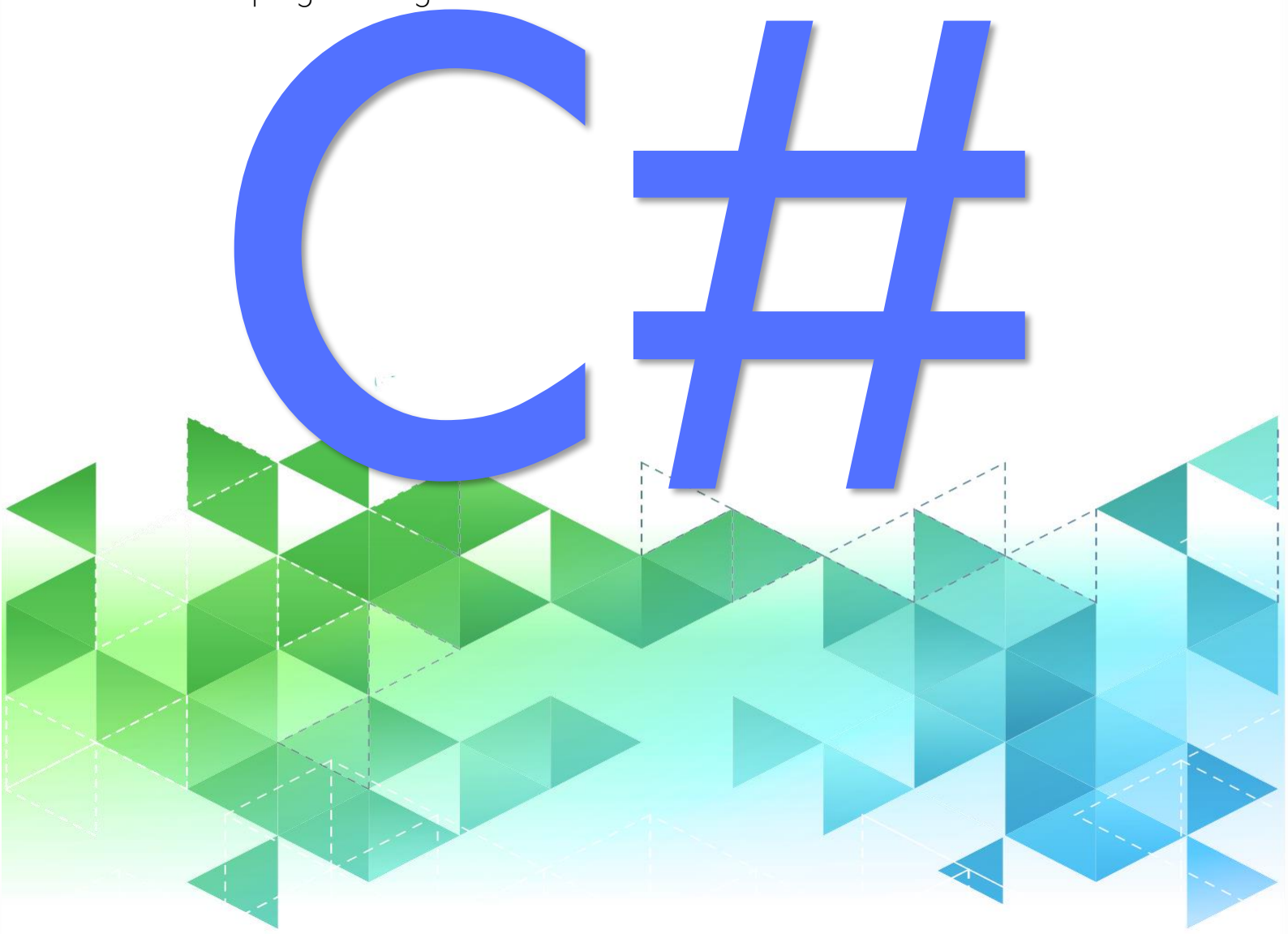
Functional Programming

In C#

{with: categories}

Gain advanced understanding of the mathematics behind modern functional programming.



dimitris papadimitriou v 2.2.1

FUNCTIONAL PROGRAMMING IN C# WITH CATEGORIES

Gain advanced understanding of the mathematics
behind modern functional programming

Dimitris Papadimitriou

Version : 2.2.1 last updated 26-Apr-2021

Feel free to contact me at:

https://www.linkedin.com/in/dimitrispapadimitriou/

https://leanpub.com/u/dimitrispapadim

https://github.com/dimitris-papadimitriou-chr

dimitrispapadim@live.com

Resources

Source Code: https://github.com/dimitris-papadimitriou-chr/FunctionalCSharpWithCategories

Also contains https://github.com/dimitris-papadimitriou-chr/Practical-Functional-CSharp

Contents

Purpose

The tools we use have a profound (and devious!) influence on
our thinking habits, and, therefore, on our thinking abilities.
— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.
—Michael Feathers (Twitter)

One of the main reasons for this book is to transfer in the community of object-oriented developers some of the ideas and advancements happening to the functional side. This book wants to be pragmatic. Unfortunately, some great ideas are coming from academia that cannot be used by the average developer in his day to day coding.

This is not an Introductory book to functional programming, even if I restate some of the most basic concepts in the light of category theory and the object-oriented paradigm. This book covers the middle ground. Nonetheless, it is written in a way that if someone pays attention and goes through the examples, he or she could understand the concepts. An introductory book will follow in Leanpub, covering more basic ideas, along with the extended version of this book covering more advanced topics.

In this book, I will try to simplify the mathematical concepts in a way that can be displayed with code. If something cannot be easily displayed with code probably will not be something that can be readily available to a developer's arsenal of techniques and patterns.

If you think a section is boring, then skip it and maybe finish it later.

One thing I admire in Software Engineers is their ability to infer things. The ability to connect the dots is what separates the exceptional developer from the good one. In some parts of the book, I might indeed have gaps or even assume things that you are not familiar with. Nonetheless, I am sure that even an inexperienced developer will connect the dots and assign meaning, as I usually do when left with unfinished

About this book coding conventions

Let's be pragmatic here, this style of coding that promotes purity should **not be an end in itself**, and we should always be able to convince ourselves to go back into an object-oriented style of coding even if it is not that pure. The important thing is to write **functional code that provides business value** to the organization and to the end-user that will eventually use it. We must be pragmatic and utilitarian when we code and abstractionists when we think about code.

This book is aiming to present the basics of functional programming using the **language-ext** library. We will try to exhibit the usage of the basic Functional types: Option, Either, Task and Validation within an ASP.NET MVC web paradigm.

There is a **WebApplicationExample** solution that has multiple simpler examples of language-ext in a .net core web application. The intention is to build up an understanding behind the architecture of the **Sample Contoso** web Application in the **language-ext** GitHub project which is examined in the last chapter of this book.

The language-ext C# library

This book is all about the **specialization** of the concepts of functional programming with the language-ext C# library. There are some functional libraries out there for C#, but currently the only complete library that could be used in a commercial project is the language-ext. In this section we are going to see a couple of the basic crosscutting ideas in the library and then in the following Chapters we will introduce all the different structures.

In this book we are going only to **use the Nu-get package LanguageExt.Core.**

C# Functional Features

An overview

"1996 - James Gosling invents Java. Java is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Sun loudly heralds Java's novelty.

2001 - Anders Hejlsberg invents C#. C# is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Microsoft loudly heralds C#'s novelty."
-*Brief, Incomplete and Mostly Wrong History of Programming Languages*,

1 Language Functional support

1.1 Pure functions and Side effects

No matter the programming paradigm the minimal unit of processing is a function **a→b**. There would not be any motion if a value could not be transformed into another value. In Object oriented programming we call them methods. the most important thing about a function is its Type signature. A **type signature** or **type annotation** defines the inputs and outputs for a function, subroutine or method. In Object oriented programming we do not think about the function types that much, whilst in functional programming the function types are the main focus.

Look for example, the following function:

```
public double GetDiscountedPrice(double price, double discount)
{
    return price * (1 - discount);
}
```

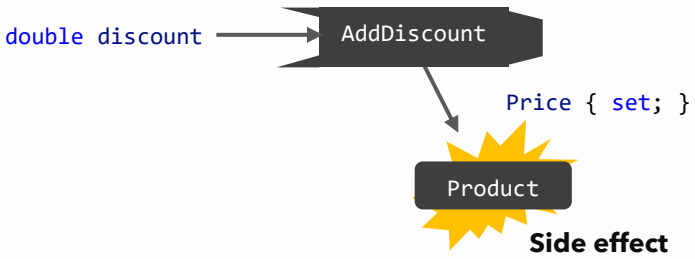
This function has a signature like this GetDiscountedPrice: (double, double) → double. This is almost a perfect function. First, it's called **total** because no matter what value you input for the `price` and `discount` it will yield a result. Does not throw exceptions or return null. If only all functions where like this one. Also, its **Pure** because it does not access any outside

variable or modifies in any way the computational environment (this means that has **no side effects**)

In object-oriented programming the importance of the Type signature is diminished by the fact that there is so much coupling going on. Let me explain. Look at the following object:

```
public class Product
{
        public double Price { get;  set; }
        public Product(double price)
        {
            Price = price;
        }
        public void AddDiscount(double discount)
        {
            this.Price = this.Price * (1 - discount);
        }
}
```

Now we placed the previous function in an object. What is the type signature of the `AddDiscount` method? is it `double`→ ( ) ? It Sure, looks like that it just takes a double and returns nothing. So why not. Well almost. The fact is that the function **access in a hidden manner** the `this.Price` so this counts as an additional input argument (`double`, `double`)→(). There is no return output that is suspicious. Because also access the `this.Price` of the Product and mutate its state.



Funtional Programming

Side effects

Side effects are operations that **change the global state of a computation**. Formally, all assignments and all input/output operations are considered side-effects. **The functional programming style tries to avoid /reduce side effects**.

First, the fact that the object exposes the Price can lead to bugs. For example, we can have this sequence:

```
var product = new Product(100);
product.AddDiscount(0.1);

//...
//somewhere else
product.Price = 80;//what now? this cancels the previous discount
```

you can even have simple multithread applications that just access the Price in a different order of what was intended. We can of course hide the Price from the outside environment (the class scope).

```
public class Product
 {
     public double Price { get; private set; }
     public Product(double price)
     {
         Price = price;
     }
     public void AddDiscount(double discount)
     {
         this.Price = this.Price * (1 - discount);
     }
 }
```

Better. Now the Product price cannot be changed directly from the outside environment.

Of course, we can just remove the setter all together and every time create a new updated product. Thus, make the `Product` **immutable**.

```
public class Product
{
    public double Price { get; }
    public Product(double price)
    {
        Price = price;
    }
    public Product GetDiscountedProduct(double discount)
    {
        return new Product(this.Price * (1 - discount));
    }
}
```

No setter

We return a `new Product`

Now we could completely extract the method from the `Product` and place it in a `PricingService`:

```
public class PricingService {
   public Product GetDiscountedProduct(Product product, double discount)
   {
```

```
        return new Product(product.Price * (1 - discount));
    }
}
var discountedProduct = new PricingService()
                            .GetDiscountedProduct(new Product(100), 0.1);
```

The type signature of the `GetDiscountedProduct` function now is

`(Product, double) → Product`

Finally, this type reflects the real intention of the Pricing computation.

1.2 Evolution of Delegates in C#

The most prominent language feature that facilitates functional programming is the existence of First-Class functions. This means the language needs to support the ability to treat functions as you would any variable and pass them around to other functions as you see fit. Lambda functions extend this concept, allowing the creation of an anonymous function, in a compressed and easy to read syntax.

Here we are going to display in rapid succession in just a section all the C# features related to lambdas. Consider this as a small reference guide. From .net 1.0 version the delegate was introduced. The delegate is a thing of the past and you probably will not see it anywhere but is behind all other concepts. A **delegate Type** is just a declaration of a function signature.

Delegates

For example, we can declare this delegate:

```
public delegate double DiscountDelegate(double price, double discount);
```

this informs us that anything that looks like the following:

```
double ___(double price, double discount)
```

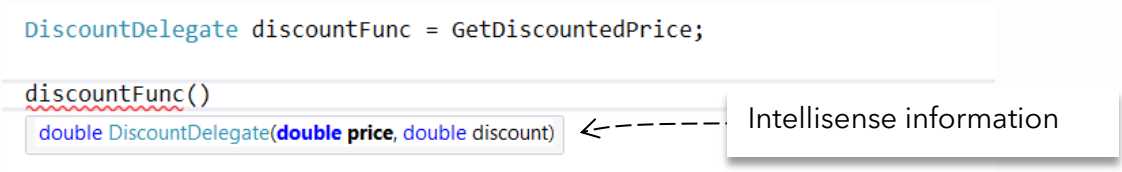
can be assigned at a variable of type `DiscountDelegate`. If we have the following function:

```
double GetDiscountedPrice(double price, double discount)
{
    return price * (1 - discount);
}
```

This is a perfectly valid declaration:

```
DiscountDelegate discountFunc = GetDiscountedPrice;
```

You will see if you try to use this `discountFunc` variable the intellisense informs you that this expects 2 arguments of double and returns a double

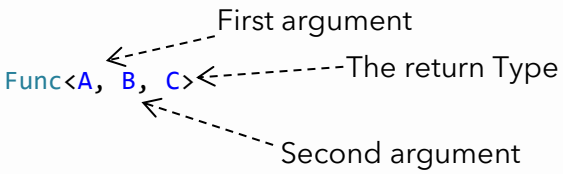


C#3.0 lambda expression

After C# 3.0 we do not have to declare delegates anymore, and we instead use the generic `Func<>`. For example, we can write the following:

```
Func<double, double,double> discountFunc = GetDiscountedPrice
```

The `Func<double, double, double>` represents a Delegate Type that has the following signature (A x, B y) → C



And we can assign a lambda expression directly to a `Func<...>`

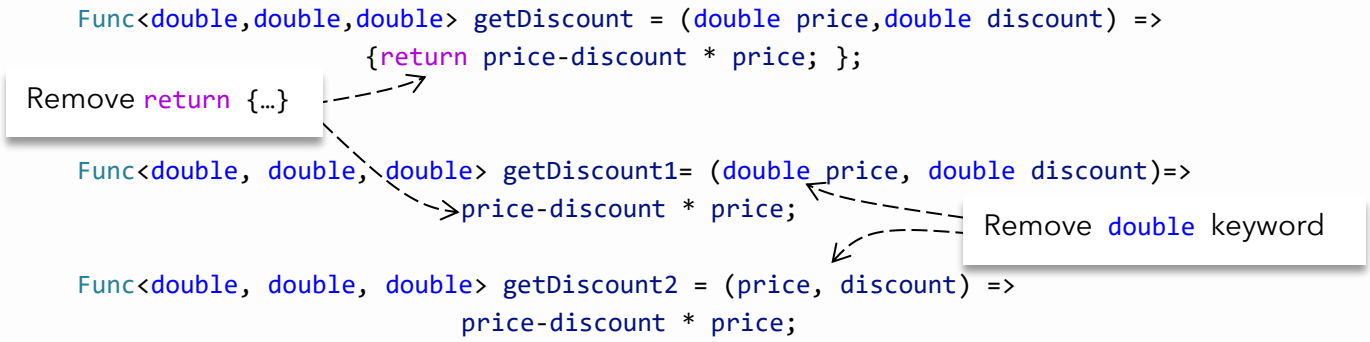
If you go to the definition you will see the framework `Func<>` implementation for different number of input arguments:

```
public delegate TResult Func<in T, out TResult>(T arg);

public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

.net source

they are just `delegate` declarations.

All the following statements are equivalent :



hopefully, you are already familiar with this notation, but in case you are not, let us go through each one by one briefly. Firstly, the following is an lambda expression :

```
(double price,double discount) =>{return price - discount * price; };
```

And is just a function. We can remove the `{return ___ };` with the resulting expression:

```
Func<double, double, double> getDiscount = (double price, double discount) =>
                                                    price-discount * price;
```

if the function returns a result immediately and we do not need multiple lines. Because in functional programming we try to chain computations usually there is no need for the braces. Finally, we can remove the types inside the input arguments:

```
(double price, double discount)=>_
```

resulting to the following:

```
Func<double, double, double> getDiscount2 = (price, discount) =>
                                                    price-discount * price;
```

Local functions

Starting with C# 7.0, C# supports local functions. With local functions you can just nest functions inside other functions, methods or even lambdas:

```
public class Demo
{
        public double GetDiscountedPrice(double price, double discount)
        {
            double getCustomerDiscount()
            {
                return discount * 0.5;
            };

            return price * (1 - discount - getCustomerDiscount());
        }
}
```

nested local function

We can nest local functions in local functions all the way down:

```
 public class Demo
    {
        public double GetDiscountedPrice(double price, double discount)
        {
            double getCustomerDiscount()
            {
                return discount * 0.5;

                double getCustomer ()
                {
                    throw new NotImplementedException();
                };
            };

            return price * (1 - discount - getCustomerDiscount());
        }
```

Double nested local function

Also, we can nest inside lambdas:

```
Func<double, double, double> getDiscount = (price,  discount) => {
  decimal getCustomerDiscount()
```

```
    {
        throw new NotImplementedException();
    };

 return price - discount * price;
};
```

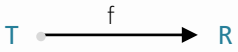
1.3 Built-in .NET Delegates

.NET contains a set of delegates designed for commonly occurring use cases, so we do not have to declare custom delegates. Those are residing in the System namespace. Here are the most important and most used :

`Func<T,R>`

The most common delegate by far will be the `Func<T, R>`, which represents functions and methods that look like this `R f(T t)` taking one argument and returning one result.

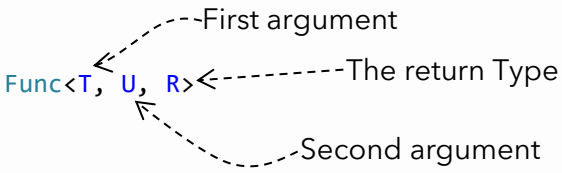
Visually this could be represented by just an arrow:



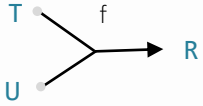
`Func<T,U,R>`

Represents a function Type that has the following signature `R f(T t1, U t2)`.

The `Func<T,U,R>` represents a Delegate Type that has the following signature **(T , U) → R**



If you wanted to visualize this could be a join:



Action<T>

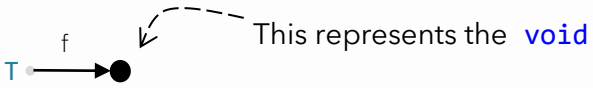
The `Action<T>` Functional interface represents functions that have this signature `void f(T t)`. In functional programming we usually do not like the `void` because it is not a Type like all the others, and this forces us to treat it differently. `Action<T>` is just a `Func<T, void>` but because `void` is not a Type .NET had to invent `Action<T>`. As an example of usage look at the following, where we "consume" a string returning nothing:

```
Action<string> print = (string x) => Console.WriteLine(x);
```

When you return `void` is an indication that you create a side effect of some sort. We can shorten the declaration, using the point free notation :

```
Action<string> print1 = Console.Write;
```

A visualization for `Action<T>` could be:

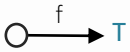


Func<T>

The `Func<T>` represents functions that have this signature `T f()` that do not take any arguments. A simple example would be:

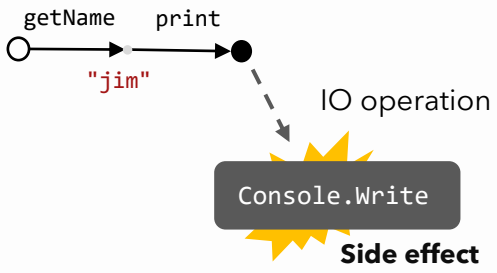
```
Func<string> getName =()=>"jim";
```

visualize this as a value coming out of nothing:



`Func<T>` is the dual of `Action<T>` and in this way we can compose them and annihilate them into a `void`.

```
Action<string> print = Console.Write;
Func<string> getName =()=>"jim";

print(getName());              // prints Jim as side effect with return type void
```



`Predicate<T>`

The `Predicate<T>` Functional interface represents functions that have this signature `bool f(T t1)` that returns a `bool`. This is equivalent to a `Func<T, bool>`.

Summary Table

`Action<T1>`	`void f(T1 a){…}`
`Action<T1,T2>`	`void f(T1 arg1, T2 arg2){…}`
`Func<TResult>`	`TResult f(){…}`
`Func<T1,TResult>`	`TResult f(T1 arg1){…}`
`Func<T1,T2,TResult>`	`TResult f(T1 arg1, T2 arg2){…}`
`Func<T1,T2,…T16,TResult>`	`TResult f(T1 arg1, T2 arg2,…,T16 arg16){…}`
`Predicate<T>`	`bool f(T arg1){…}`

Remember :

1. The `Action` delegates only have input arguments.
2. The `Func<_,,,_>` is the general case.
3. For Any `Func<_,,,TResult>` the Result is always at the Last position.
4. If you only want to return a result `()=>…` you must use a `Func<TResult>`.

.NET Conventions — ?.Invoke(…)

Since we can declare `Func` delegates as variables, it is possible that the delegate variable has not been assigned at the time of Invocation:

```
Func<string> getName = null;
var name = getName();        // we get a Null Reference Exception
```

in order to avoid this situation we usually use the ?`.Invoke()` method on the Delegates

```
getName?.Invoke();          //because of the ? operator we dont have an Exception.
```

1.4 Essential Typed Lambda calculus and C#

Lambda calculus is the original functional language. Lambda calculus is a formal system in mathematical logic, that was used to express computations. It was created by the great Alonzo Church in the 1930s.

In this section we are going to see the **simply typed lambda calculus** (the usual symbol is $\lambda^{\rightarrow}$) and how this relates to the C# lambdas. In order to do that we will keep only for this section a subset of C# equivalent to the $\lambda^{\rightarrow}$. Simple typed lambda calculus is a very simple system that does not support Generics but allows for simple types.

Terms

The formulation of the lambda expressions is constructed bottom up-in an inductive manner using as its base the term **Term**. A **Term** can be one of three things:

1. A **Variable**: **x** is a **Term**
2. An **Abstraction**: **λx:A.M** is a **Term**, where **M** is a **Term** and x is a variable.
3. An **Application**: (**M N**) is a **Term**, where **M** and **N** are **Terms**
4. A **Constant** **c** is a **Term**

From those the peculiar ones are the (2), and (3) which correspond in C# loosely to the following (we are going to elaborate on this):

1. The lambda abstraction **λx:A.M** corresponds to a C# function
   `Func<A, B> f = (A x) => ...`

2. And the application (**M N**) corresponds to the trivial function call `f(x)`

Of course, only with those expressions you cannot expect to do any meaningful computation because you must introduce syntax related with the Natural Numbers (+, -, …), Booleans (if, then…). Nonetheless, one could go down the road of defining Numerals using the Church encoding (we will skip this part in this book) which are typable in $\lambda^{\rightarrow}$ using only one type:

```
Func<Func<_, _>, Func<_, _>> zero = (Func<_, _> f) => (_ x) => x;
Func<Func<_, _>, Func<_, _>> one = (Func<_, _> f) => (_ x) => f(x);
Func<Func<_, _>, Func<_, _>> two = (Func<_, _> f) => (_ x) => f(f(x));
```

For some single arbitrary empty type `class _ { }`.

Church encoding

In mathematics the **Church numerals** are a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

0 ≅ λf. λf. x
1 ≅ λf. λf. f(x)
2 ≅ λf. λf. f(f(x))

The type of those numerals is (_→_) →(_→_), which we can define as delegate for simplicity:

```
delegate Func<_, _> N (Func<_, _> f);
```

Then you can define things like the Successor function, addition etc.

```
Func<N, N> Succ = (N n) => (f) => x => f(n(f)(x)); // add an additional f in front
```

This is just a Hacky way to do arithmetic, imagined by the genius of Alonzo Church during the 30's, which constitute the ancient history of computer science.

Now, in the untyped lambda calculus we could have a lambda application (**M N**) of any two Terms. This led to problems early on because it allowed Russel type paradoxes [Curry's paradox]. That is why **Types** were introduced to restrict the operations of Terms based on their type.

This idea of types is considered today common knowledge, in all strongly typed, languages like C#.

Types

The only types in $\lambda^{\rightarrow}$ is the Set of some base types **B**, and anything whatever is produced by this simple grammar in BNF form:

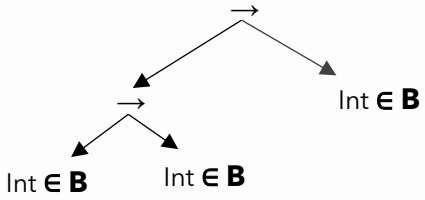
τ ::= τ→ τ | T where **T ∈ B**

this means the only types that occur in $\lambda^{\rightarrow}$ are inductively defined and are either of the form τ→ τ where τ is some already created type, or some type belonging to the base types. If for example **B** = {σ, ρ} then the types generated look like this :

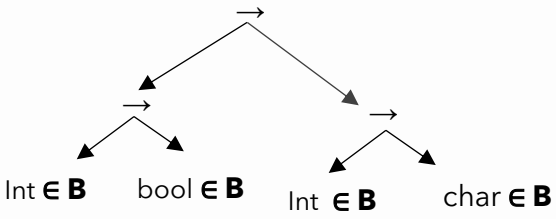
σ, ρ, σ→ σ, ρ→ρ, ρ→ σ, (ρ→ σ) → ρ, …, (ρ→ σ) →(ρ→ σ) …

If we take the C# type system with the usual primitive build-in types then a simple $\lambda^{\rightarrow}$ restricted version of C# would have as base types **B=** {**int**, **char**, **bool**, …} and everything that would be generated using the **τ ::= τ→ τ | T** grammar. This would result in syntax trees like the following:

(int→ int) → int



(int→ bool) → (int → char)



The (**int→ int**) → **int** in C# is the delegate `Func<Func<int,int>, int>` while the second one (**int→ bool**) → (**int** → **char**) would be `Func<Func<int,bool>, Func<int,char>>`

As one can see an embedded $\lambda^{\rightarrow}$ version within C# would only allow for things that look like this `Func<Func<... , ...>, Func<..., ...>>` with nested `Func` delegates.

Introduction and Elimination Rules

Introduction Rule

Now we have used an inductive definition to create a **Set of Types** and used an inductive definition to define a **Set of Terms**, the next step is to link them. We have to give an Inductive definition of how specific Terms belong to specific Types.

First there is the introduction rule which signifies the way we construct a lambda abstraction.

$$\frac{\begin{array}{c}[x{:}A]^1\\ \vdots \\ M{:}B\end{array}}{\lambda x.M{:}\ A \rightarrow B}\ \Rightarrow I_1$$

Here you see the mechanism to create a **λx:A.M** Term and why this Term is of type **A →B** and that is the only way to create **τ→ τ** types. The [**x**: **A**]^1 means that at some point we make an assumption of having a variable **x** of type **A**, or in another phrasing we have a context where we have a variable **x** of type **A.** In C# this is expressed by the declaration:

```
(A x) => ...
```

By starting a lambda, you might assume that you introduce a `x:A` out of thin air. Then in this setting we finally construct a term **M**:**B** we can **discharge** the assumption and introduce this implication sign **⇒** (equivalently the **=>** in C#) and this gives us the

```
Func<A, B> Id = (A x) => ... ;
```

We note the introduction with an arrow ⇒ I_1

$$\frac{\begin{array}{c}[x:A]^1\\ \vdots\\ \ldots\end{array}}{\ldots}\ \Rightarrow I_1$$

[Another notation for the Introduction using the entailment sign ⊢ is

$$\frac{\Gamma,\quad x:A \vdash M:B}{\Gamma \vdash \lambda x.M:A \to B}\ \Rightarrow E$$

This meant to express the same thing as the previous notation. That in a Typing context

Γ, **x**:**A** ⊢ …. (the context is comprised by the assumptions that are placed left of the ⊢) we produce a term **M**:**B** and thus we infer that in the Typing context without the x:A, we can infer a lambda Term **λx:A.M** : **A** →**B**]

Elimination Rule

The elimination rule for the lambda is equivalent to the very famous modus ponens in logic

$$\frac{M:A \to B \qquad N:A}{(MN):B}\ \Rightarrow E$$

This says you can use the application (**M N**) only with two terms that have this specific kind of types **A** →**B** and **A** and this will yield a new term (**M N**) of type **B** .

Of course, this does not explain how exactly that would happen, and for this reason we need the last type of set of rules the Computation rules.

Computation Rules

The computation rule for the Application is just what is called **β-reduction**

(**λx**:**A**. **M**)(**a**) → **M**[**a**/**x**]

The **M**[**a/x**] is the representation of substitution in type theory and it means wherever we find the variable **x** inside **M** we should replace it with **a**.

This is the standard notion of executing a function if we have the following f

```
Func<int, int> f = (int x) => x + 2;
```

And we want to compute the `f(2)` we replace **x** with 2 inside `f`.

That was a presentation of the Simply typed lambda calculus system.

Now in order to use the integer types, formally one should declare how they behave by declaring the formation, introduction, elimination, and computation rules [we would have to introduce Gödel's System T (which we briefly see in the Recursion section)]. But here we will just provide a mix of typing rules and operational semantics in order to make it work with the $\lambda^{\to}$

Assuming only addition for example we could have a rule that says we only can add integers

$$\frac{\mathbf{a}:\mathbf{int} \qquad \mathbf{b}:\mathbf{int}}{(\mathbf{a}+\mathbf{b}):\mathbf{int}}$$

And another computation rule that will say that if you have an addition **(a+b): int** you can infer the addition result which is an **int**

$$\frac{(\mathbf{a}+\mathbf{b}):\mathbf{int}}{\mathbf{n}:\mathbf{int}} \quad where\ \boldsymbol{n}=\boldsymbol{a}+\boldsymbol{b}$$

Now we can do simple derivations like the following:

$$\frac{\dfrac{[\mathbf{x}:\mathbf{int}]^{\mathbf{1}} \qquad \mathbf{2}:\mathbf{int}}{\boldsymbol{x}+\mathbf{2}:\boldsymbol{int}}}{\dfrac{\boldsymbol{\lambda x}:\boldsymbol{int}.\,(\boldsymbol{x}+\mathbf{2}):\boldsymbol{int}\rightarrow\mathbf{int}}{(\boldsymbol{\lambda x}:\boldsymbol{int}.\,(\boldsymbol{x}+\mathbf{2}))(\mathbf{2}):\boldsymbol{int}}\Rightarrow \mathrm{E}}\Rightarrow I_1$$

optional

Curry–Howard correspondence

In programming language theory and proof theory, the **Curry–Howard correspondence** is the direct relationship between computer programs and mathematical proofs. We are going to explore some of those connections as we go along. We are going to only scratch the surface of the **Curry–Howard correspondence**, in a simplistic presentation within the scope of this book.

The First clear connection is between the lambda abstraction (⇒) and logical implication (→):

	Types	**Logic**
Introduction	$\dfrac{\begin{array}{c}[\mathbf{x}:\mathbf{A}]^{\mathbf{1}}\\ \vdots\\ \mathbf{M}:\mathbf{B}\end{array}}{\boldsymbol{\lambda}\mathbf{x}.\mathbf{M}:\mathbf{A}\rightarrow\mathbf{B}}\Rightarrow I_1$	$\dfrac{\begin{array}{c}[\mathbf{A}]^{\mathbf{x}}\\ \vdots\\ \mathbf{B}\end{array}}{\mathbf{A}\rightarrow\mathbf{B}}\rightarrow I_1$
Elimination	$\dfrac{\boldsymbol{\lambda}\mathbf{x}.\,\mathbf{M}:\mathbf{A}\rightarrow\mathbf{B} \qquad \boldsymbol{t}:\boldsymbol{A}}{\mathbf{M}(\mathbf{t}):\mathbf{B}}\Rightarrow E$	$\dfrac{\mathbf{A}\rightarrow\mathbf{B} \qquad \boldsymbol{A}}{\mathbf{B}}\rightarrow E$

1. an **introduction** rule describing how elements of the type can be **created**, and
2. an **elimination** rule describing how elements of the type can be **used**.

The Correspondence is the following:

1. An implication **introduction** in Logic is like a Lambda **abstraction** in Lambda calculus.
2. An Implication **elimination** in Logic is like a Lambda **application** in Lambda calculus.

The easiest thing that we can prove is the identity, `Func<A, A> Id = (A x) => x;`

$$\text{Logic} \qquad \frac{[\mathbf{A}]^x}{\mathbf{A} \to \mathbf{A}} \to I_x$$

$$\text{Types} \qquad \frac{[\mathbf{x}\colon \mathbf{A}]}{\boldsymbol{\lambda x}.\, \boldsymbol{x}\colon \mathbf{A} \to \mathbf{A}} \Rightarrow I_x$$

The natural deduction derivation is straightforward:

1. [**A**]$^{\mathbf{x}}$: We make an **assumption** [**A**]$^{\mathbf{x}}$ that we have an **A**. In the code this means that we start the lambda `(A x)=>` in this way we are charged with an **A**, that we can use in our function implementation. The problem is that when we will need to use the **Id** we should have an **A** available. The notation of **[**…**]**$^{\mathbf{x}}$ is used within the context of deduction as a book-keeping mechanism to keep track of the assumptions.

2. And then immediately conclude **A** `(A x) => x` and **eliminate the assumption** (discharge the assumption) **[A]**$^{\mathbf{x}}$ by invoking the implication → introduction rule.

Another easy tautology is the proposition **A** → (**B** → **A**) [this is called **K** combinator] with the following straight forward implementation in C# using only lambdas:

```
Func<A, Func<B, A>> K = (A x) => ((B y) => x);
```

This corresponds to the lambda term:

λx:**A. λy**:**B**.**x**: (**A** →(**B**→**A**))

K just takes a variable **x**:**A** this returns a function which takes a **y**:**B** completely ignores it and returns the initial variable. Yes, as you might expect it is completely useless in real situations.

For a complete presentation of the combinators we are also going to briefly discuss the **S** combinator.

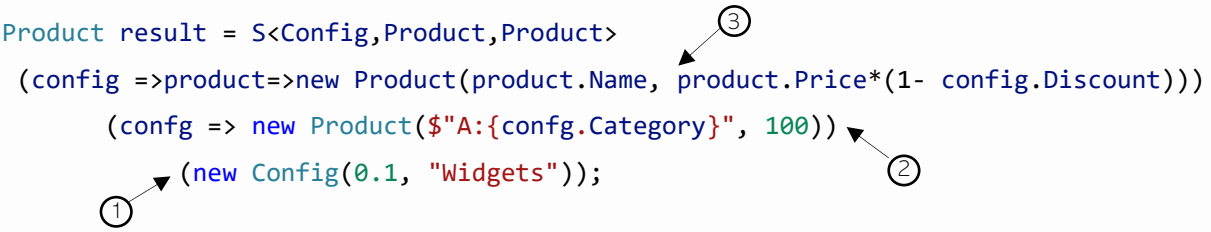
```
Func<Func<A, B>, Func<A, C>> S<A, B, C>(Func<A, Func<B, C>> f) =>
                  (Func<A, B> g) => ((A a) => f(a)(g(a)));
```

The type of this function is :

S: **(A** → (**B** → **C**)) → ((**A** → **B**) → (**A** → **C**))

And it is an interesting exercise to try and implement **S** in C# because it will give you better understanding of `Func` delegates.

S, takes x and applies it to both f and g and then applies the **g(x)** onto **f(x)**. We will find this in the applicative reader functor section. The **x** should be understood like a common environment of variables (or a configuration) that is provided to both f and g, and then we apply the result of **g(x)** to the **f(x)**, which is a function. An example would be the following:



Run This: .Net Fiddle

We used the Combinator to pass (1) the `Config(0.1, "Widgets")` to the first lambda (2) and create a new `Product` and then we passed both the configuration and the `Product` to the (3) lambda to create a discounted `Product` using the `Config` Discount.

η-reduction / expansion – Point Free notation

Let us say we have a simple lambda:

```
Func<int, int> f = (int a) => a + 1;
```

We can use this directly in any method that accepts a `Func<int, int>` delegate:

```
var x = new List<int> { 1, 2, 3 }.Select(f);
```

but we can also create a new lambda that explicitly call the function:

```
var y = new List<int> { 1, 2, 3 }.Select(x => f(x));
```

this is an **η-expansion (eta-expansion)**. It is obvious that **f** ≡ (**x** => **f**(**x**)) , that is the function **f** is equivalent to the lambda **x** => **f**(**x**). We can also keep doing this process indefinitely for example we could define the following lambdas:

```
Func<int, int> f1 = (int x) => f(x);
Func<int, int> f2 = (int x) => f1(x);
...
```

Run This: .Net Fiddle

Each of which can be used in place of the initial `f` e.g.:

```
var z = new List<int> { 1, 2, 3 }.Select(f1);
```

when we move in the opposite direction by simplifying a **x** => **f**(**x**) into just **f**, we make what is called an **η-reduction (eta- reduction)**

```
new List<int> { 1, 2, 3 }.Select(x => f(x));
```

η-reduction

```
new List<int> { 1, 2, 3 }.Select(f);
```

this syntax is called point-free because you don't have explicit reference to the object **x** =>

That is called the "point"

```
new List<int> { 1, 2, 3 }.Select(x => f(x));
```

The formal justification from Gentzen is that the introduction for lambda `(int x)=>` is (post)-inverse to the lambda elimination `f(x)`

→ E

```
Func<int, int> f1 = (int x) => f(x);
```

→ I

In the logical form we just conclude **B** from **A** and (**A** →**B**) and then immediately introduce the assumption of **A** getting back the (**A** →**B**)

$$\frac{\dfrac{\mathbf{A}\to\mathbf{B} \qquad [A]^{x}}{\mathbf{B}}\ \to \mathrm{E}}{\mathbf{A}\to\mathbf{B}}\ \to \mathrm{I}_{x}$$

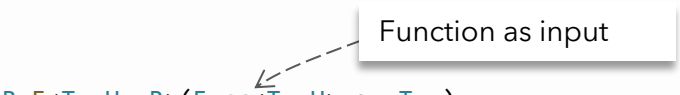
or in type theoretic derivation

$$\frac{\dfrac{[x\colon \mathbf{A}]^{1} \qquad \mathbf{f}\colon A\to\mathbf{B}}{f(x)\colon \mathbf{B}}\ \Rightarrow \mathrm{E}}{\lambda x\colon A.\,\big(f(x)\big)\colon A\to\mathbf{B}}\ \Rightarrow \mathrm{I}_{1}$$

1.5 Higher-order functions

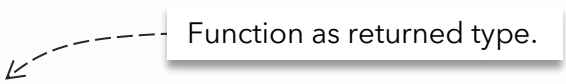
A **Higher-Order function (Hof) is a function that receives a function as an argument or returns a function as output**. But I assume that you are already familiar with this definition.

The following is a Hof because it takes a `Func<T, U>` delegate as an argument.




```
R F<T, U, R>(Func<T, U> g, T y)
{
      ...
}
```

The type of this function is **F:** ((**T**→**U**) ×**T**) →**R.** While the following is a Hof because it returns a `Func<T, U>`




```
Func<T, U> G<T, U>(T y)
{
      ...
}
```

With a type of **G: T**→(**T**→**U**) , we can also use the local function form inside other methods or functions.

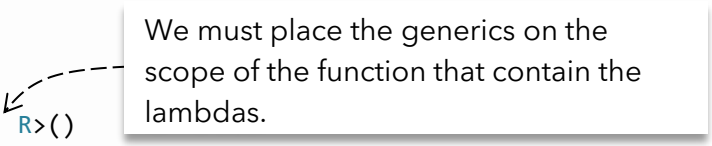
In the form of `Func` delegates the previous functions now should be declared like this:

```
Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...

Func<T, Func<T, U>> G = (T y) => ...
```

unfortunately, C# does not allow for Generic definitions at the level of the delegate. We cannot write for example :

```
Func<Func<T, U>, T, R> F = <T, U, R> (Func<T, U> g, T y) => ...

Func<T, Func<T, U>> G = <T, U>(T y) => ...
```

For this reason, we must place them in a Function or Class, and include the generic constrains there:




```
public static void Foo<T, U, R>()
{
      Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...
```

```
        Func<T, Func<T, U>> G = (T y) => ...
}
```

Again, here the type of `Func<Func<T, U>, T, R>` corresponds to ((**T**→**U**) × **T**) →**R** and the type of `Func<T, Func<T, U>>` corresponds to **T**→(**T**→**U**)

Obviously, we can have as many levels of Higher order Functions by allowing the Functions passed as arguments or return as results being themselves of Higher type. For example:

This is a Hof passed as an argument.

```
R F<T, U, R>(Func<Func<T, U>,U> g, T y)
{
        ...
}
```

With type **F:** (((**T**→**U**) →**U**) × **T**) →**R**. In practice this type of Higher-Higher-order-functions becomes too convoluted too fast, and it is not recommended. In a delegate variable declaration this already looks way to complex.

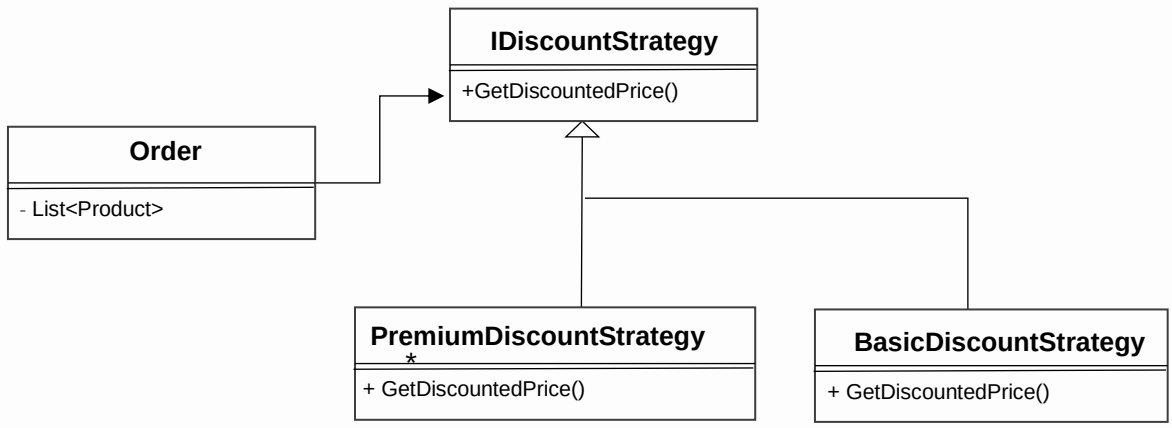
```
Func<Func<Func<T, U>, U>, T, R> F = (Func<Func<T, U>, U> g, T y) => ...
```

What makes it worse is that we cannot replace the variable type with the `var` keyword.

Strategy design pattern

The strategy design pattern is the closest thing to Higher order function form a purely OOP perspective since it allows to pass around functions as Objects. With the strategy pattern, we can modify the behavior of an object based on the different strategy object that we pass to it.

If for example, we have two different discounting scenarios for our product, we can abstract the discounting calculation from the product object and place it in a strategy object.



```
public interface IDiscountStrategy {
	double GetDiscountedPrice(Product product);
}

public class PremiumDiscountStrategy : IDiscountStrategy {
	public double GetDiscountedPrice(Product product) => product.Price * 0.7;
}

public class BasicDiscountStrategy : IDiscountStrategy {
	public double GetDiscountedPrice(Product product) => product.Price * 0.9;
}
```

If we have defined an `Order` class, responsible for handling a collection of `Product` items, we can place there the calculation of the Total amount, by using a `IDiscountStrategy` that might vary:

```
public class Order
{
	IList<Product> Products { get; }

	public Order(IList<Product> products) => Products = products;

	public double GetTotalPrice(IDiscountStrategy activeDiscount) =>
		Products
		  .Select(activeDiscount.GetDiscountedPrice)
		  .Sum();
}
```

In his way if we have an order containing some Products, we can instruct how the discount will be computed by providing **at run time** a concrete class implementing `IDiscountStrategy`

```
var total = order.GetTotalPrice(new PremiumDiscountStrategy());
```

In a functional manner we could refactor this to directly pass the discount calculation function `Func<Product, double>` as an argument. We can thus define a `GetTotalPrice` function:

```
double GetTotalPrice(List<Product> products,Func<Product,double> activeDiscount)=>
    products.Select(activeDiscount).Sum();
```

and define some predetermined Discounts:

```
Func<Product,double> PremiumDiscountStrategy =(Product product)=>
                                                    product.Price* 0.7;

Func<Product, double> BasicDiscountStrategy = (Product product) =>
                                                    product.Price * 0.9;
```

We could even define the `GetDiscountedPrice` function, as an extension method attached on the `List<Product>` type:

```
public static partial class FunctionalExtensions
{
    public static double GetDiscountedPrice(this List<Product> @products,
                                Func<Product, double> activeDiscount) =>
                                      products.Select(activeDiscount).Sum();
}
```

This would allow for the following syntax:

```
var total = products.GetDiscountedPrice(PrimumDiscountStrategy);
```

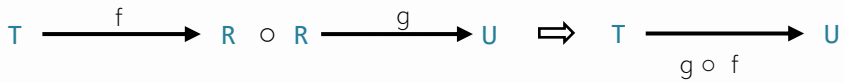
In this example `GetDiscountedPrice(...)` is the Higher-order Function. We can call this refactoring move "**Replace Strategy with Higher-order Function**." For simple scenarios, we do not have to create a full class in order to use the strategy pattern. In functional programming Higher-order functions are everywhere. This is the core idea of **functional programming; to treat functions as data**.

1.6 Simple Function Composition

One can argue that composing functions is the most important idea in programming and functional programming in particular. Functional composition is an instance of the concept of compositionality. Creating systems by composing smaller systems that are easier to implement and design can be found in all levels of programming. Staring from the Syntax of the language, to Design patterns, architectural patterns, larger System design etc.

In this section we are going to take a brief look at one of the most basic ones; Function composition.

In mathematics, function composition is the application of one function to the result of another to produce a third function. For instance, the functions f : T→ R and g : R→ U can be composed to yield a function **h**: **g**(**f**(**x**)) with type T→ U we usually represent that as **g** ∘ **f** and we read that as "**g after f** " the symbol ∘ usually represents "**after**".




