"Tars", "Kipp", "Case"};
Func<string> randomRobot = () =>
{
    var shouldFail = random.Next(10) > 5;

    return shouldFail ?
        throw new InvalidProgramException("Plenty of slaves for my robot colony")
        : robots[random.Next((3))];
};
Func<string, string> upperCase = str => str.ToUpperInvariant();

Try(randomRobot()).Map(upperCase); // Failure -> System.InvalidProgramException: Plenty of slaves for my robot colony
Try(randomRobot()).Map(upperCase); // "KIPP"
Try(randomRobot()).Map(upperCase); // "CASE"
Try(randomRobot()).Map(upperCase); // Failure -> System.InvalidProgramException: Plenty of slaves for my robot colony
Try(randomRobot()).Map(upperCase); // Failure -> System.InvalidProgramException: Plenty of slaves for my robot colony
```

Map, Bind, Filter, Do, Match, IfFail, IfSucc, …

EITHER

Either L R Holds one of two values 'Left' or 'Right'.
Usually **'Left'** is considered **'wrong'** or 'in **error**', and **'Right'** is, well, right.

When the Either is in a Left state, it cancels computations like bind or map, etc.

```
record Account(decimal Balance);

static Either<Error, Account> Withdraw(this Account account, decimal amount)
    => amount > account.Balance ?
        Left(Error.New("Insufficient Balance")) :
        Right(account with {Balance = account.Balance - amount});


new Account(9000).Withdraw(10_000); // Left(Insufficient Balance)
new Account(100_000).Withdraw(10_000); // Right(Account { Balance = 90 000 })
```

Map, Bind, Filter, Do, Match, IfFail, IfSucc, …

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 items = List(1, 2, 3, 4, 5);
var fold = items
    .Map(x => x * 10)
    .Fold(0, (acc, x) => acc + x); // 150

var reduce = items
    .Map(x => x * 10)
    .Reduce((acc, x) => acc + x); // 150
```

- **Fold :** the accumulator and result type can differ as the accumulator is provided separately.
- **Reduce** : the accumulator and therefore result type must match the list element type.

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

var reduce = items
    .Map(x => x * 10)
    .Reduce((acc, x) => Convert.ToDecimal(acc + x)); // Do not compile
```

BEFORE WE START

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);
```

One namespace to rule them all

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

https://github.com/ythirion/language-ext-kata

@YOI88

EXERCISES

Language-ext kata

This is a hands on session on language-ext : a C# functional language extensions (base class library for functional programming)

- You will learn different concepts incrementally.
- Prerequisites is to understand functional programming paradigm.

Do the exercises in this order :

- `CollectionsExercises`
- `OptionExercises`
- `TryExercises`
- `EitherExercises`
- `PlayWithFuncExercises`
- `RealLifeExample`

REAL LIFE EXAMPLE

```
public class AccountService
{
    private readonly IBusinessLogger _businessLogger;
    private readonly TwitterService _twitterService;
    private readonly UserService _userService;

    public AccountService(
        UserService userService,
        TwitterService twitterService,
        IBusinessLogger businessLogger)
    {
        _userService = userService;
        _twitterService = twitterService;
        _businessLogger = businessLogger;
    }

    public string Register(Guid id)
    {
        try
        {
            var user = _userService.FindById(id);

            if (user == null) return null;

            var accountId = _twitterService.Register(user.Email, user.Name);

            if (accountId == null) return null;

            var twitterToken = _twitterService.Authenticate(user.Email, user.Password);

            if (twitterToken == null) return null;

            var tweetUrl = _twitterService.Tweet(twitterToken, "Hello I am " + user.Name);

            if (tweetUrl == null) return null;

            _userService.UpdateTwitterAccountId(id, accountId);
            _businessLogger.LogSuccessRegister(id);

            return tweetUrl;
        }
        catch (Exception ex)
        {
            _businessLogger.LogFailureRegister(id, ex);

            return null;
        }
    }
}
```

Step-by-step guide

Async version available in the solution-async branch

TO GO FURTHER

- Functional Core, Imperative shell
- You can play with true FP languages on JVM or .NET
  - F# on .NET
  - Clojure, Kotlin, Scala

Paul Louth
lang-ext

Daniel Dietrich
Vavr

Rich Hickey
Clojure yoda

Scott Wlaschin
F# guru

@YO188