```
Func<T, R> Compose<T, U, R>(Func<T, U> f, Func<U, R> g) => (T x) => g(f(x));
```

if we have some Functions we can compose them:

```
Func<Product, string> GetName = product => product.Name;
Func<string, string> ToUpper = name => name.ToUpper();

Func<Product, string> GetNameUpperCase = Compose(GetName, ToUpper);
```

Or by using Extension methods, we can fluently chain function delegates:

```
public static partial class FunctionalExt
{
  public static Func<A, C> Compose<A, B, C>(this Func<A, B> @f, Func<B, C> g) =>
                (A x) => g(f(x));
}
```

This allows us to write the composition in this way :

```
Func<Product, string> GetNameUpperCase = GetName.Compose(ToUpper);
```

What follows is a light exploration of function composition in the C# type system. We are not going to bother here with the pragmatics **instead we are going to just explore the morphology of composition**.

optional

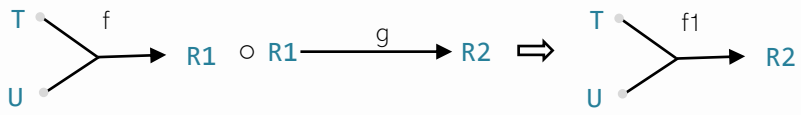
We are going to explain the intuition behind the construction of this function.

Since we want to construct a `Func<T,R>` we must return a function (lambda) that takes an argument arguments `T x` and return something of type `R`, using only the two functions that we want to compose `Func<T, R>` and `Func<R, U>`.

1. We start by writing this function as lambda `(T x) =>...` now we have a `T x` and two delegates `Func<T, R> f`, `Func<R, U> g`

2. We use the `T x`, to invoke the `Func<T, R> f` `f(x)` this will return an `R` and then we can use the functional delegate `Func<R, U>` which represents a function g: `R→U` from `R` to `U` in order to get back a `U` and return this.

If the first function was a two-argument function, we have another example of function composition:



The implementation is again trivial:

```
Func<T, U, R2> Compose<T, U, R1, R2>(Func<T, U, R1> f, Func<R1, R2> g) =>
                (T x, U y) => g(f(x, y));
```

The intuition behind the construction of this function is also simple:

Since we want to construct a `Func<T,U,R2>` we must return a function that takes two arguments `T x` and `U y` and return something of type `R2`, using only the two functions that we want to compose `Func<T, U, R1>` and `Func<R1, R2>`.

1. We start by writing this function as lambda `(T x, U y) =>...` now we have a `T x`, and `U y` and two delegates `Func<T, U, R1> f`, `Func<R1, R2> g`
2. We use the `T x`, and `U y` to invoke the `Func<T, U, R1> f` `f(x, y)` this will return a `R` and then we can use the functional delegate `Func<R1, R2>` which represents a function g: `R1`⟶ `R2` from `R1` to `R2`

optional

Curry-Howard correspondence

The construction of this simple function corresponds into a Proof of a Term like the following : ((**T**∧**U**) → **R1**) →(**R1** → **R2**) →((**T**∧ **U**) → **R2**)

When we implement the `Compose` function we actually construct a Deductive Proof.

$$\dfrac{\dfrac{\dfrac{[\boldsymbol{T}]^{x}\quad[\boldsymbol{U}]^{y}}{\mathbf{T}\wedge\mathbf{U}}\,I\wedge\quad[\mathbf{T}\wedge\mathbf{U}\rightarrow\mathbf{R1}]^{f}}{\mathbf{R1}}\rightarrow\mathrm{E}\qquad[\boldsymbol{R1}\rightarrow\mathbf{R2}]^{g}}{\mathbf{R2}}\rightarrow\mathrm{E}}{\dfrac{\boldsymbol{T}\rightarrow\mathbf{R2}}{\boldsymbol{U}\rightarrow\boldsymbol{T}\rightarrow\mathbf{R2}}\rightarrow I_{y}}\rightarrow I_{x}$$

1. We make two assumptions **[T]**^{x} and **[U]**^{y} this is pretty much the introduction of the two arguments `(T x, U y)` .

$$\dfrac{[\boldsymbol{T}]^{x}\quad[\boldsymbol{U}]^{y}}{\mathbf{T}\wedge\mathbf{U}}\,I\wedge$$

2. We then use them together with the function **f: T∧U→R1** in what is called implication elimination →**E,** which is just the function application `f(x, y)`.

$$\dfrac{\mathbf{T}\wedge\mathbf{U}\qquad[\mathbf{T}\wedge\mathbf{U}\rightarrow\mathbf{R1}]^{f}}{\boldsymbol{R1}}\rightarrow\mathrm{E}$$

3. Finally, we use another implication elimination →**E** which is the application `g(f(x, y))`.

$$\frac{\mathbf{R1} \quad [\boldsymbol{R1} \rightarrow \mathbf{R2}]^{\boldsymbol{g}}}{\boldsymbol{R2}}$$

We proved in a strong deductive sense that by having as Hypotheses the **T, U, T∧U→R1, R1→R2** we can get a **R2**.

Type Theory

Inhabited Types

If we had some other function **g** for example **R1→T**, there would not be any way to prove **R2**. The meaning of this in programming terms, is that there would not be any way to have a valid `Compose` of the following type without Compiler Errors:

```
Func<T, U, R2> Compose<T, U, R1, R2>(Func<T, U, R1> f, Func<R1, T> g) =>{  }
```

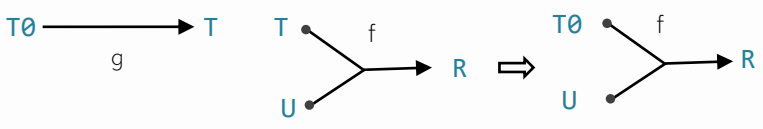
In type theory we say that the type ((**T**×**U**) → **R1**) →(**R1** → **T**) →((**T**×**U**) → **R2**) is **not inhabited**. Of course, it might hold if we would add some additional "facts" about T and R2 for example that **T** is subtype **R2**. By adding this constraint `where T:R2` this is typable:

```
Func<T, U, R2> Compose1<T, U, R1, R2>(Func<T, U, R1> f, Func<R1, T> g)
                                      where T:R2 =>
                                        (x, y) => g(f(x, y));
```

Additional "Facts"

In the case of simply typed lambda calculus, a type has an inhabitant if and only if its corresponding proposition is a tautology in Logic.

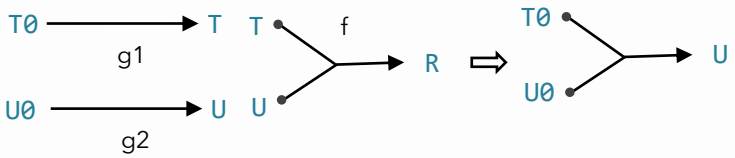
Another composition could be to pre-compose on the first argument some other function.




```
Func<T0, U, R> PreCompose<T, U, R, T0>(Func<T, U, R> f, Func<T0, T> g) =>
                (T0 x, U y) => f(g(x), y);
```

Run This: .Net Fiddle

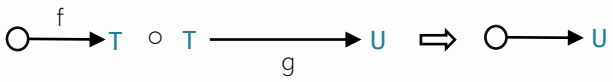
We could also pre-compose on both arguments :




```
Func<T0, U0, R> PreCompose<T, U, R1, T0, U0>(Func<T, U, R> f, Func<T0, T> g1, Func
<U0, U> g2) =>  (T0 x, U0 y) => f(g1(x), g2(y));
```

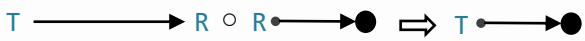
Run This: .Net Fiddle

Another possibility is to have a simple value provider `Func<T>` as the first function. In this case we can only post-compose:



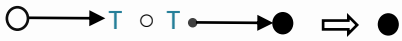

```
Func<U> Compose<T, U>(Func<T> f, Func<T, U> g) => () => g(f());
```

Similarly, we can compose into a " value sink" or destructor `Action<R>` that consumes a value `R`:




```
Action<T> PreCompose<T, R>(Func<T, R> g, Action<R> f) => (T x) => f(g(x));
```

Another situation is to have a value supplier followed immediately by a consumer of this value:



```
void Compose<T>(Func<T> f, Action<T> g) => g(f());
```

Terminal Operations

Unofficially we usually call methods that do not return anything as a result, terminal Operations.

The most common example is the `List<T>.ForEach(...)` method that has the following declaration:

```
public void ForEach(Action<T> action);
```

for example, we can pass inside the `ForEach` an `Action<T>`

```
Action<Product> printProductName = product => Console.WriteLine(product.Name);

new List<Product>{
       new Product("widgetA",100 ),
       new Product( "widgetB",1100),
       new Product( "widgetA",2000),
       new Product( "widgetB",300)
}.ForEach(printProductName);
```

when something returns a void, you might suspect that might be accepting an `Action<>` as an argument.

1.7 Callbacks

Let us close this section with another familiar Higher Order Function composition type: The Callback. We are going to use Callbacks in various forms thought-out this book (especially together with the concept of pattern matching, as we ill see in the next chapter).

Callback is a Func delegate that is passed as an argument to some Function. Other code inside the function is expected to *invoke the* Func *delegate* argument at a given time.

The simplest form of a callback is the following :

```
void G<T, R>(Action<R> callback, T y)
```

```
{
    //... do some stuff

    callback(default(R)); // call the action passing inside some result.

    //... continue or return
}
```

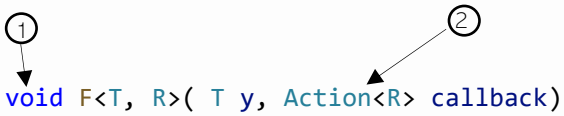
It is the closest thing to Firing a .NET Event, in a purely Object-Oriented setting.

Callbacks come up often in Asynchronous situations and Event driven architectures and are the core components of concept like the Task<T>, Rx.NET Observables<T>, and the Observer design pattern.

Every function can be converted into an equivalent one that uses callbacks. This is called Continuation passing style transformation. If we have a Function:

```
R F<T, R>(T y)
{
    return default(R);
}
```

We can convert it to the equivalent :




```
void F<T, R>( T y, Action<R> callback)
{
    callback(default(R));
}
```

1. The function now returns void **(1)**.
2. And has an additional argument `Action<R>` **(2)** in which we can pass the result.

Using `Func` delegates syntax a Continuation passing Transformation now takes a `Func<T,R>`:

```
Func<T, R> F = (T y) => default(R);
```

Into an `Action<T, Action<R>>`:

```
Action<T, Action<R>> G = (T y, Action<R> callback) => callback(default(R));
```

And instead of the initial function call :

```
R result = F(default(T));
```

Now we access the result using the following:

```
G(default(T), (R result) => { });
```

As a simple example we can convert a simple function that return the length of a string

```
Func<string, int> F = (string y) => y.Length;
```

```
Console.WriteLine(F("Foo"));
```

into this :

```
Action<string,Action<int>>G =(string y,Action<int> callback) =>callback(y.Length);

G("Foo", (int result) => { Console.WriteLine(result); });
G("Foo", result => Console.WriteLine(result));
G("Foo", Console.WriteLine);
```

Run This: .Net Fiddle

optional

Continuations

Furthermore, if you Curry (we will see currying in a later section) the callback argument you get something like the following :

```
Func<T, Action<Action<R>>> F =(T y)=>(Action<R> callback) => callback(default(R));
```

That is used in this manner:

```
F(default(T))((R result) => { });
```

In the case of the previous simple example, we now have this transformation.

```
Func<string, Action<Action<int>>> F =(string y) => (Action<int> callback) =>
                                                        callback(y.Length);

F("Jim")(Console.WriteLine);
```

In C# instead of continuations we use Task<T>. The previous can be rewritten using Tasks:

```
Func<string, Task<int>> F = (string y) => Task.Run(() => y.Length);

F("Jim").ContinueWith(x => Console.WriteLine(x.Result));
```

1.8 Named parameters and Tuples

Another late functional feature of C# is the **named parameters** feature. Named parameters are meant to increase readability. Using named parameters makes code way less ambiguous. If you have a method like the following for example:

```
void PrintOrderDetails(int orderNum, string productName, string sellerName) {...}
```

We can call it by explicitly naming the parameter names along with the values:

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
```

Instead of the following:

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Tuples

Sometimes instead of creating new classes we might use tuples. Tuples are lightweight, and do not pollute our model with unnecessary intermediate classes. For example, when you have a long parameter list for a method It is common practice to refactor it and Introduce a parameter object. For example, in this simple method:

```
public void PrintOrderDetails(int orderNum, string productName, string sellerName)
```

we could create the following type:

```
public struct PrintOrderDetailsCommand
{
     public int OrderNum { get; set; }
     public string ProductName { get; set; }
     public string SellerName { get; set; }
}
```

and replace the arguments:

```
public void PrintOrderDetails(PrintOrderDetailsCommand printCommand) { }
```

and use it like this:

```
var printCommand = new PrintOrderDetailsCommand
{
  OrderNum = 1,
  ProductName = "widget",
  SellerName = "ace"
};

new Printer().PrintOrderDetails(printCommand);
```

even if Introduce a parameter object refactoring sometimes is an improvement, we might not want to create a new class. We can instead use a tuple as an argument:

```
void PrintOrderDetails((int OrderNum,string ProductName,string SellerName)command)
```

amazingly this is a valid syntax the triple:

```
(int OrderNum, string ProductName, string SellerName)
```

Represents an object:

```
var printCommandTuple = (
                           OrderNum: 1,
                           ProductName: "widget",
                           SellerName: "ace"
                         );
```

we can pass this into the method above

```
void PrintOrderDetails((int OrderNum, string ProductName,
string SellerName) command)

var printCommandTuple = (
                          OrderNum: 1,
                          ProductName: "widget",
                          SellerName: "ace"
                       );

PrintOrderDetails(printCommandTuple);
```

The Tuples make it easier to also **return multiple** results from methods or Func Delegates. For example if we wanted to Return a Result object from the `PrintOrderDetails` method we could modify the method as follows:

```
(int orderPlacementId, int orderStatus) PrintOrderDetails((int OrderNum, string Pr
oductName, string SellerName) command)
```

1.8.1 Lambda expressions and tuples

Tuples in Func delegates

We can also mix the tuples with the `Func<>` delegates by using tuples as an argument type or a result type. This means we can write `Func` delegates of this form `Func<(_,_), (_,_)>` and create expressions like the following :

```
 Func<(int OrderNum, string ProductName), (int orderPlacementId, int orderStatus)>
printOrderDetails = command => { ... };
```

This is equivalent to the method `PrintOrderDetails`

```
public class Printer
{
  public (int orderPlacementId, int orderStatus) PrintOrderDetails(
     (int OrderNum,   string ProductName) command)
      {
```

```
            ...
        }
}
```

And we can even assign this method to a delegate of that type:

```
printOrderDetails = new Printer().PrintOrderDetails;
```

this is equivalent to:

```
printOrderDetails = command => new Printer().PrintOrderDetails(command);
```

if one wants to explicitly mention the command input argument.

Func delegates in Tuples

Also, you can have tuples that contain `Func<>` delegates. We are going to use those kinds of tuples allot. For example, look at this type:

```
(Func<int> zero, Func<int, int, int> add) monoid = (zero:()=>0, add:(x,y)=> x + y)
```

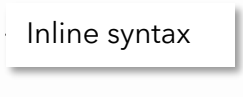
Is a collection of two functions `zero: () => 0` and `add: (x, y) => x + y`

Which is equivalent to the following class:

```
public class Monoid
{
        public int Zero() { return 0; }
        public int Add(int x, int y) { return x + y; }
}
```

Or if we use the inline function notation:

```
public class Monoid
{
        public int Zero() => 0;
        public int Add(int x, int y) => x + y;
}
```



Unfortunate we cannot assign tuples to interfaces:

```
interface IMonoid
{
        int Zero();
        int Add(int x, int y);
}

IMonoid monoid1 = (Zero: () => 0, Add: (x, y) => x + y);
```

Also, we cannot destructure tuples with Func delegates like this :

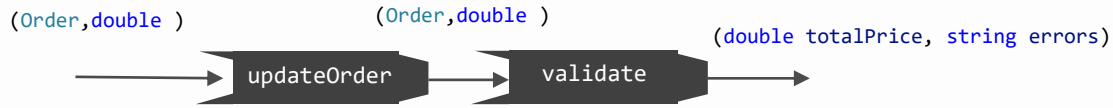
```
(Func<int>, Func<int, int, int>)(zero, add) =(zero: () =>0, add: (x, y) => x + y);
```

Tuples fit perfect the linear model of functional programming because we can easily adjust the inputs and outputs of consecutive function pipelines.

For example, we can trail along the pipeline some state data without the need to access the outside scope and thus maintaining the purity of the functions:

```
Func<(Order, double discount), (Order updatedOrder,double totalPrice)> updateOrder
 = state => { … };


Func<(Order, double totalPrice), (double totalPrice, string errors)> validate = st
ate => { … };

Func<(Order, double discount), (double totalPrice, string errors)> proccess =
                order => validate(updateOrder(order));
```

here for example, we transfer the updated Order across the pipeline.



Sometimes if you must pass too much data across, it might be an indication that you should break down even more the functions, or even use Object oriented instead of Functional design.

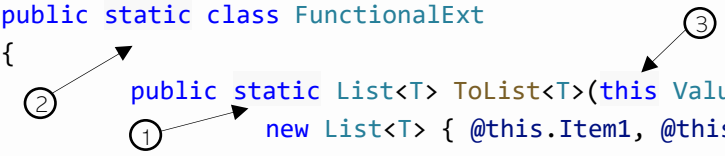
There are functional patterns like the State monad that facilitate this kind of composition. We are not going to see state monad in this book since C# is not purely functional and there is no need to introduce complexity when not needed.

1.9 Extension Methods

we are going to use Extension methods a lot. Extension methods might diverge from the traditional Object-oriented model of programming but allow us to provide custom functionality to .NET native types or extend our classes in ways that are not possible (or very difficult) through the standard language capabilities.

For example, we can give useful methods to native types to simplify our code and provide method chaining. Take for example this extension of Tuples that we are going to use. By adding the Following static class that contains an Extension method on the `ValueTuple`

```
public static class FunctionalExt
{
        public static List<T> ToList<T>(this ValueTuple<T, T> @this) =>
                    new List<T> { @this.Item1, @this.Item2 };
}
```



[Extension methods should be static (1) inside a static class (2) and you must use the `this` keyword (3) before the argument, which acts as a reference to the type you want to attach the method.]

We can now go ahead and use the `ToList<T>` method on any 2-item `ValueTuple`

```
var list = (4, 5).ToList();
```

since `ValueTuple<T,T>` is equivalent to a list `List<T>` with 2 items.

To take this one step further we will rewrite our Map definition on `ValueTuple<T,T>` from the Higher order Functions section :

```
ValueTuple<T1, T1> Map<T, T1>(ValueTuple<T, T> @this, Func<T, T1> f) =>
    new ValueTuple<T1, T1>(f(@this.Item1), f(@this.Item2));
```

and instead create an extension method on the `ValueTuple<T,T>`

```
public static class FunctionalExt
{
  public static ValueTuple<T1, T1> Map<T, T1>(
       this ValueTuple<T,T> @this,
       Func<T,T1> f) => new ValueTuple<T1, T1>(f(@this.Item1), f(@this.Item2));
}
```

The above declaration now will allow us to write:

```
var result  = (1,3).Map(x=>x*x);
```

instead of this:

```
var result = Map((1, 3), x => x * x);
```

in this way we obtain a certain control over the Native types and force our functional style of writing.

1.10 Using static directive

With the static directives one can use static methods without referring to the class. For example, because `WriteLine` is a `static` method of the `System.Console`

We can directly write :

```
WriteLine("message");
```

Instead of :

```
Console.WriteLine("message");
```

If we put as a using statement the following

```
using static System.Console;
```

like this :

```
using static System.Console;

namespace PracticalCSharp
{
    public class Demo
    {
        public void Run()
        {
            WriteLine("message");
        }
    }
}
```

In functional programming (and in language-ext) library we can use this to provide factory methods and avoid using the `new` keyword. For example, imagine that we have this class:

```
public class IO<T>
{
        public Func<T> Fn { get; set; }
        public IO(Func<T> fn) => Fn = fn;
        public T Run() => Fn();
}
```

Instead of writing:

```
var t = new IO(() => 1);
```

We can define a class with a static method that creates `new IO()`

```
public class Factories
{
   public static IO<T> IO<T>(Func<T> fn) => new IO<T>(fn);
```

```
}
```

In this way by `using static Factories` we can write directly:

```
var t = IO(() => 1);
```

Without the `new` keyword. We can argue about the usefulness of this idea. But it is a tradition coming from functional languages and minimizes the syntax. Also allows us to hide the `new` inside a single point. In this way if we wanted to change the class that is created on every `IO(() => …)` function call, we only have to change the `static` definition, like we do when we use the standard factory design pattern, or an IoC container.

In language-ext library most of the Types have static "factories", so we can usually write expressions like the following:

```
var optional = Some(123);
```

Instead of :

```
var optional = new  Some(123);
```

1.11 LINQ support for custom Data structures

Did you know that if you implement Select method in any type you can use the `from… in …select` Linq syntax??? The C# community quickly saw in this LINQ feature a way to emulate a syntactic pattern for chaining computation called Do notation looks something like this in Haskell:

```
 do { x1 <- action1
    ; x2 <- action2
    ; action( x1,x2 )}
```

This would translate to something like this in LINQ

```
var r = from x1 in new Id<int>(5)
        from x2 in new Id<int>(2)
        select x1 + x2;
```

we will see the way to add Linq support for custom data structures in more detail in the Functors and Monads Chapters. We also can use extension methods to provide LINQ support for native C# types like `Lazy<T>` for example:

```
public static Lazy<T1> Select<T, T1>(this Lazy<T> @this, ...) =>{ ... }
```

that would allow for something like this:

```
var r = from x1 in new Lazy <int>(()=>5)
         select x1 + 1;
```

1.12 Currying and partial application

Currying is a considerably basic and profound concept in functional programming. That is why it carries the name of the great Haskell Curry. Currying means that a function that takes multiple arguments can be translated into a series of function calls that each take a single argument. An expression like this:

```
public string F(int x, int y) {        }
```

that we invoke like this

```
var resultString = F(4,5);
```

can be transformed to the equivalent :

```
public Func<int, string> F(int x)
{
   string g(int y) { }
   return g;
}
```

Which we can call like this:

```
var resultString = F(4)(5);
```

or if we just use lambda the previous transformation would look like this:

```
Func<int, int, string> F = (x, y) => {   };

Func<int, Func<int, string>> F1 = x => y => {  };
```

We say that the second is the Curried version of the first. I know that using `Func` delegates it makes it difficult to grasp. For example, in JavaScript where we don't have types the previous looks like this:

```
var f = (x, y, z) => {…}

var f = x=>y=>z=> {…}
```

we replace the commas **(,)** with arrows **=>**.

The language-ext library provides Currying functions. A simple implementation of a generic Higher-order function that transforms a two-argument function, into its curried version looks like this:

```
Func<A, Func<B, C>> Curry<A, B, C>(Func<A, B, C> f) => (A x)=>(B y) => f(x, y);
```

the type of this function is ((**A** × **B**) → **C**) →(**A** →(**B** →**C**)). In a more specific example, a function that has a discount and a price argument:

```
Func<double, double, double> discountedPrice = (discount, price) =>
```

```
   price - discount * price;
```

the equivalent curryied function would look like this:

```
Func<double, Func< double, double>> discountedPrice = (discount) => (price) =>
  price - discount * price;
```

If we wanted to use the un-curried function with an array of prices, using a given discount normally one would write:

```
var discountedPrices = new List<double> { 10, 20, 30 }
                                        .Select(x =>discountedPrice(0.1, x));
```

here is the curryied version:

```
var discountedPrices1 =new List<double>{ 10, 20, 30 }
                                        .Select(discountedPriceCurried(0.1));
```

The difference here is that in the second case, we do not have the `.Select(x =>` because there is no need to refer to the array object explicitly. For this reason, the first style of writing is called pointed and the second one **point free**.

After we fixed an argument to a specific discount 0.1 we got a specialized function:

```
var discountedPricePartial = x => discountedPrice(0.1, x);
```

this process of reducing the arguments of a function it is **called partial application**, (or sometimes we might refer to this process as a **specialization**) and is usually mentioned alongside with currying, because after currying a function, we can get a partial application on the fly like this :

```
Curry(discountedPrice)(0.1)
```

Partial Application

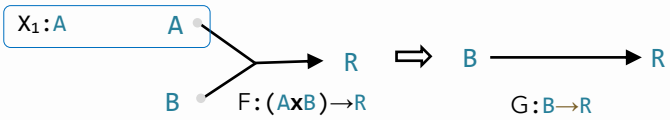
Partial application is a very simple concept to understand. Lets say we have a two-argument function:

```
R F<A, B, R>(A a, B b) => default(R);
```

We can create a new function:

```
R G<B, R>(B b) => F<A, B, R>(x1, b);
```

That takes a single argument by providing some default value `x1` for the `A` argument.

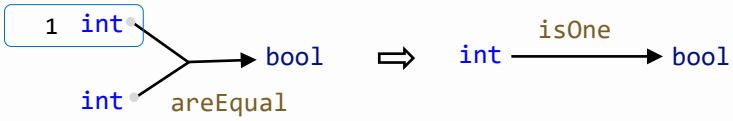


As a simple example consider the predicate, that tests if two integers are equal:

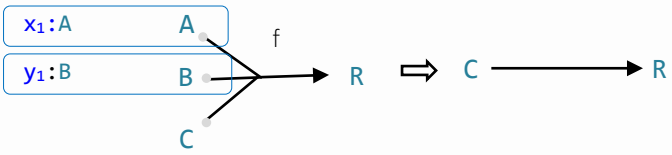
```
bool areEqual(int a, int b = 0) => a == b;
```

we can create a range of specialized predicates by partial applying on any of the arguments:

```
bool isZero(int a) => areEqual(a, 0);

bool isOne(int a) => areEqual(a, 1);
```



obviously, we can partially apply values to more than one argument (or even all of them)



currying always leaves the function at hand in the same syntactical state of x=>y=>z=>… every time we use **compose** or **partially apply** a value, we consume an arrow (=>), but the result maintained the same syntactical form; y=>z=>… thus promoting uniformity. This is idempotent design in practice. The downside is that the order of arguments becomes now important, and simple reorder refactoring of arguments might have side effects.

Algebras of Programming

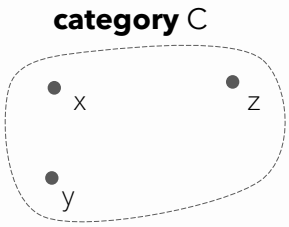
2 Algebras of Programming

2.1 Categories

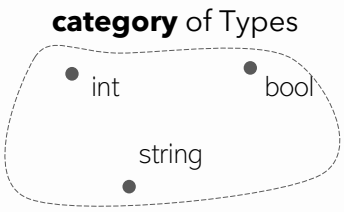
Category Theory is a mathematical discipline with a wide range of applications in theoretical computer science. Concepts like Category, Functor, Monad, and others, which were originally defined in Category Theory, have become pivotal for the understanding of modern Functional Programming (FP) languages and paradigms.

The **Category** is just an extension of the classical notion of **Set** of Objects. A category *C* consists of the following three mathematical entities:

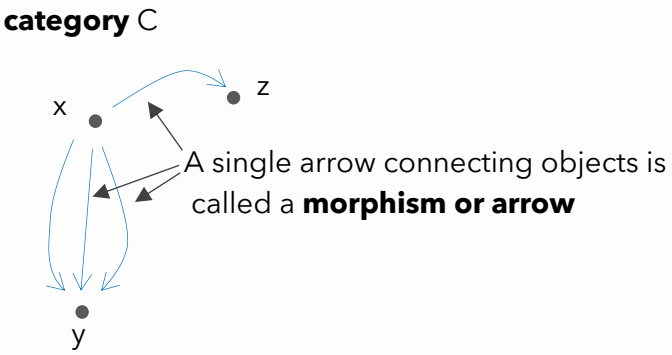
1. A class ob(*C*), whose elements are called **objects**. Any object-oriented programmer would find this as a great way to start a definition.



Our favorite category in programming is the **category of types** int, bool, char, etc. There are many interesting categories in programming and in this book, we will explore some of them.



2. A collection of elements called **morphisms** or arrows. Each morphism **f** has a source object **a**, and target object **b**.

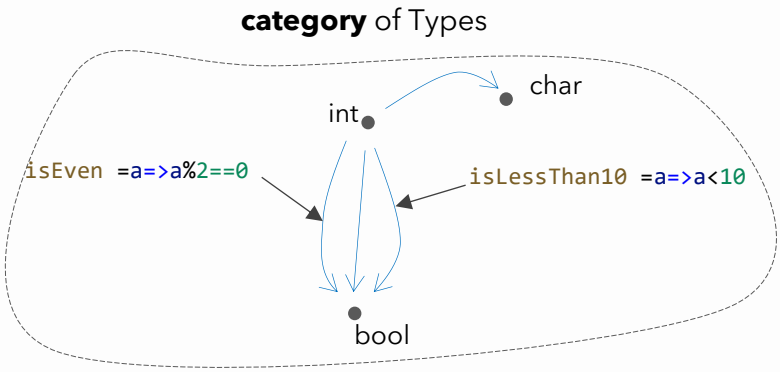


In our Type category, any arrow from `int` to `bool` for example, represents a function `int` → `bool`. Some examples might be:

```
bool isEven(int a)=> a%2==0;
```

Or this one:

```
bool isLessThan10(int a)=> a<10
```



The basic focus of category theory **is the relations between objects** and not the objects per se, in contrast with the Set theory that primarily focuses on sets of objects. Functional programmers quickly endorsed this unique perspective of category theory.

2.2 Monoids

"Alternatively, the fundamental notion of category theory is that of a Monoid"
– Categories for the Working Mathematician

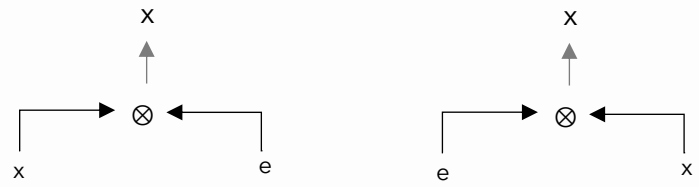
Monoids are one of those profound ideas that are easy to understand and are everywhere. Monoids belong to the category of Algebras. There are a couple of particularly important ways to organize different structures that appear in functional programming. Algebras is one of them. Fantasy land has some of the most important algebras in its specification.

Wikipedia says a **monoid** is an algebraic structure with a single associative binary operation and an identity element. Any structure that has those two elements is a monoid.

Suppose that S is a structure and ⊗ is some binary operation **S** ⊗ **S** → **S**, then **S** with ⊗ is a **monoid** if it satisfies the following two axioms:

1. **Associativity**: For all a, b and c in S, the equation **(x** ⊗ **y)** ⊗ **z** = **x** ⊗ **(y** ⊗ **z)** holds.

2. **Identity element**: There exists an element e in S such that for every element a in S, the equations **e** ⊗ **a** = **a** ⊗ **e** = **a** hold.



Let us take the integers we can take addition as a binary operation between two integers. The addition is associative. This means there is no importance on the order of the addition.

```
(x+(y+z)) = ((x+y) +z)
```

And 0 is the identity element for addition since:

```
x + 0 = x
```

For those reasons, the pair **(0, +)** forms a monoid over the integers. However, we can form different other monoids over the integers. For example, multiplication ***** also can be viewed as a monoid with its respective identity element the **1**. In this way the pair **(1, *)** is another

monoid over the integers. Strings in C# have a default monoid, which is formed by the concatenation and the empty string (`""`, `Concat`).

We will use monoids in our code by defining simple object literals that have two functions, one that returns the identity element called **Empty** and one that represents the binary operation called **Concat** .

[In this book, we will use the fantasy land specification of the monoid regarding naming conventions, especially since C# already uses `Concat` for strings and arrays.]

```
public interface IMonoid<T>
{
     T Empty { get; }
     T Concat(T x, T y);
}
```

This interface has a type of : `IMonoid<T>`: (`Empty`: T , `Concat`: (T × T)⟶ T), alternatively the curried version for `Concat` is (T ⟶ T)⟶ T which is a common typing.

we will use monoids in this way:

```
var sum = new Sum();
var total = sum.Concat(1, sum.Concat(3, 4));
```

This definition would allow us to use it with the `Aggregate` function of a list in a direct manner:

```
total = new[] { 1, 3, 4, 5 }.Aggregate(sum.Empty, sum.Concat);
```

Run This: Fiddle

We can also define a monoid as a tuple:

```
(int empty, Func<int, int, int> concat) sum = (0, (x, y) => x + y);
```

Alternatively, if we want to have closure [meaning that the return type of the `Concat` and `Empty` is of the same type with the initial object], we could define one new class for each monoid.

```
public interface IMonoidAcc<T>
{
        T Identity { get; set; }
        IMonoidAcc<T> Concat(IMonoidAcc<T> m);
}

public class Sum : IMonoidAcc<int>
{
       public Sum(int value) => Identity = value;
       public int Identity { get; set; }
       public IMonoidAcc<int> Concat(IMonoidAcc<int> m) =>
                                    new Sum(m.Identity + Identity);
}

var total = new Sum(0).Concat(new Sum(1)).Concat(new Sum(2)).Concat(new Sum(3));
```

The return type is `IMonoidAcc` allowing for fluent chaining

Run This: Fiddle

This formulation is nice in the sense that allows for fluent chaining `var total = new Sum(0) .concat(new Sum(1)) .concat(new Sum(2))` because of the fact that empty and concat both return a result of the same type. This also is more congruent with the mathematical definition that says that if S is a structure (in a programming language, this translates to a Type), then $S \times S \rightarrow S$, means that the operation returns a structure of the same kind.

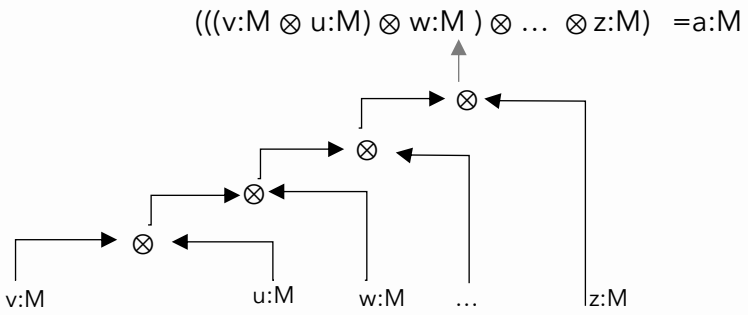
In practice we do not care about strict definitions as long as we are aware that we are dealing with a monoid.

2.2.1 Folding monoids

Monoids have this very desirable property that if we keep applying this binary operation ⊗, we can always reduce the computation to a single element of the same type:

$(M \otimes \ldots (M\ (\otimes (M \otimes M))) \rightarrow M$

this is called folding.



Let us say we have some sequence of integers `2,4,5,6` and we want to add them this a beginner level problem:

```
var list = new[] { 1, 2, 3, 4 };
var accumulation = 0;
foreach (var current in list)
    accumulation = accumulation + current;
```

(0,+) monoid

What I highlighted there are just an initial element 0 and the addition which is an operation between two integers that return an integer

What if we wanted to multiply them?

```
var accumulation = 1;
foreach (var current in list)
    accumulation = accumulation * current;
```

(1,*) monoid

both **+** and ***** can be abstracted as lambdas. For the **+** case

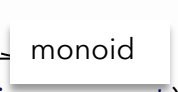
```
Func<int, int, int> add = (int x, int y) => x + y;
```

And in the second case we start with an initial element the 1 and we use the multiplication as an operation between the two integers

```
Func<int, int, int> add = (int x, int y) => x * y
```

Because both have the same signature we can abstract them and create a Higher order function that we can reuse.:

```
public T Fold<T>(List<T> list, T accumulation, Func<T, T, T> concat)
{
        foreach (var current in list)
                accumulation = concat(accumulation, current);
        return accumulation;
}
```



Run This: .Net Fiddle

Now we can use this function in both cases:

```
var sum = Fold<int>(new[] { 1, 2, 3, 4 }, 0, (x, y) => x + y);
var product = Fold<int>(new[] { 1, 2, 3, 4 }, 1, (x, y) => x * y);
```

but also, we can use other combinations of `(T accumulation, Func<T, T, T> concat)` parameters (aka monoids) to Fold the array. For example:

```
 var max = Fold(new[] { 1, 2, 3, 4 }, int.MinValue, (x, y) => x > y ? x : y);
```

Run This: .Net Fiddle

Obviously, this Fold function is so important that is implemented in all programming languages for the array data structure. In .NET it is called Aggregate:

```
var sum = new List<int> { 1, 2, 3, 4 }
             .Aggregate(0, (accumulation, current)=> accumulation + y);

var product = new List<int> { 1, 2, 3, 4 }
             .Aggregate(1, (accumulation, current) => accumulation * y);
```

If you look the type signature by navigating to the metadata definition (with F12) on any List you will see the following

```
TAccumulate Aggregate<TSource, TAccumulate>(TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func);
```

[side note the aggregate is something we call usually FoldMap which is the same with Fold but also includes a Map aka Select operation this allows us to have the result `TAccumulate` to be of different type from the type of the List `TSource`]

2.3 Folding monoids showcase

In this section we are going to display some examples of monoid folding using the `Aggregate` function. In all the examples we are going to:

1. Identify some type S and find some way to combine two S into a single S.
2. Then having that given a List of S we can use `Aggregate` to fold them.

Folding delegates Func<,>

We will study this case extensively later in this chapter because it is the base of many classical object-oriented patterns. Here we will start with two `Func<int, int>` delegates:

```
Func<int, int> A = (x) => x + 1;
Func<int, int> B = (x) => x + 2;
```

Having those two we can find a function that combines them into a single `Func<int, int>`

```
Func<int, int> compose  = (x) => A(x) + B(x);
```

Having this function, we can now fold a whole list of `Func<int, int>` into one.

```
List<Func<int, int>> list = new List<Func<int, int>>() { A, B };

Func<int, int> accumulation = (x) => x;
foreach (var current in list)
{
    accumulation = (x) => accumulation(x) + current(x);
}
```

Run This: .Net Fiddle

This for loop can be abbreviated using `the Aggregate` method:

```
var folding = list.Aggregate(
                (int x) => x,
                (Func<int, int> accumulation, Func<int, int> current) =>
                            (x) => accumulation(x) + current(x)
            );

Console.WriteLine(folding(2));
```

Folding callbacks

Having two callbacks, we can find a way to compose them into a single callback:

```
Action<int, Action<int>> square = (x, callback) => callback(x * x);
Action<int, Action<int>> cube = (x, callback)   => callback(x * x * x);

Action<int, Action<int>> Compose(Action<int, Action<int>> c1,
```

```
                                    Action<int, Action<int>> c2) =>
                                    (int x, Action<int> callback) =>
                                        c1(x, r => c2(r, callback));

var f = Compose(square, cube);

f(4, r => Console.WriteLine(r));
```

Run This: .Net Fiddle

having this idea of composition, we can fold a list of callbacks.

Folding Tasks

Having two Tasks, we can compose:

```
Task<int> A = Task<int>.FromResult(5);
Task<int> B = Task<int>.FromResult(7);

Task<int> Compose(Task<int> t1, Task<int> t2) =>
                          t1.ContinueWith(a => a.Result + t2.Result);
```

Having a List of Tasks we can Fold them:

```
var list = new List<Task<int>>() { A, B };

Task<int> total = Task<int>.FromResult(0);
foreach (var current in list)
{
    total = A.ContinueWith(a => a.Result + B.Result);
}
```

Or using the `Aggregate`

```
Task<int> t = list.Aggregate(                    // Kleisli Fold  - FoldM
    Task.FromResult(0),
    (accumulation, current) => A.ContinueWith(a => a.Result + B.Result)
  );
```

Run This: .Net Fiddle

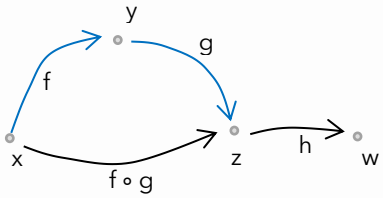
We will go in more detail in the Monads Chapter.

2.4 Function composition as a monoid

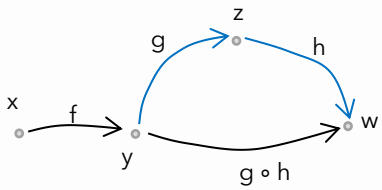
Function composition is itself a more abstract and interesting example of a monoid. Let us say that we have a category where the objects are the functions `Func<A, B>`, and the binary operation is the **composition** of two functions _ ∘ _

```
Func<A, C> Compose<A, B, C>(Func<A, B> f, Func<B, C> g) => (A x) => g(f(x));
```

then associativity still works since: (**f** ∘ **g**) ∘ **h** = **f**∘ (**g** ∘ **h**)



(f ∘ g) ∘ h



f∘ (g ∘ h)

This means that those two Paths :

```
Func<A, D> Path1<A, B, C, D>(Func<A, B> f, Func<B, C> g, Func<C, D> h) =>
                  Compose(Compose(f, g), h);

Func<A, D> Path2<A, B, C, D>(Func<A, B> f, Func<B, C> g, Func<C, D> h) =>
                  Compose(f, Compose(g, h));
```

will be equal for all inputs (unfortunetly we cannot express the **forall** quantification within the Type system). Furthermore, `A Id<A>(A x) => x` is the identity element for the function composition ∘ :

(**f** ∘ **id**) = **f**

Or in code:

```
Func<A, B> ReflA<A, B>(Func<A, B> f) => Compose<A, B, B>(f, Id<B>);
```

This would be equal with the function **f forall** inputs for **f**. In this way, the composition is a monoid (**x=>x,** ∘) over the `Func<,>`.

Sidenote:

Because of the monoidal nature of composition, we can extend the definition that takes only two functions on any number of functions by just keep applying the composition. For example, for four functions would be `compose(h)(compose(g)(compose(g)(k)))`, given that their types are appropriate.

2.5 Composing monoids

Monoids can also be composed in various ways and give more monoids. In this section, we will see three such examples of composition. Our first monoidal composition is the one that is formed by functions that have the same type signature:

$(\mathbf{A}\rightarrow\mathbf{B}) \otimes (\mathbf{A}\rightarrow\mathbf{B}) \rightarrow (\mathbf{A}\rightarrow\mathbf{B})$

[Note: this is different from functional composition where we have the composition operation $(\mathbf{A}\rightarrow\mathbf{B}) \circ (\mathbf{B}\rightarrow\mathbf{C}) \rightarrow (\mathbf{A}\rightarrow\mathbf{C})$]

Let me illustrate how this monoid arises in a practical way. Let us say for example we have this discount function `(double price) => 0.1 * price` that type of this function is `double →double` we can define a higher order function `ComposeDiscounts` that compose two functions of this type:

```
Func<double, double> basicDiscount = (double price) => 0.1 * price;
Func<double, double> repeatCustomerDiscount = (double price) => 0.2 * price;


Func<double, double> ComposeDiscounts(Func<double, double> firstDiscount,
                                      Func<double, double> secondDiscount) =>
      (double price) => firstDiscount(price) + secondDiscount(price);
```

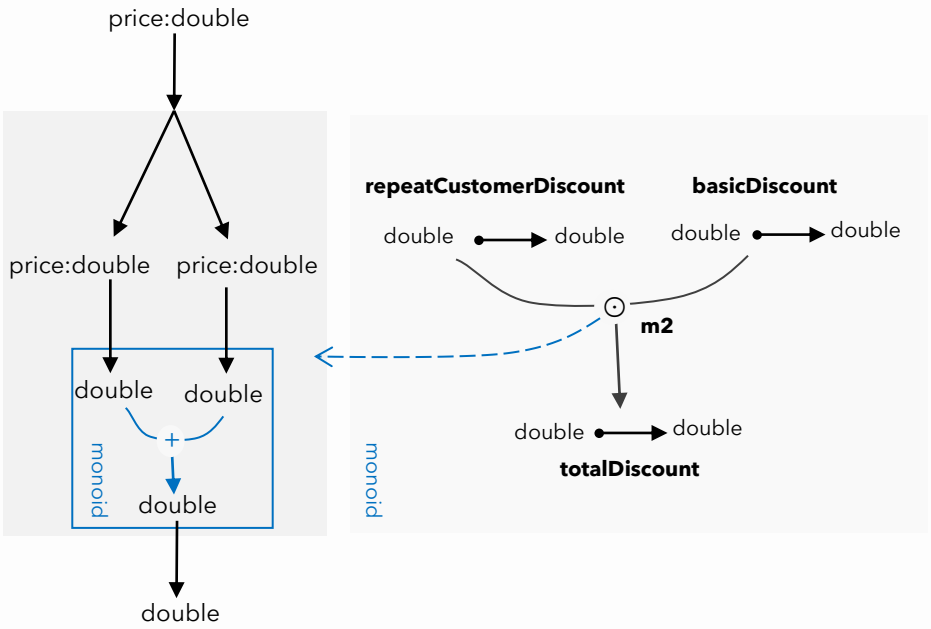
Monoid over the `double`

In this way by using the **+** monoid operation over the `double` type, we have created a function `ComposeDiscounts` which is a monoid over the `Func<double, double>`

```
Func<double, double> GetFinalPrice(Func<double, double> discount) =>
                        (double price) => price - discount(price);

var basicAndRepeatedDiscount = ComposeDiscounts(basicDiscount, repeatCustomerDisco
unt);

var finalPrice = GetFinalPrice(basicAndRepeatedDiscount)(100);
```

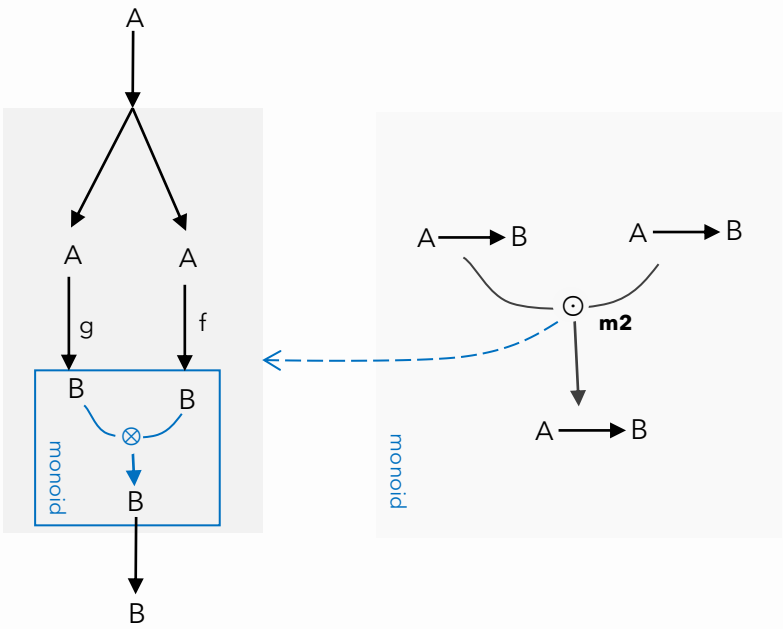
Run This: .Net Fiddle

The mechanics of this composition is extremely easy to understand :



1. We pass the input price to both `Func<double, double>` and we get the results,
2. We then use the **+** to get a single `double`.

One can prove that we can create a monoid ⊙ from functions (**A**→**B**) `Func<A, B>` when **B** has a monoid ⊗.



We can create a new **A→B**, by just applying both **f** and **g** and then gluing their **B** results.

This idea is in the core of the famous **Composite and Decorator [also interpreter] design patterns** from the Gang of Four design patterns.

Immediately you can see that if we had a List of Discounts (`Func<double, double>`) we could use `ComposeDiscounts` in order to Fold them into a single discount delegate:

```
Func<double, double> ComposeAll(List<Func<double, double>> @discounts) =>
    @discounts.Aggregate(
            (x) => 0,
            (totalDiscount, current) => ComposeDiscounts(totalDiscount, current)
      );
```

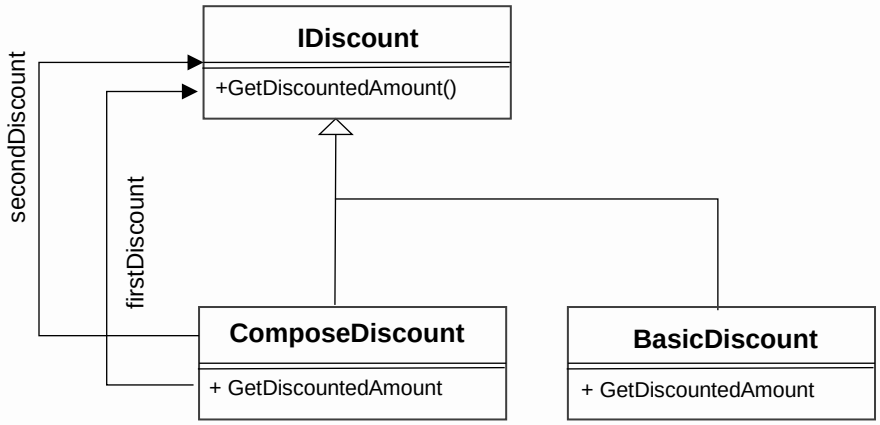
Monoid on the `discounts`

First let us rewrite the above in its full Object-Oriented syntax :

```
public interface IDiscount {
    double GetDiscountedAmount(double price);
}


public class BasicDiscount : IDiscount
{
    private readonly double discount;
    public BasicDiscount(double discount) => this.discount = discount;
    public double GetDiscountedAmount(double price) => discount * price;
}

public class ComposeDiscounts : IDiscount
{
    private readonly IDiscount firstDiscount;
    private readonly IDiscount secondDiscount;

    public ComposeDiscounts(
        IDiscount firstDiscount,
        IDiscount secondDiscount)
    {
        this.firstDiscount = firstDiscount;
        this.secondDiscount = secondDiscount;
    }
    public double GetDiscountedAmount(double price) =>
        firstDiscount.GetDiscountedAmount(price) +
        secondDiscount.GetDiscountedAmount(price);

}
```

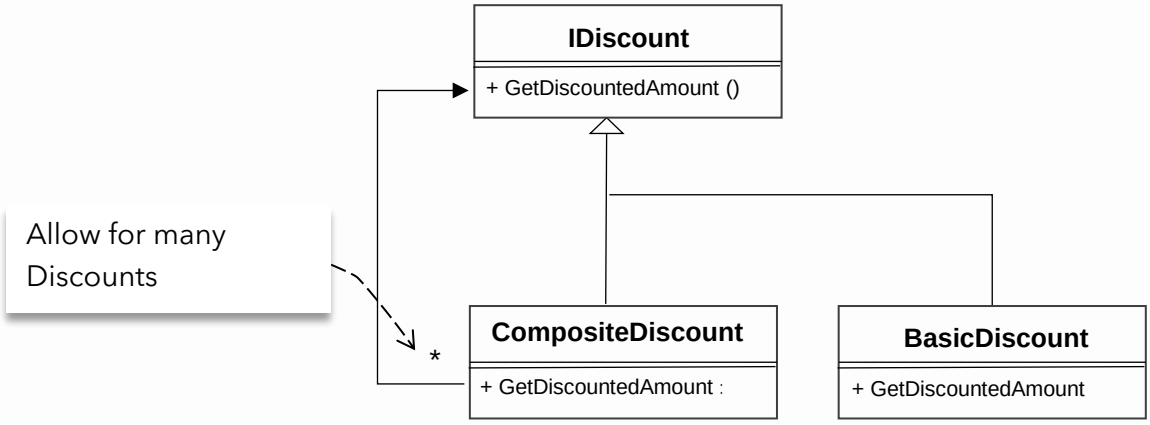
Monoid on the `discounts`

Here the `GetDiscountedAmount` is the equivalent of the anonymous `Func<double, double>`.



2.6 Composite Design pattern - Functional Perspective

The following is an implementation of a Composite design pattern that generalize the compose discount model. This is easily done by creating a new sub-class that contains a `List<IDiscount>` list of discounts:

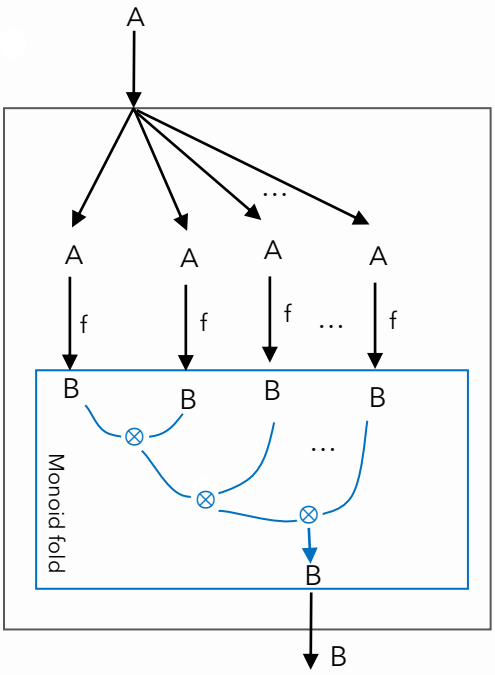



```
class CompositeDiscount : IDiscount
{
    List<IDiscount> discounts { get; set; } = new List<IDiscount>();

    public CompositeDiscount AddDiscount(IDiscount discount)
    {
        discounts.Add(discount);
        return this;
    }
```

The composite discount uses the composition of monoids in `GetDiscountedAmount`

```
    public double GetDiscountedAmount(double price)
    {
        return discounts.Aggregate(
                seed: 0.0,
                func: (sum, discount) => sum + discount.GetDiscountedAmount(price)
            );
    }
}
```

as you can see in the following diagram using a fold you can convert an arbitrary number (list) of functions with the same type **A→B** (eg `GetDiscountedAmount`) into a single function **A→B** (eg `GetDiscountedAmount`) when the **B** has a monoid that we can fold with.



We can create a new **A→B**, by just applying the same A to all **f** and then gluing their **B** results.

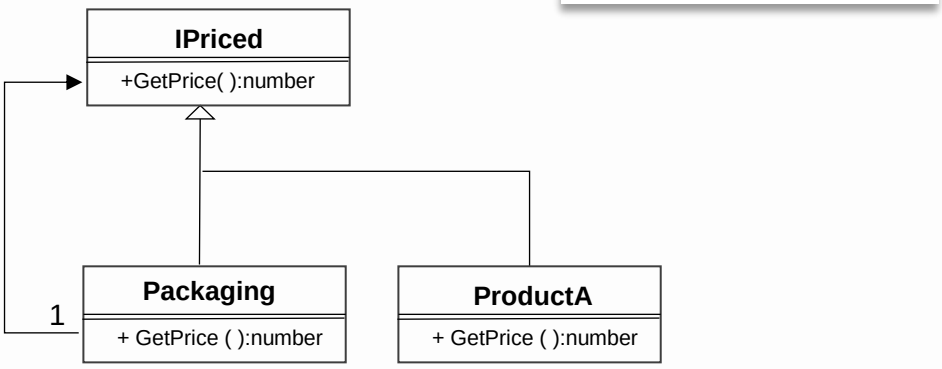
2.7 Decorator Design pattern - Functional Perspective

Functional composition is in the heart of many object-oriented best practices. Here we will see an idiomatic expression of the Decorator design pattern, which is one of the most used object-oriented design patterns from the Gang of Four collection.

Let us start with simple example where we have a `ProductA`, and we can decorate it with `Packaging`. Adding a packaging would affect the price, but we want to treat a product and a packaged product interchangeably in our code. The decorator pattern allows us to do just that by unifying both `ProductA` and `Packaging` under a common `IPriced interface` :

```
public interface IPriced {
	double GetPrice();
}

public class ProductA : IPriced {
	public double GetPrice() => 100;
}

public class Packaging : IPriced
{
	private readonly IPriced product;
	public Packaging(IPriced product) => this.product = product;
	public double GetPrice() => product.GetPrice() + 10;
}
```

Notice the Monoidal operation



The decorator pattern is based on the idea/theme that the decorator exposes the same interface with the decorated object.

In this example, the `Packaging` exposes the same `GetPrice()` method with the decorated `Product.` This allows the rest of the application to handle a simple product or a decorated product in the same way without needing to know more details in order to get the price [this is a nice display of **information hiding/encapsulation** in object-oriented programming, since the decorated object hides the details of implementation from the outside world and only reveals the price].

From the Functional perspective this again is possible because the **+** monoid on the return type of the `GetPrice` function.

Only functions notation

Finally, we can strip of all the object-syntax completely and only keep the functions.

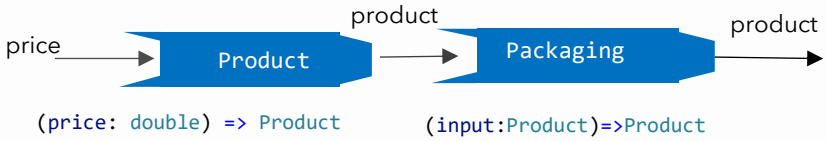
```
Func<double> ProductA = () => 100;
Func<double> ProductB = () => 200;
Func<Func<double>, Func<double>> Packaging =
             (product) => () => product() + 10;

Func<double> packagedProduct = Packaging(ProductA);
```

So, In a way any function of this form `(T a) => (R b) => {}` **is equivalent to an object** that takes an **`a:T` in the constructor** and has **only one method** that takes a **`b:R`** as argument.

Only functions: `(Func<double> product) =>` `()=>{...}`

OOP: `class Packaging {constructor(IPriced product) {…}` `double getPrice() {...} }`

The idea here is for the decorating composing functions that come after the decorated function that has a return type Product.



Tuple notation

Another way to represent the Object is to use tuples, Unfortunately C# does not allow Tuples with only a single member. So, for presentation purposes we add another method to our product that returns the name `getName: () => name`. Now we can define a product, by a function that acts as a Constructor:

```
Func<string, double, (Func<string> getName, Func<double> getPrice)> Product =
    (string name, double price) =>
            (getName: () => name, getPrice: () => price);
```

We can use this in order to create products:

The Tuple (…) can be seen as a minimal Object

The `Func` delegates can be seen as methods.

```
(Func<string> getName, Func<double> getPrice) ProductA = Product("A", 100);
(Func<string> getName, Func<double> getPrice) ProductB = Product("B", 200);
```

Now we can define the Decorator function as follows:

```
Func<(Func<string> getName, Func<double> getPrice),
    (Func<string> getName, Func<double> getPrice)>
    Packaging =
```

```
        product =>
        (
            getName: () => product.getName() + "Packaging",
            getPrice: () => product.getPrice() + 0.1
        );
```

Run This: .Net Fiddle

Which then we can use like this :

```
(Func<string> getName,Func<double> getPrice) packagedProduct= Packaging(ProductA);

(Func<string> getName, Func<double> getPrice) doublePackagedProduct =
                                                Packaging(Packaging(ProductA));
```

Obviously the Tuples as Object minimal syntax here is only for presentation purposes. In C# 9.0 immutable record types have been added in order to cover the cases where we want to adhere to a more minimal functional coding style.

2.8 Predicate monoidal Composition

The most Famous **contravariant** functional type is the Predicate which is any function that has a type of _→ `bool` (returns a Boolean) we already saw that the .NET provides a native Predicate delegate :

```
public delegate bool Predicate<in T>(T obj);
```

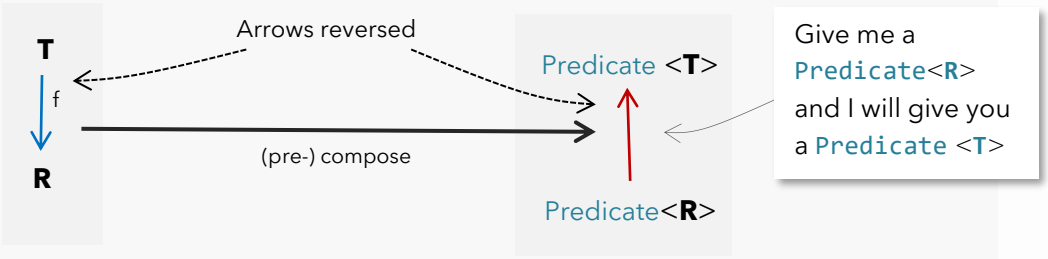
in practice we usually we use the delegate `Func<T,bool>` to represent predicates. Most of the .NET framework internally also uses the `Func<T,bool>` as predicate type. For example, for the .NET framework definition for the `List<T>.Where` has the following signature:

```
IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                Func<TSource, bool> predicate)
```

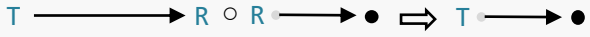
Category Theory — Contravariant Types

Those kind of types are called **contravariant** because they have the "changing (Type)" in the "negative side of the function (which is the input) ". The input argument is what varies (T x) → `bool`. When we say here that the input argument varies we mean that the **Type of the input argument is Generic** (we have **parametric polymorphism** on the input argument) whereas the return Type is fixed to `bool`.

To understand why its contravariant, try to compose a function `T⟶ R` with a predicate `(R a)=>bool` to get a predicate `(R a) => bool`. The only way is to have a composition, is to pre-compose the `T⟶ R` to the `(R a) => bool` to get a `(T a) => bool`



This visually looks like this :



Where ● is the `bool` in our case.

In this way for the Predicate, we can define a `PreCompose` function:

```
Predicate<R> PreCompose<T, R>(Func<R, T> f, Predicate<T> g) => (R x) => g(f(x));
```

The whole contravariant pre-composition becomes obvious when you take a look at this simple example where we have a `Predicate<double>` on the Price and we want to produce a `Predicate<Product>` on the Product using a Projection for the Price `GetPrice` :

```
Predicate<double> IsHightPriced = (price) => price > 100;

Func<Product, double> GetPrice = (product) => product.Price;

Predicate<Product> IsHightPricedProduct = PreCompose(GetPrice, IsHightPriced);
```

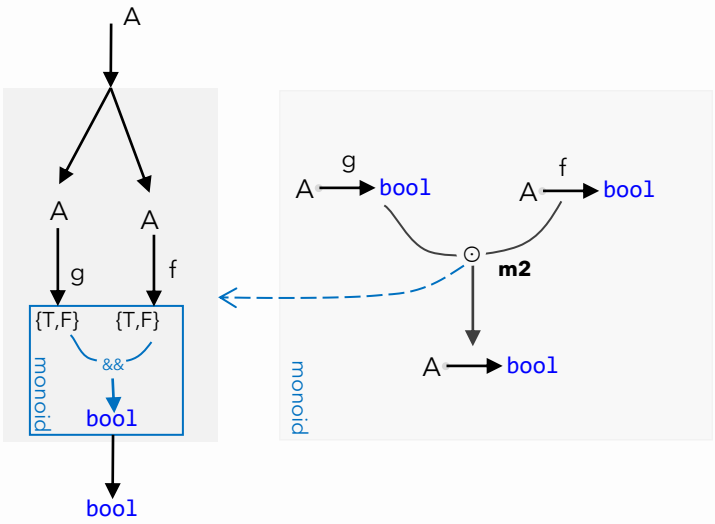
This is one way to compose Contravariant types Like `Predicate`.

Another way is the **monoidal composition**. Using the monoidal operations on the `bool`, `&&`, `||` , `~` , we can define various compositions `And`, `Or`, and `Not` that would allow us to form larger Predicates from simpler.

The specification design pattern is based on this idea by defining:

```
Func<T, bool> And<T>(Func<T, bool> p1,Func<T, bool> p2) => (T x) => p1(x) && p2(x)

Func<T, bool> Or<T>(Func<T, bool> p1,Func<T, bool> p2) => (T x) => p1(x) || p2(x);

Func<T, bool> Not<T>(Func<T, bool> p1) => (T x) => !p1(x);
```

Run This: .Net Fiddle



> We can create a new predicate, by just applying both predicates and then gluing their `bool` results.

Then we can use some elementary basic predicates on the product :

```
Func<string, Func<Product, bool>> nameContains = pattern => product =>
                                                 product.Name.Contains(pattern);

Func<Product, bool> highPriced = product => product.Price > 1000;
```

and then we can compose them using `And`, `Or`, and `Not` to **create more complex predicates** like :

```
Func<Product, bool> highPricedAndContaisA = And(highPriced, nameContains("A"));
```

we can use those predicates with the filter method of the array :

```
var products = new List<Product>{
                new Product("widgetA",100 ),
                new Product( "widgetB",1100),
                new Product( "widgetA",2000),
                new Product( "widgetB",300)
            };

var notHighPricedAndContaisA = Not(highPricedAndContaisA);

var filiteredProducts = products.Where(notHighPricedAndContaisA);
```

Run This: .Net Fiddle

or try filtering the List with the following `Predicates`

```
var filter  = Or(highPriced, nameContains("B"));

var filter  = Not(Or(highPriced, nameContains("B")));
```

We can further simplify the syntax by providing Static extensions, to allow for Fluent composition:

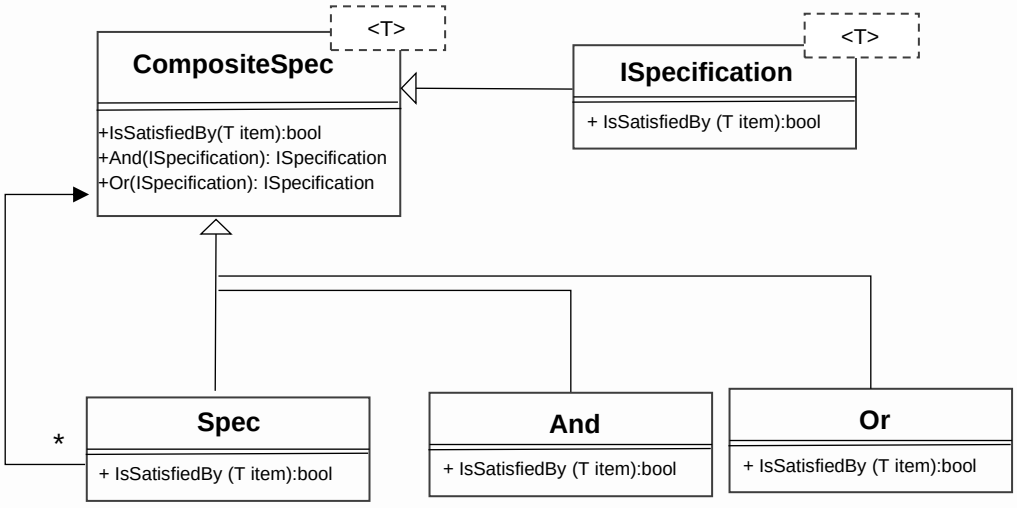
```
public static partial class FunctionalExt
{
public static Func<T, bool> And<T>(this Func<T, bool> @this, Func<T, bool> p) =>
        (T x) => @this(x) && p(x);

public static Func<T, bool> Not<T>(this Func<T, bool> @this) =>
            (T x) => !@this(x);
}
```

Now we can write:

```
Func<Product,bool>notHighPricedAndContaisA=highPriced.And(nameContains("A")).Not()
```

2.9 Specification Pattern - Functional Perspective

We can extract those mechanics in a fluent building construction that allow us to create predicate expressions. This is the vary famous Specification pattern. The specification pattern often invokes mixed reactions, and it is even considered as anti-pattern by many. Nonetheless is a part of the Domain Driven Design tactical patterns as envisioned by Eric Evans.



The basic idea for the specification pattern is **the composition of Predicates over the over the Boolean monoid that we saw previously**. The main Object-oriented enhancement **is the addition of a fluent builder** that allows us to chain expression that look like the following:

```
 var spec1  = new Spec(nameContains(`A`)).And(highPriced);
```

Instead of the “functional”

```
And(highPriced, nameContains("A"))
```

the core is the `CompositeSpecification` from which all other expression inherits from

```
public interface ISpecification<T> {
    bool IsSatisfiedBy(T item);
}

//in order to add fluent builder semantics onto the And, Or,Not
public abstract class CompositeSpecification<T> : ISpecification<T>
{
    public CompositeSpecification<T> And(ISpecification<T> spec) =>
             new And<T>(this, spec);

    public CompositeSpecification<T> Or(ISpecification<T> spec) =>
             new Or<T>(this, spec);

    public CompositeSpecification<T> Not() =>
             new Not<T>(this);
    public abstract bool IsSatisfiedBy(T item);
}
```

`CompositeSpecification` provides a fluent interface.

Run This: .Net Fiddle

Here we are going to have a base `Spec<T>`

```
public class Spec<T> : CompositeSpecification<T>
{
    private Func<T, bool> expression;
    public Spec(Func<T, bool> expression) => this.expression = expression;
    public override bool IsSatisfiedBy(T item) => expression(item);
}
```

Take a look at the `And<T>`:

```
public class And<T> : CompositeSpecification<T>
{
    private ISpecification<T> left;
    private ISpecification<T> right;
    public And(ISpecification<T> left, ISpecification<T> right)
    {
        this.left = left;
        this.right = right;
    }

    public override bool IsSatisfiedBy(T item) =>
        left.IsSatisfiedBy(item) && right.IsSatisfiedBy(item);
}
```

The fact that everything inherits from the `CompositeSpecification` means that we can build fluent expressions that provide chaining. In essence this is not the composite pattern but a Builder design pattern which allows us to write and use specifications like this:

```
var products = new List<Product>{
    new Product("widgetA",100),
    new Product("widgetB",1100),
    new Product("widgetA",2000),
    new Product("widgetB",300)
};
```

```
ISpecification<Product> notHighPricedAndContaisA =
                new Spec<Product>(nameContains("A"))
                .And(new Spec<Product>(highPriced))
                .Not();

var filiteredProducts = products.Where(product =>
                        notHighPricedAndContaisA.IsSatisfiedBy(product)
                    );
```

Run This: .Net Fiddle

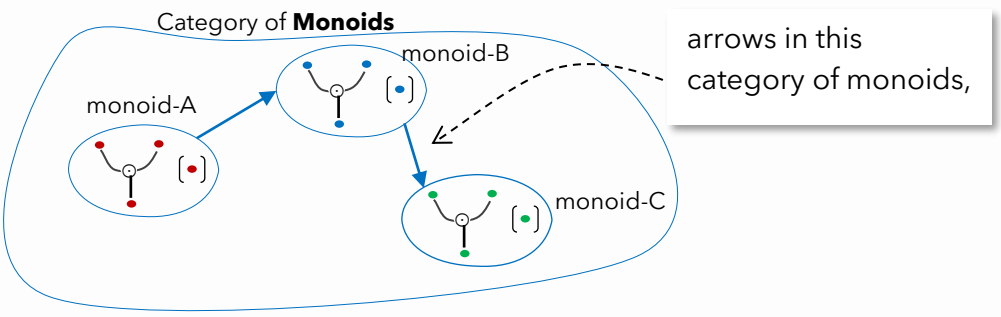
2.9.1 Monoid homomorphisms and Parallelism

The length of a concatenation of two arrays, equals the sum of the length of the arrays:

```
([1, 2, 3, 4].Concat([7, 8])).Length == ([1, 2, 3, 4]).Length +([7, 8]).Length
```

The length property can distribute over the array concatenation if the (`Concat`) is replaced with (+). This means that if **we break an array in small chunks** and compute a property like length and then add the results this whould be the same as we computed the result to our initial array. This fact comes from the following idea of Monoid Homomorhism.

First imagine a category of monoids. As any category there should be **objects**, and **arrows**. The objects are all the various monoids we can construct. An interpretation for the **arrows** between those monoids is to consider them as **monoid-homomorhisms**. The arrows should respect the monoid structure.



Monoid Homomorhism

Monoid Homomorphism is a thing of beauty and a profound concept, that can be found everywhere in mathematics, logic, and programming. It captures the idea of **Preserving the Structure** when we map something from one monoid structure to another. The word *homomorphism* comes from the ancient Greek language: *ὁμός (homos)* meaning "same" and *μορφή (morphe)* meaning "form" or "shape".

A monoid homomorphism between two monoids ($\otimes$, M) and ($\odot$,N), **is a function :**

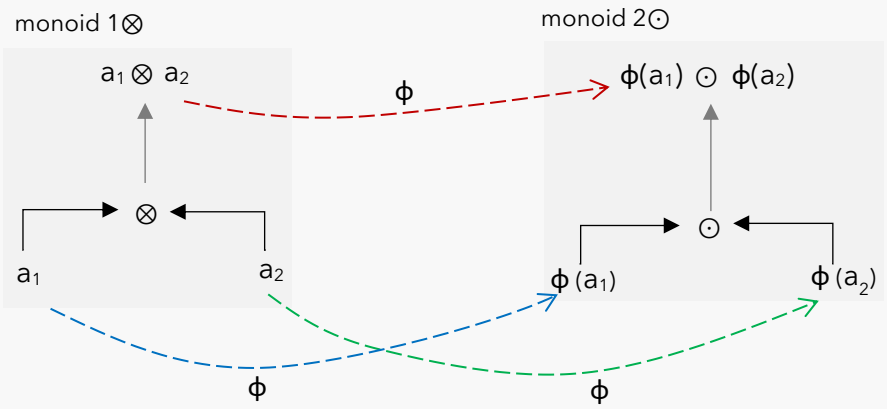
ϕ: M → N

Such that:

$\phi(a_1 \otimes a_2)=\phi(a_1) \odot \phi(a_2)$

$\phi(1_M)=1_N$

where 1_M and 1_N are the identities of N and M aka our monoid empties.



This means that if we map the a_1 ,a_2 and $a_1 \otimes a_2$ to the $\phi(a_1)$, $\phi(a_2)$ and $\phi(a_1 \otimes a_2)$ then the $\phi(a_1) \odot \phi(a_2)$ multiplication with the second monoid $\odot$ should be exactly equal to $\phi(a_1 \otimes a_2)$. The structure should be preserved.

Also the ϕ maps the identity element of the M monoid 1_M to the identity element of the N monoid.

Let us see another example:

```
("monoid" + "homomorphism ").Length == ("monoid".Length + "homomorphism ".Length);
```

Or this one:

```
var totalSum = array => array.Aggregate( 0 , (a, b) => a + b)

totalSum([1, 2, 3, 4].concat([7, 8])) ===totalSum([1, 2, 3, 4]) + totalSum([7, 8])
```

where the `totalSum` function is a Homomorhism over the Sum monoid `(0, (x, y) => x + y)` and the Array Monoids . The point of all this is that we can parallelize over the list by splitting it to smaller chunks and process them independently. This is the idea behind the famous pattern called mapReduce

if we have a big list, we can slice it and apply a function to each chunk, then fold each slice and then fold the results of the chunks.

```
var bigList = Enumerable.Range(1, 1000000);

Func<IEnumerable<int>, Func<int, IEnumerable<IEnumerable<int>>>> slice = list => s
ize =>
{
    var accumulation = new List<IEnumerable<int>>();
    for (int i = 0; i < list.Count(); i += size)
        accumulation.Add(list.Skip(i).Take(size));
    return accumulation;
};

var mapReduce = list => monoid => map =>
list.Select(chunk =>
chunk.Select(Convert.ToDecimal).Select(map).Aggregate(monoid.Empty, monoid.Concat)
   ).Aggregate(monoid.Empty, monoid.Concat);

var chunks = slice(bigList)(10000);
var sum = mapReduce(chunks)(new SumDecimal())(x => x + 1);
```

Run This: Fiddle

Of course C# already has embedded a degree of parallelism in TPL(Task Parallel Library) and the PLINQ extension of LINQ. The point of this section is why monoids are related with parallelism.

The theoretical support of the map reduce pattern came from the work of Richard Bird and Lambert Meertens that come up with the Bird–Meertens formalism and the Homomorhism lemma specifically for lists that say that a function *h* on lists is a list homomorphism :

h([])=e

h(a.concat(b))=h(a) • h(b)

if only if there exist a function f and an operation • so the following hold

h = reduce (•) ∘ map(f)

reduce (•) here means the operation (e1•…(e2(• (e3 • e_n))) folding the elements on a list. If there is a Homomorhism it should **be able to be decomposed** in a **map** followed by a reduce.

A Homomorhism in a list means Parallelism. Let's see another example.

Counting words in a large document is also something that can be done in parallel.

1. We can split the document in chunks and
2. then count the words in each chunk and add them in a dictionary.
3. Then we can merge the dictionaries because they form a monoid

Here the monoid is the dictionary as a `Dictionary`. We can prove that if we have a `IDictionary<TKey, TValue>` with keys of type `TKey` and values of type `TValue` and `TValue` **is a monoid** then we can create a `IDictionary` **monoid**:

```
    var dictionary1 = new Dictionary<string, int> {
                { "or", 2  },
                { "and", 3 },
                { "the", 4 }
            };
            var dictionary2 = new Dictionary<string, int> {
                { "or", 1  },
                { "and", 5 },
                { "jim", 1 },
                { "his", 1 }
            };


var merged = new DictionaryMonoid<string, int>(new Sum())
                                    .Concat(dictionary1, dictionary2);

public class DictionaryMonoid<TKey, TValue> : IMonoid<IDictionary<TKey, TValue>>
{
        private IMonoid<TValue> ValueMonoid { get; }
        public DictionaryMonoid(IMonoid<TValue> monoid) => ValueMonoid = monoid;
        public IDictionary<TKey, TValue> Empty => new Dictionary<TKey, TValue>();

public IDictionary<TKey, TValue> Concat(IDictionary<TKey, TValue> x, IDictionary<T
Key, TValue> y)
      {
            var merge = y.ToDictionary(entry => entry.Key, entry => entry.Value);
            foreach (var keyValue in x)
                if (merge.ContainsKey(keyValue.Key))
                    merge[keyValue.Key] =
                      ValueMonoid.Concat(merge[keyValue.Key], keyValue.Value);
                else
                    merge[keyValue.Key] = keyValue.Value;
            return merge;
```

```
        }
}

public class Sum : IMonoid<int>
{
   public int Concat(int x, int y) => x + y;
   public int Empty => 0;
}
```

Run This: Fiddle

this idea means that we **can parallelize the word counting problem, just because we were able to recognize a monoid.**

This page intentionally left blank.

3 Algebraic Data Types

"Lists come up often when discussing monoids,
and this is no accident: lists are the "most fundamental" Monoid instance."
Monoids: Theme and Variations (Functional Pearl)

Many of the most popular data structures like a list have definitions like this:

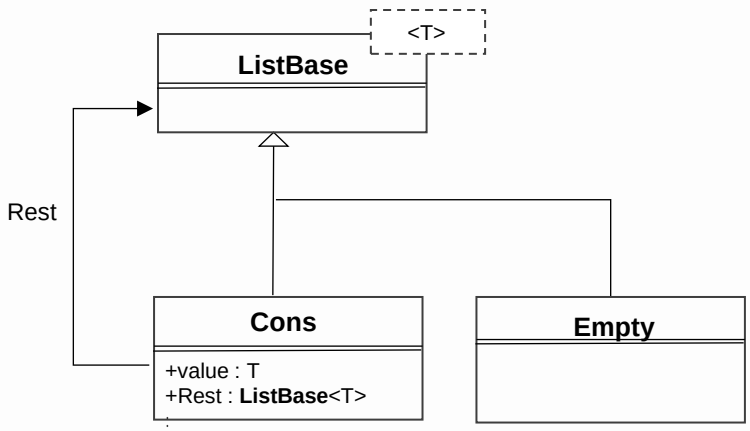
"A list is **either** an **empty list** or a **concatenation** of an **element** and a **list.**"

In the above sentence if we replace the word element with a, the either with +, and the concatenation with * we get the algebraic definition of a list:

List (**a**)= [ ] + **a** * **List**(**a**)

This definition is recursive because the term List appears on both sides of the definition. In C#, this translates to something like the following:

```
public abstract class ListBase<T> { }

public class Cons<T> : ListBase<T> {

        public ListBase<T> Rest { get; set; }
        public T Value { get; set; }

        public Cons(T value, ListBase<T> rest)
        {
            this.Value = value;
            this.Rest = rest;
        }
}

public class Empty<T> : ListBase<T> { }
```



```
var list = new Cons<int>(value: 2,
                         rest: new Cons<int>(value: 5,
                                             rest: new Empty<int>()));
```

This definition is the base definition of a List. Furthermore, because of its significance, we will often use this simple algebraic form of the List to display many functional ideas.

In practical terms giving recursive definitions means that we can progressively compose more complex entities out of simpler objects by applying some operations between them. The following three ways to combine structures are the most pervasive in programming.

1. **Product** [object-oriented equivalent: **composition**]: In our definition of the list the object literal that has two properties is a product
   `Cons(T value, ListBase<T> rest)`

2. **Coproduct** [object-oriented equivalent: **inheritance**]: In our definition of the list, `Cons<T> : ListBase<T>`, and `Empty<T> : ListBase<T>` form a co-product, commonly called a union type. In object-oriented terms, simple inheritance is a union type.

3. **Recursive Type** [object-oriented equivalent: **a type that contains objects of the same Type or inheriting type**]: The fact that in

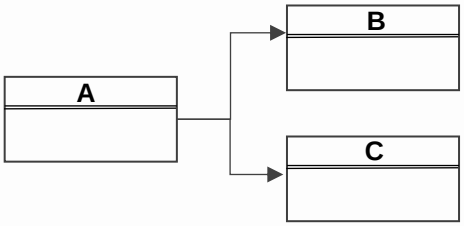
   `Cons<T> : ListBase<T> {public ListBase<T> Rest { get; set; }}`

   the `Rest` is a `ListBase<T>`Type means that we have a recursive definition.

The fact that a data type can be formed by applying some operations between them gives them the name **algebraic**. An **algebraic data type** is a kind of composite type.

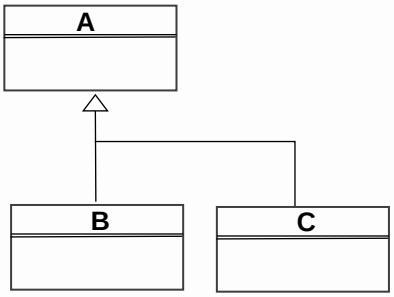
Now, if we define those three operations in any language or programming paradigm, we can define a list by following the algebraic definition **"**A list is **either** an **empty list** or a **concatenation** of an **element** and a **list."** . If someone tells us how to form a product a coproduct and a recursive type in Java or python or Elm we can instantly write a definition of a list without the need to be experts of the language.

Product type A= (B, C)



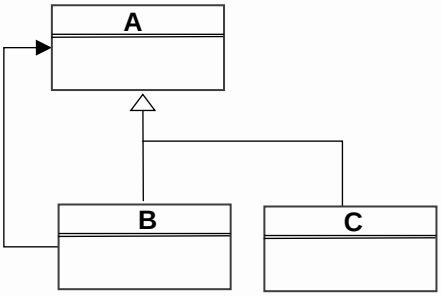
Has-a: **Composition**

Union type A= B + C



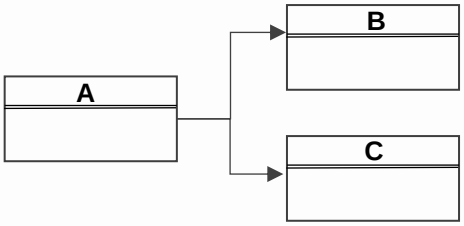
Is-a: **Inheritance**

Recursive Type A= (B, A) +C



3.1 The Product

The product of a family of objects is the "most general" object which admits a morphism to each of the given objects. The product must be both A and B and must be the least thing containing both those elements. In object-oriented programming we know product as composition.



We can form products using the available C# syntax in many ways, the following are products or arity-2 formed by two objects (arity is the number of items) but there can be products of any size

```
(int A, int B) product = (A: 10, B: 20);  //tuples are products.

public class Product                      //objects are products.
{
   public int A { get; set; }
   public int B { get; set; }
}

var product = new List<int> { 10, 20 };  //a list can be seen as a product.
```

All those are isomorphic, containing **the exact same information**, and we could construct functions that convert the one to the other.

That is why we can replace for example the argument list on a method with a Tuple or an object. The following are all equivalent:

```
public static void GetDiscount(int A, int B) { }
public static void GetDiscount((int A, int B) product) { }
public static void GetDiscount(Product product) { }
```

This transformation commonly called **Introduce Parameter Object** refactoring move. And it is based on the fact that both (,,,,) and Tuples `((,,,))` are products. The most important law in order to consider something as a product is that we must be able to write two functions that will take a product and gives us the component parts. Those functions are called **projections**.

For example, for an array definition of a product [,] the projections would be:

```
var product = new List<int> { 10, 20 };

Func<List<int>, int> first = p => p[0];
Func<List<int>, int> second = p => p[1];
```

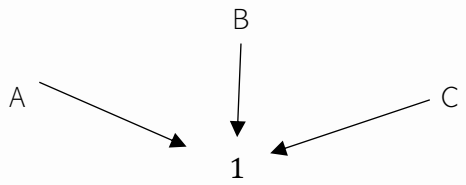
For the Tuple `(int A, int B)` definition of a product the projections would be:

```
(int A, int B) product = (A: 10, B: 20);

Func<(int A, int B), int> first = p => p.A;
Func<(int A, int B), int> second = p => p.B;
```

3.2 One optional

If we defined + and • maybe, we could define a 1 item that will be consistent with our usual algebra rules of a•1 = a for C# Arrays if we assume • is the Concat operation then 1 should be [] in order to hold a.Concat([]) = a . The term **1** is called the terminal object of a category.

Terminal object: The terminal object is the object with one and only one morphism coming to it from any object in the category.



In the category of Lists the terminal object can be the empty list [ ] because we can get a function to the empty list from any other list if we drop all the elements. However, we cannot have a single function to any other list without any ad hoc specific instructions. It is not a very interesting idea, but its a display of the consistency of algebra across different domains. For example for the algebraic definition of a list, we have that

$x = 1 + a \bullet x$

we can replace the x at the left part with the definition and get:

$x = 1 + a \bullet (1 + a \bullet x) = 1 + a + a \bullet a \bullet x$

and if we keep doing this, we get:

$x = 1 + a + a \bullet a + a \bullet a \bullet a \bullet \mathbf{x...}$

Which says that a list can be either empty 1 = [] or just an element [a] or two elements [a, a] or three elements [a ,a, a], and so on, which is very consistent with the definition of the list. Also, we could try to solve it like this:

$x \bullet (1 - a) = 1 \Rightarrow x = 1/(1 - a)$

However, since we do not have a definition of a division, we can use Taylor expansion on 1/(1-x), which is expanding to the geometric series $1 + a + a^2 + a^3 + a^4 + \ldots$

The consistency of the algebra is remarkable.

3.2.1 Lambda Calculus with Product Types optional

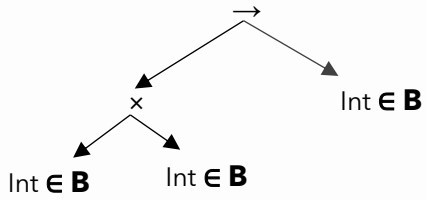
The next useful extension of the simply typed lambda calculus λ→ is to add products _×_ λ→,×

Types

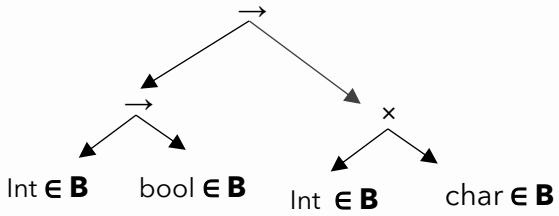
The only types in λ→ is just a set of base types **B** and anything else that is produced by this simple grammar in BNF form:

τ ::= τ→ τ | τ × τ |T where **T ∈ B**

(int × int) → int



(int→ bool) → (int × char)



The (**int** × **int**) → **int** in C# is the delegate:

```
Func<(int, int), int> f = ((int, int) a) => default(int);
```

Or in lambda notation now we can have lambdas where the variables are of product type

λx:(int× int).B: (**int**× **int**)→**int**

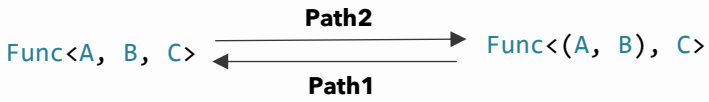
while the second one (**int**→ **bool**) → (**int** × **char**) would be:

```
Func<Func<int, bool>, (int a, char b)> g = (Func<int, bool> a) =>
                                              (a: default(int), b: default(char);
```

Also, we must treat `Func<(int,int), int>` and `Func<int, int, int>` as equivalent, because we can find two paths converting one to the other:

```
Func<A, B, C> Path1(Func<(A, B), C> f) => (A x, B y) => f((x, y));
```

```
Func<(A, B), C> Path2(Func<A, B, C> f) => ((A, B) p) => f(p.Item1, p.Item2);
```



For this reason, we must say those two syntaxes are equivalent within the C# type-system:

```
Func<(A, B), C> ≈ Func<A, B, C>
```

In lambda notation those two are represented by allowing multiple argument lambdas **λ(x,y).B** and also single variable lambdas where the variable is of product type **λx**:(**A** × **B**):

```
Func<(A, B), C>
```
λx:(**A**×**B**). **B**: (**A** × **B**)→**C**

```
Func<A, B, C>
```
λ(x:**A, y**:**B).B**: (**A** × **B**)→**C**

Introduction Elimination Rules

For the product there a single introduction rule :

$$\frac{\mathbf{x}:\mathbf{A} \qquad \mathbf{y}:\mathbf{B}}{(\mathbf{x},\mathbf{y}):\mathbf{A}\times\mathbf{B}}\times I$$

This is just a Constructor that takes an **A** and a **B** and returns an **A**× **B**

```
Func<A, B, (A, B)> Intro = (A x, B y) => (x, y);
```

The eliminations rules are two projections:

$$\frac{\mathbf{M}:\mathbf{A}\times\mathbf{B}}{\pi_1(\mathbf{M}):\mathbf{A}}\times E_1 \qquad\qquad \frac{\mathbf{M}:\mathbf{A}\times\mathbf{B}}{\pi_2(\mathbf{M}):\mathbf{B}}\times E_2$$

```
Func<(A, B), A> Pr1 = ((A, B) p) => p.Item1;
Func<(A, B), B> Pr2 = ((A, B) p) => p.Item2;
```

Computation Rules

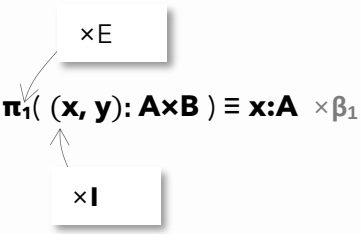
β-reductions

β-reductions for the product mean that if we have a product in its canonical form and we project the First we should get the First element and similarly for the second projection:

π₁((**x, y**): **A×B**) ≡ **x:A** ×β₁

π₂((**x, y**): **A×B**) ≡ **y:B** ×β₂

this says the coming from Gentzen**,** that the **elimination is post-inverse to the introduction** rule:



if we have a pair of Terms and use the constructor (aka Introduction) first to get a **(,)** and then immediate use the projection **π₁(·)** (aka Elimination, aka Destructor) then those should cancel out and get back the original element, for the appropriate Elimination. For C# Tuples this means that we can Provide constructions like the following:

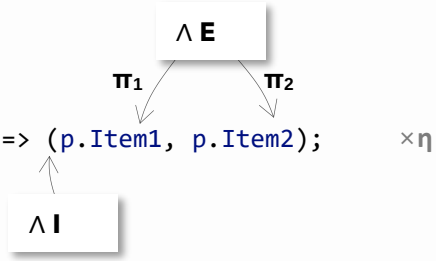
```
A Beta1<A, B>(A x, B y) => (x, y).Item1;
B Beta2<A, B>(A x, B y) => (x, y).Item2;
```

And ideally in some other setting we would like to be able to express that:

`(x, y).Item1 ≡ x` for-all x

η-reduction

```
var t = (1, 2);

var y = (t.Item1, t.Item2);

(A, B) Eta<A, B>((A, B) p) => (p.Item1, p.Item2);
```



$$\frac{\frac{A \wedge B}{A}\wedge E_1 \quad \frac{A \wedge B}{B}\wedge E_2}{A \wedge B} I\wedge$$

Recursor

> Rather than invoking this principle of function definition every time we want to define a function, an alternative approach is to invoke it once, in a universal case, and then simply apply the resulting function in all other cases. That is, we may define a function of type:
>
> Homotopy Type Theory: Univalent Foundations of Mathematics

Rec $_{A\times B}$**:** $(A\to B\to C) \to A\times B\to C$

This says if you give me a function (**A→B→C**) that takes an **A** and then a **B** and produces a **C** then if you give me something of a Product type **A×B** i could give you a **C**. This is the definition of **un-currying**. And its easy to implement in C#:

```
Func<(A, B), C> Rec(Func<A, Func<B, C>> f) => ((A, B) p) => f(p.Item1)(p.Item2);
```

Having this Recursor we can define everything else. For example, we can define the projections using the **Rec** $_{A\times B}$

```
Func<(A, B), A> Pr1<A, B>(Func<A, Func<B, A>> f) => Rec<A, B, A>(x => y => x);
Func<(A, B), B> Pr2<A, B>(Func<A, Func<B, A>> f) => Rec<A, B, B>(x => y => y);
```

We will use this product Recursor very often in practice, in the form of **Pattern matching** on objects that contain **compositions**.

optional

Curry–Howard correspondence

The Product is the equivalent to a logical AND or a conjunction **A ∧ B** of two propositions A and, B. In order to conclude that the proposition **A ∧ B** holds we must know that both A and B are true. Or equivalently if we can prove A and prove B then we can Prove **A ∧ B**. This rule is commonly named **introduction** in mathematical logic because it introduces a conjunction **∧** out of the previous propositions.

$$\frac{\mathbf{A} \qquad \mathbf{B}}{\mathbf{A} \wedge \mathbf{B}} \wedge I$$

In the same spirit, in order to create a product of a type A and a type B then we must have **an instance of both types** in order to call the constructor of product `var product = (x, y) => _`. This equivalence originates from the Curry–Howard isomorphism, which is probably the most profound connection of computer programs and mathematical proofs. If we have an A ∧ B then we can deduce any of the A or B. This is called conjunction elimination. From **A ∧ B** the following hold

$$\frac{\boldsymbol{A} \wedge \mathbf{B}}{\boldsymbol{A}} \wedge E_1 \text{ and } \frac{\boldsymbol{A} \wedge \mathbf{B}}{\boldsymbol{B}} \wedge E_2$$

or the propositions **A ∧ B → A** , **A ∧ B → B** are tautologies.

A product is "costly to construct" since we need all components but easy to use since by taking any of the components through a projection (`first`, `second`…) would be valid.

	Types	Logic
Introduction	$\dfrac{\mathbf{x}:\mathbf{A} \quad \mathbf{y}:\mathbf{B}}{(\mathbf{x},\mathbf{y}):\mathbf{A}\times\mathbf{B}}\times I$	$\dfrac{\mathbf{A} \quad \mathbf{B}}{\mathbf{A}\wedge\mathbf{B}}\wedge I$
Elimination	$\dfrac{\mathbf{M}:\mathbf{A}\times\mathbf{B}}{\boldsymbol{\pi_1}(\mathbf{M}):\mathbf{A}}\times E_1$	$\dfrac{\boldsymbol{A}\wedge\mathbf{B}}{\boldsymbol{A}}\wedge E_1$
	$\dfrac{\mathbf{M}:\mathbf{A}\times\mathbf{B}}{\boldsymbol{\pi_2}(\mathbf{M}):\mathbf{B}}\times E_2$	$\dfrac{\boldsymbol{A}\wedge\mathbf{B}}{\boldsymbol{B}}\wedge E_2$

Let us construct the `Curry` function from the previous chapter:

```
Func<A, Func<B, C>> Curry<A, B, C>(Func<A, B, C> f) => (A x)=>(B y) => f(x, y);
```

From the Logic perspective the implementation of the above `Curry` function corresponds in a Deductive Proof for a term $(\mathbf{A}\wedge\mathbf{B}\rightarrow\mathbf{C})\rightarrow(\mathbf{A}\rightarrow(\mathbf{B}\rightarrow\mathbf{C}))$

$$\dfrac{\dfrac{\dfrac{[\mathbf{A}]^x\ [\mathbf{B}]^y}{\mathbf{A}\wedge\mathbf{B}}I_{x,y}\wedge \quad [\mathbf{A}\wedge\mathbf{B}\rightarrow\mathbf{C}]^f}{\boldsymbol{B}\rightarrow\mathbf{C}}\rightarrow I_y}{\boldsymbol{A}\rightarrow(\boldsymbol{B}\rightarrow\mathbf{C})}\rightarrow I_x}{((\mathbf{A}\wedge\mathbf{B})\rightarrow\mathbf{C})\rightarrow(\boldsymbol{A}\rightarrow(\boldsymbol{B}\rightarrow\mathbf{C}))}\rightarrow I_f$$

Similarly, from the perspective of Type theory using simple Typed lambda calculus this corresponds to the following construction:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\boldsymbol{x}:\mathbf{A}]^1\ [\mathbf{y}:\mathbf{B}]^2}{(\mathbf{x},\mathbf{y}):\mathbf{AxB}}I_{x,y}\wedge \quad [\mathbf{f}:(\mathbf{AxB})\rightarrow\mathbf{C}]^3}{(\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})):\mathbf{C}}\Rightarrow \mathrm{E}}{\boldsymbol{\lambda y}:\boldsymbol{B}.(\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})):\boldsymbol{B}\rightarrow\mathbf{C}}\Rightarrow I_x}{\boldsymbol{\lambda x}:\boldsymbol{A}.\boldsymbol{\lambda y}:\boldsymbol{B}.(\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})):(\boldsymbol{A}\rightarrow(\boldsymbol{B}\rightarrow\mathbf{C}))}\Rightarrow I_f}{\boldsymbol{\lambda f}.\boldsymbol{\lambda x}:\boldsymbol{A}.\boldsymbol{\lambda y}:\boldsymbol{B}.(\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})):((\mathbf{A}\times\mathbf{B})\rightarrow\mathbf{C})\rightarrow(\boldsymbol{A}\rightarrow(\boldsymbol{B}\rightarrow\mathbf{C}))}$$

1. We make two assumptions **[x**:**A]** and **[y**:**B]** this is pretty much the introduction of the and we immediately invoke the Product constructor, to get a product **(x,y)**

$$\frac{[x\colon \mathbf{A}]^1\ [\mathbf{y}\colon \mathbf{B}]^2}{(\mathbf{x},\mathbf{y})\colon \mathbf{A}\times\mathbf{B}}\, I_{x,y}\wedge$$

2. We also assume a function **f**: **A**×**B** → **C**
3. We use the product to call the function f getting a **f(x,y)**
4. We then introduce a **λ** over the **y:B**

$$\frac{\big(f(x,y)\big)\colon \mathbf{C}}{\lambda y\colon B.\,\big(f(x,y)\big)\colon B\to \mathbf{C}}$$

5. Followed immediately by an **λ** introduction over the **x**:**A** and then **f**

$$\frac{\dfrac{\lambda y\colon B.\,\big(f(x,y)\big)\colon B\to \mathbf{C}}{\lambda x\colon A.\,\lambda y\colon B.\,\big(f(x,y)\big)\colon \big(A\to(B\to \mathbf{C})\big)}}{\lambda f.\,\lambda x\colon A.\,\lambda y\colon B.\,\big(f(x,y)\big)\colon \big((\mathbf{A}\times\mathbf{B})\to \mathbf{C}\big)\to\big(A\to(B\to \mathbf{C})\big)}$$

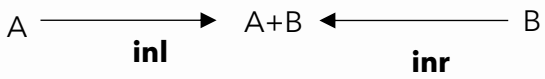
3.3 The Co-Product (aka Union)

"…If you only have studied conventional programming languages,
you never heard of **Sum types** before, because there is this gigantic blind spot…"

– Robert Harper
Oregon Programming Languages Summer School

The coproduct (or either-or sum type or **+**) is the dual structure of the product. The coproduct of a family of objects is essentially the "least specific" object [that is in linguistic terms an attempt to describe something similar to all of the objects it sums] to which each object in the family admits a morphism. It is the category-theoretic dual notion to the categorical product, which means the definition is the same as the product but with all arrows reversed.

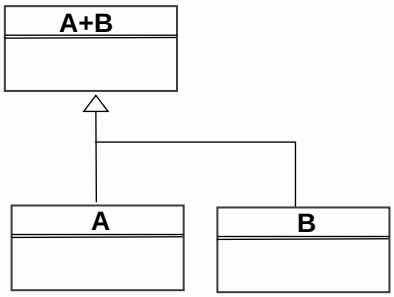
if a type S can be any of a set of types {A, B } then we can say that S is a sum type of A,B or **S**=**A+B**. In a strong typed language in order to assign any of **A**, **B to a variable S the A and B should be a Subclass of S.** This definition of a coproduct can be derived from the definition of the product if we reverse the arrows in the diagram.



Instead of projection the coproduct is defined by its **injections inl (for inject left) and inr (for inject right)**. There must be two functions that provide inclusion to the coproduct. **Injections are just the constructors of the subclasses in an Object-Oriented setting**.

To create a coproduct, we can provide **any one** of the Types it sums, in contrast to the product where we need **all** of them to call the constructor.

If we bend the arrows a little and name injections to subtype this looks like a valid diagram resembling the definition of coproduct



If we define as <: the notion of subtype, that is if A <: B, then **A can be used in place of B instead of B,** then <: this is a valid notion of coproduct. In C# terms, a valid subtype scheme is B<: A if A = {x_1, x_2 , x_3 ,…,x_n } and B= {x_1, x_2 , x_3 ,…,x_m } **where the top class has less properties n<m than the subtype object.**

This definition of subtyping, often seen as the Liskov substitution principle, became famous as the L letter of the SOLID principles.

Another way to discriminate would be to add a **tag** to each type and we are not going to do that here. There was always a demand in the C# community especially while functional programming gaining ground for Pattern Matching. Finally, in C# 8.0 pattern matching capabilities where included, allowing C# developers to write more elegant C# when it comes to union types.

Type Theory

Liskov Substitution Principle

Barbara Liskov described the principle in a 1994 paper as follows:

«*Subtype Requirement*: Let φ(χ) be a property provable about objects χ of type T. Then φ (y) should be true for objects y of type S where S is a subtype of T.»

This requirement defines a subtype relation as we saw previously **S<:$_L$ T** that can be called Behavioral Subtyping.

If ϕ(x) where x: T then if S<:$_L$ T => ϕ(y) should also hold for all y: S

Behavioral subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type

3.3.1 Lambda Calculus with Co-Product Types

We are going also to expand our simple Type system by adding a union types **A + B**.

Types

All possible types now are generated by the extended grammar:

τ ::= τ→ τ | τ × τ| τ + τ |T where **T ∈ B**

Introduction/Elimination Rules

we are going to use the inheritance as Union Type convention for this section because, this is how we will be using union- types through the book.

Introduction

There are two introduction rules:

$$\frac{\mathbf{x}: \mathbf{A}}{\mathbf{inl}(\mathbf{x}) : \mathbf{A} + \mathbf{B}} + \mathbf{I}_1 \qquad \frac{\mathbf{y}: \mathbf{B}}{\mathbf{inr}(\mathbf{y}) : \mathbf{A} + \mathbf{B}} + \mathbf{I}_2$$

To simulate those rules in C# we will define a simple hierarchy.

```
public class AB { }
public class A : AB { }
public class B : AB { }
```

where the `AB` behaves like the union of A and B: **A+B** under this assumption the In-Left (**inl**) is the A constructor `new A()`, and the In-Right (**inr**) is the B constructor `new B()`:

```
Func<A, AB> Inl = (A x) => new A();
Func<B, AB> Inr = (B x) => new B();
```

Elimination

The elimination rule for **+** is a bit more complex. Elimination means we must indirectly find out with which type was the union instantiated **A**+**B**. The elimination is called **Case** :

$$\frac{\mathbf{f}:\mathbf{A}\to\mathbf{C} \qquad \mathbf{g}:\mathbf{B}\to\mathbf{C} \qquad \mathbf{x}:(\mathbf{A}+\mathbf{B})}{\mathbf{Case}(\mathbf{x},\mathbf{f},\mathbf{g}):\mathbf{C}} +E$$

The C# implementation using `switch-case`

```
Func<AB, Func<A, C>, Func<B, C>, C> Case = (x, f, g) =>
    {
        switch (x)
        {
            case A a: return f(a);
            case B b: return g(b);
            default: throw new Exception();
        };
    };
```

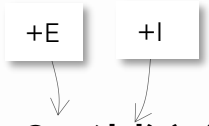
Run This: .Net Fiddle

Or using pattern matching with `switch`

```
Func<AB, Func<A, C>, Func<B, C>, C> Case = (x, f, g) =>
      x switch
          {
              A { } => f((A)x),
              B { } => g((B)x),
          };
```

Computation Rules

If we use the elimination after the introduction, they should cancel each other out:



Case(Inl(x) , f, g) ≡ f(x) :C $+\beta_1$

Case(Inr(y) , f, g) ≡ g(x) :C $+\beta_1$

This means if we use the case:

1. on a Left injection we get the f applied on the x.
2. on a Right injection we get the g applied on the y.

in C# using the above intro/elim definitions, we get two beta reductions:

```
Func<A, C> Beta1(Func<A, C> f, Func<B, C> g) => (A x) => Case(Inl(x), f, g);
Func<B, C> Beta2(Func<A, C> f, Func<B, C> g) => (B y) => Case(Inr(y), f, g);
```

Recursor

The recursor of the coproduct **Rec** $_{A+B}$ is exactly the **Case** seen above because the **+** is already generalized in its definition. The **Case** or **Rec** $_{A+B}$ type is:

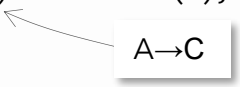
Rec $_{A+B}$: **(A→C)→(B→C) →A+B→C**

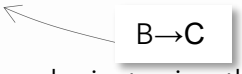
We are going to use the **Rec** $_{A+B}$ extensively again in the form of **Pattern Matching** on hierarchies.

Connection with Object-Oriented Polymorphism

In classic Object-oriented programing instead of giving the two functions f: **A→C** and g: **B→C** we **embed them onto the objects** with the same name, as the overrides of a base class method:

```
public abstract class AB<C> {
       public abstract C F();
}

public class A<C> : AB<C> {
    public override C F() => default(C);
}

public class B<C> : AB<C> {
    public override C F() => default(C);
}
```





Then we can rewrite the **Case** by just using the polymorphic method **F()** :

```
Func<AB<C>/,* Func<A, C>, Func<B, C>*/, C> Case = (/*f, g,*/x) =>
{
    switch (x)
    {
        case A<C> a: return a.F();
        case B<C> b: return b.F();
        default: throw new Exception();
    };
};
```

optional

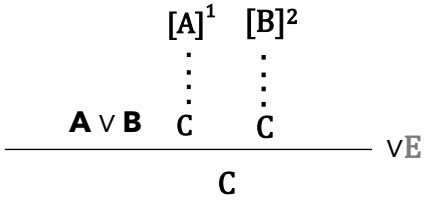
Curry–Howard correspondence

Co-Product is the equivalent to a logical OR - disjunction **A** ∨ **B** of two propositions A, B. In order to conclude that the proposition **A** ∨ **B** holds we must know that Either A or B is true. Or equivalently if we can prove A **or** prove B then we can Prove **A** ∨ **B**. This rule in commonly named disjunction ∨ **introduction**.

$$\frac{A}{A \vee B} \quad \text{or} \quad \frac{B}{A \vee B}$$

in the same spirit in order to create a coproduct of a type A and a type B then we must have **an instance of Any** of them. Going back to the logical OR - **A** ∨ **B** if we have an **A** ∨ **B** then we cannot deduce any of the A or B because we cannot know which one was initially

injected to the Union type. But we can eliminate disjunction indirectly over a third type C in the Following way:



or with an alternative notation:

$$\frac{A \vee B \qquad A \rightarrow C \qquad B \rightarrow C}{C}$$

This states that if we have a proof that gives us a **C** from an **A** (**A** →**C**) **and** a proof that gives us a **C** from a **B**, (**B** → **C**), **and** we have either of **A** or **B**, (**A** ∨ **B**) **then** we can deduce **C**.

3.4 Extending Union Types

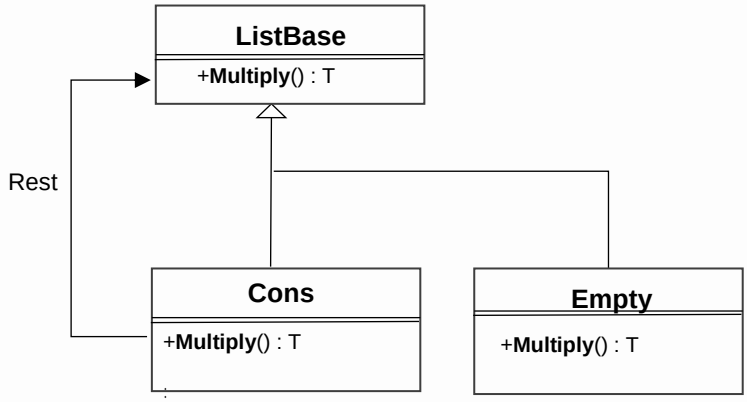
We will see in this book some ways to add behavior on Union types as we go along but for now let us see the default way: **using polymorphism**.

As i mentioned previously, If we want to add a new property/method to a Union of types, we must add the variations of the property/method in **all** the sub-types that can be in the Union. This is the use of default polymorphism in order to **discriminate** between the parts of the union:

```
public class Cons<T> : ListBase<T>
{
        ...
        public override T Multiply() => Value * Rest.Multiply();
}

public class Empty<T> : ListBase<T>
{       ...
       public override T Multiply() => 1;
}
```

Run This: .Net Fiddle

Here we have added `Multiply` in both parts of the union.



3.4.1 Adding Pattern Matching extensions to Union Types

Now let us add a new method called MatchWith, which we also sometimes come across as Case or Cata or even Fold.

```
public abstract class ListBase<T>
{
 public abstract T1 MatchWith<T1>(
       (Func<T1> Empty, Func<T, ListBase<T>, T1> Cons)  pattern);
}

public class Cons<T> : ListBase<T>
{
         ...
   public override T1 MatchWith<T1>(
      (Func<T1> Empty, Func<T, ListBase<T>, T1> Cons) pattern)  =>
          pattern.Cons(Value, Rest);
}

public class Empty<T> : ListBase<T>
{
        ...
   public override T1 MatchWith<T1>(
      (Func<T1> Empty, Func<T, ListBase<T>, T1> Cons) pattern) =>
         pattern.Empty();
}
```

Run This: .Net Fiddle

This allows us to distinguish between the two subtypes. We can use this `MatchWith` to extend **the base type without** modifying it.

3.4.2 Rewriting Union Type methods with MatchWith

Think of `MatchWith` like a universal definition that allows us to write Polymorphic method that must be added in all the branches of a union type **in a single place**.

We can now write any method definition in our `ListBase<T>` Union Type using just `MatchWith`. Here I rewrite the Multiply using an extension method:

```
public static T Multiply<T>(this ListBase<T> @this) =>
    @this.MatchWith(pattern: (
        Empty: () => 1,
        Cons: (value, rest) =>  value* rest.Multiply()
    ));
```

If you do not want to use extension methods, you can add the method at the Base class:

```
public abstract class ListBase<T>
{
     public T Multiply() =>
         this.MatchWith(pattern: (
             Empty: () => 1,
             Cons: (value, rest) => value* rest.Multiply()
   ));

  public abstract T1 MatchWith<T1>(
                      (Func<T1> Empty, Func<T, ListBase<T>, T1> Cons) pattern);
}
```

Run This: .Net Fiddle

We are going to use this way of extending Union Types extensively in this book.

3.4.3 The C#8.0 Pattern matching Feature

-"function follows form."

After C# 8.0 we have case-analysis capabilities on polymorphic types natively. We can now skip the custom implementation of the `MatchWith` and instead use the `switch` statement :

```
public static T Multiply<T>(this ListBase<T> @this,
            (Func<T, T, T> concat, T empty) monoid) =>
              @this switch
              {
                  Empty<T> { } => monoid.empty,
                  Cons<T> { Value: var value, Rest: var rest } =>
                           monoid.concat(value, rest.Multiply(monoid))
              };
```

Run This: .Net Fiddle

We are going to discuss more C# 8.0 pattern matching in the Functors example. But immediately you can see that you don't need to modify your types at all in order to add polymorphic functions. The only drawback is that you have to expose the properties that you want to access as `public` (alternatively you have to define type destructors that will decompose the type into the values)

3.5 A brief intro to Recursion

"Did you mean: **recursion**"
-Google Search

"**Hofstadter's Law**: It always takes longer than you expect,
even when you take into account Hofstadter's Law."
- Hofstadter's Law

Recursion occurs when a thing is defined in terms of itself or of its type. The importance of recursion in human thought, computer science and functional programming in particular is immense. We will be using Recursion throughout the book. For this reason, I won't go into an introductory explanation. Instead, we are going to have a light exploration of the Recursion in the C# type system. The easiest way to give a Recursive definition in C# would be :

```
public T F<T>() => F<T>();
```

unsurprisingly, if you try to call the function F() you get a Stack overflow. Since it generates an infinite …F(F(F(F (…..)))) call sequence. But we are not going to Run anything for now and **just explore the morphology of the recursion**. First let us add some input parameters. We could add an integer for example :

```
public T F<T>(int n) => F<T>(n);
```

it type-checks so we are ok. In the same sense we can add as many generic Types or functions e.g:

```
public T F<T, T1>(Func<T, T> f, Func<T, T1> g) => F(f, g);
```

Any change we make to the Definition we have to find something of the appropriate type to call the Function.

Let us use the function f. The only Requirement is that whatever we write Type-Checks. In order to use the f we need to call it by providing a T, well F(f) is of type T:

```
public T F<T>(Func<T, T> f) => f(F(f));
```

trivially you could use f as many times you wanted :

```
public static T F<T>(Func<T, T> f) => f(f(f(F<T>(f))));
```

we can now add also a T1 parameter. Nothing changes:

```
T F<T, T1>(Func<T, T> f, T1 n) => f(F(f, n));
```

now let's change the signature of the f in order to also takes another argument of type T1

```
T F<T, T1>(Func<T, T1, T> f, T1 n) => f(F(f, n), n);
```

Adding a T1 and use the n to call f

we can add as many arguments on f as we wanted:

Adding 3 T1 and use the n 3 times.

```
T F<T, T1>(Func<T, T1, T1, T1, T> f, T1 n) => f(F(f, n), n, n, n);
```

Run This: .Net Fiddle

Curried Versions

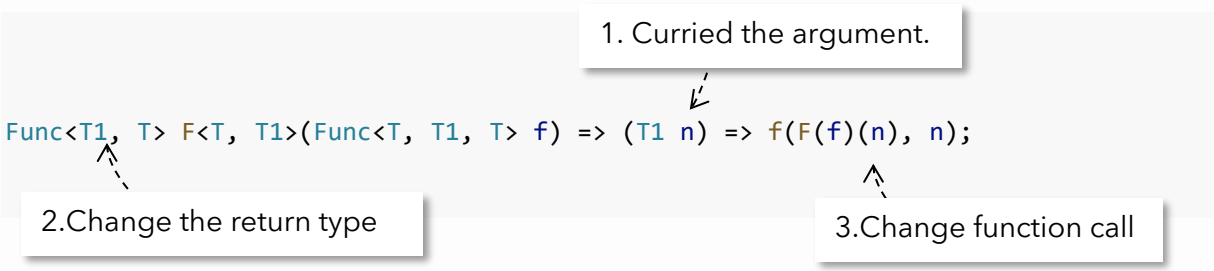
Let us see just for fun the Curried version of the Recursive function:

```
T F<T, T1>(Func<T, T1, T> f, T1 n) => f(F(f, n), n) ;
```

Curry this argument.

The steps are:

1. Curry the argument `(T1 n) =>`
2. Change the return type into `Func<T1, T>`
3. Change the function call `F(f)(n)`



Run This: .Net Fiddle

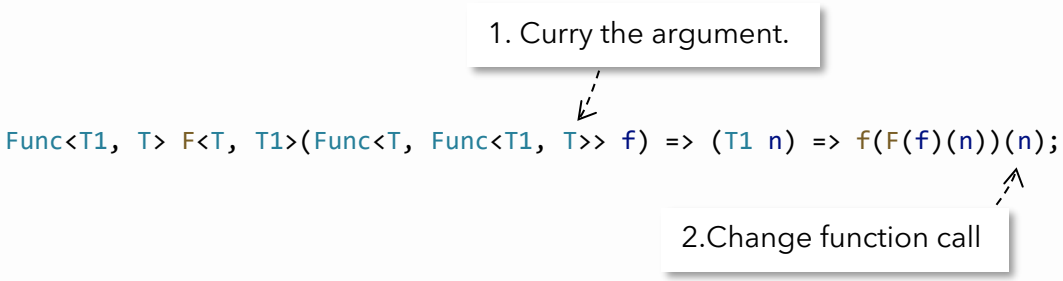
For maximum confusion let's also curry the function `Func<T, T1, T> f` inside the `F`

```
Func<T1, T> F<T, T1>(Func<T, T1, T> f) => (T1 n) => f(F(f)(n), n);
```

Let's Curry this `f` also.

The steps are:

1. Change the f type `Func<T, T1, T>` into the curried version : `Func<T, Func<T1, T>>`
2. Change the function call `f(..., n)` into: `f(...)(n)`



Termination

Of course, those recursive functions **do not terminate** if you decide to call them, you will get the usual stack overflow exception. To stop the recursion, we should have some sort of decreasing counter, that will signify that we reached a terminating state.

We can use many things as a counter. Any recursive data structure that we can unpeel in each recursive call will do. For example, we could use as a counter an array [ ] or a tree. Here we will use the integers to connect it with the next section.

For simplicity let us say that the `T1` type is `int`

```
T F<T>(Func<T, int, T> f, int n) => f(F(f, n), n);
```

we can stop the recursion by checking if `n` is 0 (or any other constant)

```
T F<T>(Func<T, int, T> f, int n) => n == 0 ? default(T) : f(F(f, n), n);
```

Obviously returning `default(T)` (that is `null`) is just a possibility but we can abstract this by adding another argument that we have to pass on the Function call:

```
T F<T>(Func<T, int, T> f, int n, T v) => n == 0 ? v : f(F(f, n, v), n);
```

Adding a default value `v` parameter for `T`

Run This: .Net Fiddle

Here if we abstract the _?_:_ with the following:

```
public static T1 MatchWith<T1>(
            this int @this,
            (Func<T1> Zero, Func<int, T1> Succ) pattern) =>
                (@this == 0) ?
                pattern.Zero() :
                pattern.Succ(@this - 1);
```

we can rewrite the function `F` as follows:

```
T F<T>(Func<T, int, T> f, int n, T v) =>
            n.MatchWith(
                pattern: (
                    Zero: () => v,
                    Succ: (predecessor) => f(F(f, predecessor, v), predecessor)
                   ));
```

Run This: .Net Fiddle

Using a List as a counter

I already mentioned that we could use an Array as a counter instead of an integer. This will make sure that the recursion terminates by passing in the recursive call smaller and smaller sections of the initial array. This will assure the termination of the recursion eventually.

Changed the `int` into `List<T>`

```
T F<T>(Func<T, List<T>, T> f, List<T> n, T v) =>
                n.MatchWith((
                        Empty: () => v,
                        Cons: (current, rest) => f(F(f, rest, current), rest)
                    ));
```

Terminating when List is empty.

3.5.1 Lambda Calculus Again- Gödel's System T optional

Gödel's System T is the simply typed λ-calculus with products $\lambda^{\rightarrow,\times}$, where a basic type $\mathbb{N}$ for numbers and a **Recursor** for the Naturals have been added.

For the Naturals numbers the typing rules of introduction are simple we have a zero 0 which is in Naturals $\mathbb{N}$ or if we have a term that is in naturals then its successor also is a Natural.

$$\frac{}{\mathbf{0} : \mathbb{N}}\,\mathbf{Zero} - \mathbb{N} \qquad \frac{\mathbf{n}: \mathbb{N}}{\boldsymbol{S(n)} : \mathbb{N}}\,\mathbf{Succ} - \mathbb{N}$$

While it has a single elimination rule, called Recursor **Rec**

$$\frac{\mathbf{n} : \mathbb{N} \qquad \mathbf{a}: \mathbf{A} \qquad \mathbf{c}: \mathbf{A} \rightarrow \mathbb{N} \rightarrow \mathbf{A}}{\mathbf{Rec(n, a, c)}: \mathbf{A}}\,\mathbf{Rec} - \mathbb{N}$$

The Recursor again here is the classic aggregate fold the **a:A** is the "seed" element (or the accumulator) and the **c: A →ℕ→A** is the aggregation function that takes the previous accumulation the current state of the counter **n:ℕ** and returns a new accumulation.

The Evaluation/β-reductions/computation rule for the Recursor is a case analysis on the current counter:

Rec(0, a, c):A $\rightarrow_{\beta}$ **a:A** β-ℕ

Rec(n, a, c):A $\rightarrow_{\beta}$ **c(Rec(n-1), n):A**

And that is precisely the C# Recurive function:

```
public static A Rec<A>(int @num, A a, Func<A, int, A> c) =>
            num == 0 ?
              a :
              f(Rec(num - 1, acc, c), num - 1);
```

Let's refactor this in order to take a familiar form. First we will add our Custom pattern matching (case analysis) on the definition of the Naturals:

```
public static T1 MatchWith<T1>(this int @this,
            (Func<T1> Zero, Func<int, T1> Succ) pattern) =>
                (@this == 0) ?
                pattern.Zero() :
                pattern.Succ(@this - 1);
```

this just hides the `_?_:_` operation. Now we can rewrite the Recursion above using the custom pattern matching `MatchWith` function:

```
public static A Rec<A>(int @num, A a, Func<A, int, A> c) =>
            @num.MatchWith(pattern: (
                        Zero: () => a,
```

```
                    Succ: (n) => c(Rec(n, a, c), n)
                ));
```

Run This: .Net Fiddle

The elimination Rule represents in System T the **Schema of induction**

The simplest form of mathematical induction infers that a statement **C** involving a natural number *n* holds for all values of *n*, by proving two cases:

1. The **initial** or **base case**: prove that the statement holds for 0. This means **C**(0) holds.
2. The **induction step**: prove that for every *n*, if the statement holds for *n*, then it holds for *n* + 1. **C**(**n**) → **C**(**n+1**)

The logical formal abbreviation of the schema is:

C(0) ∧ **∀n**. (**C**(**n**) → **C**(**n+1**)) → **∀n**. **C**(**n**)

to understand the reasoning of why the above function represents the induction keep in mind that when we program we are working in Type theory. In this setting proving a term of Type **A** is inhabited (aka can be constructed) means that we proved in logic that a Statement **C** holds.

Let us dissect the above function `Rec<A>` using the Induction steps as comparative guideline:

1. The **initial** or **base case**: prove that the statement holds for 0, **a:A** that is the `Zero: () => a` case.
2. The **induction step**, **inductive step**: prove that for every n, if the statement holds for n, then it holds for n + 1. **∀n.(C(n)** → **C**(**n+1**)) the type of this can be understood in the following way:
    a. If you give me any n [that is **∀n** which in type theory means a function that takes a n as input : **n: ℕ ⇒...**] in code: `Succ: (n) =>`
    b. And then you give me something where the property C holds **C**(**n**)**:A ⇒** In code `Rec(n, a, c)`.
    c. I can construct the next **C**(**n+1**)**:A** in code: `c(Rec(n, a, c), n)`.
    d. Stringing all this together we get : **ℕ ⇒ A ⇒ A** which is the **A⇒ ℕ ⇒ A** reordered. This is the type of the `Func<A, int, A> c`.

Finally, if we just attach this as an extension method on the integers we get a familiar method:

```
public static T Aggregate<T>(this int @this, T acc, Func<T, int, T> f) =>
                                                Rec(@this, acc, f);
```

Now we can write stuff like this:

```
var range = (5).Aggregate(new List<T>(), (acc, n) => (n, acc).AsList());
```

this starts with an empty list and then appends all the integers as the counter decreases. Let us run this computation step by step. Here i denote as * the operation of concatenating to the list:

Rec(5,[], *) → //we can apply.

[5, **Rec**(4,[], *)] →

[5 ,4, **Rec**(3,[], *)] → //we can

… //at some point we reach zero

[5,4,3,2,1, **Rec**(0,[], *)] → //we reached the base case, and the recursion stops.

[5,4,3,2,1,0]

Obviously, there is no reason to use (and should not use) the **Recursor**, since we can use a `for` loop. The intension of the section was to bring together the ideas of Gödel System T, mathematical induction, pattern matching and recursion.

3.6 Recursive Algebraic Types

The fact that our list is a recursive type means that when we try to construct a new method for our type, we must give an inductive definition (or recursive definition). So, let me create a map function that will apply a function f to all the values of the list:

```
public class Cons<T> : ListBase<T>
{
	...

	public override ListBase<T1> Map<T1>(Func<T, T1> f) => { ... }
}

public class Empty<T> : ListBase<T>
{
	public override ListBase<T1> Map<T1>(Func<T, T1> f) => { ... }
}
```

Firstly, because it is a coproduct, we must define the **map** in **both subtypes of the coproduct**. We start with the `Empty`, which the easiest. `Empty` is the base case of the structural induction. We know that the map must return something of the same type, so we will return a new `Empty` since there is nothing else that we could do with f.

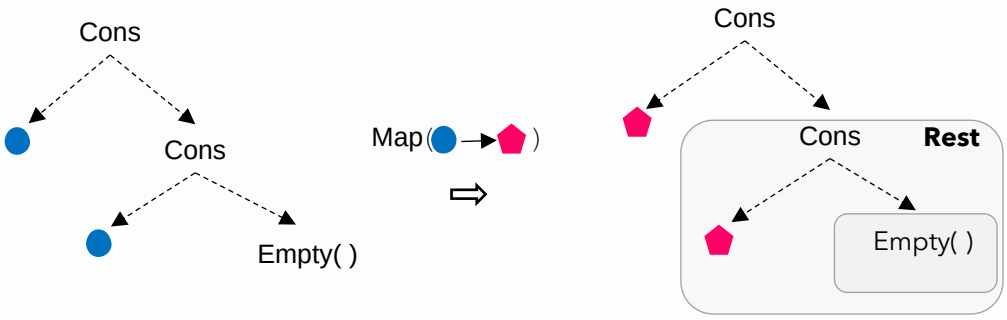
```
public override ListBase<T1> Map<T1>(Func<T, T1> f) => new Empty<T1>();
```

we can represent that symbolically with **Map** (f, [ ]) = [ ]

Now, let us move to `Cons`. `Cons` is a product (`Value`, `Rest`) the `Value` is something concrete so we can apply the f directly and get the lifted value `f(value)`. The second part `Rest` is a list, and the only thing we know is that it must have a **Map** function working as we

expect. This claim constitutes the inductive hypothesis, so we assume that `Rest.Map(f)` returns a lifted list for the `Rest.` We must finally return something of the same type, so we return a new `Cons`

```
ListBase<T1> Map<T1>(Func<T, T1> f) => new Cons<T1>(f(Value), Rest.Map(f))
```



we can represent that symbolically:

Map (f, [v, rest]) = [f(v), **Map** (f, rest)]

Furthermore, that is the correct implementation-defined recursively.

```
public class Cons<T> : ListBase<T>
{
       ...
       public override ListBase<T1> Map<T1>(Func<T, T1> f) =>
                              new Cons<T1>(f(Value), Rest.Map(f));
}

public class Empty<T> : ListBase<T>
{
       public override ListBase<T1> Map<T1>(Func<T, T1> f) => new Empty<T1>();
}
```

```
 var list = new Cons<int>(value: 2,
             rest: new Cons<int>(value: 5,
                   rest: new Empty<int>()));

var list2 =list.Map(x=>x+2);
```

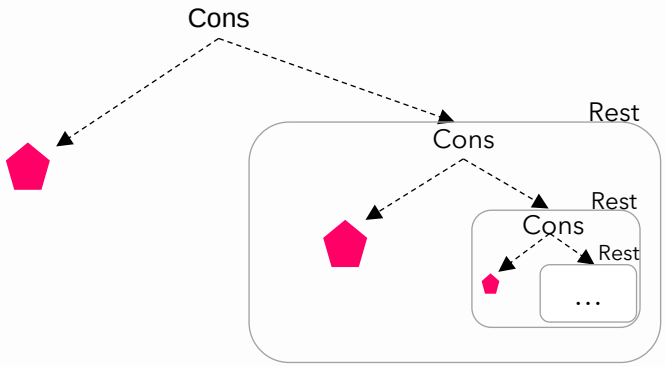
Run This: .Net Fiddle

3.7 Practical Structural Induction optional

"Informal proofs are algorithms; formal proofs are code."
-Logical Foundations

This section aims to convey the general idea behind Structural induction and how we can use it when we want to formulate methods from Recursive Algebraic Data structures.

Structural induction is used to prove that some proposition *P*(*x*) holds for all *x* of some sort of recursively defined structure, such as formulas, lists, or trees.

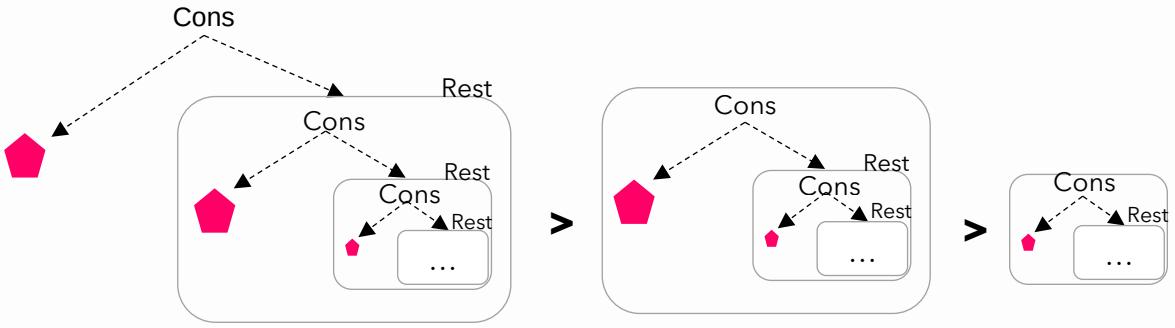


The structural induction generalizes the simple mathematical induction on Natural numbers, can be generalized within the context of Inductive types. Mathematicians are obsessed with the Natural numbers and it is of no surprise, that Gödel started with his **System T** Recusor from the Naturals. Nonetheless it soon became clear that from the point of view of Computer Science the Natural numbers is simple a recursive structure 0, S0, SS0…, which does not have any difference from other recursive structures like the lists and trees.

! For the structural induction there is a generalized form of the induction that says : If **x** is a Type of a Recursive Type T and **P(y)** is true for all **substructures y** , then **P(x)** must also be true.

[∀x: T [∀y: T [y **<:** x → P(y)]] → P(x)

Structural induction is a more general version of the Mathematical induction on the naturals. The nice thing with programming is that you can actually test your "proof" immediately by seeing that the types "fit" using the type-checking of the compiler and also that some example cases have the expected result at runtime.

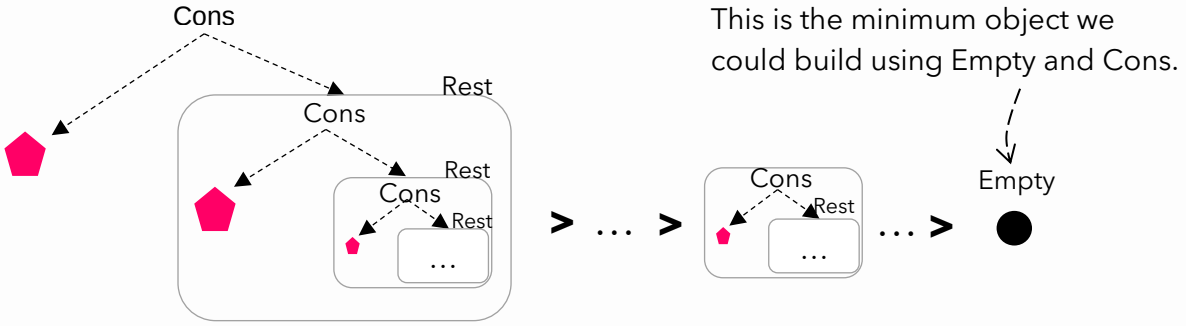


Let's revisit the Custom List implementation seen above, from the point of view of structural induction.

Base Case:

For the tree structure the **terminal element** is the empty list [] represented by the `Empty<T>` sub-class. This is **the smallest possible element** `Empty<T> <: List<T>`

```
public class Empty<T> : ListBase<T>
{
        public override ListBase<T1> Map<T1>(Func<T, T1> f) => new Empty<T1>();
}
```



Inductive Hypothesis:

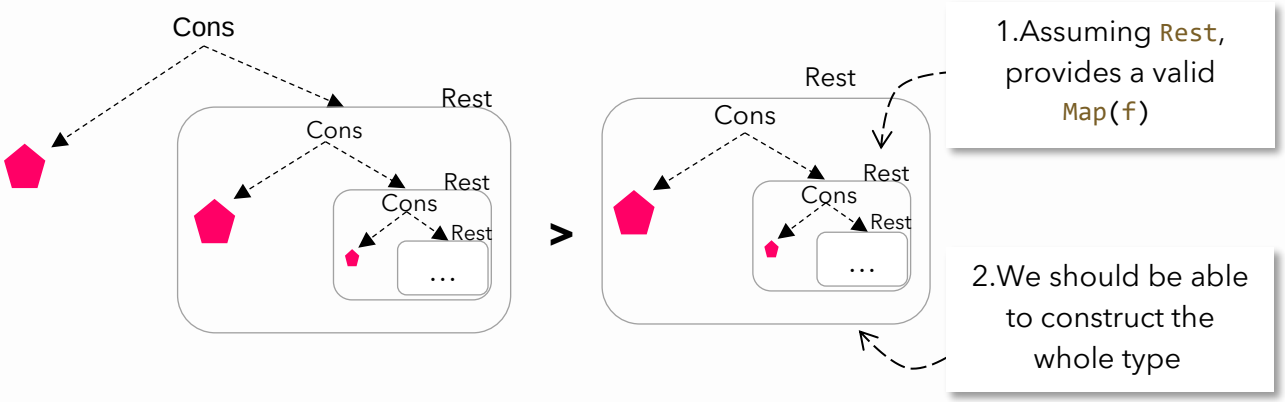
We want to "prove" `Map<T1>(Func<T, T1> f)` holds for `Cons<T>`. We do that unknowingly by providing a valid implementation for the method. Writing a program implementation is equivalent to a proof in mathematical logic, due to the Curry–Howard correspondence.

1. Assuming that we have, as a given that the `this.Rest.Map(f)` works correctly, and it's of Type `List<T1>`

2. We want to give a valid implementation that respects the types for the following:

```
public class Cons<T> : ListBase<T>
{
        public override ListBase<T1> Map<T1>(Func<T, T1> f) => □
}
```

We must fill this to fit the types.



3.7.1 Rewriting Map with MatchWith

And since we have a `MatchWith` method in place we can rewrite Map as an extension method:

```
public static ListBase<T1> Map<T, T1>(this ListBase<T> @this, Func<T, T1> f) =>
            @this.MatchWith<ListBase<T1>>(pattern: (
                Empty: () => new Empty<T1>(),
                Cons: (value, rest) => new Cons<T1>(f(value), rest.Map(f))
            ));
```

Or using the `switch`:

```
public static ListBase<T1> Map<T, T1>(this ListBase<T> @this, Func<T, T1> f) =>
            @this switch
            {
                Empty<T> { } => new Empty<T1>(),
                Cons<T> { Value: var value, Rest: var rest } =>
                            new Cons<T1>(f(value), rest.Map(f))
            };
```

Run This: .Net Fiddle

3.7.2 On the value of the symbolic representation

"In its most general form, algebra is the study of mathematical symbols and the rules for manipulating these symbols."
-Wikipedia:Algebra

You might want to ignore the symbolic representation, and focus on the code, or you can try to write your own **map** equation by induction. Let me write some more map definitions that i just came up with right now:

1. here I just added a → sign to point the term mapped, and I replaced concat with * the multiplication

map(f) → ([ ]) =[ ]

map(f) → ([value * rest]) = [f(v) * **map**(f)→ (rest)]

2. or this one where I have replaced fmap f with an underlining $(\_\_)_f$ because why not:

$([])_f = []$

$([value\ ,\ rest])_f = [\ f(v),\ \ ([\ rest])_f\ ]$

We can come up with a hundred different ways to represent the same thing. Thinking in graphical-symbolic terms while trying to figure out the mechanics behind some of the functional concepts, has been very valuable for me and I think it can help you too. Please take some minutes to write down your own map(f) equations for the list, on a piece of paper. The ability to abstract an idea into a symbolic representation allows us to move between paradigms and get a deeper understanding of the mechanics.

3.7.3 Adding Pattern Matching extension to List<T>

We will now add a `MatchWith` extension on the **native .net List<T> type in order to be able to use recursive definitions:**

```
public static T1 MatchWith<T, T1>
      (this List<T> @this, (Func<T1> Empty, Func<T, List<T>, T1> Cons) pattern) =>
                (@this.Count == 0) ?
                pattern.Empty() :
                pattern.Cons(@this[0], @this.GetRange(1, @this.Count - 1));
```

This definition will allow us to use pattern matching syntax for `List<T>` types:

```
public static int Multiply(this List<int> @this) =>
  @this.MatchWith (
     pattern: (
         Empty: () => 1,
          Cons: (v, r) => v * r. Multiply()
      ));
```

In functional languages is quite common to write definitions by simply pattern matching. Look at the Haskell definition of list-map:

map _ [] = []
map f (x:xs) = f x : map f xs

this is the way we defined it in section of recursive algebraic types

Map (f, [ ]) = [ ]

Map (f, [v, rest]) = [f(v), **Map** (f, rest)]

If we can rewrite map using pattern matching with `MatchWith`

```
public static List<T1> Map<T,T1>
            (this List<T> @this, Func<T,T1> f) => @this.MatchWith(
             pattern: (
                  Empty: () => new List<T1> { },
                  Cons: (v, r) => new List<T1> {f(v)}.Concat(r.Map(f))
         ));
```

When you start rewriting code using pattern-matching with `MatchWith` with lists gives an insight on the functional mindset. Let's see how to zip two Lists. In an imperative style we would have a for loop that

```
  for (let index = 0; index < a1.Length; index++) {
    const element1 = a1[index];
    const element2 = a2[index];
```

```
    zipped.Add(element1);
    zipped.Add(element2)
  }
```

In this situation I assume that the arrays are of the same length. If this is not true, we must use more if (…) statements. Now if we see our arrays in the form [x:xs] [y:ys] this reveals a very different perspective. The general case where `zip([x:xs],[y:ys]) =[x,y:zip(xs,ys)]`

```
public static List<T> Zip<T>(this List<T> @this, List<T> @a2) =>
            @this.MatchWith(pattern: (
                Empty: () => @a2,
                Cons: (x, xs) =>
                    @this.MatchWith(pattern: (
                        Empty: () => xs,
                        Cons: (y, ys) =>
                        new List<T> { x, y }.Concat(xs.Zip(ys))
                    ))));
```

Run This: .Net Fiddle

3.8 Recompose the List

This is the end of this chapter and what for the rest of the section we are going to see various Recursive definition on the List. All the examples have two cases:

1. The base case for the empty List **[]**
2. And a recursive case where we break up the List **[v , Rest]**, use the first element (head) **v** of the list, and then Repeat the recursive operation for the rest of the list **Rest**.

Let us first start with something simple. We will break down the list and reassemble it. We define symbolically a function **Rebuild** that just rebuilds the list without modifying it:

Rebuild ([ ]) = [ ]

Rebuild ([v , rest]) = [v, **Rebuild** (rest)]

An implementation would be:

```
List<int> Rebuild(List<int> @list)
{
     return @list.MatchWith((
                    Empty: () => new List<int> { },
                    Cons: (v, rest) => (v, Rebuild(rest)).AsList()
                   ));
}
```

Rebuild ([ ]) = [ ]

[v, **Rebuild** (rest)]

Here the `.AsList()` is a custom extension on types:

```
    public static List<T> AsList<T>(this ValueTuple<T, List<T>> @this) =>
        @this.Item1.AsList().Concat(@this.Item2);
```

Let us make an example "execution" of this simple Term Rewriting system with its two rules. Let's start with an initial state [1, 4,6] on which we will apply the reduction sequentially.

Rebuild ([1,4,6]) →

[1, **Rebuild** ([4,6])] →

[1,4, **Rebuild** ([6])] →

… //at some point we empty the list

[1,4,6, **Rebuild** ([])] → //we reached the base case and the recursion stops

[1,4,6]

As you can see what we did is just traverse the list one element at a time by recursion and used the list monoid (**[],***) to rebuild it.

Using the `Aggregate` Linq method on the `List<T>`

```
List<T> Rebuild<T>(List<T> @list) => list.Aggregate(
                    new List<T> { },
                    (acc, current) => (acc, current).AsList()
                );
```

Unfortunately, this reverses the List (this will happen even if you reverse the order of concatenation `(current, acc).AsList()`). The `Aggregate` is weaker than the recursion since we lose the order of traversal. `Aggregate` is great for the cases where we want to fold to things into less information e.g. Fold a list into a bool or an integer.

3.9 Passing More arguments

We can always pass more arguments into the recursive function trivially.

```
List<int> Rebuild(List<int> @list, Func<int,int> f)
{
    return @list.MatchWith((
                    Empty: () => new List<int> { },
                    Cons: (v, rest) => (v, Rebuild(rest, f)).AsList()
                ));
}
```

declare some f

pass down the f

Symbolically:

Rebuild ([ ], **f**) = [ ]

Rebuild ([v , rest], **f**) = [v, **Rebuild** (rest, **f**)]

If we use this on the head of the sub-list in each step we get the default Map:

Map ([ ], f) = [ ]

Map ([v , rest], f) = [f(v), **Map** (rest, f)]

3.10 Returning More items

In a similar manner we can modify the **Rebuild** function to return a tuple `(v, v * v)` pairing the value with its square

```
List<(int, int)> Squares(List<int> @list) =>
         @list.MatchWith((
                           Empty: () => new List<(int, int)> { },
                           Cons: (v, rest) => ((v, v * v), Squares(rest)).AsList()
                         ));
```

Symbolically:

Squares ([ ]) = [ ]

Squares ([v , rest]) = [(v, v^2), **Squares** (rest)]

Of course, we can do the same using only the **Map** function:

```
List<(int, int)> Squares(List<int> @list) => Map(@list, v => (v, v * v));
```

Or directly using `Select`:

```
List<(int, int)> Squares(List<int> @list)=> @list.Select(v =>(v, v*v)).ToList();
```

3.11 Filtering

Here we will add another `Func<T, bool> @predicate` parameter to the recursive Function, we are gong to use this `@predicate` to filter the Array using recursion. Symbolically, the **Filter** function:

Filter ([ ], predicate) = [ ]

Filter ([v , rest], predicate) = `if`(predicate (v))
[v , **Filter** (rest , predicate)]
`else`
Filter (rest , predicate)

This says:

1. If you have an Empty array just Return an Empty array.
2. If you have any other array:
    a. Check if the head of the array satisfy the predicate, if it does then then keep the head and append the Remaining Filtered array:
    [v , **Filter** (rest , predicate)]

    b. If it does not then just return the Remaining Filtered array and discard the element :
    Filter (rest , predicate)

```
List<T> Filter<T>(List<T> @array, Func<T, bool> @predicate)
    where T : IEquatable<T> =>
            @array.MatchWith((
                        Empty: () => new List<T> { },
                        Cons: (v, rest) =>  @predicate(v) ?
                                Filter(rest, @predicate) :
                                (v, Filter(rest, @predicate)).AsList()
                        ));
```

Filter ([ ], predicate) = [ ]

Filter (rest , predicate)

[v , **Filter** (rest , predicate)]

Run This: .Net Fiddle

3.12 Inserting items in an Ordered List

In this case we want to add an Item on an Already ordered list while preserving the order of the list.

Insert ([ ], s) = [ ]

Insert ([v , rest] , s) = if(v>s) [s, v, rest] else [v , **Insert** (rest , s)]

This says:

1. if you have an Empty array just Return an Empty array.
2. if you have any other array:
    a. Check if the head of the array **v** is greater that the element to insert **s**, **v>s**, if it is then we must pre-pend the element and finish the recursion [**s, v, rest**].
    b. Else if the head of the array **v** is smaller that the element to insert **s**, **v<s it** must be inserted somewhere in the rest of the list.

The implementation is straightforward:

```
List<T> InsertInOrdered<T>(List<T> @ordered, T @elementToInsert)
where T : IComparable<T> =>
@ordered.MatchWith((
```

```
        Empty: () => elementToInsert.AsList(),   //[t]
        Cons: (currentValue, restOfList) =>
          currentValue.CompareTo(elementToInsert) > 0 ?
              (elementToInsert,currentValue).AsList().Concat(restOfList):
              (currentValue,InsertInOrdered(restOfList, elementToInsert)).AsList()
        ));
```

[s, v, rest]

[v , **Insert** (rest , s)]

Run This: .Net Fiddle

3.13 Insertion Sort

Finally using the `InsertInOrdered` we can give a recursive implementation for the insertion sort algorithm on the list:

```
public List<T> Sort<T>(List<T> @unordered)
       where T : IComparable<T> =>
           @unordered.MatchWith((
                           Empty: () => new List<T> { },
                           Cons: (current, rest) =>
                               InsertInOrdered(
                                       ordered: Sort(rest),
                                       elementToInsert: current
                                   )
                               ));
```

By induction we assume that this is already sorted

Run This: .Net Fiddle

1. If you have an Empty array just Return an Empty array.
2. If you have any other array, just take the head of the list `current` and use the `InsertInOrdered` to add it in the ordered rest of the list `Sort(rest)` by calling `InsertInOrdered(Sort(rest), current )`

The magic here is that by induction we assume that `Sort(rest)` is already shorted.

In the following table, there is a summary of a progression of algebraic data types from simplest to more complex Zero => numerals=>Lists=>Trees=>..

Algebraic data Types

Name	Signature
Natural numbers	`public class Succ : Numeral` `{`

[Peano numerals]

```
  public Numeral Rest { get; set; }
}
public class Zero : Numeral{}

 var fourNumeral = new Succ(rest: new Succ(rest: new Succ(
rest: new Succ(rest: new Zero())))
```

List

```
public class Cons<T> : ListBase<T>
{
   public ListBase<T> Rest { get; set; }
   public T Value { get; set; }
}

public class Empty<T> : ListBase<T>{}

var list = new Cons<int>(value: 2,
              rest: new Cons<int>(value: 5,
                   rest: new Empty<int>()))
```

Tree

```
public class Node<T> : Tree<T>
{
  public Tree<T> Left { get; set; }
  public T Value { get; set; }
  public Tree<T> Right { get; set; }
}

public class Leaf<T> : Tree<T>
{
  public T V { get; }
}

var tree = new Node<int>(new Node<int>(new Leaf<int>(1), 2
, new Leaf<int>(3)), 4,
new Node<int>(new Leaf<int>(5), 6, new Leaf<int>(7)))
```

Sidenote:

Those algebraic data structures can become more and more complex but always have a terminating condition represented by the Zero, Empty, Leaf etc. On the opposite side, there are the so-called co-inductive or co-recursive data structures, which are the dual constructions and usually are infinite.

Functors

"It should be observed first that the whole concept of a category is essentially an auxiliary one; Our basic concepts are essentially those of a functor and of a natural transformation."
S. Eilenberg and S. MacLane

The Idea:

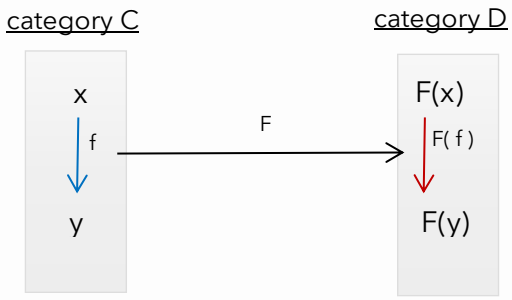
In C# the most famous functional programming idea is to use List.Select to replace iterations instead of *for loops* in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section.

"Practically a Functor is anything that has a valid .**Map(f)** method."

Functors can be considered the core concept of category theory.

4 Functors

In mathematics, a **functor** is a map between categories.



This Functor F must map two requirements.

1. Map each object x in *C* with an object F(x) in *D*,
2. Map each morphism f in *C* with a morphism F(f) in *D*

For object-oriented programming, the best metaphor for functors is a ***container***, together with a **mapping** function. The Array as a **data structure** is a Functor, ***together*** with the **map** method. The **map** is the array method that transforms the items of the array by applying a function f.

4.1 The Identity Functor

We will start by looking at the minimum structure that qualifies as a functor in C#:

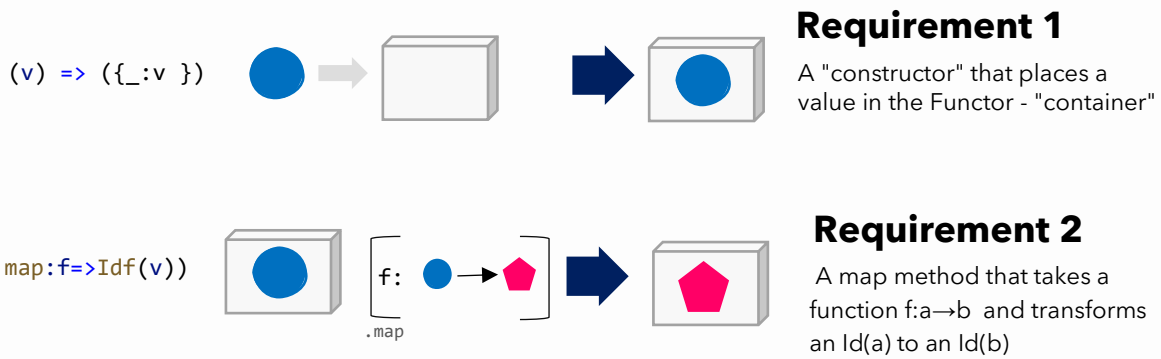
```
public class Id<T>
{
      public T Value { get; set; }
      public Id(T value) =>  Value = value;
      public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));

}
```

that's requirement 1

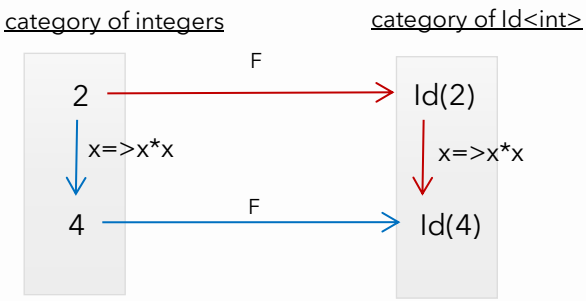
that's requirement 2

This is the minimum construction that we could call a functor because it has exactly two things:

1. A "**constructor**" that maps a value v to an object literal `Id<T>`
2. and it has a **mapping** method `Map<T1>(Func<T, T1> f)` that lifts functions f



Because it's the minimal functor structure it goes by the name **Identity functor**. Let us see a simple example where we have two integers 2 and 4 (here we take for simplicity the category of integers as our initial category C) also in this category there is the function `square = x=> x*x` that maps 2 to 4.



If we apply the `Id(_)` constructor we can map each integer to the `Id<int>` category. For example, 2 will be mapped to `Id(2)` and 4 maps to `Id(4)`, the only part missing is the correct lifting of the function **f** `Id( f )` to this new category. It is easy to see intuitively that the correct mapping is:

```
Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value))
```

Because if we use this map method with the squaring function `x=>x*x` we will get as a result, an `Id (4)`:

```
var square  = x=>x*x;

Id(2).Map(square) === Id(square(2))
```

Discussion on the Types

Usually when one looks at the map method in the context of Pure functional languages is represented - **map**: (**a** → **b**) → **f**(**a**) → **f**(**b**) where **f** is a Functor and **a**, **b** are types - (eg. Haskell, Scala[Cats], Sanctuary.js, Ramda.js).

We can get this type if we completely extract the method as a function that acts on the object like below:

```
Func<Id<T>,Id<T1>> Map<T,T1>(Func<T, T1> f) =>
                                        (Id<T> @this) =>
                                              new Id<T1>(f(@this.Value));
```

Here the function map has the type (**T** → **T1**) → **Id<T>** → **Id<T1>**

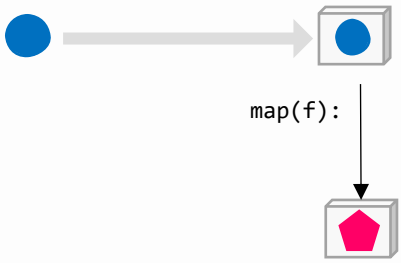
if we rearrange the terms into this form **Id**<**T**> → (**T** → **T1**) → **Id**<**T1**> (this is the form in Fantasy land spec) which is the Object oriented version which we can be interpreted in this way : if you give me a function from a to b (**a** → **b**) and I have an **Id**<**T**>, I can get an **Id**<**T1**>. The **Id**<**T**>, in this case is the object (`this`) that contains the method.

4.2 Commutative Diagrams

There is one important thing about the mapping function, though, the mapping should reach the same result for Id(**4**) if we take any of the two possible routes to get there. This means **map** (aka lifting of a function from C to D) **should preserve the structure of C.**

1. We could first get the Functor and then map **f**. This is the red path on the diagram.

```
Id(y) = Id(x).Map(f);
```

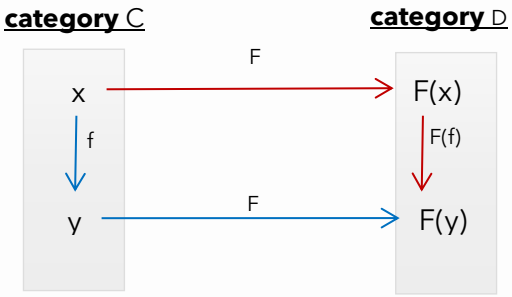


2. Or first lift 2 with **f** and then get the Functor.

```
Id(y) = Id(f(x));
```

Two objects that were connected with an arrow in category **C**, should also be connected with an arrow in category **D**. And reversely, if there was no connection in **C** there should be no connection in **D**.

When the red and blue paths give the same result, then we say that the diagram **commutes**



Moreover, that means that the lifting of morphisms (aka arrows, aka functions in programming) preserves the structure of the objects in **C**.

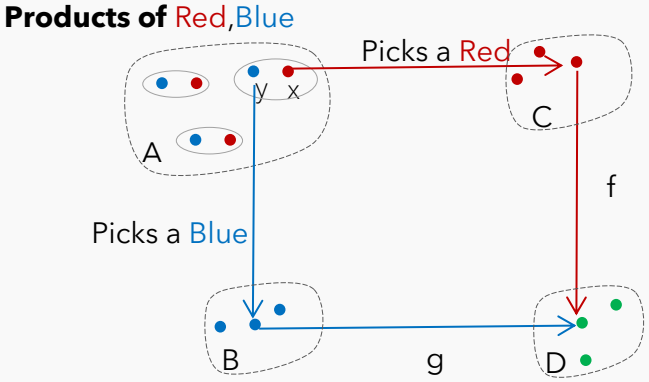
In practical day to day programming, commutating diagrams means that **we can exchange the order of operations and still get the same result.** It is not something that happens automatically and is something extremely helpful to have when coding.

Category Theory

Commuting Diagram

In category theory the commuting diagram represents in a way the **concept of equality**. A **commutative diagram** is a diagram such that all directed paths in the diagram with the same start and endpoints lead to the same result. It is said that commutative diagrams play the role in category theory that **equations** play in algebra.

For example in the case of a Pullback where the you can imagine that the A is some kind of a product between Red and blue items. in order for both paths to map to the same green item the following must hold **f(x)=g(y)**



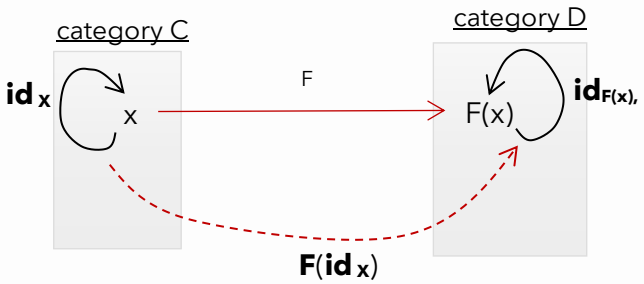
4.3 The Functor Laws

Every data structure has not just a signature, but some expected behavior. For example, a Stack has a **push** and a **pop** operation that specify a Stack functionality, and we expect these operations to behave in a certain way:

1. for all x, it holds that **pop** (**push** x **empty**) = **empty** - if the stack is empty then push something and then pop the last item the stack must remain empty.
2. etc

The laws for a given data structure usually follow from the specifications for its operations. Most of the constructions we are going to discuss in this book come with laws. Any functor must obey two laws when lifting the morphisms:

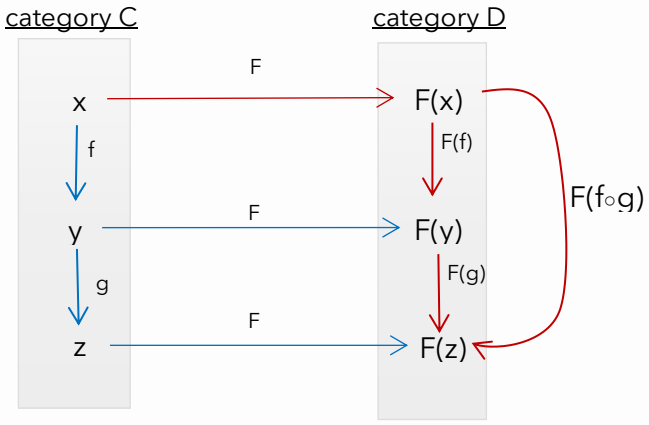
1. **$F(id_x) = id_{F(x)}$**, This means that the functor **preserves the Identity**.
   The functor should map the id function x=>x correctly



this simply means that `Id(value).Map(x=>x) ≡ Id(value).` Or in a Visual form:

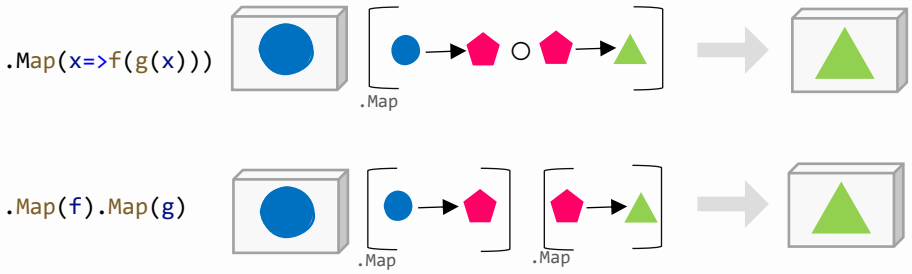
Who would expect that? This is so Obvious. This is a testament of the ability of mathematicians to complexify things.

2. **F(f∘g) = F(f) ∘ F(g)** This means that the functor **preserves the composition of functions**.



F(f∘g) = F(f) ∘ F(g)

This simply states that for any functions f and g the following must hold `functor(value).map(x=>f(g(x))).value ≡ functor(value).map(f).map(g)`.
Or in a Visual form:



Again, those two laws seem that are mathematically strict, but in real software development, those two conditions express a **well-designed map method that behaves as someone would expect it to behave**.

we expect for example that those two expressions should give the same result:

```
new[] { 2.0, 3, 4, 5 }.ToList().Map(discountedPrice.Compose(title));
```

```
new[] { 2.0, 3, 4, 5 }.ToList().Map(discountedPrice).Map(title);
```

More generally for all f and g the following should hold:

```
// Law 1-identity preserving fmap id = id
Id(value).Map(x=>x).value ≡ (value)

// Law 2-composition of functions is preserved fmap (f . g)== fmap f . fmap g
Id (value).Map(x=>f(g(x))).value ≡ Id (value).Map(f).Map(g).value
```

Run This: .Net Fiddle

by the way, I hope the composition law $\mathbf{F(f} \circ_{\mathbf{f}} \mathbf{g)} = \mathbf{F(f)} \circ_{\mathbf{F}} \mathbf{F(g)}$ reminds you of the Homomorhism equation $\phi(a_1 * a_1)=\phi(a_1) \bullet \phi(a_1)$ because that is what it is. Homomorhism are everywhere.

4.4 Pattern Matching

Pattern matching is a computer science concept that means different things in different contexts. And in many languages, one cannot explain it within the syntax of the language because of the syntax itself. This is again the Whorf hypothesis stating that *"the structure of a language affects its speakers' world view or cognition, and thus people's perceptions are relative to their spoken language."* If we transfer that in the realm of software engineering, **we can say that a developer can only think about code in relativity to the syntax of the programming language they use**.

Nonetheless since C# 8.0, finally the concept of pattern matching has been acknowledged as a true feature, we can hope that it will become an intuitive concept from the mainstream C# developer in the future.

After we wrap a value in a functor and manipulate it with the map, **eventually, we want to get the value out of the functor** elegantly and consistently. Until now, we could directly access the value because we had stored it explicitly in our object literal:

```
public class Id<T>
{
    public T Value { get; set; }
    public Id(T value) => Value = value;
}
```

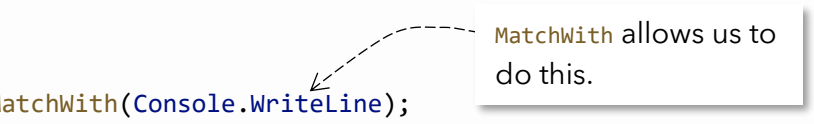
Not very pretty. Especially when everyone can access it and mutate it. As object-oriented programmers, we do not see any problem with that, but it could drive a functional programmer mad.

Nonetheless, **exposing the value is always an option if we do it consciously**. However, there are some more orthodox ways to extract a value from a Functor, and one of them is pattern matching. We will also see Folds.

For example, how can we extract the value out of an Id functor? Provide a method (`cata`) that takes a callback (`alg`) [alg is a short name for algebra, hinting the maths behind folds], to which we pass the value:

```
public class Id<T>
{
```

```
        public T Value { get; set; }
        public Id(T value) => Value = value;

        public T1 MatchWith<T1>(Func<T, T1> callback) => callback(Value);

        public void MatchWith(Action<T> callback) => callback(Value);
}
```




```
new Id<int>(3). MatchWith(Console.WriteLine);

Console.WriteLine( new Id<int>(3).value)    // Instead of this
```

Run This: .Net Fiddle

If we want to get just the value we can pass the identity function inside

```
var value  =new Id<int>(3).MatchWith (x=>x);
```

here we are going to use the for the first time the C#8.0 pattern matching in a trivial way. Take a look :

```
string result = new Id<int>(5)
                    switch
                     {
                         Id<int> { Value: var v } => v.ToString(),
                     };
```

Run This: .Net Fiddle

Using the following `switch` syntax:

```
switch
{
      _=>
};
```

We can peak inside the `Id<int>( )`

```
switch
{
    Id<int> {   } =>  _
};
```

and match the value (access the value, make a claim about how the `Id<int>( )` looks if dissected)

```
switch
{
    Id<int> { Value: var v } =>  _

};
```

In some pure Functional language this could look like this :

Id(5) = Id(x)

And immediately the x inferred as a 5 because the left side must match as a pattern the right side. The C# syntax is more verbose.

Pattern match over Composition

Let us add another Property inside Id and Make it into a Pair "container" (basically that is a custom Tuple)

```
public class Pair<T>
{
        public T Value1 { get; set; }
        public T Value2 { get; set; }
        public Pair(T value1, T value2) { Value1 = value1; Value2 = value2; }
        public Pair<T1> Map<T1>(Func<T,T1> f) =>new Pair<T1>(f(Value1),f(Value2));
}
```

we can define `MatchWith` as:

```
public T1 MatchWith<T1>(Func<T, T, T1> callback) =>
            callback(Value1, Value2);
```

this is just an Object-oriented composition of `Value1` **and** `Value2`, or in other interpretation a product type of `Value1` **and** `Value2`. We can use now the `MatchWith` to extract the values:

```
string pairResult1 = new Pair<int>(4, 5)
                     .MatchWith((v1, v2) => (v1 + v2).ToString());
```

Using C# 8.0 pattern matching

If we use the native syntax this becomes :

```
 string pairResult = new Pair<int>(4,5)
             switch
               {
                 Pair<int> { Value1: var v1, Value2:var v2} => (v1+v2).ToString(),
               };
```

Run This: .Net Fiddle

Pattern match over Inheritance

Let us say now that you we have a simple hierarchy of shapes:

```
abstract class Shape { }
```

```
class Rectangle : Shape
{
     public int Width { get; set; }
     public int Height { get; set; }

}

class Circle : Shape
{
      public int Radius { get; set; }
}
```

Now if i make an assignment to a `Shape` variable . **I can assign either a rectangle or a circle.** Here i will go with the rectangle:

```
Shape shape = new Rectangle { Width = 100, Height = 100, };
```

How can you find out during runtime what kind of shape did i put in ? Use pattern matching of course. From the inheritance perspective usually, we use words like **case analysis**.

We can go with the traditional usage of `switch` since C# 7.0

```
  string GetShapeDescription(Shape shape)
        {
            switch (shape)
              {
                        case Circle c:
                            return $"found a circle with radius {c.Radius}";
                        case Rectangle s:
                            return $"Found {s.Height }x {s.Width} rectangle";
                        default:
                            throw new System.Exception(nameof(shape));
               }
        };
```

Great but in the same spirit with the previous section we can define a `MatchWith` that take some Callbacks that expose the values:

```
abstract class Shape
{
     public abstract T MatchWith<T>(Func<Rectangle, T> rectangleCase,
                                    Func<Circle, T> circleCase);
}

class Rectangle : Shape
{
  public int Width { get; set; }
  public int Height { get; set; }

  public override T MatchWith<T>(Func<Rectangle, T> rectangleCase,
```

```
                                    Func<Circle, T> circleCase)
                                      => rectangleCase(this);
}

class Circle : Shape
{
   public int Radius { get; set; }
   public override T MatchWith<T>(Func<Rectangle, T> rectangleCase,
                                  Func<Circle, T> circleCase)
                                     => circleCase(this);
}
```

Which we can use like this:

```
Shape shape = new Rectangle { Width = 100, Height = 100, };

var description = shape.MatchWith<string>(
            rectangleCase: r => $"Found {r.Height}x {r.Width} rectangle",
            circleCase: c => $"found a circle with radius {c.Radius}"
      );
```

Using C# 8.0 pattern matching

As you can imagine we can write the same in C#8.0 finally without having to create custom pattern matching definitions like this:

```
  string GetShapeDescription(Shape shape) =>
       shape switch
       {
          Rectangle { Height: var h,Width: var w} => $"Found {h }x {w} rectangle",
           Circle { Radius: var r } => $"found a circle with radius {r}",
       };
```

Run This: .Net Fiddle

And use it like this:

```
Shape shape = new Rectangle{Width = 100, Height = 100,};

var description = GetShapeDescription(shape);
```

Run This: .Net Fiddle

We are going to see pattern matching in many Functors that have an Inheritance structure.

4.5 Id<T> Functor on the Fly

We can use extension methods to allow the creation an `Id<T>` from a value `T` and use the Map immediately and then Fold.

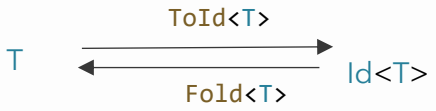
```
public static Id<T> ToId<T>(this T @this) => new Id<T>(@this);
```

And thus write:

```
(5).ToId()
.Map(x => x + 1)
.MatchWith(Console.WriteLine);
```

Is this useful? maybe, maybe not. but it signifies a way of writing that is very common in Functional programming using chaining and using transformation methods like `.ToId()` In order to move from one data structure into another.

The Id<T> and the T **are isomorphic**. **They contain the exact same information** and thus we can go back and forth without losing any information or needing any additional information by using the methods `ToId<T>` and `Fold()` (which are inverses in a sense)



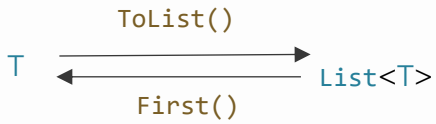
throughout this book we will see many isomorphisms. Those **isomorphisms in the case of Functors are called Natural Transformations**. We will see them in more detail in a latter chapter. For now, we are going to see four more isomorphisms (Task, Func, Lazy, IO) with the plain value T and thus provide you with a unifying abstraction when dealing with them.

4.5.1 Some more Isomorphisms

But first lats create another functor related to T . The value just T can be seen as a List with only one element:

```
public static List<T> ToList<T>(this T @this) => new List<T> { @this };
```

its inverse is the First () since for example `5.ToList().First()==5;`



T also is a ValueTuple with one element

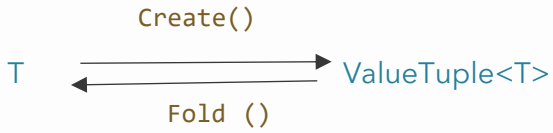
```
ValueTuple<int> t1 = ValueTuple.Create(5);
```

Which has as its inverse the generic Fold implementation by Microsoft for Tuples:

```
static S Fold<A, S>(this System.ValueTuple<A> self, S state, System.Func<S, A, S>
fold);
```

this is the standard Fold signature which is the equivalent of the List.Aggregate and we can use it to get the value back like this:

```
var value = t1.Fold(0, (a, e) => a + e);
```



Finally, obviously ValueTuple<T> isomorphic to `List<T>` so we can create some extension methods to go from one to the other

```
static List<T> ToList<T>(this ValueTuple<T, T> @this) => new List<T> {
      @this.Item1,
      @this.Item2
};
```

`ValueTuple` is a Functor since is equivalent with the `List` () but just for fun we can create a `ValueTuple` .Map extension method directly :

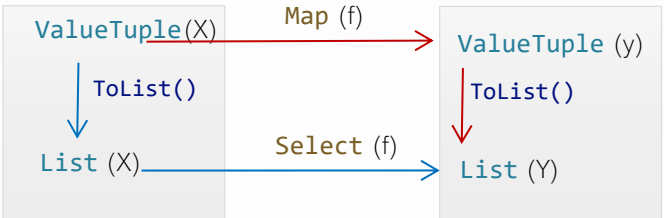
```
static ValueTuple<T1, T1> Map<T, T1>(this ValueTuple<T,T> @this, Func<T, T1> f) =>
             (f(@this.Item1), f(@this.Item2));
```

SideNote:

A very obvious observation is that if we implemented `ValueTuple<T, T>.ToList<T>()` and `ValueTuple<T1, T1>.Map<T, T1>(f)` correctly then the results of the following expressions should be equal:

```
(3, 4).ToList().Select(x => x + 4);
(3, 4).Map(x => x + 4).ToList();
```

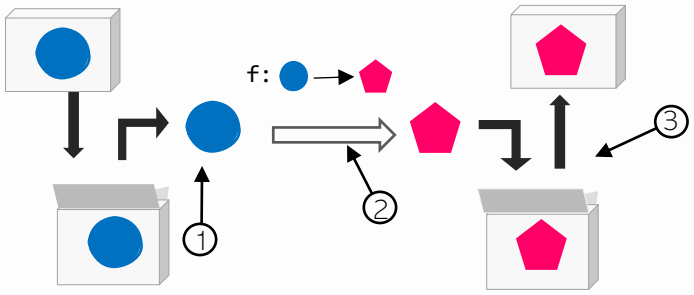
those two expressions represent the different paths in the following diagram:



thus, the diagram must commute. This is the hallmark of a valid isomorphism between Functors the structure should be preserved after the Map and Iso transformations.

4.5.2 The Basic Functor Mechanics

The following picture represents the whole idea behind how we construct the map of a functor. **If we accept the Functor as a container metaphor**, which is an acceptable way to visualize functors especially in an Object-oriented programming setting.



The mapping function that lifts the function f: int →int It follows the steps:

1. Open the container **F(a)** and access the value **a**
2. Applying the function f on the value **a** and get a new value **b**
3. wraps the resulting value again into a new container **F(b)** and return this **F(b)**

in the simplest form of the Identity functor the map follows those steps

```
public class Id<T>
{
        public T Value { get; set; }
        public Id(T value)=>  Value = value;
        public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
}
```

4.6 Extending Task<T> to Functor

In the following sections, we will see some examples of other popular Functors in C#. We will start by extending `Task<T>` , by providing a **Map** method that obeys the functor laws and thus promote native `Task<T>` into a functor.

We have said that the usual metaphor for a functor is "a container." A Task can be seen as a container that takes a value and wraps it, until it is resolved. In order to promote `Task<T>` to Functor, there must be a **mapping** function that would be able to lift any function and give a new `Task<T>` with the lifted value.

Here is one possible mapping function that preserves structure:

```
public static Task<T1> Map<T, T1>(this Task<T> task, Func<T, T1> mapping)
{
      var value = task.GetAwaiter().GetResult();
      return Task<T1>.Run(() => mapping(value));
}
```

Run This: .Net Fiddle

```
Task<int>task2 = Task<int>.Run(() => 2).Map(x => x * x);
```

The mapping function that lifts the function f: `T` → `T1` It follows the steps:

4. Waits for the result of the `Task` (so in a way unwraps the contained value) `task.GetAwaiter().GetResult()`
5. Applying the function f : `f(value)`
6. wraps the resulting value again into a new `Task`: `Task<T1>.Run(() => f(value));` because when we implement a **Map**, we always return something of the same type in order to belong to the same category (in this case type) or Promise.

Keep those steps in mind because they are common in the implementation of many Functors and monads.

SideNote : From now on we will write the `Task<T>` Map Method in this way

```
public static Task<T1> Select<T, T1>(this Task<T> task, Func<T, T1> mapping)=>
     task.ContinueWith(t => mapping(t.Result));
```

where we will utilize the ContinueWith do continue the task.

! Something worth pointing out here is the possibility of exception **this part belongs to the path of failure and ignores the f mapping** (one could argue that if there is an exception then this does not behave like a functor we will deal with this issue when we talk about the Either functor) We are going to come back to this observation many times throughout this book.

We can use our map function now:

```
  public class MockClientRepository
    {
        public Task<Client> GetById(int id) =>
            Task<Client>.Run(() =>
                new List<Client>{
                    new  Client{Id=1, Name="Jim"},
                    new  Client{Id=2, Name="John"}
                }
                .FirstOrDefault(x => x.Id == id)
            );
    }

var repository = new MockClientRepository();
var clientName = await repository
       .GetById(4)
       .Map(x => x != null ? x.Name : "Nothing found");
```

Run This: .Net Fiddle

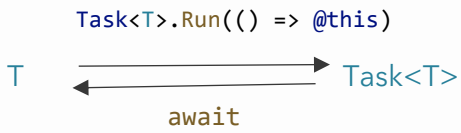
Obviously, the `Task<T>` is equivalent with the `T`.

```
public static Task<T> ToTask<T>(this T @this) => Task<T>.Run(() => @this);
```

The reverse morphism from `Task<T>` → `T` is different because `Task<T>` and `T` live in different time frames. But we know from a theorem called Yoneda lemma that those must be considered equivalent . The Yoneda lemma for the simple case of the Id functor says that a value `T` is the same as a callback that takes a `T` for example a value 5 is the same as this expression:

```
Action<Action<int>> continuation = (callback) => callback(5);
```

The `Task<T>` in its core is the same with this `Action<Action<int>>`



For the occasions where the `Task<T>` is given a `Func<T>` delegate to await, then we might get an exception, so we have to say that is the same as T+ • where the • is the null or the possibility of not returning a T.

4.7 IO Functor, a Lazy Id Functor

Another simple functor which extends the identity functor by adding Laziness is the IO functor. The IO is a lazy Id functor. IO comes handy when we want to Objectify a function, pass it around and use it later to produce a side effect as we will see.Lazy<T> as Functor

We can extend the native C# lazy type in order to provide a Map method :

```
public static Lazy<T1> Map<T, T1>(this Lazy<T> @this, Func<T, T1> f) =>
                            new Lazy<T1>(() => f(@this.Value));
```

Run This: .Net Fiddle

The mechanics behind the mapping function that lifts the function f: T →T1 follows the same stapes as Task<T>:

1. Waits for the result of the `Lazy` (so in a way unwraps the contained value) `@this.Value`
2. Applying the function f : `f(@this.Value)`
3. wraps the resulting value again into a new `Lazy`: `Lazy<T1>(() => f(@this.Value))` because when we implement a `Map`, we always return something of the same type in order to belong to the same category (in this case type).

You must appreciate that this preserves the Laziness of the `Lazy<T>` at no point the Lazy must collapse without trying to access the outer `Value.` At the step 3 we apply a lazy wrapper () => around what seems like an evaluation, `@this.Value`, and thus is not executed.

4.7.1 Func<T> Delegates as Functor

We can similarly extend the Func<T> delegate in order to provide a Map method:

```
public static Func<T1> Map<T, T1>(this Func<T> @this, Func<T, T1> f) =>
                            new Func<T1>(() => f(@this()));
```

By now you probably have figure out a pattern between The Task, Lazy and Func implementation of a Map. **The map is a powerful abstraction over the specific mechanics of each type**. Map abstracts over the need to transform the value inside a data structure.

4.7.2 IO Functor

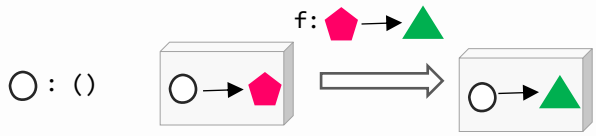
Functional programming in the same lines provides the IO functor:

```
public class IO<T>
 {
     public Func<T> Fn { get; set; }
     public IO(Func<T> fn) => Fn = fn;
     public IO<T1> Map<T1>(Func<T, T1> f) => new IO<T1>(() => f(Fn()));
     public T1 MatchWith<T1>(Func<T, T1> callback) => callback(Fn());
     public T Run() => Fn();
 }
```

And use it in this way:

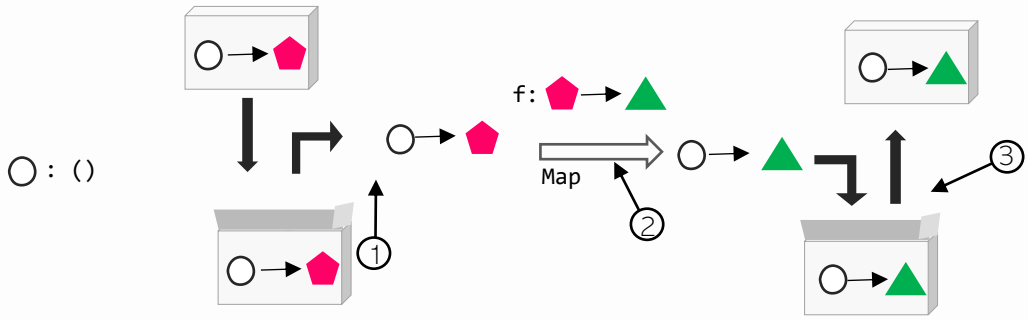
```
new IO<int>(() => 3)
       .Map(x => x + 3)
       . MatchWith(Console.WriteLine);
```

Run This: .Net Fiddle



the map implementation should remind you slightly of the Task Map line of reasoning while implementing it.

1. We first have to "run" the previous computation `Fn()`
2. Then use the function f to get the lifted value `f(Fn())`
3. And then finally rewrap this new value in a new `new IO<T1>(() => f(Fn()))`.



An example application would be to use an IO, to isolate the effects of accessing the `Console.` This is considered **Side effect** because there is no predictability on the Input. It is not Referential Transparent.

```
var readKeyInstruction = new IO<ConsoleKeyInfo>(() => Console.ReadKey());

var UserInputAsInt = readKeyInstruction
            .Map(x =>  int.Parse(x.KeyChar.ToString()))
            .Run();
```

Run This: .Net Fiddle

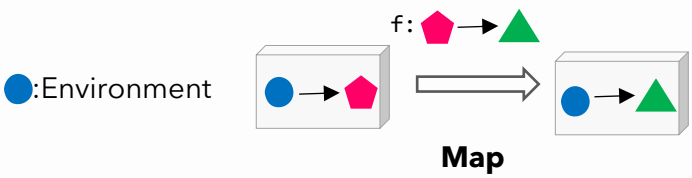
It's the same as the Id functor but encloses the value that we provide in a thunk - () => { } just to make it lazy. We would have to collapse the function Fn() in order to get back the value, most IO implementations around usually provide a Run () function that does just that. Also, with an extension method, we can create IOs out of everything on the fly.

```
public static IO<T> ToIO<T>(this T @this) => new IO<T>(() => @this);
```

4.8 Reader Functor

The Reader Functor represents a computation, which can read values from a shared environment. The reader functor conceptualizes the idea of having stored expression that we can evaluate in different configurations at a later time. We can represent this like that ((→) env), which means we give an environment, and we get out the thing that we want.

```
public class Reader<Env, T>
{
  public Func<Env, T> Fn { get; set; }
  public Reader(Func<Env, T> fn) => Fn = fn;
  public Reader<Env, T1> Map<T1>(Func<T, T1> f) =>
                                  new Reader<Env, T1>((env) => f(Fn(env)));
  public T Run(Env env) => Fn(env);
}
```



For a simple example we might have a `Config` type that contains various parameters and we want to create a Reader that stores a simple expression that depends on the

`Config`. With the reader we can do computations on the expression Lazily without having the `Config` yet:

```
public class Config
{
    public string Name { get; set; }
}

var getName = new Reader<Config, string>(environment => environment.Name).
  Map(name => $"Name: {name }").
  Run(new Config { Name = "Sql" });
```

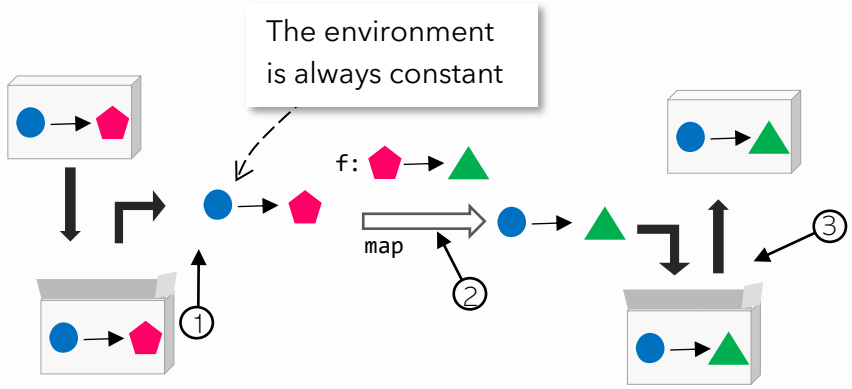
Run This: .Net Fiddle

the map here takes a function (a → b) and then lifts the initial reader that contains an a (((→) e) a) to a reader that contains a b (((→) e) b)

map: (a → b) → (((→) e) a) → (((→) e) b)

the implementation is more convoluted but again follows the lines of Task<T> and IO

1. First, we evaluate the inner reader `Fn(env)` (so in a way unwraps the contained value)
2. Applying the function `f`: `f(Fn(env))`
3. wraps the resulting value again into a new `new Reader((env) => f(Fn(env)))`



4.9 LINQ Native Query syntax: Functor Support

Did you know that if you implement a **Select** method in any type you can use the *from …in …select* Linq syntax???

Here I will rename the Map() method of our `Id<T>` to Select()

```
public class Id<T>
{
    public T Value { get; set; }
```

```
    public Id(T value) => Value = value;
    public Id<T1> Select<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
}
```

Run This: .Net Fiddle

This will allow the compiler to recognize as valid the following `from_in _select` statement:

```
var r = from x in new Id<int>(5) select x + 3;

r.MatchWith(Console.WriteLine);
```

Run This: .Net Fiddle

The result of the select is the one that comes from `Id<T1> Select<T1>(Func<T, T1> f)` there is no need to be of The same type in our example `Id<T1>`. We could for example define a select method that looks like the following:

```
public T1  Select<T1>(Func<T, T1> f) =>  (f(Value));
```

The `Id<T>` does not qualify anymore as a functor but the Linq statement compiles just fine. The only requirement is that the argument is of the form `(Func<T, T1> f)` and the name of the method to be Select.

In the same way if we rename the `Task` .Map() to Select() then we get a compiler support

```
public static Task<T1> Select<T, T1>(this Task<T> task, Func<T, T1> f)
=>  task.ContinueWith(t => f(t.Result));
```

This allows us to write Linq on Tasks

```
var repository = new MockClientRepository();

var clientName = await (from client in repository.GetById(4)
                        select client != null ?  client.Name : "Nothing found");
```

Run This: .Net Fiddle

Conventions: Map and Select

As a convention we will name our mapping method **Map** nonetheless in some instances we might also use **Select if we need LINQ support.** Map and Select will be used interchangeably in this book. In the Real world in any proper Functional library there should be a **Select** method as a name alias for **Map**.

4.10 Maybe Functor aka Option<T>

4.10.1 Dealing with null

"…The absence of Sum types is what Tony Hoare called the **Billion Dollar Mistake** … all the superstitious nonsense about null is all about the failure of language design, because they don't have Sum types. "

– Robert Harper
Oregon Programming Languages Summer School

The problem of null or undefined is one of the biggest problems in computation theory. If we want to write meaningful programs, we must accept the fact that some computations might yield no result. The usual way object-oriented languages deal with this is by checking the types for not being undefined in order to use them. Any developer knows the number of bugs that have as source the problem of undefined.

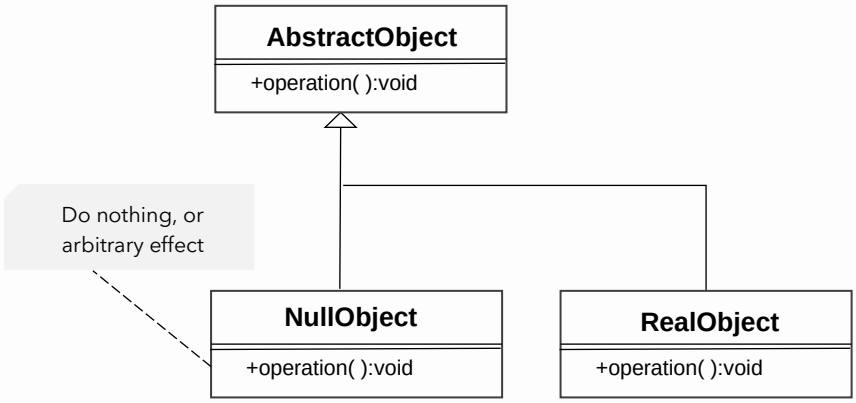
The classical solution is using conditionals everywhere to check for null.

```
if (result) {
   result.operation();
} else {
   //do nothing
}
```

This reduces the cohesion of the codebase, because we must tightly couple to code relating to the cross-cutting concern of null checking.

4.10.2 The Null Object Design pattern

A more elegant solution to the problem of Null is the "Null Object" design pattern. The null object pattern is a special case of the strategy design pattern. The idea here is to replace conditional with strategy]. Instead of setting something as null we can set it as `NullObject`. Were the `NullObject` methods are empty, or they do not do anything when called. `NullObject` is in fact designed in a way that if it is fed to any method that waits for a `RealObject`, there should not be any unexpected side-effect, or any exceptions thrown.



A simple implementation of the `NullObject` design pattern with C# could be something like this :

```
public abstract class AbstractObject<T>
{
	public abstract void Operation();
}

public class NullObject<T> : AbstractObject<T>
{
	public NullObject() { }
	public override void Operation() {//Do nothing here }
}

public class RealObject<T> : AbstractObject<T>
{
	private readonly T value;
	public RealObject(T value) => this.value = value;//instructor injection

	public override void Operation()
	  {   // do something with value
	      Console.WriteLine($"{value}");
	  }
}
```

Run This: .Net Fiddle

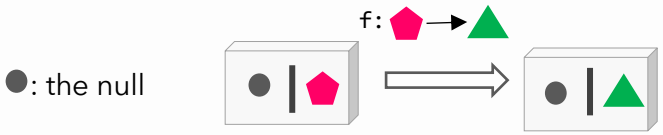
This implementation is unobtrusive. It should not affect the rest of the code and should remove the need to check for null.

The major problem with this pattern is that for each object, we need to create a nullObject with the operations that we are going to use inside the rest of the code. **This will force us to duplicate all the domain objects in our model. It is a big trade-off**. There is no easy way to abstract the mechanics of this implementation in an object-oriented world, mainly because we cannot have specific information about the operations that we want to isolate and create a nullObject that provides those operations.

4.10.3 The Functional equivalent - Maybe as Functor

In Functional programming, we use the **Maybe Type** [sometimes also called Option] in order to represent the possibility of something being null. The Maybe solves the null problem by moving the mechanics of null checking inside a functor `map`. **Thus, instead of applying the objects on functions, we reverse the flow and apply the functions onto objects**. Now, we can isolate the effect inside a single point; the `map`.

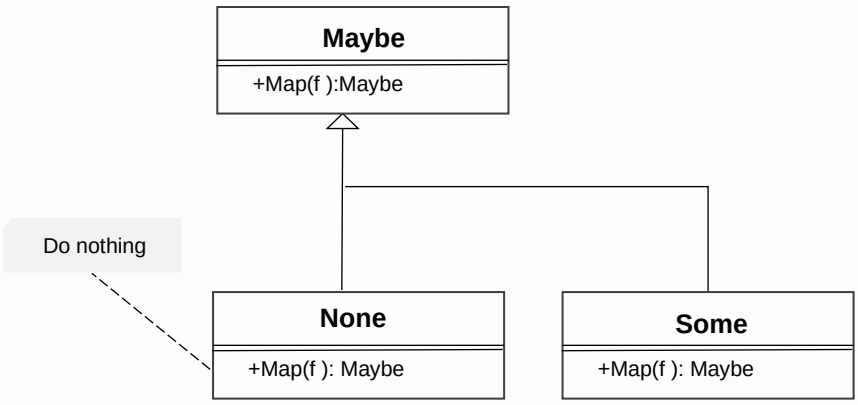
The standard implementation of the maybe/option Type is a hierarchy with a base class and two sub-types: the `Some` and the `None`. **The None represents the case of null**



After all this discussion, hopefully, the implementation of maybe functor would be apparent

```
public abstract class Maybe<T>
{
     public abstract Maybe<T1> Map<T1>(Func<T, T1> f);
     public abstract TResult MatchWith<TResult>(
                        (Func<TResult> None, Func<T, TResult> Some) pattern);
}

public class None<T>: Maybe<T>
{
     public override TResult MatchWith<TResult>(
          (Func<TResult> None, Func<T, TResult> Some) pattern) => pattern.None();

     public override Maybe<T1> Map<T1>(Func<T, T1> f) => new None<T1>();
}

public class Some<T> : Maybe<T>
{
    private readonly T value;
    public Some(T value) => this.value = value;

    public override TResult MatchWith<TResult>(
(Func<TResult> None, Func<T, TResult> Some) pattern) => pattern.Some(value);

    public override Maybe<T1> Map<T1>(Func<T, T1> f) => new Some<T1>(f(value));
}
```

Run This: .Net Fiddle



Maybe `MatchWith` implementation for `Maybe` is straightforward. Since it is a hierarchy, we have to implement for each subtype the `MatchWith`, and we are going to call different callbacks, so we can distinguish (pattern match) between `Some` and `None`.

And we can use it like this:

```
var result = new None<int>( )
               .Map(v=> $"number is  : {v}")
               .MatchWith<string>(pattern: (
                   None: () => "Not found",
                   Some: v => $"Answer - {v}"
           ));
```

Run This: .Net Fiddle

4.10.3.1 Using C# 8. pattern matching

We will display both `MatchWith` and `switch` pattern matching just for completeness. We can skip the custom definition by using the native C# 8.0 pattern matching by defining

```
public abstract class Maybe<T>
{
      public Maybe<T1> Map<T1>(Func<T, T1> f) =>
       this switch
       {
           None<T> { } => new None<T1>(),
           Some<T> { Value: var v } => new Some<T1>(f(v)),
       };
}

 public class None<T> : Maybe<T>
 {
     public None() { }
 }
```

```
    public class Some<T> : Maybe<T>
    {
        public readonly T Value;
        public Some(T value) => this.Value = value;
    }
```

If we want to hide the value `public readonly T Value` and make it into `private` then we can use the `Deconstruct` to gain access for the pattern match :

```
public abstract class Maybe<T>
{
        public Maybe<T1> Map<T1>(Func<T, T1> f) =>
         this switch
         {
             None<T>() => new None<T1>(),
             Some<T>(var v) => new Some<T1>(f(v)),
             _ => throw new NotImplementedException(),
         };
}


public class None<T> : Maybe<T>
{
     public None() { }
     public void Deconstruct() { }
}

public class Some<T> : Maybe<T>
{
     private readonly T value;
     public Some(T value) => this.value = value;
     public void Deconstruct(out T value) => value = this.value;
}
```

Run This: Fiddle

With this formalism we can now write as an example :

```
 string result = new None<int>()
            .Map(x => x + 1)
            switch
            {
                None<int>() => "nothing",
                Some<int>(var x) => x.ToString(),
                _ => throw new NotImplementedException(),
            };
```

Run This: Fiddle

4.10.4 Maybe Functor Example

Now let us get a client asynchronously with a certain Id from a repository.

```
public class MockClientRepository
{
    List<Client> clients = new List<Client>{
                    new  Client{Id=1, Name="Jim"},
                    new  Client{Id=2, Name="John"}
                };

       public Maybe<Client> GetById(int id) =>
                    clients.FirstOrNone(x => x.Id == id);

}
```

Run This: .Net Fiddle

Here we have replaced the result of the array filtering:

```
.FirstOrDefault(x => x.Id == id)
```

with something that will return a `Maybe<Client>`. If there is no client for a specific id, we should return `None()`. If there is a client, we will return `Some(client)`.

```
public static partial class FunctionalExtensions
{

public static Maybe<T> FirstOrNone<T>(this List<T> @source,Func<T,bool> predicate)
{
            var firstOrDefault = @source.FirstOrDefault(predicate);
            if (firstOrDefault != null)
                return new Some<T>(firstOrDefault);
            else
                return new None<T>();
 }
}
```

Now we can write:

```
var repository = new MockClientRepository();

var result = repository.
                GetById(1)
                  .MatchWith(pattern:(
                        none: () => "Not Found",
                        some: (client) => client.Name
```

```
));
```

Or by using native pattern matching

```
var repository = new MockClientRepository();
var result = repository.
                GetById(1) switch
                {
                    None<Client>() => "Not Found",
                    Some<Client>(var client) => client.Name,
                    _ => throw new NotImplementedException(),
                };
```

Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle

SideNote:

You can run a simple ASP.NET MVC Maybe Functor Example .NET Fiddle. Also you can download the Practical-Functional-CSharp project from Github and run WebApplicationExample.sln the from the **WebApplicationExample** folder**.**

4.10.5 Maybe in Language-ext / Option

The usual name for this functor F=1+X (disjoint union with a base point) in most functional languages is called **Maybe** but in Language-ext is called **Option** which is another common naming for Maybe that comes closer to the OOP paradigm. Lets say that **Maybe** is the name of the concept and **Option** the implementation. It is the same thing. For the rest of this book **we will use language-ext Option** monad in our code examples.

4.10.6 Maybe Functor Example With language-ext Option

Fortunately for us the language-ext have defined **an implicit operator on the Option that does the conversion of null to Option immediately**:

```
public static implicit operator Option<A>(A a);
```

this allows us to write something like the following:

```
public Option<Client> GetById(int id) => clients.SingleOrDefault(x => x.Id == id);
```

the compiler expects an `Option<Client>` so it uses the operator `Option<A>(A a)` in order to do the conversion on the spot. Now we can write:

```
public class MockClientRepository
 {
        List<Client> clients = new List<Client>{
                    new  Client{Id=1, Name="Jim", },
                    new  Client{Id=2, Name="John",}
                };

       public Option<Client> GetById(int id) =>
                           clients.SingleOrDefault(x => x.Id == id);
    }
```

Run This: .Net Fiddle

We can write again:

```
public string GetNameById(int clientId) =>
       clients
        .GetById(clientId)
        .Map(client => client.Name)
        .Match(Some: (name) => name,
               None: () => "no client");
```

Run This: .Net Fiddle

4.10.7 C# 8. pattern matching Support for Option

Because of the structure of the Option implementation in language-ext library, it's not possible to use `switch` directly on the `Option<>` but fortunately it exposes an `Option<>.Case` property that allow us to use `switch`:

```
  public string GetNameById1(int clientId) =>
        clients
```

```
.GetById(clientId)
.Map(client => client.Name)
 .Case switch
{
    SomeCase<string>(var name) => name,
    NoneCase<string> { } => "no client",
    _ => throw new NotImplementedException()
};
```

The Option exposes a `Case` property.

Run This: .Net Fiddle

The two subtypes that we must match on, are the `SomeCase<>` and `NoneCase<>` and we are going to use this form instead of the `Match` a lot. Both provide the same behaviour, and it comes down to preference.

The only drawback is that **you must write explicitly the Types** of the `SomeCase<>` and `NoneCase<>` because the compiler cannot infer those.

4.10.8 Folding Maybe

For the maybe Fold is extremely easy to implement we start with the accumulator and if we have a None then we just return itself or in the case of the Some we apply the reducer to the `accumulator` and the value in order to merge the values:

```
public S Fold<S>(S accumulator, Func<S,T,S> reducer) =>
          this.MatchWith((
                  none: () => accumulator,
                  some: (v) => reducer(accumulator,v)
              ));
```

Most of the times (almost all) we will prefer the MatchWith to get the value out of the Maybe. We can use Fold like this:

```
public string GetNameById(int clientId) =>
        clients
        .GetById(clientId)
        .Map(client => client.Name)
         .Fold("", (a, name) =>
         {
             return name;
         });
```

Run This: .Net Fiddle

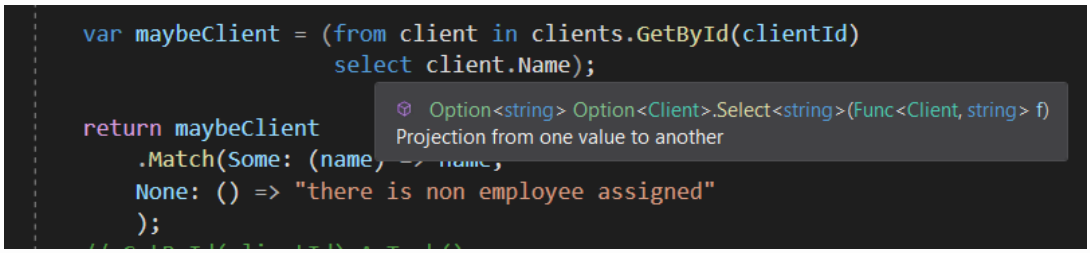
4.10.9 Using the Linq syntax

The language-ext library provides a Select method that allows us to use the Linq syntax with the Option:

```
public string GetNameById (int clientId) =>
        (from client in clients.GetById(clientId)
         select client.Name)  .Case switch
        {
            SomeCase<string>(var name) => name,
            NoneCase<string> { } => "no client",
            _ => throw new NotImplementedException()
        };
```

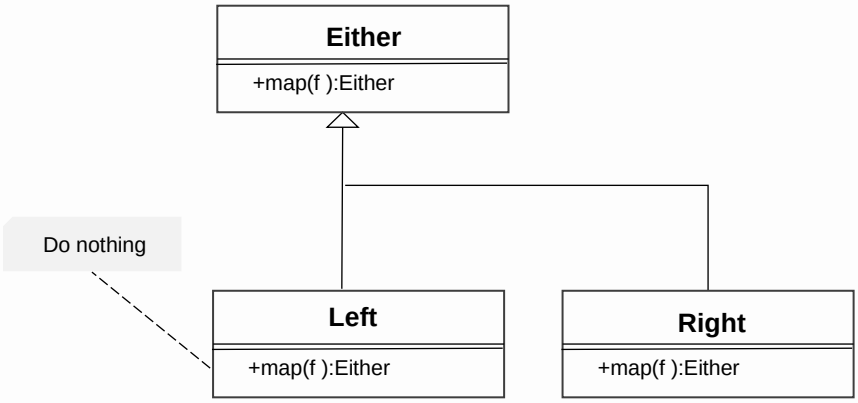
Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle

you can see the signature of the select if you place the cursor over the select Keyword.



4.11 Either Functor

Either Functor is an extension of Maybe. The `Right` is equal to the Maybe.Some, and the `Left` is the equivalent of Maybe.None, **but now we can allow it to carry a value** `Left =(v)=>{}`. This value can be viewed as a **message** (usually, it represents an error message).



```
public abstract class Either<TL, TR>
{
    public abstract Either<TL, T1> Map<T1>(Func<TR, T1> f);
}

public class Left<TL, TR> : Either<TL, TR>
{
    public TL Value { get; }
    public Left(TL value) => Value = value;
    public override Either<TL, T1> Map<T1>(Func<TR, T1> f) =>
             new Left<TL, T1>(Value);
}

public class Right<TL, TR> : Either<TL, TR>
{
    public TR Value { get; }
    public Right(TR value) => Value = value;
    public override Either<TL, T1> Map<T1>(Func<TR, T1> f) =>
               new Right<TL, T1>(f(Value));
}
```

Run This: .Net Fiddle

The thing we must pay attention to here is that the `Left` is a functor that **dismisses the mapped function f** and returns itself - `Map:(f)=>Left(Value)`. In essence, it preserves the value that it holds. After we reach a `Left` **all the subsequent transformation through the map(f) are ignored**.

4.11.1 Pattern matching for Either <T>

The `MatchWith` implementation for Either is similar to Maybe's.

```
public abstract class Either<TL, TR>
{
        public abstract T1 MatchWith<T1>((Func<TR, T1> right, Func<TL, T1> left) p
attern);
}

public class Left<TL, TR> : Either<TL, TR>
{
        public TL Value { get; }
        public Left(TL value) => Value = value;
        public override T1 MatchWith<T1>((Func<TR, T1> right, Func<TL, T1> left) p
attern)  => pattern.left(Value);
```

```
}

public class Right<TL, TR> : Either<TL, TR>
{
        public TR Value { get; }
        public Right(TR value) => Value = value;
        public override T1 MatchWith<T1>((Func<TR, T1> right, Func<TL, T1> left) p
attern) => pattern.right(Value);

}
```

Run This: .Net Fiddle

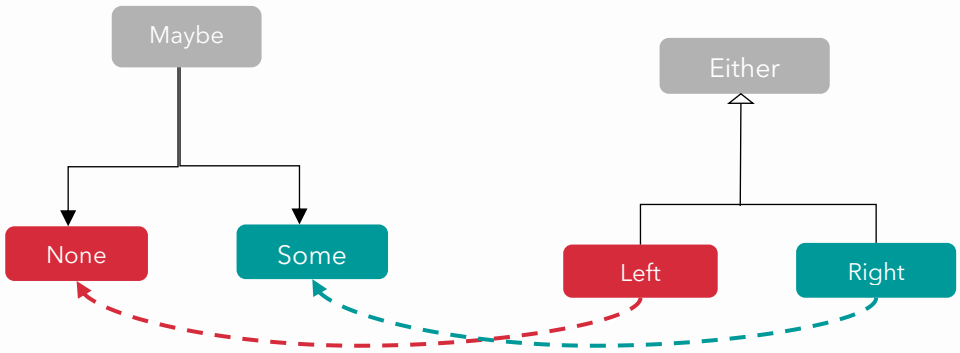
4.11.2 Using C# 8.0 pattern matching

Like in the case of Maybe we can skip the `MatchWith` and use the `switch` to pattern match. Defining Either using the native C# 8.0 pattern matching would give us something like this:

```
public abstract class Either<TLeft, T>
{
        public Either<TLeft, T1> Map<T1>(Func<T, T1> f) =>
           this switch
           {
               Left<TLeft, T>(var v) => new Left<TLeft, T1>(v),
               Right<TLeft, T>(var v) => new Right<TLeft, T1>(f(v)),
           };
}

public class Left<TLeft, T> : Either<TLeft, T>
{
        private readonly TLeft value;
        public Left(TLeft value) => this.value = value;
        public void Deconstruct(out TLeft value) => value = this.value;
}

public class Right<TLeft, T> : Either<TLeft, T>
{
        private readonly T value;
        public Right(T value) => this.value = value;
        public void Deconstruct(out T value) => value = this.value;
}
```

4.11.3 Either Functor Example

Since the Either is a Maybe that curries a value inside the `Left` subtype, we can create a method that would convert a Maybe into an Either.



The following method is **a natural transformation** from a `Maybe<T>` to an `Either<TLeft,T>`

```
public static Either<TLeft, T> ToEither<TLeft, T>(this Maybe<T> @source,TLeft defa
ultLeft)
 {
            return @source.MatchWith<Either<TLeft, T>>((
                    none: () => new Left<TLeft, T>(defaultLeft),
                    some: (v) => new Right<TLeft, T>(v)
                ));
 }
```

We pattern match on the maybe and in the case of a None we return a Left with some default value. For example, for the Client repository example in the case that we couldn't find any client with the given id we can return something more specific than simply None:

```
    public class MockClientRepository
    {
        …
        public Either<string, Client> GetById(int id) =>
            clients.FirstOrNone(x => x.Id == id)
            .ToEither("No client Found");
    }
```

The following of the computation is a usual

```
    public class Controller
    {
```

```
    MockClientRepository clients = new MockClientRepository();
    public string GetClientNameById(int clientId) =>
        clients
        .GetById(clientId)
        .Map(client => client.Name)
        .MatchWith((
                left: (e) => "error:" + e,
                right: (name) => name
            ));
}
```

the `left` case now will have as an error `e` `"No client Found"` if matched

Run This: .Net Fiddle

Run This: ASP.NET MVC Fiddle

4.11.4 C# 8. pattern matching Support for language-ext Either

The language-ext provides a standard Either type from now on we will be using this Either in our code examples instead of the custom Either. The Map and Match methods are pretty much the same.

Similar to the Option the Either implementation in language-ext library, does not allow to use `switch` directly on the `Either<>` but it exposes an `Either<>.Case` property that allow us to `switch`:

```
public string GetAssignedEmployeeNameById(int clientId) =>
          clients
                .GetById(clientId)
                .Map(client => client.EmployeeId)
                .Bind(employees.GetById)
                .Case switch
          {
              LeftCase<string, Employee>(var error) => error,
              RightCase<string, Employee>(var employee) => employee.Name,
              _ => throw new NotImplementedException(),
          };
```

We now return an `Option<Client>`

The two subtypes that we must match on are the `LeftCase<>` and `RightCase<>`.

4.11.5 Using Either for exception handling

As we saw the Either can be reduced to maybe if we discard the value passed on the left side. **The left side of either represents a faulty path where the Error** is kept and does not participate in any other computation; it just passes along the Error information. Let us refactor a try/catch to its's Either equivalent. We will start with this try-catch block:

```
double finalPrice;
 try
  {
   var discount = 0.1;
    finalPrice = 10 - discount * 10;

  }
  catch (Exception e)
  {
    Console.WriteLine(e);
    throw;
  }
```

we could extract the outline of the try catch and inject the two segments of try… and catch, with delegates [this is a functional flavored version of the **template method design pattern**].

so we can rewrite it like bellow, in a more general way.

```
    public class Try<T>
    {
        public T Result { get; set; }
        public Try(Func<T> @try, Action<Exception> exeptionHandler)
        {
            try
            {
                Result = @try();
            }
            catch (Exception e)
            {
                exeptionHandler(e);
            }
        }
    }
```

and use it like that :

```
 var tryResult =new Try<double>(() =>
 {
     var discount = 0.1;
     finalPrice = 10 - discount * 10;
```

```
    return finalPrice = 10 - discount * 10;

},
e => Console.WriteLine(e));
```

Run This: .Net Fiddle

Finally, we return an Either:

```
public class Try<T>
    {
        public Either<Exception, T> Result { get; set; }
        public Try(Func<T> @try)
        {
            try
            {
                Result = new Right<Exception, T>(@try());
            }
            catch (Exception e)
            {
                Result = new Left<Exception, T>(e);
            }
        }
    }
```

Run This: .Net Fiddle

And we can either use it by exposing the result

```
new Try<double>(() =>
{
    var discount = 0.1;
    finalPrice = 10 - discount * 10;
    return finalPrice = 10 - discount * 10;
})
.Result
.MatchWith(
pattern: (
    Right: v => Console.WriteLine(v),
    Left: v => Console.WriteLine("Either Error :" + v)
));
```

Or we could have one more Functor named Try Functor by just hiding the result and exposing a Map and a MatchWith. Here my quick implementation:

```
public class Try<T>
{
    private Either<Exception, T> Result { get; set; }
    public Try(Func<T> @try)
    {
```

```
        try
        {
            Result = new Right<Exception, T>(@try());
        }
        catch (Exception e)
        {
            Result = new Left<Exception, T>(e);
        }
    }

   public void MatchWith((Action<T> Right, Action<Exception> Left) pattern) =>
                   Result.MatchWith(pattern);

    public Try<T1> Map<T1>(Func<T, T1> f) => new Try<T1>(() =>
    Result.MatchWith<T1>(pattern: (
        Right: x => f(x),
        Left: e => throw e))
    );
}
```

We could also postpone execution until instead trying to run the `Func<T> @try` immediately `Right<Exception, T>(@try())`.

Now that we have our Try in place, we can modify some of the functors that represent delayed computations like the Lazy, IO and Reader to Execute the contained computation in a safe manner that would yield an Either:

```
public static Try<T> ToTry<T>(this Lazy<T> @this) => new Try<T>(() => @this.Value)
;
```

Run This: .Net Fiddle

Language-Ext

Try<T>

There is an explicit implementation of Try<T> in the library you can find here

4.12 Explicitly Compositing Functor

Functors are closed under composition. The term **closed** means that if we have two functors and we compose them, the resulting structure is also a functor. This means that we should be able to construct a well-behaved Map method for the composition, **using only the Map methods of the composing functors** and nothing more.

For an id in a Task for example `new Id<Task<int>>(Task<int>.Run(() => 4))` we can define as an overall map for the composition with this extension method:

```
public static Id<Task<T1>> MapT<T, T1>(this Id<Task<T>> @this, Func<T, T1> f)
       => @this.Map(t => t.Map(f));
```

Run This: .Net Fiddle

It is easy to understand how this works. You first pass the function inside with the map of the first functor, and then you map the resulting value again using the map of the inner (second) functor.

This works for all the functors. The problem is that with C# we cannot write a valid Generic Functor composition type. This is because we cannot represent the "Functor" kind as a generic. For example, it would be great if we could write an interface

```
public interface IFunctor<T>
{
     IFunctor<T1> Map<T1>(Func<T, T1> f);
}
```

To abstract over the concept of Functors and then Maybe write something like the following to abstract over all Functor compositions

```
public static F<G<T1>> MapT<T, T1>(this F<G<T>> @this, Func<T, T1> f)
where F:IFunctor<G>
where G:IFunctor<T>
=> @this.Map(t => t.Map(f));
```

Unfortunately, C# Type system is not that expressive enough. You can try to implement something like that by yourself to see why this is impossible. This is because C# does not support **Higher Kinded polymorphism** you can follow some discussions on dotnet github repository. Scala and Haskell in contrast support Higher Kinded Types.

The Problem is that if you want to use a specific combination of Functors to compose you will have to give your own extension

```
public static Task<Id<T1>> MapT<T, T1>(this Task<Id<T>> @this, Func<T, T1> f)
          => @this.Map(t => t.Map(f));
```

Which is the exact same `@this.Map(t => t.Map(f))` for all combinations except from the types.

Language-Ext

MapT

The good thing is that in language-ext library there are implemented various Composite Functor definitions. So when using Functors inside Functor you will probably have an extension method MapT available

We can now revisit the repository example from the Maybe section where we have an asynchronous operation of a repository that returns a maybe of an array:

```
var result = new MockClientRepository()
                .GetById(1)
                .Map(client=> client.Map(v => $"Answer - {v.Name}"))
                .ToEitherTask();
```

```
result = new MockClientRepository()
               .GetById(1)
               .MapT(v => $"Answer - {v.Name}")
               .ToEitherTask();
```

In which **MapT** is now defined as :

```
public static Task<Maybe<T1>> MapT<T, T1>(this Task<Maybe<T>> @this,Func<T, T1> f)
      => @this.Map(t => t.Map(f));
```

Run This: .Net Fiddle

4.13 Combining Task and Option – Task<Option<T>>

We started by presenting some simple example usages of the Option and Either types. But as we are going to see in the Contoso application example chapter, in real life situations, composite functors arise naturally. For example an Option within a Task `Task<Option<Client>>` or an Either inside a task `Task<Either<string,Client>>`. In this section we will modify the Option example to include asynchrony. For example, when we access a Database from EntityFramework, MondoDb etc, or wait for an `HttpClient` GET request.

Using Entity Framework an Async Get repository operation could look like this:

```
    public class EFClientRepository
    {
        public async Task<Option<Client>> GetByIdAsync(int id)
        {
            using (var context = new DbContext())
            {
                var clients = await context.Clients.ToListAsync();
                Option<Client> client = clients.SingleOrDefault(x => x.Id == id);
                return client;
            }
        }
    }
```

Or using an Async operations of the `HttpClient` to get a `Client` from another Service in a Domain Driven or Microservice setting:

```
public class HttpClientRepository
{
   public async Task<Option<Client>> GetByIdAsync(int id)
    {
      HttpClient httpClient = new HttpClient();
      HttpClient response = await httpClient.GetAsync("");

      if (response.IsSuccessStatusCode)
      {

      var streamTask = httpClient.GetStreamAsync($"https://myapi/client/{id}");
      var client = await JsonSerializer.DeserializeAsync<Client>(await streamTask
      );
        return Some(client);

      }
       else
      {
          return None;
      }
   }
 }
```

To emulate a general case, we will use this mock `MockClientRepository`

```
public class MockClientRepository
{
        List<Client> clients = new List<Client>{
                    new  Client{Id=1, Name="Jim" },
                    new  Client{Id=2, Name="John" }
```

```
                };

        public Task<Option<Client>> GetByIdAsync(int id) =>
System.Threading.Tasks.Task.Run(() =>
            {
              System.Threading.Thread.Sleep(1000);
              Option<Client> maybeClient = clients.SingleOrDefault(x => x.Id== id);
              return maybeClient;
             });
    }
```

The important thing here is that we have a Get method signature

```
OptionAsync<Client> GetByIdAsync(int id)
```

We can now compose `GetNameByIdAsync`

```
public class Controller
{
        MockClientRepository clients = new MockClientRepository();

        public Task<string> GetNameByIdAsync(int clientId) =>
            clients
            .GetByIdAsync(clientId)
            .Map(x => x.Map(client => client.Name))
            .Match(Some: (name) => name,
                   None: () => "no client");

}


var assignedEmployeeName = await new Controller().GetNameByIdAsync(1);
```

Run This: .Net Fiddle
Github: source

here we used the composite Map to bypass the `Task` and use the Map of the `Option` inside

```
.Map(x => x.Map(client => client.Name))
```

Fortunately for us language-ext has a `MapT` on almost every Type. So, we can use this instead:

```
public class Controller
{
        MockClientRepository clients = new MockClientRepository();

        public Task<string> GetNameByIdAsync(int clientId) =>
            clients
            .GetByIdAsync(clientId)
```

```
            .MapT(client => client.Name)
            .Match(Some: (name) => name,
                   None: () => "no client");
}
```

SideNote:

In this example for the language-ext does provide us with an extension method that provides a `Match` on the task that contains an Option:

```
public static Task<B> Match<A, B>(this Task<Option<A>> self, Func<A, B> Some, Func
<B> None)
```

```
public static OptionAsync<B> MapAsync<A, B>(this Task<Option<A>> self, Func<A, B> f);
public static Task<B> Match<A, B>(this Task<Option<A>> self, Func<A, B> Some, Func<B> None);
public static Task<Unit> Match<A>(this Task<Option<A>> self, Action<A> Some, Action None);
public static Task<B> MatchAsync<A, B>(this Task<Option<A>> self, Func<A, Task<B>> Some, Func<Tas
public static Task<B> MatchAsync<A, B>(this Task<Option<A>> self, Func<A, B> Some, Func<Task<B>>
```

Nonetheless in the following section we will see that there is no Match extension method for `Task<Either<,>>`.

4.14 Combining Task and Option – OptionAsync<T>

`OptionAsync<T>` is another way to see the combination of Task and Option and belongs to the category of Monad Transformers we haven't reached the monad chapter yet and we are not going to see the internal mechanics of Monad transformers in this book. But it is very intuitive to use even without theory. `OptionAsync<T>` is just an Option that has a Task on top of it and provides some valid methods Map and Match.

We can convert a `Task<Option<Client>>` into an `OptionAsync<Client>` by using the `.ToAsync()` method

```
public OptionAsync<Client> GetByIdAsync(int id)
{
    Task<Option<Client>> r = System.Threading.Tasks.Task.Run(() =>
    {
        System.Threading.Thread.Sleep(1000);
        Option<Client> t = clients.SingleOrDefault(x => x.Id == id);
        return t;
    });
    return r.ToAsync();
}
```

(extension) OptionAsync<Client> Task<Option<Client>>.ToAsync<Client>()

After we have an `OptionAsync` we can use the `Map` and `Match` as usual

```
public class Controller
{
        MockClientRepository clients = new MockClientRepository();

        public Task<string> GetNameByIdAsync(int clientId) =>
            clients
            .GetByIdAsync(clientId)
            .Map(client => client.Name)
            .Match(Some: (name) => name,
                   None: () => "no client");
    }
```

Github: source

Here the `Match` returns a `Task<string>`

4.15 Combining Task and Either – Task<Either<,>>

As one would expect, if we want to propagate an error message instead of just the None we will end up with the case of an Either inside a Task. We can modify the Repository again to return a `Task<Either<string, Client>>`

```
public class MockClientRepository
{
 ...
public Task<Either<string, Client>> GetByIdAsync(int id)
{

Task<Either<string, Client>> maybeClientTask =
System.Threading.Tasks.Task.Run(()  =>
  {
   System.Threading.Thread.Sleep(1000);
     Option<Client> maybeClient = clients.SingleOrDefault(x => x.Id == id);
     return maybeClient.ToEither("no Client Found");
    });
    return maybeClientTask;
   }

 }
```

Now we can use that in a controller (again the two different versions. One with the MapT):

```
public class Controller
{
```

```
    MockClientRepository clients = new MockClientRepository();
    public Task<Either<string, string>> GetNameByIdAsync(int clientId)
            => clients.GetByIdAsync(clientId)
                 .MapT(client => client.Name);

    public Task<Either<string, string>> GetNameByIdAsync1(int clientId)
         => clients.GetByIdAsync(clientId)
              .Map(x => x.Map(client => client.Name));
}
```

Github: source

And then we could finally call it by awaiting the task and then doing a case analysis

```
var clientName = (await new Controller().GetNameByIdAsync(3))
                    .Case switch
                    {
                        LeftCase<string, string>(var error) => error,
                        RightCase<string, string>(var name) => name,
                        _ => throw new NotImplementedException(),
                    };
```

After `await` we left with the Either

Github: source

This is the same as

```
var clientName = await new Controller().GetNameByIdAsync(3)
                    .Map(x => x.Case) switch
                    {
                        LeftCase<string, string>(var error) => error,
                        RightCase<string, string>(var name) => name,
                        _ => throw new NotImplementedException(),
                    };
```

Or even

```
var clientName = (await new Controller().GetNameByIdAsync(3))
                    .Match(
                     Right: (name) => name,
                     Left: (e) => $"error {e}");
```

That's the `Either`'s `Match`

Github: source

4.16 Combining Task and Either – EitherAsync<>

`EitherAsync<>` is another Monad Transformer that helps us with combinations of Task and Either. Again `EitherAsync < >` is just an Either that has a Task on top of it and provides some valid methods **Map** and **Match**.

```
public class MockClientRepository
{

    public EitherAsync<string, Client> GetByIdAsync(int id)
        {
            Task<Either<string, Client>> maybeClientTask =
                        System.Threading.Tasks.Task.Run(() =>
          {
            System.Threading.Thread.Sleep(1000);
            Option<Client> maybeClient = clients.SingleOrDefault(x => x.Id == id);
            return maybeClient.ToEither("no Client Found");
          });
            return maybeClientTask.ToAsync();
        }

    }
```

Github: source

After that we can use again **Map**, and **Match**. The Match is the difference of `EitherAsync` compared to Task<Either<>>

```
public class Controller
{
        MockClientRepository clients = new MockClientRepository();
        public Task<string> GetNameByIdAsync(int clientId)
                => clients.GetByIdAsync(clientId)
                     .Map(client => client.Name)
                     .Match(
                      Right: (name) => name,
                      Left: (e) => $"error {e}");
}


var clientName = await new Controller().GetNameByIdAsync(3);
```

an alternative writing would be:

```
public class Controller
{
        MockClientRepository clients = new MockClientRepository();
```

```
        public async Task<string> GetNameByIdAsync(int clientId)
           => await clients.GetByIdAsync(clientId)
             .Map(client => client.Name).Case switch
        {
            LeftCase<string, string>(var error) => error,
            RightCase<string, string>(var name) => name,
            _ => throw new NotImplementedException(),
        };
    }
```

Note :The use of `.Case` without the need to first await

Github: source

4.17 Functors from Algebraic Data Types

We have seen an implementation of the map function for our algebraic definition of the list data structure in section 3.4. I will restate the implementation here:

```
public class Cons<T> : ListBase<T>
{
       ...
        public override ListBase<T1> Map<T1>(Func<T, T1> f) =>
                            new Cons<T1>(f(Value), Rest.Map(f));
}

public class Empty<T> : ListBase<T>
{
        public override ListBase<T1> Map<T1>(Func<T, T1> f) => new Empty<T1>();
}
```

Now that we know much more about functors, we can extrapolate the implementation of the map for a binary tree. So, let us go from Linear to quadratic.

The tree is also **a recursive data structure**. The definition of a binary tree is this:

Tree (**a**)**= leaf**(**a**) **+ Node**(**Tree**(**a**) **, Tree** (**a**))

Which states **that a Tree(a) can be a Leaf (a) or a Node of two Trees of type a**.

[Sidenote: The list can be viewed as a tree, which has its Right side of all nodes are always leaves: Tree a = Leaf a + Node (Leaf a) * (Tree a) ≅ List a = a + a * (List a)]

```
public abstract class Tree<T>
{
        public abstract Tree<T1> Map<T1>(Func<T, T1> f);
}
```

```
public class Node<T> : Tree<T>
{
        public Tree<T> Left { get; set; }
        public T Value { get; set; }
        public Tree<T> Right { get; set; }

        public Node(Tree<T> left, Tree<T> right)
        {
            this.Left = left;
            this.Right = right;
        }
}

public class Leaf<T> : Tree<T>
{
        public T V { get; }
        public Leaf(T v) => V = v;
}
```

In order to create a functor based on this algebraic data type we define a valid **map** method. We will again follow our **method of structural induction** in order to define the **map**.

First, because it is a **coproduct**, we must define the map method in both subtypes :`Leaf` and `Node` of the coproduct.

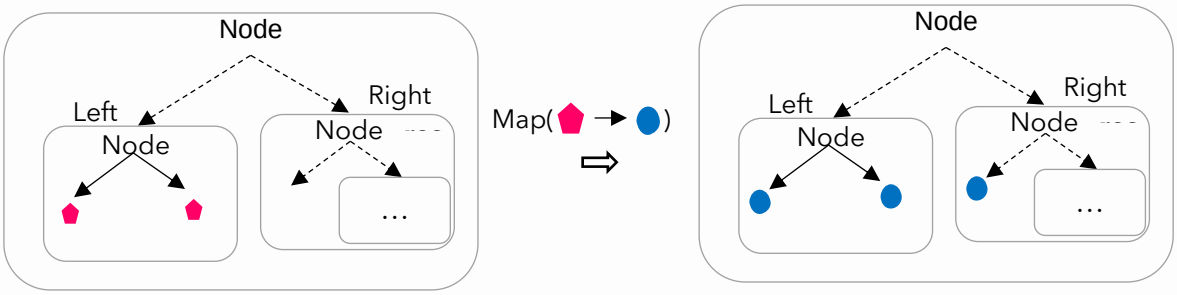
1. We start with the `Leaf` which is the base case of the structural induction. We must return something of the same type. That's why we return a new `Leaf` with the function applied to the value

   ```
   Tree<T1> Map<T1>(Func<T, T1> f) => new Leaf<T1>(f(V))
   ```

   A leaf is just an Identity functor. Hopefully, you recognized it.

2. The `Node` part is a product of two trees (`Left, Right`) both are of the same type as the one for which we want to define the map. The only way to deal with it is to assume that both must have a map function already in place that works as expected (this is the inductive hypothesis again),

   ```
   Tree<T1> Map<T1>(Func<T, T1> f) => new Node<T1>(Left.Map(f), Right.Map(f))
   ```



The whole tree, with the map method, now becomes:

```
public class Node<T> : Tree<T>
{
        public Tree<T> Left { get; set; }
        public T Value { get; set; }
        public Tree<T> Right { get; set; }

        public Node(Tree<T> left, Tree<T> right)
        {
            this.Left = left;
            this.Right = right;
        }

        public override Tree<T1> Map<T1>(Func<T, T1> f) =>
                                    new Node<T1>(Left.Map(f), Right.Map(f));
}

public class Leaf<T> : Tree<T>
{
        public T V { get; }
        public Leaf(T v) => V = v;
        public override Tree<T1> Map<T1>(Func<T, T1> f) => new Leaf<T1>(f(V));
}
```

Run This: .Net Fiddle

One usual variation of the tree data structure is to **include an additional value at the node level**:

```
public class Node<T> : Tree<T>
{
        public Tree<T> Left { get; set; }
        public T Value { get; set; }
        public Tree<T> Right { get; set; }

        public Node(Tree<T> left, T value ,Tree<T> right)
        {
```

```
            this.Left = left;
            Value = value;
            this.Right = right;
        }
}
```

Tree (a)= leaf(a) + Node(Tree(a) , a , Tree (a))

Can you extend the previous definition of map to also lift the value of the node?

Can you write a symbolic equation for the definition of the map for this tree version?

C# 8.0 Pattern Matching

We can rewrite the map using the `switch` and extension methods in this way we skip the custom MatchWith method

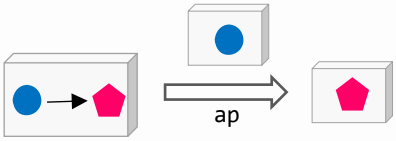
```
public static Tree<T1> Map<T, T1>(this Tree<T> @this, Func<T, T1> f) =>
             @this switch
             {
                 Leaf<T> { Value: var v } => new Leaf<T1>(f(v)),
                 Node<T> { Left: var l, Right: var r } =>
                                 new Node<T1>(l.Map(f), r.Map(f)),
             };
```

Run This: .Net Fiddle

4.18 Applicative Functors

Applicative is a relatively easy concept to grasp. It is a **functor F that contains a function** (**a→ b**) instead of just a value, and we must provide a valid method that can take another functor with a value F(a) and give an F(b). We call this **apply** (usually the method is named **Ap**) :

Ap: F(a → b) → F (a) → F(b)



you can compare this with the map signature **Map :** (**a** → **b**) → **F**(**a**) → **F**(**b**) the only difference is the initial **F.**

If we take the Identity functor as an example, we can implement the apply as follows:

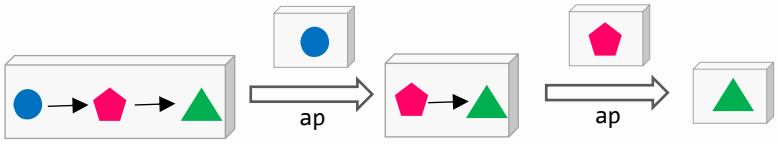
```
public static class FunctionalExtensions
{
    public static Id<T1> Ap<T, T1>(this Id<Func<T, T1>> @this, Id<T> fa)  =>
             @this.Map(f => fa. MatchWith (f));
}

new Id<Func<int, int>>(x => x + 1).App(new Id<int>(1))
```

run this: fiddle

if we use currying, we can chain multiple applicatives like that:

```
var chain = new Id<Func<int, Func<int, int>>>(x => y => x + y)
.Ap(new Id<int>(1))
.Ap(new Id<int>(1));
```

run this fiddle



What is happening here? Well the first app applies the Id(5) to the `Id ( x =>y=> x +y)` this gives us an `Id ( y=> 1+y),` and since we still have a function inside a functor, we can use `Ap` again `Id ( y=> 1+y).Ap(Id(1))` which eventually gives us `Id(2).`

Brace yourselves because we are going to use this extensively throughout this book. You must appreciate how nice this property of the applicative is. **In a way, we can chain multiple operations without passing any values around but binding them through a partial application.**

Obviously if there was a curried function of arity 10 we could chain 10 applicatives:

```
Id(a =>b=>c=>d=>e=>…=>{}).ap(Id(_)).ap(Id(_)) .ap(Id(_))…. .ap(Id(_))
```

run this fiddle

We will represent this as ⟨f _ _ _ …_ ⟩

So, let's take this one step further than the simple Id functor and define applicative on a reader functor.

4.19 Reader Applicative Functor optional

A reader applicative is a reader that has a function stored as the expression `expr`. The implementation is a bit complex. Try to imagine that you want to combine a reader that has a function and a reader that has just a value. This is my implementation:

```
public static class FunctionalExtensions
{
   public static Reader<Env, T1> App<Env, T, T1>
                (this Reader<Env, Func<T, T1>> @this, Reader<Env, T> fa) =>
   new Reader<Env, T1>(env => { return @this.Map(f => fa.Map(f).Run(env)).Run(e
nv); });

}

var r = new Reader<(int, int), Func<int, int>>(g => (r) => g.Item1 + r)
                .App(new Reader<(int, int), int>(g => 1))
                    .Run((1, 2));
```

Run This: .Net Fiddle

if we take the applicative operation

Ap: f(a⟶ b)⟶f(a)⟶f(b) (this is the name for ⟨⊗ … ⟩)

for the reader specifically by substituting the f with the ((⟶) env) [that's (env ⟶_)] we get

⟨⊗⟩: ((⟶)env) (a⟶ b) ⟶((⟶)env) a⟶((⟶)env) b

With some reductions we get the form

(env ⟶ (a⟶ b)) ⟶ (env ⟶ a) ⟶ (env ⟶ b)

If for example, we have a curried function of arity 2 like `x=>y=>y+x` we can do that

⟨add ⊗1 ⊗2 ⟩⟶ ⟨ 1+2 ⟩ we will get a reader of the addition

```
var t = new Reader<object, Func<int, Func<int, int>>>(g => x1 => y1 => x1 + y1)
 .App(new Reader<object, int>(g => 1)) .App(new Reader<object, int>(g => 1))
   .Run(default);
```

Run This: .Net Fiddle

Category Theory

Monoidal Functor

Let $(C,\otimes_C,1_C)$ and $(D,\otimes_D,1_D)$ be two monoidal categories. A lax monoidal functor between them is

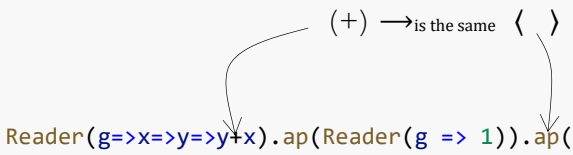
1. a functor $F:C\longrightarrow D$,
2. a morphism $\epsilon:1_D\longrightarrow F(1_C)$

3. a natural transformation $\mu_{x,y}: F(x) \otimes_D F(y) \longrightarrow F(x \otimes_C y)$ for all $x,y \in C$

$\epsilon: 1_D \longrightarrow F(1_C)$ represents the pure operation of the applicative, that allows us to get into an applicative

$F(x) \otimes_D F(y) \longrightarrow F(x \otimes_C y)$ is the applicative operation of **ap**. The interesting thing with this definition is that it connects the "multiplication" $\otimes_D$ of one category with "multiplication" $\otimes_C$ of the other category.

In our example above the equivalence is that the multiplication $\otimes_{int}$: (+) of the integers is now the same as the $\otimes_{Reader}$, applicative operation ap ⟨ ⟩ of the reader :



Two completely different domains but actually. **ap** is equivalent with + in a different level of abstraction. Cool right.

Also as we will see in the traversable section the Arrays "multiplication" $\otimes_{Array}$ concat (:) will be lifted to the now the same as the $\otimes_{Reader}$, applicative operation ap ⟨ ⟩ of the reader

(:) → ⟨ ⟩

```
Reader(g=>x=>y=>[y].concat(x)).ap(…).ap(Reader(…)) ;
```

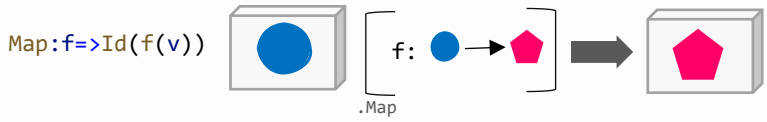
(Hopefully you would recognized that $\mu_{x,y}: F(x) \otimes_D F(y) \longrightarrow F(x \otimes_C y)$ is a monoid homomorphism)

Monads

«Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem ?»
-Brief, Incomplete and Mostly Wrong History of Programming Languages,

5 Monads

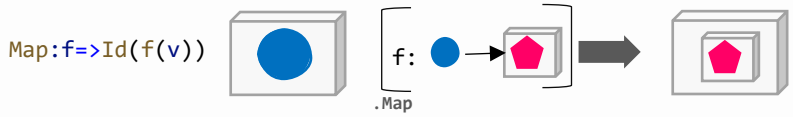
If you remember in the section "Intro to Functors" we stated that when we use the `map` on a function **f**: **A** → **B** we transform the value A inside the functor **F**<**A**> to a new type **F**<**B**>.



Requirement 2

A map method that takes a function f:a→b and transforms an Id(a) to an Id(b)

Sometimes it might happen that the type **B** is inside a Functor **F** is itself a **F<B>**, this means that the function f might look like this **f: A** → **F**(**B**)**.** In those situations, after the map we end up with something like **F**<**F**<**B**>>. A **Functor-In-A-Functor** situation.



You can see this in the following example

```
Id<Id<int>>IdInId = new Id<int>(5).Map(x => new Id<int>(x + 1));
```

We will now extend our Identity functor to make it into an identity monad.

```
public class Id<T>
{
    public T Value { get; set; }
    public Id(T value) => Value = value;
    public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
```

```
    public Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value);
}
```

The new thing here is the bind method. `Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value)`

```
Id(5).Bind(x=>Id(x+1))   == Id(6)
```

Run This: .NET Fiddle

this bind method has a type of **Bind:(T→ M <T1>) →M<T1>** that takes a function **T→M <T1>**, which, when provided with the contained value v it returns a new monad **M<T1>** of type T1.

```
    var result = new Id<int>(2)
                          .Bind(x => new Id<int>(2)
                             .Bind(y => new Id<int>(x + y)));
```

Run This: .NET Fiddle

Let's put a monad in a monad and Bind with the identity x=>x to get the initial Monad back.

```
var IdInId = new Id<int>(5).Map(x => new Id<int>(x + 1));
var justId = IdInId.Bind(x => x);
```

Run This: .NET Fiddle

The difference between bind and map in the identity monad is that map passes the lifted value to a monad constructor `Id(f(v))` while bind does not: `f(v)`. In other monads the map will be significantly different than the bind.

The same reasoning goes for all functors, here some other examples leading to the same situation of a Functor in a Functor if we use the Map:

```
New IO(()=>5).Map(x=> new IO(()=> x+1)) === new IO(()=>new IO(()=> 6))

var a = new[] { 4 }.Select(x => new[] { x + 1 });
var b = new[] { new[] { 4 } };

new  Some<int>(4).Map(x=> new Some<int>(x+1)) === new Some (new Some (5))
```

so its usual to define a Bind in most Functos and thus upgrade them into monads.

Conventions: Bind and SelectMany
As a convention we will name our mapping method **Bind** nonetheless in some instances we might also use **SelectMany if we need LINQ support.** Map and Select will be used interchangeably in this book. The language-ext library provides **Bind** and **SelectMany** methods aliases

5.1 Monads in Category Theory optional

"A monad is just a monoid in the category of endofunctors"
- A Brief, Incomplete, and Mostly Wrong History of Programming Languages

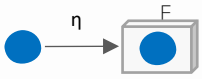
This definition has become an internal joke in the functional community. Because it mystifies a concept that is extremely easy to grasp from a programming perspective. However, in category theory because of the abstraction involved, the definition of a monad can rightfully be difficult to grasp. We already have seen all the components of this definition.

The equivalent definition of a monad is a **monad** is an endofunctor **F** (a functor mapping a category to itself), together with two natural transformations required to fulfil certain coherence conditions (aka laws) .

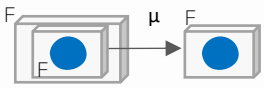
Monads have two operations called return and join (those are the **monoid** operations):

1. $\eta: v \longrightarrow F(v)$ //Return (this is the empty of the monoid)
2. $\mu: F(F(v)) \longrightarrow F(v)$ //Join (this is the concat of the monoid)

1) In this first operation v → F(v) we can start from an object v and get a container F(v). This is our way to get monads. This is just a fancy way to say **Monad constructor** in programming terms :



2) The second one F(F(v)) → F(v) means that we can join/flatten the two containers in one.

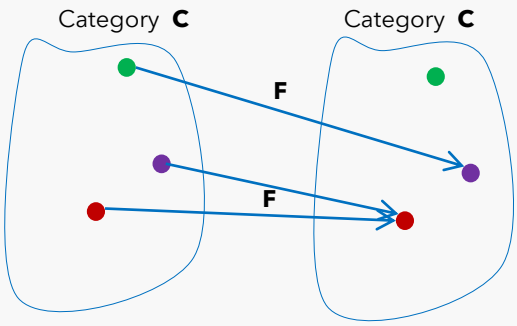


You might feel that this was straightforward, and you can see the connection with the Id and array monads already seen. The concept that we have hidden here is that those are natural transformations. So, let us try to see what this mean in a visual way. First, we are going to see what Endofunctor means. This might need some mathematical imagination.

Endofunctor

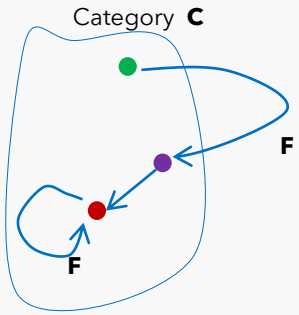
A Functor from a category to itself is called an endofunctor. Given any category C, we can create a new category End(C) is called the endofunctor category of C. The objects of End(C) are endofunctors F:C → C, and the morphisms are **natural transformation** between such endofunctors.

In a visual form this would look like the following. First, we have a functor Between the same Category **C**. Since this section is more abstract, I am not going to give a programming example just yet. Let's first say that we have this category C that has some Objects and arrows (we omit the arrows here).The objects are just those three ●, ●, ●

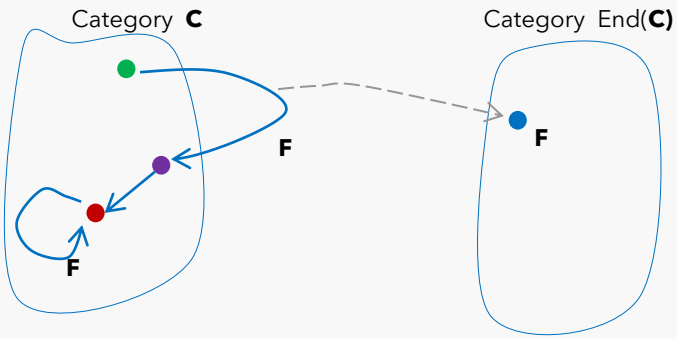


The functor F transforms the green ●→● the purple ●→● and the red ●→●

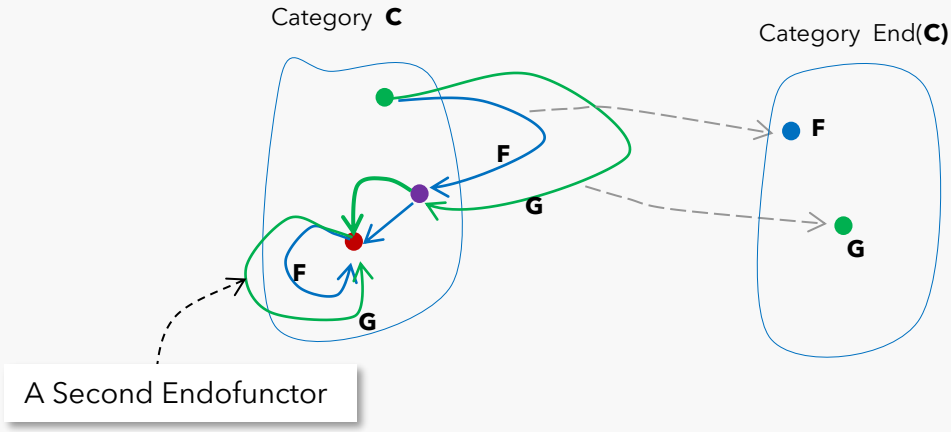
But since the object are the same, we could depict it in the same diagram.



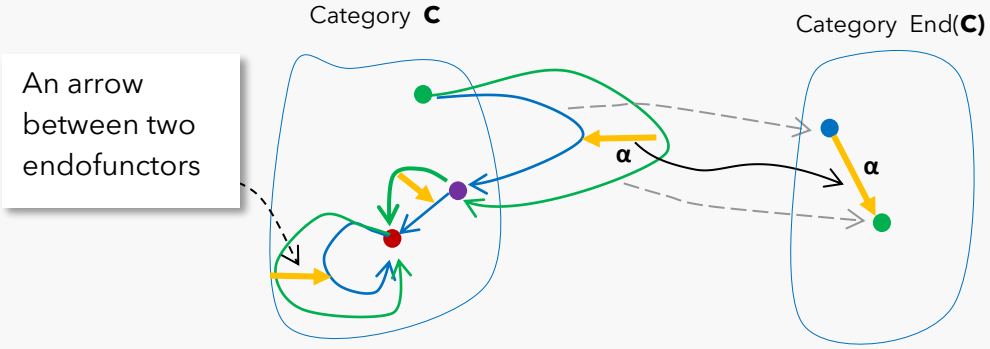
Now we can place this F as a point in the Category of Endofunctors where the Objects are Functors in C



Now we could add some second endofuctor. [The arrows should not be the same, but I used arrow between the same objects for the second functor in order for the diagram to not get chaotic visually.]

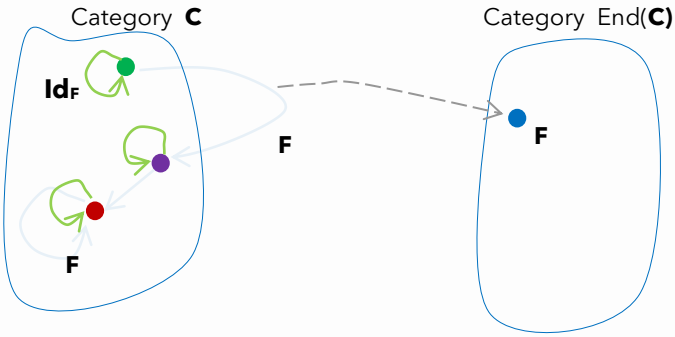


and Finally a natural transformation **α** between those Endofunctors **F,G** is depicted with the yellow arrow

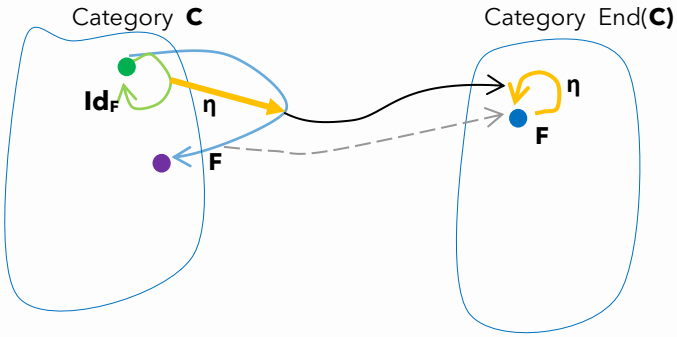


A monoid in this endofunctor category is called a monad on C.

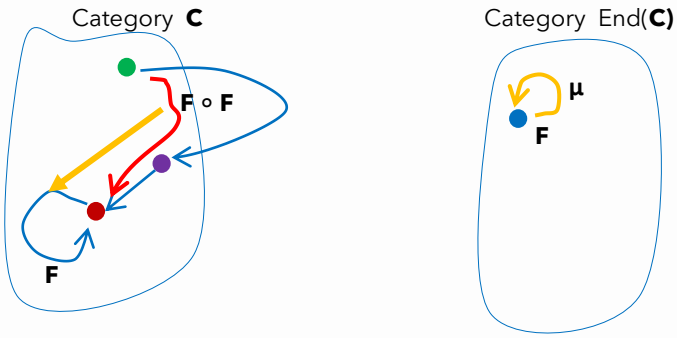
For the first operation which is η: $1_F \rightarrow F$ from the identity $\mathbf{Id_F}$ of the Functor F depicted as $\mathbf{Id_F}$:●→● the light green arrow around the green object ●



A natural transformation from $\mathbf{Id_F}$ to **F** looks like the yellow arrow below :



Now for the second natural transformation μ: $F \circ F \rightarrow F$ which is from a composition of F with another **F**. We can visually see what that means if we can replace the two arrows ●→● and ●→● with the composition ●→●→● which is seen as the red arrow bellow **F ∘ F :** ●→● now there should be a natural transformation from this arrow to another **F** arrow. This is the operation $F \circ F \rightarrow F$



5.2 The List Monad

If we have a list of integers and try to get a list of the squares, cubes and quads, we might try to go about it by using map:

```
var a = new[] { 2,3 }.Select(x => new[] {  x * x, x * x * x , x * x * x *x});
```

Run This: .NET Fiddle

this would create a list- in-a-list `var list = [[4,8,16], [9,27,91]]`

It would be great if there was a way to just get the union of those sub-list elements. Well that's exactly the purpose of the list as a monad throught the use of SelectMany.

```
var b = new[] { 2,3 }.SelectMany(x => new[] { x * x, x * x * x , x * x * x *x});
```

Let's revisit one by one the operations of a monad specifically for the list.

1. $\eta : c \longrightarrow T(c)$ represents the array construction by the element `new[] { _ }`. If we apply list constructor once more we get a list in a list [[ ]]
2. $\mu : T(T(c)) \longrightarrow T(c)$ is the join operation that reverses the one level of containment that was added and gives us the T(c) again [ ]
   `new[] { new[] { _ }}.SelectMany(x=>x) = new[] { _ }`

Run This: .NET Fiddle

5.3 The Identity Monad

Identity Monad is the simplest form of a monad.

```
public class Id<T>
{
    public T Value { get; set; }
    public Id(T value) => Value = value;
    public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
    public Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value);
}
```

It does not do much. It doesn't have any computational value. The new thing with the Identity Monad is the bind method as we already saw:

```
Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value)
```

```
 var result = new Id<int>(2)
                          .Bind(x => new Id<int>(2)
                             .Bind(y => new Id<int>(x + y)));
```

Run This: .NET Fiddle

5.3.1 Optional Monad laws for Identity Monad

A monad has 3 straightforward laws, which are:

1. Left identity, applying the unit function to a value and then binding the resulting monad to a function f is the same as calling f on the same value.

```
Func<int, Id<int>> f = x => new Id<int>(x + 2);
var m1 = new Id<int>(5).Bind(f);
var m2 = f(5);
```

Run This: .NET Fiddle

2. Right identity, binding the unit function to a monad doesn't change the monad

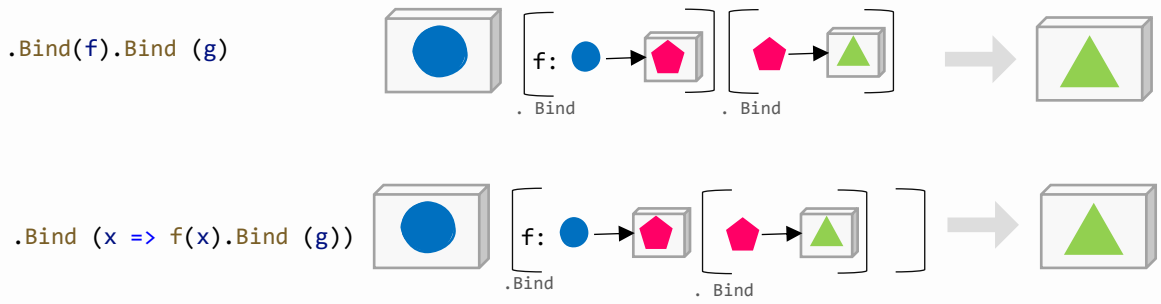
```
var value = 5;
var m1 = new Id<int>(value).Bind(x => new Id<int>(x));
var m2 = new Id<int>(value);
```

Run This: .NET Fiddle

3. Associativity, if we have a chain of monadic function applications, it doesn't matter how they are nested: `m.bind(f).bind(g) == m.bind(x => f(x).bind(g));`

```
Func<int, Id<int>> f = x => new Id<int>(x + 2);
Func<int, Id<int>> g = x => new Id<int>(x * x);
var value = 5;

var m1 = new Id<int>(value).Bind(f).Bind(g);
var m2 = new Id<int>(value).Bind(x => f(x).Bind(g));
```

Run This: .NET Fiddle



Both left and right identity guarantee that applying a monad to a value will just wrap it: the value won't change, nor the monad will be altered. The last law guarantees that monadic composition is associative. All laws together make code more resilient, preventing counter-

intuitive program behaviour that depends on how and when you create a monad and how and in which order you compose functions.

5.4 Maybe Monad

Maybe monad is extension of the Maybe Functor. To extend our maybe functor implementation into a Monad we must provide a `Bind` method that combines two Maybe monads into one. **There are 4 different ways to combine the 2 possible states of maybe (some,none)** one can see that a valid implementation would be the following :

```
public abstract class Maybe<T>
{
      public abstract T1 MatchWith<T1>((Func<T1> none, Func<T, T1> some) pattern);

      public Maybe<T1> Bind<T1>(Func<T, Maybe<T1>> f) =>
            this.MatchWith((
                    none: () => new None<T1>(),
                    some: (v) => f(v)
                ));
}
```

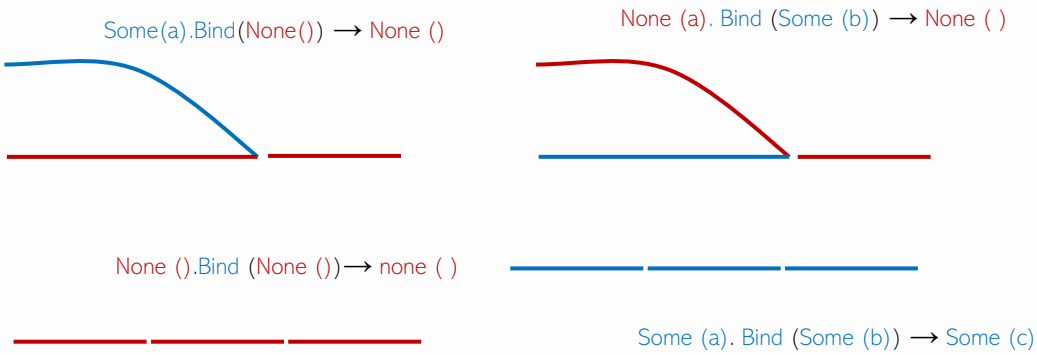
Run This: .Net Fiddle

Or By using the switch instead:

```
 public abstract class Maybe<T>
 {
        public Maybe<T1> Bind<T1>(Func<T, Maybe<T1>> f) =>
         this switch
         {
             None<T>() => new None<T1>(),
             Some<T>(var v) => f(v),
             _ => throw new NotImplementedException(),
         };
}
```

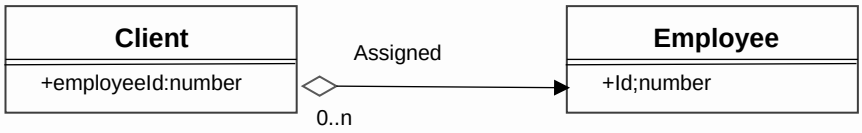
Of course the langiage-ext library has a `Option<>.Bind` with similar implementation. The only combination that leads to a new some (_) is when the two paths that both have values are being combined.

```
Some (x). Bind ( None )
None ( ). Bind ( Some )
Some (x). Bind ( Some )  //only this gives a Some result
None ( ). Bind ( None )
```



Let us say that we have this scenario where we have a list of clients and each client is assigned to an employee. Then let us say we want to get the name of the assigned employee for that client.

This is the standard one-to-many relationships and the way to model it is to add on the Client entity a property that will store the value of the Id of the related employee :




```
public class Client
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int EmployeeId { get; set; }
}
```

We use the bind `.Bind(employees.GetById)` to capture the `Client` value from the `Maybe<Client>` that is returned from the `MockClientRepository` and pass it to the `MockEmployeeRepository.getById` which returns a `Maybe<Employee>`.The final result of the bind is in fact a `Maybe <Employee>`. This is the implementation using the simple our custom Maybe

```
public class MockClientRepository
{
   …
   public Maybe<Client> GetById(int id) => clients.FirstOrNone(x => x.Id == id);
}

public class MockEmployeeRepository
{
   …
   public Maybe<Employee> GetById(int id) => clients.FirstOrNone(x => x.Id == id);
}

public class Controller{
```

```
        MockEmployeeRepository employees = new MockEmployeeRepository();
        MockClientRepository clients = new MockClientRepository();

        public string GetAssignedEmployeeNameById(int clientId) =>
          clients
                .GetById(clientId)
                .Map(client => client.EmployeeId)
                .Bind(employees.GetById)
                .MatchWith((
                    none: () => "there is non employee assigned",
                    some: (employee) => employee.Name
                ));
}
```

Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle

Github: source

5.4.1 Using language-ext Option

Rewriting the same using the **language-ext Option** we get

```
public class MockClientRepository
{
   …

public Option<Client> GetById(int id) => clients.SingleOrDefault(x => x.Id == id);
}

public class MockEmployeeRepository
{
   …

public Option<Employee> GetById(int id) => clients.SingleOrDefault(x => x.Id == id
);
}

public class Controller
{
        MockEmployeeRepository employees = new MockEmployeeRepository();
        MockClientRepository clients = new MockClientRepository();

        public string GetAssignedEmployeeNameById (int clientId) =>
              clients
                .GetById(clientId)
                .Map(c => c.EmployeeId)
```

```
                .Bind(employees.GetById)
                .Match(
                  Some: (employee) => employee.Name,
                  None: () => $" No Employee Found"
                );
}
```

Of course instead of `Match` we could use the `.Case switch`

```
    public string GetAssignedEmployeeNameById(int clientId) =>
              clients
                .GetById(clientId)
                .Map(c => c.EmployeeId)
                .Bind(employees.GetById)
                .Case switch
                    {
                        SomeCase<Employee>(var employee) => employee.Name,
                        NoneCase<Employee> { } => "No Employee Found",
                        _ => throw new NotImplementedException()
                    };
```

Github: source

5.4.2 Using the LINQ syntax with Option Monad

For the final variation of the maybe example will use the LINQ syntax. By defining a `SelectMany` method with the following signature

```
Maybe<T2> SelectMany<T1, T2>(Func<T, Maybe<T1>> f, Func<T, T1, T2> selector)
```

this is the classic Bind but has an intermediate step and a selector. Here is the implementation in case you are interested.

```
public abstract class Maybe<T>
{ …
   #region Linq support Aliases
       public Maybe<T1> Select<T1>(Func<T, T1> f) => Map(f);

       public Maybe<T2> SelectMany<T1, T2>(Func<T, Maybe<T1>> f, Func<T, T1, T2> s
       elector) =>
             MatchWith((
                    none: () => new None<T2>(),
                    some: (v) => f(v).Map(x => selector(v, x))
                ));
   #endregion
}
```

The important thing is that we can now write something like this

```
var r = from x1 in new Some<int>(2)
          from x2 in new Some<int>(3)
          select x1 + x2;
```

Instead of this :

```
var r1 = new Some<int>(2).Bind(x => new Some<int>(3).Map(y => x + y));
```

we could also define `SelectMany` using the existing `Bind`

```
public Maybe<T2> SelectMany<T1, T2>(
       Func<T, Maybe<T1>> f,
       Func<T, T1, T2> selector) =>
             Bind(x => f(x).Map(y => selector(x, y)));
```

the language-ext has LINQ support and we can write now the `GetAssignedEmployeeNameById` function from our example like this :

```
 public string GetAssignedEmployeeNameById(int clientId) =>
                 (from client in clients.GetById(clientId)
                  from employee in employees.GetById(client.EmployeeId)
                  select employee) .Case switch
                       {
                           SomeCase<Employee>(var employee) => employee.Name,
                           NoneCase<Employee> { } => "No Employee Found",
                           _ => throw new NotImplementedException()
                       };
```

Run This: ASP.NET MVC Fiddle
Github: source

5.5 Either Monad

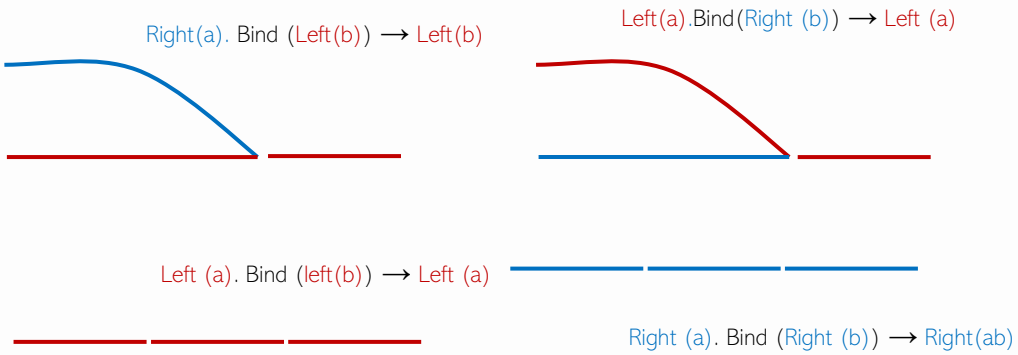
In the same way we can extend the Either functor to a monad by providing a valid bind method:

```
public abstract class Either<TLeft, T>
{
        public abstract Either<TLeft, T1> Map<T1>(Func<T, T1> f);

        public Either<TLeft, T1> Bind<T1>(Func<T, Either<TLeft, T1>> f) =>
            MatchWith((
                left: (e) => new Left<TLeft, T1>(e),
                right: v => f(v)
```

```
            ));
}
```

Run This: .Net Fiddle

The `Left` always stops the computation and returns itself `Left(v)`. This means that for the four possible bind combinations between `Right` and `Left` again only in the case of `Right .Bind(Right )` we get a `Right` . In all other cases the computation stops, and the first `Left` is return



5.5.1 Using Either Monad for exception handling

In this section I want to build up, on our previous discussion on how to use either to address the issue of error handling. This is suitable for cases that we want to return the first failure signified by a `left` (_)

With a minor refactoring from Maybe to Either we can modify our previous section example to display a more specific error message if we could not find an employee to whom the `client` is assigned. Here we have used the **Maybe. `ToEither`.**

```
public static Either<TLeft,T> ToEither<TLeft,T>(this Maybe<T> @source, TLeft defau
ltLeft)
{
    return @source.MatchWith<Either<TLeft, T>>((
                none: () => new Left<TLeft, T>(defaultLeft),
                some: (v) => new Right<TLeft, T>(v)
          ));
 }
```

```
public class MockClientRepository
```

```
{
        public Either<string, Client> GetById(int id) =>
            clients.FirstOrNone(x => x.Id == id)
            .ToEither("No client Found");

}

public class MockEmployeeRepository
{
        public Either<string, Employee> GetById(int id)
        => clients.FirstOrNone(x => x.Id == id)
            .ToEither("No employee Found");
}

public class Controller
{
        MockEmployeeRepository employees = new MockEmployeeRepository();
        MockClientRepository clients = new MockClientRepository();

        public string GetAssignedEmployeeNameById(int clientId) =>
            clients
            .GetById(clientId)
            .Map(client => client.EmployeeId)
            .Bind(employees.GetById)
            .MatchWith((
                    left: (e) => "there was an issue:" + e,
                    right: (employee) => employee.Name
                ));
}
```

Using `toEither` in order to convert the possible None, to an Either. `Left` with a more specific message

Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle

5.5.2 Using language-ext Either<>

Using the language-ext library Either gives us the same code since it has the same implementation for `Bind`

```
public string GetAssignedEmployeeNameById(int clientId) =>
          clients
                .GetById(clientId)
                .Map(client => client.EmployeeId)
                .Bind(employees.GetById)
                .Match(
                     Right: (employee) => employee.Name,
                     Left: (e) => $"error{e}"
                );
```

And of course, we can natively pattern match instead :

```
public string GetAssignedEmployeeNameById(int clientId) =>
          clients
                .GetById(clientId)
                .Map(client => client.EmployeeId)
                .Bind(employees.GetById)
                .Case switch
                {
                    LeftCase<string, Employee>(var e) => $"error{e}",
                    RightCase<string, Employee>(var employee) => employee.Name,
                    _ => throw new NotImplementedException(),
                };
```

Github: source

5.5.3 Using the LINQ syntax with Either Monad

Like the maybe/Option monad, Either has a SelectMany implementation that allows us to use LINQ syntax :

```
var r =  from x1 in Right(2)
         from x2 in Right(3)
         select x1 + x2;
```

which is the same as :

```
var r1 = Right(2).Bind(x => Right(3).Map(y => x + y));
```

which means that we can rewrite `GetAssignedEmployeeNameById`

```
public string GetAssignedEmployeeNameById(int clientId) =>
            (from c in clients.GetById(clientId)
             from e in employees.GetById(c.EmployeeId)
             select e)  .Case switch
                  {
                      LeftCase<string, Employee>(var error) => error,
                      RightCase<string, Employee>(var employee) => employee.Name,
                      _ => throw new NotImplementedException(),
                  };
```

Github: source

5.5.4 Using Either for Validation

We can use the Either for validation. For example, we can chain validation functions that when are not validated return a `Left` with the appropriate validation error. For example, let us say that we have an `isDigit` function that checks if a given input string consists of digits only:

```
Either<TestError, string> isDigit(string str) =>
            str.ForAll(Char.IsDigit)
            ? Right<TestError, string>(str)
            : Left<TestError, string>(new TestError("must be a valid number"));
```

We can also have a function that checks for the maximum length of a string (notice that this is a curried function)

```
Func<string, Either<TestError, string>> maxStrLength(int maxlenght) => str =>
               str.Length < maxlenght
            ? Right<TestError, string>(str)
            : Left<TestError, string>(new TestError("failed max length"));
```

Then could chain those two validations, making sure that a string input satisfies both validations:

```
var res = maxStrLength(3)("abcd").Bind(s => isDigit(s));

res.Match(
      Right: _ =>
            {
                  Console.WriteLine("no errors");
             },
      Left: error =>
            {
                   Console.WriteLine(error);
            });
```

Run This: .Net Fiddle

The problem with the use of either for validation is that we cannot accumulate errors from each validation, instead the composite validation fails at the first unsuccessful validation and returns the error of this validation.

To accumulate errors, we must use the Validation Type from the Language-ext library.

5.6 Validation Monad

In this this section we are going to take a brief look at the validation type. The Validation behaves like the Either but also has an operation to accumulate validation erros.

Firstly, we could rewrite the validation example to use `Validations` instead of Either. In the place of Left we use the `Fail` and in the place of a Right we use a `Success`:

```
Validation<TestError, string> isDigit(string str) =>
                str.ForAll(Char.IsDigit)
              ? Success<TestError, string>(str)
              : Fail<TestError, string>(TestError.New("failed isDigit"));

 Func<string, Validation<TestError, string>> maxStrLength(int maxlenght) => str =>
               str.Length < maxlenght
              ? Success<TestError, string>(str)
              : Fail<TestError, string>(TestError.New("failed maxlenght"));

var res = maxStrLength(3)("aaa").Bind(s => isDigit(s));

res.Match(
       Succ: _ =>
       {
              Console.WriteLine("no errors");
       },
       Fail: errors =>
       {
              errors.ToList().ForEach(error => Console.WriteLine(error));
       });
```

Run This: .Net Fiddle

5.6.1 Using Validation.Apply to Collect validations

`Validation` also has this operator **|** that combines multiple `Validation` types into to one while collecting errors. So now we can write something like this:

```
Validation<TestError,string>bothFailed= maxStrLength(3)("aadda")| isDigit("1e12");
```

now the `bothFailed` is a `Validation<TestError, string>` that contains a list of errors that we can extract with pattern matching :

```
Validation<TestError, string> bothFailed = maxStrLength(3)("aadda") | isDigit("1e1
2");
bothFailed.Match(
              Succ: _ =>
              {
```

```
                Console.WriteLine("no errors");
            },
            Fail: errors =>
            {
                errors.ToList().ForEach(error => Console.WriteLine(error));
            });
```

Run This: .Net Fiddle

Another way to use the validation within language-ext library is to use the `.Apply()` method with a Tuple of `Validations` for example:

```
(maxStrLength(3)("aadda"), isDigit("1e23")).Apply((name, cardNumber) => …)
```

Here all the validations are successful then the result of each one is placed in the lambda inside the `Apply((name, cardNumber) => …)` and we can collect them and return any object by composing them :

```
var result = (maxStrLength(3)("aadda"), isDigit("1e23"))
   .Apply((name, cardNumber) => new Card { Name=name, CardNumber=cardNumber });

result.Match(
            Succ: _ =>
            {
                Console.WriteLine("no errors");
            },
            Fail: errors =>
            {
                errors.ToList().ForEach(error => Console.WriteLine(error));
            });
```

Run This: .Net Fiddle

5.7 Task<T> as Monad

In the Following sections we are going to see how to use Monads in Asynchronous situations. Firstly, we can extend the native `Task<T>` into a monad by providing a Bind method that would allow us to combine Tasks using async/await :

```
public static async Task<T1> BindAsync<T, T1>(
      this Task<T> @source,
      Func<T, Task<T1>> f)
       {
           var value = await @source;
           var continuation = await f(value);
```

```
            return continuation;
        }
```

Or alternatively we can use the `ContinueWith`.

```
public static Task<T1> Bind<T, T1>(this Task<T> @source, Func<T, Task<T1>> f) =>
           source.ContinueWith(task => f(task.Result).Result);
```

this will allow us to write

```
var result = await Task<int>.Run(() => 4)
                .Map(x => x * x)
                .BindAsync(x => Task<int>.Run(()=>x+2))
                .MatchWithAsync((
                left: (e) =>
                {
                    return e.Message;
                },
                right: (v) =>
                {
                    return "result:" + v;
                }
            ));
```

Where the `MatchWithAsync` is defined in order to provide a normalized functional syntax that looks the same for Maybe , Either:

```
public static async Task<T1> MatchWithAsync<T, T1>(this Task<T> @this,
 (Func<Exception, T1> left, Func<T, T1> right) pattern) =>
         await @this.ContinueWith(t =>
         {
             if (t.IsFaulted)
                 return pattern.left(t.Exception);
             else
                 return pattern.right(t.Result);
         });
```

Language -ext has no similar implementation of `Match` and we are not going to see this method again instead we will await the Task to extract the results.

5.8 The Task - Option Monad Combination -Task<Option<>>

We are going to build up the previous example of Option by having both repositories return Task<Option<>> types

```
  public class MockClientRepository
    {
```

```
        public Task<Option<Client>> GetByIdAsync(int id)
        {
            Task<Option<Client>> maybeClientTask =
             System.Threading.Tasks.Task.Run(() =>
            {
                System.Threading.Thread.Sleep(1000);
                Option<Client> maybeClient =
                    clients.SingleOrDefault(x => x.Id == id);

                return maybeClient;
            });

            return maybeClientTask;
        }
    }

    public class MockEmployeeRepository
    {

        public Task<Option<Employee>> GetByIdAsync(int id)
        {
        Task<Option<Employee>> maybeEmployeeTask =
             System.Threading.Tasks.Task.Run(() =>
            {
             System.Threading.Thread.Sleep(1000);

             Option<Employee> maybeEmployee =
                    employees.SingleOrDefault(x => x.Id == id);

             return maybeEmployee;
            });

            return maybeEmployeeTask;
        }
    }
```

And we can compose them using the **BindT** which is a Monad transformer extension of **Bind** on top of the Type `Task<Option< >>`

```
public class Controller
{
        MockEmployeeRepository employees = new MockEmployeeRepository();
        MockClientRepository clients = new MockClientRepository();

   public Task<Option<Employee>> GetAssignedEmployeeNameById(int clientId)
           => clients.GetByIdAsync(clientId)
               .BindT(client => employees.GetByIdAsync(client.EmployeeId));
}
```

GitHub: source

This is the same as

```
public Task<Option<Employee>> GetAssignedEmployeeNameById(int clientId)
    => clients
   .GetByIdAsync(clientId)
   .MapT(client => client.EmployeeId)
   .BindT(employees.GetByIdAsync);
```

Those `GetAssignedEmployeeNameById` returns a `Task<Option<Employee>>` but we could resolve the value early and return `Task<string>`

```
public Task<string> GetAssignedEmployeeNameById2(int clientId)
        => clients
       .GetByIdAsync(clientId)
       .MapT(client => client.EmployeeId)
       .BindT(employees.GetByIdAsync)
       .Map(e => e.Case
       switch
        {
            SomeCase<Employee>(var client) => client.Name,
            NoneCase<Employee> { } => "No Client Found",
            _ => throw new NotImplementedException()
        });
```

Or even use the Linq syntax

```
public Task<string> GetAssignedEmployeeNameById3(int clientId) =>
    (from c in clients.GetByIdAsync(clientId)
     from e in employees.GetByIdAsync(c.EmployeeId)
     select e)
        .Map(e => e.Case
        switch
        {
            SomeCase<Employee>(var client) => client.Name,
            NoneCase<Employee> { } => "No Client Found",
            _ => throw new NotImplementedException()
        });
```

5.9 The Task - Option Monad Combination -OptionAsync<>

Coming back again to the `OptionAsync` we can Modify the previous example to allow the repositories to return `OptionAsyncs`

```
public class MockClientRepository
{
    public OptionAsync<Client> GetById(int id)
    {
        Option<Client> t = clients.SingleOrDefault(x => x.Id == id);
        return t.ToAsync();
    }
}

public class MockEmployeeRepository
{
    public OptionAsync<Employee> GetById(int id)
    {
        Option<Employee> t = employees.SingleOrDefault(x => x.Id == id);
        return t.ToAsync();
    }
}
```

Using `ToAsync` to Convert to `Task<Option< >>` to `OptionAsync< >`

And we can compose them using **Bind** since both are of the same Type of Monad transformer `OptionAsync<Employee>`. **Again, Monad transformers** Provide a valid **Bind (also Match and Map) method** for this new composite type that is formed if we put a Monad on top of another monad.

```
public class Controller
{
        MockEmployeeRepository employees = new MockEmployeeRepository();
        MockClientRepository clients = new MockClientRepository();

        public OptionAsync<Employee> GetAssignedEmployeeNameById(int clientId)
          => clients
        .GetByIdAsy
        .Map(client
        .Bind(employees.GetByIdAsync);
}
```

Standard binding

GitHub: source

We can consume the method it like this :

```
var employee = await new Controller().GetAssignedEmployeeNameById(3).Case
               switch
               {
                   SomeCase<Employee>(var client) => client.Name,
                   NoneCase<Employee> { } => "No Client Found",
                   _ => throw new NotImplementedException()
               };
```

Also we could return a `Task<Option<Employee>>` at the controller level instead of `OptionAsync<Employee>`

```
public Task<Option<Employee>> GetAssignedEmployeeNameById1(int clientId)
          => clients
              .GetByIdAsync(clientId)
              .Map(client => client.EmployeeId)
              .Bind(employees.GetByIdAsync)
              .ToOption();
```

Using the **`.ToOption().`** The `.ToOption()` is the inverse conversion from `OptionAsync< >` to `Task<Option< >>`

We could also just pull the case analysis early inside the `Controller` and return a `Task<string>`

```
public Task<string> GetAssignedEmployeeNameById2(int clientId)
               => clients
              .GetByIdAsync(clientId)
              .Map(client => client.EmployeeId)
              .Bind(employees.GetByIdAsync)
              .ToOption()
              .Map(e => e.Case
              switch
               {
                   SomeCase<Employee>(var client) => client.Name,
                   NoneCase<Employee> { } => "No Client Found",
                   _ => throw new NotImplementedException()
               });
```

GitHub: source

Or just use LINQ syntax :

```
public Task<string> GetAssignedEmployeeNameById3(int clientId)
        => (from c in clients.GetByIdAsync(clientId)
            from e in employees.GetByIdAsync(c.EmployeeId)
            select e)
              .ToOption()
              .Map(e => e.Case
              switch
              {
                  SomeCase<Employee>(var client) => client.Name,
                  NoneCase<Employee> { } => "No Client Found",
                  _ => throw new NotImplementedException()
              });
```

5.10 The Task - Either Monad Combination -Task<Either<,>>

Finally, a combination of Task and Either as monads. Hopefully, those last sections come with no surprise. Again, we modify the Repositories to return a Task<Either<>> each

```
public class MockClientRepository
    {
        public Task<Either<string, Client>> GetByIdAsync(int id)
        {
            Task<Either<string, Client>> eitherClientTask =
                        System.Threading.Tasks.Task.Run(() =>
             {
                 System.Threading.Thread.Sleep(1000);
                 Option<Client> maybeClient =
                         clients.SingleOrDefault(x => x.Id == id);
                 return maybeClient.ToEither("no Client Found"); ;
             });
            return eitherClientTask;
        }
    }

    public class MockEmployeeRepository
    {
        public Task<Either<string, Employee>> GetByIdAsync(int id)
        {
         Task<Either<string, Employee>> eitherEmployeeTask =
       System.Threading.Tasks.Task.Run(() =>
             {
                 System.Threading.Thread.Sleep(1000);
                 Option<Employee> maybeEmployee =
                       employees.SingleOrDefault(x => x.Id == id);
                 return maybeEmployee.ToEither("no Employee Found"); ;
             });
            return eitherEmployeeTask;
        }
    }
```

The composition using `MapT` and `BindT` is straightforward:

```
public class Controller
{
  MockEmployeeRepository employees = new MockEmployeeRepository();
  MockClientRepository clients = new MockClientRepository();
  public Task<Either<string, Employee>> GetAssignedEmployeeNameById(int clientId)
          => clients
              .GetByIdAsync(clientId)
              .MapT(client => client.EmployeeId)
              .BindT(employees.GetByIdAsync);
}
```

GitHub: source

Or use early case analysis to return a Task<string>

```
public Task<string> GetAssignedEmployeeNameById1(int clientId) =>
                  (from c in clients.GetByIdAsync(clientId)
                   from e in employees.GetByIdAsync(c.EmployeeId)
                   select e).Map(e => e.Case switch
                     {
                       LeftCase<string, Employee>(var error) => error,
                       RightCase<string, Employee>(var employee) => employee.Name,
                        _ => throw new NotImplementedException(),
                      });
```

GitHub: source

SideNote : the case is inside the map.`Map(e => e.Case switch…)` because there is no monad transformer support for this combination `Task<Either<string, Employee>>` of a Match. This brings us to the real Monad transformer `EitherAsync<string, Client>` that has its own Bind and Match methods.

5.11 The Task - Either Monad Combination -EitherAsync<>

Using the `.ToAsync()` we can convert any `Task<Either<string, Client>>` into an `EitherAsync<string, Client>`.

Now we could have repositories with methods :

```
EitherAsync<string, Client> GetByIdAsync(int id)
```

And

```
EitherAsync<string, Employee> GetByIdAsync(int id)
```

Which we can compose them :

```
public class Controller
{

  MockEmployeeRepository employees = new MockEmployeeRepository();
  MockClientRepository clients = new MockClientRepository();

  public EitherAsync<string, Employee> GetAssignedEmployeeNameById1(int clientId)
       => clients
           .GetById(clientId)
           .Map(client => client.EmployeeId)
           .Bind(employees.GetById);
}
```

GitHub: source

And use it like this :

```
var employeeName = await new Controller().GetAssignedEmployeeNameById1(1)
                    .Case switch
                {
                    LeftCase<string, Employee>(var error) => error,
                    RightCase<string, Employee>(var employee) => employee.Name,
                    _ => throw new NotImplementedException(),
                };
```

Some other variation could be :

```
public async Task<string> GetAssignedEmployeeNameById2(int clientId)
                      => await clients
                          .GetById(clientId)
                          .Map(client => client.EmployeeId)
                          .Bind(employees.GetById).Case switch
                   {
                      LeftCase<string, Employee>(var error) => error,
                      RightCase<string, Employee>(var employee) => employee.Name,
                    };
```

GitHub: source

We can also use the `.ToEither()` to convert the `EitherAsync<string, Employee>` into a `Task<Either<string, Employee>>` for example a variation could be this :

```
public Task<string> GetAssignedEmployeeNameById2(int clientId) =>
                     (from c in clients.GetById(clientId)
                      from e in employees.GetById(c.EmployeeId)
                      select e)
                       .ToEither()
                       .Map(e =>
                       e.Case switch
                    {
                      LeftCase<string, Employee>(var error) => error,
                      RightCase<string, Employee>(var employee) => employee.Name,
                    });
```

This is the Map of the `Task`

Traversables

Optional Chapter

6 Catamorphisms Again

This Chapter is about Folding, Decomposing, various structures it's all about **getting values out of data structures** in an elegant and consistent manner. The path that we will follow will involve many concepts from category theory ,like the concept of **catamorphism** (from the Greek: κατά "downwards" and μορφή "form, shape") its dual anamorphism (from the Greek: ανά "upwards" and shape), and their combination hylomorphism (from the Greek: ὕλη *hyle* "matter" and shape) and probably is the most difficult part of this book. Feel free to skip any part you don't like or find interesting and maybe revisit in a later time.

6.1 A brief mentioning of F-algebras

If F is a Functor, then an **F-algebra** is a pair (*A*, α), where *A* is called carrier type and α is a function F(A) → A called **structure map**. The F-Algebra is a way to get an **A out of a F(A).** It is an *evaluation*.

For our Peano Numerals functor one algebra with carrier type the integers would be this

```
public abstract class Numeral    {    }

public class Succ : Numeral
{
    public Numeral Rest { get; set; }
    public Succ(Numeral rest)
    {
       this.Rest = rest;
    }
}

public class Zero : Numeral    {    }
```

```
(Func<int> Zero, Func<int, int> Succ) algebraInt  = (
             Zero: () => 0,
             Succ: (n) => n + 1
          );
```

That's called the **structure map** but when we say algebra, we will mean this function

This algebra tells us how to *interpret* a `Numeral` Expression into an integer. This algebra is the *intended interpretation* (also called a *standard model* a term introduced by Abraham Robinson in 1960) because it gives us the intended "meaning" of the Expression, which in this case was to represent numbers using a formal language[1]. Nevertheless, we can have many algebras based on this Functor. Here some more:

The algebra below needs no further explanation.

```
(Func<string> Zero, Func<string, string> Succ) algebraString = (
     Zero: () => $"Zero( )",
     Succ: (Rest) => $"Add one -({Rest })"
);
```

The thing is that we cannot apply the algebra just yet to an expression like `Succ(Succ(Succ( Succ(Zero())))))` for example. The algebra is for evaluating only one **"step" or "layer"** of the expression. We must somehow push the algebra downwards, perform an evaluation, and then compose the results. **The catamorphism is this exact process**.

Category Theory — F-Algebra

If C is a category, and $F : C \to C$ is an endofunctor of C, then an **F-algebra** is a pair (A, α), where A is an object of C and α is a morphism $F(A) \to A$ the a often called *structure map*.

6.2 Catamorphisms

Now we are going to define a `Cata` method for the `Numeral` union type, which will take an algebra and evaluate the `Numeral`. Here the implementation will be ad hoc-specifically for the `Numeral` structure.

```
T1 Cata<T1>(this Numeral @this, (Func<T1> Zero, Func<T1, T1> Succ) algebra)
     =>@this.MatchWith<T1>(pattern: (
                         Zero: () => algebra.Zero(),
                         Succ: (Rest) => algebra.Succ(Rest.Cata<T1>(algebra))
                 ));
```

pass the algebra to the `Rest`

Use the algebra to evaluate

[1] Lectures on Semantics: The initial algebra and final coalgebra perspectives

Run This: .Net Fiddle

[**SideNote**: We have seen Recursive definitions before using `MatchWith.`

For the first time we are creating a method that takes as an argument `(Func<T1> Zero, Func<T1, T1> Succ) algebra` which is of the same type as the the `MatchWith` argument. **Inception**. The `Cata` is a generalised `MatchWith` for composite Algebraic types. Or `MatchWith` is just the `Cata` for elementary Types]

This definition follows *the structural induction* and should be easy to follow by now. Now we can evaluate a numeral using our algebras above.

```
(Func<int> Zero, Func<int, int> Succ) algebraInt  = (
             Zero: () => 0,
             Succ: (n) => n + 1
          );

var four = fourNumeral.Cata<int>(algebraInt); //"4
```

Run This: .Net Fiddle

The `Numeral` functor has two parts the type of `Zero` is () ⟶ **1** this means that we can get an initial object (“”,[],0 etc) depending on A, and the type of `Succ` is A ⟶ A since the `Numeral` is a union we should add them `Numeral:`A ⟶ 1+ A

You might already have seen the `ToString` () implementation for `Numeral`

```
public static string ToString(this Numeral @this) =>
            @this.MatchWith(pattern: (
                Zero: () => $"Zero( )",
                Succ: (Rest) => $"Succ({Rest.Show()})"
            ));
```

The Cata just generalize this by extracting the Recursion mechanics and allow `Cata` to be used with different algebras without defining any additional extension methods.

For example, We could provide a different `ToString` by providing custom patten matching algebras. For example, this is a different mapping from `Numeral` ⟶ `string`

```
 (Func<string> Zero, Func<string, string> Succ)  justExclamationmarks  = (
                Zero: () => $"",
                Succ: (Rest) => $"!{Rest }"
            );
```

C# 8.0 pattern matching

We can instead use C#8.0 pattern matching instead we can unobtrusively attach `Cata<T1>` on the `Numeral` Type

```
public static partial class FunctionalExt
   {
       public static T1 Cata<T1>(this Numeral @this,
                           (Func<T1> Zero, Func<T1, T1> Succ) algebra) =>
               @this switch
               {
                  Zero { } => algebra.Zero(),
                  Succ { Rest: var rest } => algebra.Succ(rest.Cata<T1>(algebra))
               };
   }
```

Run This: .Net Fiddle

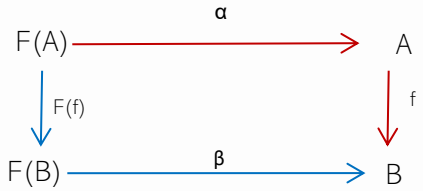
6.3 Initial algebra optional

This section is optional. You might want to revisit the whole 6th section even if you don't feel yet that everything "connects".

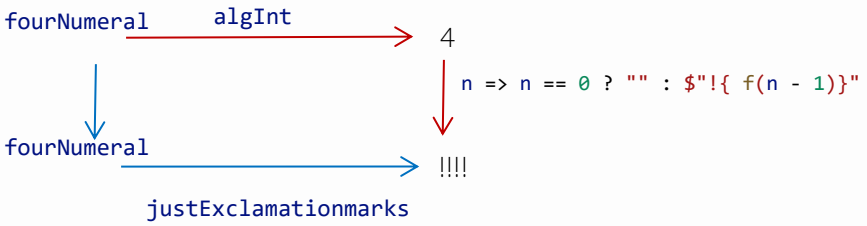
6.3.1 F-Algebras Homomorhisms

Category theorists will form a category out of anything so why not get a category of F-Algebras. So, let's take two F-Algebras and see the relationships.

A homomorphism from an *F*-algebra (*A*, α) to an *F*-algebra (*B*, β) is a morphism $f: A \rightarrow B$ such that $f \circ \alpha = \beta \circ F(f)$, according to the following diagram:



What that practically means is that for example we had two algebras for the `Numeral` functor




```
justExclamationmarks  = (Zero: () => $"", Succ: (Rest) => $"!{Rest }"  );

var algInt = (Zero: () => 0, Succ: (Rest) => Rest + 1   );
```

Run This: .Net Fiddle

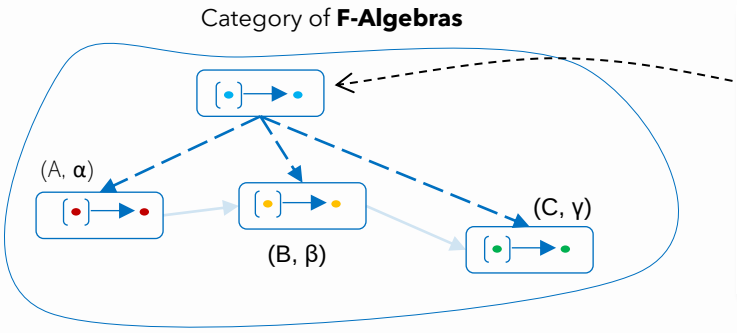
we can go from the `Succ(Succ(Succ(Succ(Zero()))))` to 4 obviously , but also we can get !!!! by using the `justExclamationmarks` to consume this `fourNumeral.` And then use this function `f = n => n == 0 ? "" : $"!{ f(n - 1)}"` to go from 4 to !!!!

The truth is that we can go from the 4 to any other possible interpretation of any functor with type 1+A for any A because the integers and the `algInt` is an initial algebra for our `Numeral` functor. Initial algebra means that for any other algebra there is an arrow (which represents a Homomorhism in the category of F-algebras) to that object from the Initial algebra. In Essence this proves that we **can represent Iteration** with this

```
Func<int, string> recursion = null;
recursion = n => n == 0 ? algebra.E() : $"!{ algebra.F(recursion(n - 1))}";
```

for any algebra : `(Func<string> E, Func<string, string> F) algebra = (E: () => $"", F: (Rest) => $"!{Rest }");`

Run This: .Net Fiddle



The natural numbers together with the tuple (0, n=>n+1) form the **initial object in the category of F-Algebras** for the `Numeral`

The point of all this is **that the initial algebra of the Peano functor `Numeral` is semantically equivalent to a for () loop.**

Category Theory Natural Number System

This generalization of the simple Peano arithmetic to other domains, first presented by Lawvere and its known as the Peano-Lawvere axiom. This triple (`Numeral` , `Succ`, `Zero`) because it is "Equivalent" to the natural numbers (N,S,0) is what is called a **natural number system** (in TS Type category in our case), and `Numeral` a natural number object.

We have seen another natural number system residing in lambda calculus; the church encodings:

```
0 ≅ λf. λf.  x
1 ≅ λf. λf.  f(x)
2 ≅ λf. λf.  f(f(x))
```

6.4 Catamorphisms for Trees

We already have seen the Peano Numerals as the simplest recursive Algebraic types. We will also look at the binary tree. Let us say we have a simple algebra for this tree. This algebra sums up the values of a node or returns the value of a leaf if it is a leaf.

```
var algSumInt = (

        Leaf: (v) => v,
        Node: (l, v, r) => l + v + r
)
```

Our Cata method would look like this:

```
public static T Cata<T>(this Tree<T> @this,
       (Func<T, T> Leaf, Func<T, T, T, T> Node) algebra) =>
               @this.MatchWith(algebra: (
                 Leaf: v => algebra.Leaf(v),
                 Node: (l, v, r) =>
                 algebra.Node(r.Cata<T>(algebra), v, l.Cata<T>(algebra)
             )
));
```

Run This: .Net Fiddle

Hopefully, the implementation of a method, like Cata by structural induction on a tree, should be familiar by now. In case you don't still feel confident I will go through with the construction one more time below.

```
var tree = new Node<int>(new Node<int>(new Leaf<int>(1), 2, new Leaf<int>(3)),
                4,
                new Node<int>(new Leaf<int>(5), 6, new Leaf<int>(7)));

            //(Func<T, T> Leaf, Func<T, T, T, T> Node) algebra
            Console.WriteLine(
                tree.Cata(algebra: (
                Leaf: (v) => v,
                Node: (l, v, r) => l + v + r))
            );
```

Run This: .Net Fiddle

Optional

This is almost identical with the previous definitions in order to highlight the fact that this is an automated process for simple algebraic structures.

Again, because it is a coproduct, we must define the cata method in both subtypes `Leaf` and `Node`.

1. We start with the `Leaf`, because this is the base case of the structural induction. We can apply the Algebra directly to this leaf since there is nothing else to do.

   ```
   Leaf: v => algebra.Leaf(v),
   ```

2. The `Node` is a product of two trees `(left, v, right)` and a value. The only way to deal with it is to assume that those must have a `Cata<T>` function that works as expected (that is is the inductive hypothesis again) if we pass the algebra `algebra` on those nodes the result must be an integer in this way we will have three integers -one from the left node : `l.Cata<T>(algebra)` [this returns a T type] the node `v`, and t one from the right node: `r.Cata<T>(algebra)` [this also returns a T type], and we can use the `algebra` to call the `algebra.Node(___)` In order to evaluate all those three results that are of the same type T now

   ```
   Node: (l, v, r) => algebra.Node(r.Cata<T>(algebra), v, l.Cata<T>(algebra)
   ```

Run This: .Net Fiddle

C# 8.0 pattern matching

Again, we can reformulate the `Cata` using `switch`

```
public static T1 Cata<T, T1>(this Tree<T> @this,
            (Func<T, T1> Leaf, Func<T1, T1, T1> Node) algebra) =>
                @this switch
                {
                    Leaf<T> { Value: var v } => algebra.Leaf(v),
                    Node<T> { Left: var l, Right: var r } =>
                           algebra.Node(l.Cata(algebra), r.Cata(algebra)),

                };
```

Run This: .Net Fiddle

6.5 Catamorphisms with the Visitor Design pattern

We can also use our algebra on a tree with the visitor design pattern, If we define an `accept` method on the tree structure. The visitor offers one more degree of freedom because we can use different kinds of visitors.

```
public class AlgebraSumVisitor : Visitor<int, int>
    {
        public override int VisitLeaf(Leaf<int> leaf) => leaf.V;
        public override int VisitNode(Node<int> node) =>
            node.Left.Accept(this) + node.Value + node.Right.Accept(this);
    }
```

And we implement the standard accept method in each component

```
public abstract class Tree<T>
{
        public abstract T1 Accept<T1>(Visitor<T, T1> visitor);
}

public class Node<T> : Tree<T>
{
        public Tree<T> Left { get; set; }
        public T Value { get; set; }
        public Tree<T> Right { get; set; }

        public Node(Tree<T> left, T value, Tree<T> right)
        {
            this.Left = left;
            this.Value = value;
            this.Right = right;
        }

        public override T1 Accept<T1>(Visitor<T, T1> visitor) =>
                                        visitor.VisitNode(this);
}

public class Leaf<T> : Tree<T>
{
        public T V { get; }
        public Leaf(T v) => V = v;
        public override T1 Accept<T1>(Visitor<T, T1> visitor) =>
                                        visitor.VisitLeaf(this);
}
```

Run This: .Net Fiddle

As you can see the Visitor is pretty much isomorphic to the catamorphism implementation. Visitors use the **double dispatch** to dispatches a function call to different concrete functions

depending on the runtime types of two objects involved in the call. Double Dispatch is a Kind of discrimination between the Members of the Union (aka Inheritance)

The visitor methods are almost identical with the algebra of `MatchWith` mechanism

```
public abstract class Visitor<T, T1>
{
        public abstract T1 VisitNode(Node<T> node);
        public abstract T1 VisitLeaf(Leaf<T> leaf);
}

MatchWith(algebra: (
                  Leaf: v =>  {}
                  Node: (l, v, r) => {}
                 ))
```

6.5.1 The Base Functor of the List<T>

optional

I was not sure if I wanted to include this section because it only adds unnecessary complexity and probably will leave you more confused because of the C# Types system. This section is partially about Fix points and Base Functors.

Let me introduce you to another Form of the List. the algebraic definition of a list is this

```
List (a)= [] + a * List(a)
```

Since we already have the `Maybe` as a Coproduct types that has a subtype `None` (that can represent the concept of Nothing or Empty or a Terminal object) and a subtype `Some` that can have a Value we might use this to represent this above definition.

```
List (a)= None() + Some(a * List(a))
```

This definition is isomorphic to the definition of a list. Let's see another way to prove this. Let's say that we have a class for product

```
public class Product<T,E>
{
      public T Value { get; set; }
      public E Rest { get; set; }
}
```

If I take the E parameter to be a `Maybe<Product<T>>` this becomes

```
public class Product<T>
{
      public T Value { get; set; }
      public Maybe<Product<T>> Rest { get; set; }
}
```

Let's say that we try to write what a `Maybe<Product<T>>` looks like (in a kind of symbolic representation)

```
Maybe<Product<T>> = None<Product<T>> + Some<Product<T>>({ Value=a:T, Rest:
Maybe<Product<T>>})
```

And if we replace `Maybe<Product<T>>` with the name List<T> then we get:

```
List<T> = None<Product<T>> + Some<Product<T>>({ Value=a:T, Rest: List<T>}).
```

Which is has the correct meaning of a List.

Let me try to explain the same from another perspective. Lets now start from this class

```
public class SomeF<T, E>
{
    private readonly Product<T, E> value;
    public SomeF(Product<T, E> value) => this.value = value;
}
```

Which looks like the `SomeF<T, E>` where the value is of type `Product<T, E>`

Let's write some instances of this class

In the first one `T` is `int` and `E` is `int`

```
SomeF<int, int> s = new SomeF<int, int>(new Product<int, int>() { Value = 1, Rest
= 1 });
```

Now let's take `E` as `SomeF<int, int>` This is valid:

```
SomeF<int, SomeF<int, int>> s1 = new SomeF<int, SomeF<int, int>>(new Product<int,
SomeF<int, int>>() { Value = 1, Rest = s });
```

Now if we take `E` as `SomeF<int, SomeF<int, int>>` we get

```
SomeF<int, SomeF<int, SomeF<int, int>>> s2 = new SomeF<int, SomeF<int, SomeF<int,
int>>>(
new Product<int, SomeF<int, SomeF<int, int>>>() { Value = 1, Rest = s1 });
```

the thing is we suspect that s1 should be a Subtype of s2, s1<: s2 which is in the Behavioral sense if for example we were writing in C# it would be safe to replace s1 with s2.

We suspect that if we were to repeat this replacement of `E` with

`SomeF<int, SomeF<int, SomeF<int, …>>>` this should converge somewhere. Let's call this infinite expression as X then it would not make any difference if we iterate one more time (like adding +1 at infinite is still infinite .This is a usual tactic when reasoning with infinite structures) so `SomeF<int, X> == X` we say that this X type would be a Fix Type for this Type `SomeF<int, X>` the only thing we can do in C# is write this equivalence this way

```
public class SomeFixed<T,E>  where E : SomeFixed<T, E>
```

```
{
        private readonly Product<T,E> value;
        public SomeFixed(Product<T, E> value) => this.value = value;
}
```

Nd finally completely replace E like this:

```
public class SomeFixed <T>
{
        private readonly Product<T, SomeFixed <T>> value;
        public Some(Product<T, SomeFixed <T>> value) => this.value = value;
}
```

Run This: .Net Fiddle

The point of all this is that `Maybe<Product<T>>` is equivalent to a List<T>

Type Theory

Base Functor and Fix points

In the list definition

List (a)= [] + a * List(a)

We can get rid of recursion by replacing the `List(a)` in the right side with a new type E

ListF (a, E) = [] + a * E

That's the base functor. *The functor that we get when we replace recursive occurrences with a new type*. Now the map function with respect to **E** for this new Base Functor are the ones that we create previously.
Imagine this as a type F(_) where F(_)=ListF (a, _)=[] + a * _

If we replace E with any Type we get a ListF (a, E) Type for example we can get E as int then we get the Type ListF(a,int)=[] + a * b where b is int .We can take as E whatever we want but for this example there is one Type X that if E is of that type then something special happens :

ListF (a, X) = X **(!)**

We want to find this specific X because if we replace it in the definition, we get:

ListF (a, X) = [] + a * X = [] + a * ListF (a, X)

Replace from **(!)**

And now if we replace ListF (a, X) with a name like list(a) for example we get

list(a) = [] + a * list(a)

well we have the list definition. We started from a non-recursive definition
ListF (a, E) = [] + a * E and we found an E that gives us the recursive type that we wanted.

This X is called A Fix point of F(_). A fix point must satisfy an equation like this:

F(X) = X

Fix points arise everywhere and are a thing of beauty. We are not going to get in any more detail here. But in the **C# type systems the Recursive Types are Fix points of other Higher order types that we never see**. *The ability of a language to have Recursive types hides all the complexity we just discussed*. We use Fix points all the time that's why we cannot even see them.

Finally, **the `List<T>` is a Fix point of the Functor `Maybe<Product<T,E>>`**

Lets now use our Base list definition as `Maybe<Product<T>>` where

```
public class Product<T>
{
      public T Value { get; set; }
      public Maybe<Product<T>> Rest { get; set; }
}
```

We can define our usual List Methods on this structure

```
public static class FunctionalExtensions
 {
     public static int Multiply(this Maybe<Product<int>> @this) =>
         @this.MatchWith(pattern: (
                 None: () => 1,
                 Some: (product) => product.Value * product.Rest.Multiply()
      ));
 }
```

```
var r = new Some<Product<int>>(
                new Product<int>
                {
                    Value = 5,
                    Rest = new Some<Product<int>>(
                        new Product<int>
                        {
                            Value = 3,
                            Rest = new None<Product<int>>()
                        })
                }) .Multiply();
```

Run This: .Net Fiddle

6.5.2 A brief mentioning of Anamorhisms optional

It's easy to use recursion in order to construct instances of this type `new Some({ value: 2, rest: new Some({ value: 1, rest: new None() }) })`

```
public static Maybe<Product<int>> AnaToListBase(this int @n) =>
@n == 0 ? (Maybe<Product<int>>)new None<Product<int>>() :
            new Some<Product<int>>(new Product<int>()
             {
                 Value = @n,
                  Rest = (@n - 1).AnaToListBase()
              });
```

```
(2).AnaToListBase()//{"v":{"value":2,"rest":{"v":{"value":1,"rest":{}}}}}
```

Run This: .Net Fiddle

This is an example of an unfold or anamorhisms. Starting from a single seed integer value we gradually build a Maybe functor instance.

The above definition has two entangled concepts that can be separated. We can isolate the recursive part. Writing something like this

```
 var coAlgListBase = n => n <=0 ? new None() : new Some({
       head: n - 1,
       tail: n - 1
   });

   var ana = (colAg) => oldState => colAg(oldState).cata({
       some: (v) => new Some({value:v.head, rest:(ana(colAg)(v.tail))}),
       none: (_) => new None()
   })

   var nat = ana(coAlgListBase)(2);//{"v":{"value":1,"rest":{"v":{"value":0,"rest
":{}}}}}
```

Run This: .Net Fiddle

I know that this way more complex and the only thing that we here is to be able to use different `colAg` for example using the following we get only the even numbers:

```
var coAlgListBaseEvens = n => n <=0 ? new None() : new Some({
     head: n - 2,
     tail: n - 2
});
```

We can also change the `ana` function and generate lists by using instructions made of `Maybe's`

```
var ana = (colAg) => oldState => colAg(oldState).cata({
        some: (v) => [v.head].concat(ana(colAg)(v.tail)),
        none: (_) => []
})

var nat = ana(coAlgListBaseEvens)(10);//[8,6,4,2,0]
```

Run This: .Net Fiddle

this can be further simplified using the definition of anamorhisms from the perspective of category theory, but we are not going into this here. We indirectly introduced the definition of F-Coalgebras.

6.5.3 F-Coalgebra

A F-coalgebra is the dual structure of a F-algebra. If an algebra is something like **F(A) → A** where F is a Functor then by reversing the arrows (in order to get the dual) a coalgebra is something of this form **A → F(A).** We start from an object and we get a Functor of an object.

Look again at this coalgeabr `coAlg= n => n <=0 ? new None() : new Some({…})` from an int we get a `Maybe`.

6.5.4 Corecursion

We all know that this definition is recursive:

```
var ana = n => n == 0 ? new None() :new Some({ value: n, rest: ana(n - 1) });  (!)
```

the characteristics that make it recursive are three:

1. There is a base **terminal** condition `n == 0? new None()`.
2. There is an either/union/coproduct involved. In our case the conditional operator (`n => n == 0 ? _: _ `) plays this role
3. We call the function again **with a reduced** argument that ensures that we will eventually reach the base condition (1) and this process will finally stop. So, we **de-construct** the **initial** value step by step.

But mathematicians came up with the dual concept to recursion, which is called, co-recursion. Here instead of working top-to-bottom we build things bottom-up. A corecusrsive definition of definition **(!)** would be :

```
Func<int, Maybe<Product<int>>, Maybe<Product<int>>> Coana = null;

Coana = (n, r) => Coana(n + 1, new Some<Product<int>>(new Product<int>() { Value =
 n, Rest = r }));

var stream = Coana(0, new None<Product<int>>());
```

well enjoy your StackOverflowException

the resulting structure `stream` **is infinite** and gives us **all** the natural numbers! Corecursion often gives us infinite data structures we call co-data.

The thing is that we cannot call `Coana` because C# will try to evaluate everything at once. But we can use a Func<> ()=> to make it lazy right ? well please try it for yourself forking the .net Fiddle.

I added a gap to just witness the co-iteration the first repetitions give us this

```
//(Value: 0, Rest:{ })
//(Value: 1, Rest: (Value: 0, Rest:{ }))
//(Value: 2, Rest: (Value: 1, Rest: (Value: 0, Rest:{ })))
//(Value: 3, Rest: (Value: 2, Rest: (Value: 1, Rest: (Value: 0, Rest:{ }))))
```

Run This: .Net Fiddle

the characteristics that make it co-recursive are dual to those of recursion as noted above:

1. There is a base **initial** condition `(0, new None())`.
2. There is a **product** involved for value accumulation. In our case the tuple `(n, rest)` plays this role
3. We call the function again **with an increased** argument. There is no assurance that this process is going to stop eventually. We **construct** the **final** value step by step.

[Some advanced but manageable papers on conduction can be found on the page of Jan Rutten here where the popular "An introduction to (co)algebra and (co)induction" resides[2]]

[2] https://homepages.cwi.nl/~janr/papers/files-of-papers/2011_Jacobs_Rutten_new.pdf

6.5.5 A brief mentioning of Hylomorphisms optional

"**Hylomorphism** (or **hylemorphism**) is a philosophical theory developed by Aristotle, which conceives being (ousia) as a compound of matter and form"

-Wikipedia

We can now combine `ana` and `cata`, by first using an anamorhisms `ana` to generate an expression like

```
new Some({ value: 2, rest: new Some({ value: 1,rest: new None() }) })
```

and then use a catamorphism `cata` it into a single value. This process is called Hylomorhism. In our case we can get the factorial:

```
public static int Multiply(this Maybe<Product<int>> @this) =>
@this.MatchWith(pattern: (
        None: () => 1,
        Some: (product) => product.Value * product.Rest.Multiply()
));

public static Maybe<Product<int>> AnaToListBase(this int @n) =>
@n == 0 ? (Maybe<Product<int>>)new None<Product<int>>() :
            new Some<Product<int>>(new Product<int>()
            {
                Value = @n,
                Rest = (@n - 1).AnaToListBase()
            });

Func<int, int> factorial = n => n.AnaToListBase().Multiply();
```

Run This: .Net Fiddle

the mathematical formulation behind hylomorphism is a recursive function

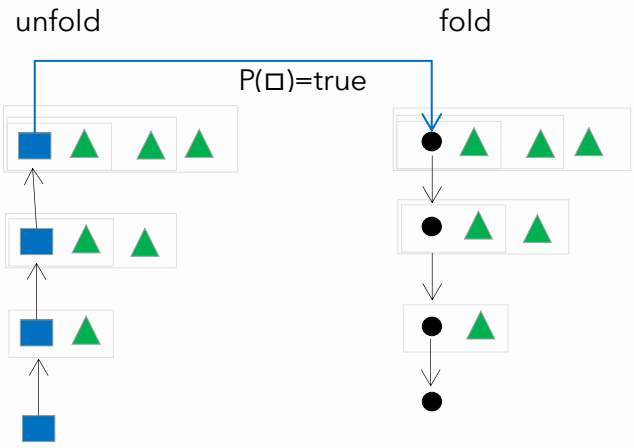
$$h(a) = \begin{cases} c & if\ p(a) == true \\ b \otimes h(a') & where\ g(a) = (a', b) \end{cases}$$

where h is a function h: A ⟶ C . This can be decomposed in an anamorphic part g: A ⟶ A × B that generates a pair of (a: A , b:B) and a catamorphic part represented by the operator

⊗ : C × B ⟶ C that folds a pair (c: C , b:B) into an a: A and also a conditional P(a) that determines if the anamorhisms has to stop and start accumulating the generated items on a base **c** element of type C . In order to fully grasp the idea, you can look at the following graphic representation and then revisit the mathematical notation.

suppose that we have a co-algebra g: □ → ⟦□ △ ⟧ and an algebra h: ⟦● △ ⟧ → ● the process starts with the g replacing squares □ with pairs of ⟦□ △ ⟧

□ → ⟦□ △ ⟧ → ⟦⟦□ △ ⟧ △ ⟧ →… at some point when a condition is reached, we replace the most inner square □ with a ● this starts the collapsing(catamorphism)



6.5.6 Hylomorphism example: Mergesort

"In any change, there must be three things:
(1) something which underlies and persists through the change;
(2) a "lack", which is one of a pair of opposites, the other of which is
(3) a form acquired during the course of the change "
Aristoteles -*Physics* »

In this section we will see a functional approach on mergesorting an array using hylomorphism (you can see in the Wikipedia page how ugly the imperative code can become when we try to solve problems that are use divide and conquer in which functional style excels). First, we will transform an array to a binary tree using an anamorhisms, then we will gradually fold the leaves of the tree while preserving a property of ordering using a catamorphism.

1) Unfolding to a tree

We recursively break the array in half until we reach single elements that we are going to use a leave to wrap them and Nodes to compose bottom up.

```
public static Tree<T> ToTree<T>(this List<T> @this)
        {
            var lenght = @this.Length();
            return lenght == 1 ? (Tree<T>)
```

```
                new Leaf<T>(@this.FirstOrDefault()) :
                new Node<T>(@this.GetRange(0, lenght / 2).ToTree(),
                @this.GetRange(lenght / 2, (int)Math.Ceiling((decimal)lenght / 2))
.ToTree()
                );
        }

new List<int> { 1, 2, 10, 6, 20, 11, 2, 3, 4 }.ToTree() ⟶
(((1,2),(10,6)),((20,11),(2,(3,4))))
```

Run This: .Net Fiddle

2) Folding the tree while preserving the order

Here there are two steps. Let's say that we have as an inductive hypothesis a function `zipOrdered` that when given two ordered lists this gives us the zipped list preserving the ordering. We will define `zipOrdered` later. For now, we can assume that we have already defined it. Now we can define a recursive sort function on the Tree using pattern matching with cata:

```
public static List<T> Sort<T>(this Tree<T> @this) where T : IComparable =>
        @this.MatchWith(algebra: (
                Leaf: v => v.ToList(),
                Node: (l, r) => l.Sort().ZipOrdered(r.Sort())
));
```

This says if we are in `Leaf` just return the value `v` which is ordered because its single. Now for the `Node` we first recursively use sort to get the sorted lists `(l.sort(), r.sort())` and then we zip them using `zipOrdered`.

Now we should define `zipOrdered` which also has a recursive definition:

```
public static List<T> ZipOrdered<T>(this List<T> @this, List<T> @a2)
where T : IComparable
  =>
        @this.MatchWith(algebra: (
            Empty: () => @a2,
            Cons: (x, xs) =>
                @a2.MatchWith(algebra: (
                    Empty: () => xs,
                    Cons: (y, ys) => (x.CompareTo(y) > 0) ?
                    x.ToList().Concat(xs.ZipOrdered((y, ys).ToList()))
                        : y.ToList().Concat(ys.ZipOrdered((x, xs).ToList()))
                        )
                )));
```

Run This: .Net Fiddle

This definition is the same as the zipp definition but also uses **a comparison at the node level in order to decide which array has the next element in order.**

```
var initial = new List<int> { 1, 2, 10, 6, 20, 11, 2, 3, 4 }.ToTree();
Console.WriteLine(initial.Show());          ////((1,2),(10,6)),((20,11),(2,(3,4))))
var sorted = initial.Sort();
Console.WriteLine(sorted.ToTree().Show()); ////((20,11),(10,6)),((4,3),(2,(2,1))))
```

Run This: .Net Fiddle

6.6 Fold's relation to Cata

Folds are also ways to collapse a structure. Catamorphisms are more general in the sense that we have more freedom on how we access the values. We will see Folds in more detail in a letter chapter. The point of this section is to highlight the similarity between cata and fold and highlight the *fact that some libraries provide fold functions that we can use in order to extract values out of certain data structures*.

we can create a fold for example for our Id functor (which is useless but its only for display purposes and to prove that array.reduce is just a subcase) In the same way that Enumerable. `Aggregate` is defined

```
new List<int>() { 2, 3, 4 }.Aggregate(0, (accumulate, element) => accumulate + ele
ment);
```

the signature of the extension method is pretty standard as you can see from the source code of the `Enumerable` :

```
static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> sourc
e,
 TAccumulate seed,
 Func<TAccumulate, TSource, TAccumulate> func)
```

The `TAccumulate seed` and the `Func<TAccumulate, TSource, TAccumulate> func` are enough to Define a Monoid the Empty is the `seed` and the Concat is the `Func<…>`

By analogy for the `Id` Functor we can define a Fold with the same signature

```
    public class Id<T>
    {
        public T Value { get; set; }
        public Id(T value) => Value = value;
        public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
        public TM Fold<TM>(TM state, Func<T, TM, TM> reducer) => reducer(Value, st
ate);
    }
```

And use it like this:

```
var toArray = new Id<int>(4).Fold<List<int>>(new List<int>(), (e, a) => a.Concat(e
.ToList()));

var multiply = new Id<int>(4).Fold<int>(1, (e, a) => a * e);
```

Run This: .Net Fiddle

Alternatively, if we had a simple Product structure, we can provide a fold like this:

```
var Pair = (a, b) => ({
    map: f => pair(f(a), f(b)),
    fold: (accumulator, reducer) => reducer(reducer(accumulator, a), b)
});

var r= Pair(2,3).fold(0, (accumulator, currentValue) => accumulator + currentValue
) ;
```

Run This: .Net Fiddle

Notice that this is the same as the monoidal folding idea in the monoids section.

Also you can see that if we add more members to a product structure (a,b,c,d,…) then the fold would be something like that :

```
fold: (accumulator, reducer) =>reducer(reducer(reducer(reducer(accumulator, a), b)
,c,d,...)
```

which approximates the list definition of the fold, because in essence, an arbitrary product (a,b,c,d,…) is a list . We can also define folds for Either, IO and the Reader functors (we will skip this part for now)

7 Traversable

I expect that you will find the Following section to be a bit difficult. I know this because it took me a while to derive the specific implementations from theory, since there are no similar examples in C#. Non the less in other functional languages (like Haskell and Scala) the concept of Traversables is considered pretty standard. Initially we will try to derive some specialized cases of the `List` being traversed by an `Id` and then generalize over this. After that we are going to reach Traversables from another path: FoldMap. Hopefully by looking the same thing from two different perspectives will allow you to recognize the pattern.

Let's say that we have a monoid for example the Array ([], concat) and the Identity functor as an applicative. We can use the applicative chaining to merge functors that contain arrays:

```
new Id<Func<List<int>, Func<List<int>, List<int>>>>(x => y => x.Concat(y))
            .Ap(new Id<List<int>>((10, 5).ToList()))
```

```
                .Ap(new Id<List<int>>((3, 6).ToList()));
// Id([10,5,3,6])
```

we can do that repeatedly by replacing the last Ap() with the whole structure since it is of the same type:

```
new Id<Func<List<int>, Func<List<int>, List<int>>>>(x => y => x.Concat(y))
      .Ap(new Id<List<int>>((10, 5).ToList()))
         .Ap(new Id<Func<List<int>,Func<List<int>,List<int>>>>(x =>y=>x.Concat(y))
                .Ap(new Id<List<int>>((10, 5).ToList()))
                .Ap(new Id<List<int>>((3, 6).ToList()))
          );
```

we can use recursion to use this in any list:

```
public static Id<List<T>> Distribute<T>(this List<Id<T>> @this,
           Func<Id<List<T>>> TypeRep) =>
             @this.MatchWith(algebra: (
                 Empty: ()     => TypeRep(),
                 Cons : (v, r) =>
                  new Id<Func<List<T>, Func<List<T>, List<T>>>>(x =>y=>x.Concat(y))
                                          .Ap(v.Map(x => x.ToList()))
                                          .Ap(r.Distribute(TypeRep))
           ));

new List<Id<int>> { new Id<int>(2), new Id<int>(3) }
       .Distribute<int>(() => new Id<List<int>>(new List<int>())); //Id([2, 3])
```

Run This: .Net Fiddle

Inside the paper that introduced the concept of applicatives "Applicative Programming with Effects" Conor McBride and Ross Paterson called this **applicative distributor** for lists:

```
distribute:: Applicative f : [f a ] ⟶ f [ a ]      // the f from inside the list goes to the outside
distribute ([]) = ⟨ [] ⟩
distribute ([value: rest]) = ⟨ (:) value (distribute (rest)) ⟩
```

compare this to the map

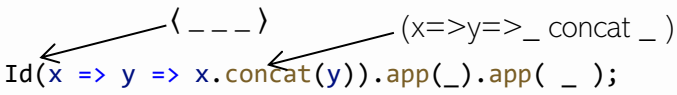
```
map f ([ ]) = [  ]
map f ([value:  rest]) = [ f(value):  map f (rest)]
```

this is a loose representation where the ⟨…⟩ represents the applicative operation.

the ⟨ (concat) _ _ ⟩ represents the following expression when our **f** is the Id applicative functor :

```
Id(x => y => x.concat(y)).app(_).app( _ );
```

It is passing a lambda inside the constructor of our data structure



Here we have placed the curried concat function (_=>_=>_.concat_) Inside an applicative in order to be able to chain the applicative calls. In Haskell, for example, this is called sequenceA :: Applicative f => t (f a) → f (t a), and what does is to take a **traversable t that contains f(a) items and evaluates them and collect the results.**

Why get into all this trouble just to get an Id ([1, 2, 3]) ? Well, we can abstract the Id and have a function a → Id(b) that will do the generation of the Id ([_]). We get this implementation:

```
public static Id<List<T1>> Traverse<T,T1>(this List<T> @this,Func<T, Id<T1>> f)
  =>
    @this.MatchWith(pattern: (
      Empty: ()     => new Id<List<T1>>(new List<T1>()),
      Cons : (v, r) =>
    new Id<Func<T1, Func<List<T1>, List<T1>>>>(x =>y =>x.AsList().Concat(y))
    .Ap(f(v))
    .Ap(r.Traverse(f))
));
var result = [2, 3, 5]. Traverse( Idf, v => Idf(v + 1));
```

Run This: .Net Fiddle

Here the f is the applicative functor constructor, aka pure (_) in the fiddle is written as TyperRep for Type Representable, (**TypeRep is the constructor of the Functor)**. In a symbolic notation, it could be described as follows:

traverse: Applicative f : (a → f(b)) → [a] → f [b]

traverse (f , []) = ⟨ [] ⟩

traverse (f ,[value, rest]) = ⟨ (:) f(value) (traverse(f, rest)) ⟩

now if you compare the signatures of the distribute and traverse you can see that we can create traverse from distribute by first mapping the (a → f(b)) on the array elements so we get:

[a]. map(a → f(b)) → [f(b)] and then apply the distribute to get the final f [b] So :

list .traverse (g) = list. map(g). distribute ( ) where g : a → f(b) :

```
var TraverseEquivalent= (list,g)=>list.map(g).distribute();
```

Run This: .Net Fiddle

7.1 Traversable Algebraic data structures

Let us now define traverse for Algebraic data structures, taking as an example again the simple Tree.

tree (a)= leaf(a) + node(tree(a) , a , tree (a))

This Is again a inductive definition. The symbolic definition would be

traverse (f , leaf(a)) = leaf (f(a))

traverse (f , node(l, x, r)) = ⟨ (node) traverse (f , r) f(x) traverse (f, r) ⟩

this translates to the following implementation.

```
public static Id<Tree<int>> Distribute(this Tree<Id<int>> @this) =>
@this.MatchWith(algebra: (
    Leaf: v => new Id<Func<int, Tree<int>>>(x => new Leaf<int>(x)).Ap(v),
    Node: (l, r) => new Id<Func<Tree<int>, Func<Tree<int>, Tree<int>>>>
                      (x => y => new Node<int>(x, y))
                    .Ap(l.Distribute())
                    .Ap(r.Distribute())
 ));
```

Run This: .Net Fiddle

Unfortunately, the need to write the Types explicitly makes C# code unreadable pretty fast.

If we strip down the types, we get this :

```
public static Id<Tree> Distribute(this Tree<Id> @this) =>
       @this.MatchWith(algebra: (
           Leaf: v => new Id(x => new Leaf<int>(x)).Ap(v),
           Node: (l, r) => new Id(x => y => new Node<int>(x, y))
                           .Ap(l.Distribute())
                           .Ap(r.Distribute())
        ));
```

⟨ (leaf) _ ⟩

⟨ (node) _ _ ⟩

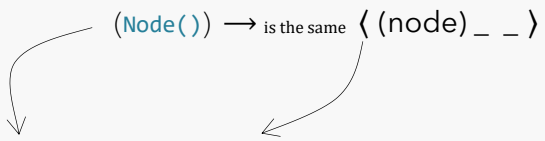
This is a bit tricky if you try to implement it for the first time. Here again in the node we have the applicative operation ⟨ (node) _ _ _ ⟩ [Id(x => y => new Node<int>(x, y))] where the node constructor new Node<int>(x, y) was curried, in order to use applicative chaining. This is a repetitive theme. Take a moment to compare this with the Array equivalent

⟨ (Concat) _ _ ⟩ and try to make sense of the mechanics involved in this concept by yourself.

Run This: .Net Fiddle

I want to make sure that you will see the monoid homomorphism that is at play here. **We use the applicative as a monoid to fold the List.**

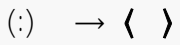
In our example above the equivalence is that the multiplication $\otimes_{node}$: `Node()` represented by the node constructor becomes equivalent with the applicative `Ap` operation of the `Id`:




```
Id(x => y => new Node<int>(x, y)).Ap(l.Distribute());
```

Two completely different domains but actually. ap is equivalent with + in a different level of abstraction. Cool right.

Also as we will see in the traversable section the Arrays "multiplication" $\otimes_{Array}$ concat (:) will be lifted to the now the same as the $\otimes_{Reader}$, applicative operation ap ⟨ ⟩ of the reader




```
Reader(g=>x=>y=>[y].concat(x)).Ap(…).Ap(Reader(…)) ;
```

7.2 Traversing with The Task<T> aka Parallel

Now let's traverse some structures with a task. First we have to make Task applicative by adding this extension method, that allows us to use Tasks that contain functions :

```
public static Task<T1> Ap<T, T1>(this Task<Func<T, T1>> @this, Task<T> fa)
        => @this.ContinueWith(f => f.Result(fa.Result));
```

Now we can define a Distribute. This method takes a `List<Task<T>>` a list of tasks and after executing all of them collects the results in a list as a Task `Task<List<T>>` which is pretty amazing

```
   public static Task<List<T>> Distribute<T>(this List<Task<T>> @this)
            =>
             @this.MatchWith(pattern: (
              Empty: () => System.Threading.Tasks.Task.FromResult(new List<T>()),
             Cons: (v, r) =>
         System.Threading.Tasks.Task
     .FromResult<Func<T, Func<List<T>, List<T>>>>(x => y => x.AsList().Concat(y))
```

```
                .Ap(v)
                .Ap(r.Distribute())
 ));
```

```
Func<int, int, Task<int>> delay = (result, time) => Task<int>.Run(() =>
{
                Console.WriteLine($"{result} started ");
                Thread.Sleep(time);
                Console.WriteLine($"{result} finished {time}");
                return result;
});

var t = await new List<Task<int>> {delay(20, 2000),delay(10, 3000) }.Distribute();
```

Run This: .Net Fiddle

we can also Define a traverse that will start from a list apply some function that return a Task and then await for all and return the results:

```
static Task<List<T1>> Traverse<T, T1>(this List<T> @this,Func<T,Task<T1>> f)
      =>
        @this.MatchWith(pattern: (
        Empty: () => Task<List<T1>>.FromResult(new List<T1>()),
        Cons: (v, r) => System.Threading.Tasks.Task
   .FromResult<Func<T1, Func<List<T1>, List<T1>>>>(x => y => x.AsList().Concat(y))
             .Ap(f(v))
            .Ap(r.Traverse(f))
          ));
```

Now with this we can for example start with a list of urls that contain the Github api urls for https://api.github.com/users some user details

```
List<string> {
                "https://api.github.com/users/mojombo" ,
                "https://api.github.com/users/defunkt"
            }
```

Now if I have an async method that makes a Http call to get the details from the url

https://api.github.com/users/{userName }

```
public static async Task<object> GetUserDetails(string uri)
        {
     using (HttpClient httpClient = new HttpClient())
       {
       httpClient.DefaultRequestHeaders.Add("User-Agent", "Test");
       var response = await httpClient.GetAsync(uri, default(CancellationToken));
```

```
        return (await response.Content.ReadAsStringAsync());
        }
  }
```

I can get in parallel for multiple users the details by using Traverse

```
var userDetailList= await new List<string> {
    "https://api.github.com/users/mojombo" ,
    "https://api.github.com/users/defunkt"
}.Traverse(GetUserDetails);
```

Run This: .Net Fiddle

7.3 Applicative Reader Isomorphism with the Interpreter Design pattern

This is another isomorphism between classic design patterns of the functional and Object-oriented world that display the interconnectedness of the two paradigms beautifully.

Let us say we have a simple data structure that represents expressions:

```
Exp v = Val i + Add (Exp v) (Exp v) + Sub (Exp v) (Exp v)
```

We only have a Val expression that contains a number, an Add that can contain two expressions and a subtract Sub. This forms again a recursive algebraic data structure.

```
    public abstract class Expr<T> { }

    public class ValExpr<T> : Expr<T>
    {
        public ValExpr(T x) { X = x; }
        public T X { get; }
    }

    public class AddExpr<T> : Expr<T>
    {
        public AddExpr(Expr<T> x, Expr<T> y) { X = x; Y = y; }
```

```
        public Expr<T> X { get; }
        public Expr<T> Y { get; }
    }

    public class SubExpr<T> : Expr<T>
    {
        public SubExpr(Expr<T> x, Expr<T> y) { X = x; Y = y; }

        public Expr<T> X { get; }
        public Expr<T> Y { get; }
    }
```

We want to create a function eval that will evaluate an expression Exp formed by combinations of (Val, Add, Sub). The type of eval whould be in this case:

Eval : (u:Expr, env:Env)⟶ result:Int

Eval should take an expression u:Expr and a context (env) and return a result of type result:Int. In our example the context will be empty for simplicity, but we will not omit it in the code.

7.3.1 Standard Catamorphism implementation

```
public abstract class Expr<T>
{
        public abstract T1 MatchWith<T1>((Func<Expr<T>, Expr<T>, T1> add,
Func<Expr<T>, Expr<T>, T1> sub,
Func<T, T1> val) pattern);
    }
    public class ValExpr<T> : Expr<T>
    {
        public override T1 MatchWith<T1>((Func<Expr<T>, Expr<T>, T1> add,
 Func<Expr<T>, Expr<T>, T1> sub,
Func<T, T1> val) pattern) => pattern.val(X);
    }
    public class AddExpr<T> : Expr<T>
    {
        public override T1 MatchWith<T1>(
            (Func<Expr<T>, Expr<T>, T1> add,
            Func<Expr<T>, Expr<T>, T1> sub,
            Func<T, T1> val) pattern) => pattern.add(X, Y);
    }
    public class SubExpr<T> : Expr<T>
    {
        public override T1 MatchWith<T1>((Func<Expr<T>, Expr<T>, T1> add,
```

```
            Func<Expr<T>, Expr<T>, T1> sub,
            Func<T, T1> val) pattern)  => pattern.sub(X, Y);
    }



public static T1 Eval<T, T1>(this Expr<T> @this,
    (Func<T1, T1, T1> add, Func<T1, T1, T1> sub, Func<T, T1> val) algebra) =>
      @this.MatchWith(pattern: (
            add: (x, y) => algebra.add(x.Eval(algebra), y.Eval(algebra)),
            sub: (x, y) => algebra.sub(x.Eval(algebra), y.Eval(algebra)),
            val: v => algebra.val(v)
    ));


var expression = new SubExpr<int>(
          new AddExpr<int>(new ValExpr<int>(2), new ValExpr<int>(2)),
          new ValExpr<int>(2));

            var result = expression.Eval(algebra: (
                    add: (x, y) => x + y,
                    sub: (x, y) => x + y,
                    val: v => v)
                );
```

Run This: .Net Fiddle

C# 8.0 Pattern matching

By using the `switch` we get the following unobstructive `Cata` decleration

```
public static T1 Cata<T, T1>(this Expr<T> @this,
  (Func<T1, T1, T1> add, Func<T1, T1, T1> sub, Func<T, T1> val) algebra) =>
            @this switch
            {
                ValExpr<T> { X: var v } => algebra.val(v),
                AddExpr<T> { X: var x, Y: var y } =>
                                    algebra.add(x.Cata(algebra), y.Cata(algebra)),
                SubExpr<T> { X: var x, Y: var y } =>
                                    algebra.sub(x.Cata(algebra), y.Cata(algebra)),
            };
```

Run This: .Net Fiddle

7.3.2 Reader applicative implementation

The idea here is to create a reader Applicative which will traverse each expression, apply the context and accumulate the result. I rewrite here the applicative reader, just to look at it again.

```
public static Reader<Env, T1> App<Env, T, T1>
            (this Reader<Env, Func<T, T1>> @this, Reader<Env, T> fa)  =>
 new Reader<Env, T1>(env => { return @this.Map(f => fa.Map(f).Run(env)).Run(env);
});
```

```
var res = Reader(g=>x=>g.x2+x). ap(Reader(g => g.x1));

var r =res.run({x1:10,x2:20});//30
```

below is my implementation, try to play around with the fiddle

```
public static int Eval(this Expr<int> @this)
       => @this.MatchWith(pattern: (
           add: (x, y) =>

new Reader<object, Func<int, Func<int, int>>>(g => x1 => y1 => x1 + y1)
                       .App(new Reader<object, int>(g => x.Eval()))
                       .App(new Reader<object, int>(g => y.Eval()))
                       .Run(default),
           val: v => v));
```

```
  var expression = new AddExpr<int>(new AddExpr<int>(new ValExpr<int>(2), new V
alExpr<int>(2)), new ValExpr<int>(2));
            var result = expression.Eval();
```

Run This: .Net Fiddle

The algebra of eval using our applicative notation ⟨ ⟩ is

eval::(Expr u) ⟶ Env v ⟶ b

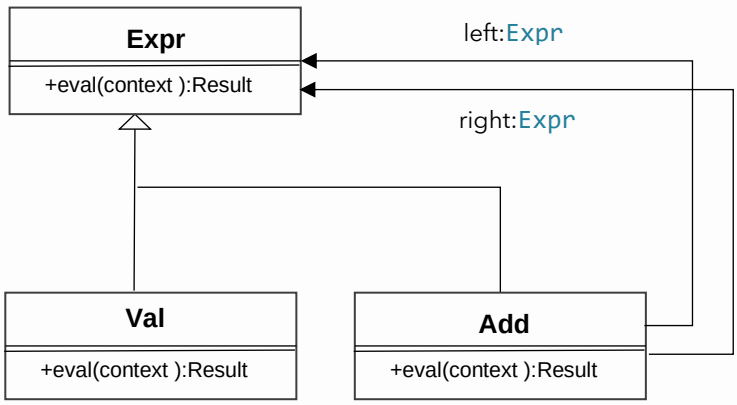
eval (Val(i), g) = ⟨ i ⟩
eval (Add(x, y),g) = ⟨ (+) eval (x, g) eval (y, g)⟩

7.3.3 Object Oriented Interpreter Pattern implementation

Take a moment to appreciate the use of the *reader applicative* to evaluate expressions. In the object-oriented world exists something isomorphic to the above implementation: the **Interpreter design pattern**. I am stating bellow an implementation based on ECS6 in order to emphasize the object-oriented characteristics:

```
    public class Context
    {
        public string Log { get; set; }
    }

public abstract class Expr
    {
        public abstract void Eval(Context context);
    }
    public class ValExpr : Expr
    {
        public ValExpr(int x) { X = x; }
        public int X { get; }

        public override void Eval(Context context)
        {
            context.Log += $"{X}";
        }

    }
    public class AddExpr : Expr
    {
        public AddExpr(Expr x, Expr y)
        {
            X = x;
            Y = y;
        }

        public Expr X { get; }
        public Expr Y { get; }

        public override void Eval(Context context)
        {
            context.Log += $"Adding(";
            X.Eval(context);
            context.Log += $",";
            Y.Eval(context);
            context.Log += $")";
        }
    }
```

Run This: .Net Fiddle



As you can see, those two implementations are pretty much equivalent. We can argue which one is better, but the object-oriented implementation looks easier. Easy, intuitive implementation is one of the advantages of object-oriented programming, even if the interpreter design pattern is not used often in development. On the opposite side, functional applicative implementation in my opinion looks prettier.

7.4 Explicitly Composing Traversables

Traversables are closed under composition. We can formulate a valid **traverse** operation for our two functor composition by just pushing the traverse operation to the inner functor and then traverse with this function the inner Functor. Unfortunately, there are no Higher Kind Types in C# so you have to rewrite the above implementation using the specific Traversables that you want to compose.

```
public static Id<List<Tree<T>>> TraverseT<T>(this List<Tree<T>> @this, Func<T, Id<
T>> f) => @this.Traverse(x => x.Traverse(f));

public static Id<Tree<List<T>>> TraverseT<T>(this Tree<List<T>> @this, Func<T, Id<
T>> f) => @this.Traverse(x => x.Traverse(f));
```

Run This: .Net Fiddle

The first can traverse this Tree of Lists<int>

```
var treeOfLists = new Node<List<int>>(new Leaf<List<int>>(new List<int>() { 1, 2 }
)
    , new Node<List<int>>(new Leaf<List<int>>(new List<int>() { 3, 4 }),
    new Leaf<List<int>>(new List<int>() { 5, 6 })));

var traversedTree = treeOfLists.TraverseT(x => new Id<int>(x + 1));
```

and the second can traverse this

```
var listOfTrees = new List<Tree<int>>() {
    new Node<int>(new Leaf<int>(2), new Node<int>(new Leaf<int>(2), new Leaf<int>(
4)))
    ,  new Node<int>(new Leaf<int>(2), new Leaf<int>(4)) };

var traversedList = listOfTrees.TraverseT(x => new Id<int>(x + 1));
```

You can try other variations like composing trees etc. by forking some of the previous Fiddles. I think that traversable is an underrated concept and there should be used more often in real situations.

7.5 Foldable

Within the same paper of "Applicative Programming with Effects" Conor McBride and Ross Paterson used **an applicative constant functor based on a monoid**, in order to use it with a traversable, to perform the accumulation. I was planning to present this way of connecting Traversables with Foldables here but because of C# typing I think it only would make things much more difficult to follow.

Instead, we will go straight to some implementations of Fold on lists and Trees .Let's start by defining Fold on a List (this is basically the Enumerable.Aggregate. Unfortunately, the .NET framework designers faced with a dilemma when naming the Aggregate function on the one hand they had to follow the whole LINQ idea and comply to the SQL naming, and on the other side was the default functional naming of Fold. They chose the SQL naming convention)

```
public static T Fold<T>(this List<T> @this, (Func<T> empty, Func<T, T, T> concat)
monoid)
            =>
            @this.MatchWith(algebra: (
                Empty: monoid.empty,
                Cons: (v, r) => monoid.concat(v, r.Fold(monoid))
            ));
```

Run This: .Net Fiddle

Here we have a List and a monoid `(Func<T> empty, Func<T, T, T> concat)` in its simple Tuple form and we end up with a single type T in the next section of FoldMap we will see the more generic case where the T does not have to be a monoid but we have to provide a function `Func<T, TM> f` that converts it into a monoid `TM`
The Type definition of Fold is this:

```
Fold: t(a) → m          // where t : traversable and m: monoid
```

Now let's define the same thing for a Tree:

```
public static T Fold<T>(this Tree<T> @this, (Func<T> empty, Func<T, T, T> concat)
monoid)
   =>
    @this.MatchWith(pattern: (
    Leaf: v => monoid.concat(monoid.empty(), v),
    Node: (l, r) => monoid.concat(l.Fold(monoid), r.Fold(monoid))
    ));


var tree = new Node<int>(new Leaf<int>(2), new Node<int>(new Leaf<int>(2), new Lea
f<int>(4)));
var r = tree.Fold(monoid: (() => 0, (x, y) => x + y));
```


alternatively we can use the Array as a monoid and thus convert a tree into a map by folding with the list monoid (: ,[])

```
var tolist = tree.Map(x => x.AsList())
.Fold(monoid: (() => new List<int>(), (x, y) => x.Concat(y)));
```

Run This: .Net Fiddle

but we had to map to a list first `Map(x => x.AsList())` in order to use this definition of Fold. We could define a new Method to do this Mapping together with the Fold. This is called traditionally FoldMap and is the topic of the next section.

7.5.1 FoldMap

Let's discuss foldMap a bit more. FoldMap takes a function that return a monoid (a→m) and apply it on a traversable in order to get a single value of type m. This is actually a variation of Fold.

Foldmap: (a→m) → t(a) → m where m is a monoid

So, we define FoldMap directly from fold like below

```
 public static TM FoldMap<T, TM>(this Tree<T> @this,
       (Func<TM> empty, Func<TM, TM, TM> concat) monoid, Func<T, TM> f)
         =>
     @this.MatchWith(pattern: (
           Leaf: v => monoid.concat(monoid.empty(), f(v)),
           Node: (l, r) => monoid.concat(l.FoldMap(monoid, f), r.FoldMap(monoid, f
))
         ));
```

Run This: .Net Fiddle

In some cases, we can just omit the demand for both

1. (Func<TM> empty, Func<TM, TM, TM> concat) monoid
2. Func<T, TM> f

But is not the case that we can always give the very convenient

```
(Func<TM> empty, Func<TM, T, TM> concat) reducer
```

That we find in the List implementation

To see why can you implement the following for a tree?

```
public static TM FoldMap<T, TM>(this Tree<T> @this,
        (Func<TM> empty, Func<TM, T, TM> concat) reducer)
         =>
         @this.MatchWith(pattern: (
         Leaf: v => monoid.concat(monoid.empty(), f(v)),
         Node: (l, r) => monoid.concat(l.FoldMap(monoid, f), ?)
        ));
```

There is no way to complete this gap

Let's move on. With the help of FoldMap we can define some operations more intuitively, for example for this conjunction monoid:

```
(empty: () => true, concat: (x, y) => x && y)
```

And ask questions about Booleans like

```
var allLargetThanThree = tree.FoldMap<int, bool>(
                (empty: () => true, concat: (x, y) => x && y), i => i > 3);
```

we can define an `All` That return true if all the members of a traversable satisfy a condition

```
public static bool All<T>(this Tree<T> @this, Func<T, bool> f)
    => @this.FoldMap((empty: () => true, concat: (x, y) => x && y), f);
```

Run This: .Net Fiddle

Another Implementation which is equivalent would be to define the Monoid we accumulate over as a different class

```
public interface IMonoidAcc<T>
{
     T Identity { get; set; }
    IMonoidAcc<T> Concat(IMonoidAcc<T> m);
}
```

And now have this equivalent formulation of FoldMap

```
public static IMonoidAcc<TM> FoldMap<T, TM>(this Tree<T> @this,
(Func<IMonoidAcc<TM>> empty, Func<T, IMonoidAcc<TM>> f) m)
    =>
        @this.MatchWith(pattern: (
            Leaf: v => m.f(v),
```

Directly chaining with the Concat that `IMonoidAcc` Exposes

```
            Node: (l, r) => l.FoldMap(m).Concat(r.FoldMap(m))
        ));

    }
```

Run This: .Net Fiddle

We just we provide an **f that return something that have a concat method** that we could use in order to accumulate over.

And use it like this with an Explicit Monoid

```
var all2 = tree.FoldMap(m: (() => new And(true), x => new And(x > 4)));

public class And : IMonoidAcc<bool>
{
    public And(bool value) => Identity = value;
    public bool Identity { get; set; }

public IMonoidAcc<bool> Concat(IMonoidAcc<bool> m) => new And(Identity && m.Identi
ty);
}
```

Run This: .Net Fiddle

7.5.2 Explicitly Composing Foldables

Obviously, we can write some explicit implementations of Fold compositions for the list and the tree.

```
public static T FoldT<T>(this List<List<T>> @this, (Func<T> empty, Func<T, T, T> c
oncat) monoid)            => @this.Select(x => x.Fold(monoid)).ToList().Fold(monoi
d);
```

Run This: .Net Fiddle

Or why not try lists of Trees:

```
public static T FoldT<T>(this List<Tree<T>> @this, (Func<T> empty, Func<T, T, T> c
oncat) monoid)             => @this.Select(x => x.Fold(monoid)).ToList().Fold(mono
id);
```

```
 var tree = new Node<List<int>>(new Leaf<List<int>>(new List<int>() { 1, 2 })
                , new Node<List<int>>(new Leaf<List<int>>(new List<int>() { 3, 4 }
),
                new Leaf<List<int>>(new List<int>() { 5, 6 })));

            var compositeFold = tree.FoldT(monoid: (() => 0, (x, y) => x + y));
```

Run This: .Net Fiddle

Or some Trees of Lists, or some Trees of Trees, all the folding compositions have the same signature `@this.Map(x => x.Fold(monoid)).Fold(monoid)` [we can write it without

Map also]You can try some and use other monoids to fold. For example, with the help of the *Array monoid we can flatten* everything into a List.

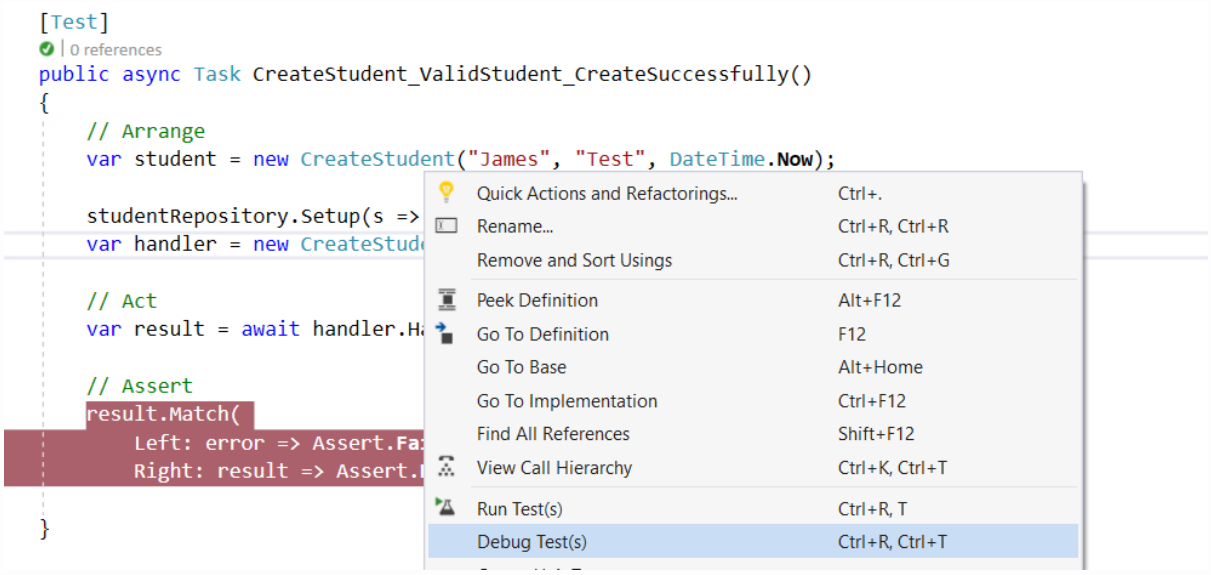
Clean Architecture in .NET

In this Section we are going to take a brief look at the possibility of **Functional Software Architecture** and how this can be integrated in the latest trends in **modern Object-oriented software architectures**. This is not a deep dive in Architectural Design but a discussion behind the architecture of the **sample Contoso** web Application in the **language-ext** GitHub project.

8 A Clean Functional Architecture Example

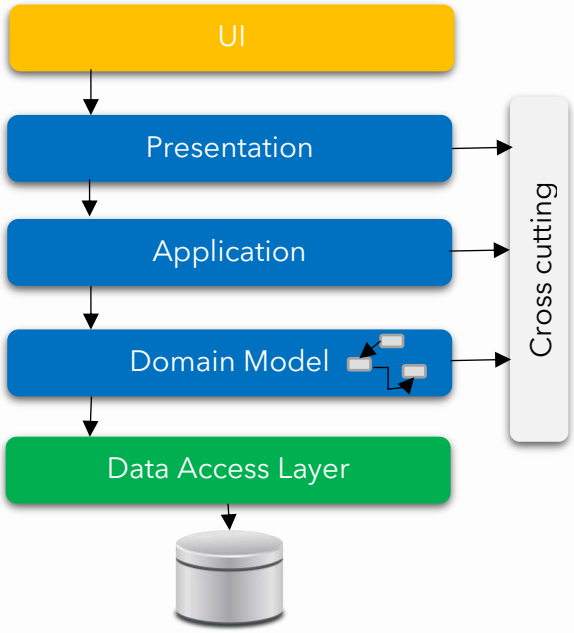
8.1 Download and Setup the Project

The project is ***headless*** and does not have any **Views** implemented. You can mock Http actions using postman or any other tool, or you might consider building your own front end UI. You can also Debug over the Tests accompanying the App as an Entry point.



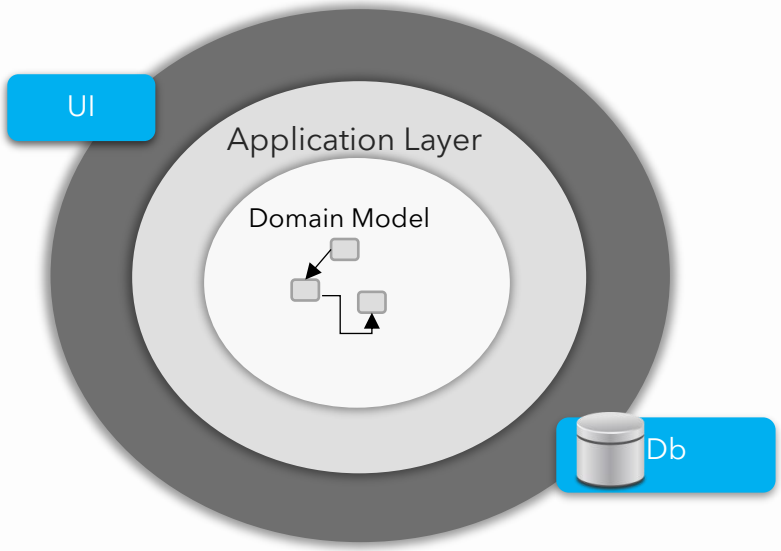
8.1.1 Clean Architecture with .NET core

The architecture of web applications depends on the scale and scope of the application. The Previous decade the dominant architecture for small and medium web applications was the **Layered Architecture.** A Layered Architecture, organizationed the project structure into four main categories: **presentation, application, domain, and infrastructure**.



Under the weight of ideas such as Domain -Driven Design from Eric Evans and Clean Code from "Uncle Bob" - Robert C. Martin the focus of the Architecture became the separation of concerns of the **Domain** from all the technicalities such as technologies, tools and implementation details.

The domain and its Use Cases now were placed in the **centre** of the architecture design with minimal dependencies. This led to a series of evolutionary transformations of the Layered structure (Hexagonal Architecture, Pipes and adapters , DDD etc) leading at to the concept of Clean architecture.



With Clean Architecture, the **Domain** and **Application** layers are at the centre of the design. This is known as the **Core** of the system.

The **Domain** layer contains enterprise logic and types and the **Application** layer contains business logic and types. The difference is that enterprise logic could be shared across many systems, whereas the business logic will typically only be used within this system.

Core should not be dependent on data access and other infrastructure concerns, so those dependencies are inverted. This is achieved by adding interfaces or abstractions within **Core** that are implemented by layers outside of **Core**. For example, if you wanted to implement the Repository pattern you would do so by adding an interface within **Core** and adding the implementation within **Infrastructure**.

All dependencies flow inwards, and **Core** has no dependency on any other layer. **Infrastructure** and **Presentation** depend on **Core**, but not on one another.

Resources:

https://github.com/maldworth/OnionWebApiStarterKit
https://github.com/josecuellar/Onion-DDD-Tooling-DotNET
Clean Architecture with ASP.NET Core 2.1 | Jason Taylor
You can also find Visual studio Templates to initialize Projects for example
https://github.com/jasontaylordev/CleanArchitecture

8.2 A Functional Applications Architecture

Where does the Functional Programming fit in the larger Architectural view? In the article form Scott Wlaschin "A primer on functional architecture" he recognizes three principles of functional programming that can be applied to the architectural level:

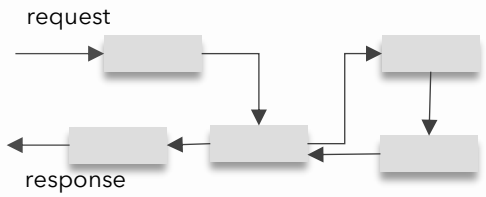
1. The first is that **functions** are standalone values. In a functional architecture, the basic unit is also a function, but a much larger business-oriented one that I like to call a **workflow**

2. Second, **composition** is the primary way to build systems. Two simple functions can be composed just by connecting the output of one to the input of another

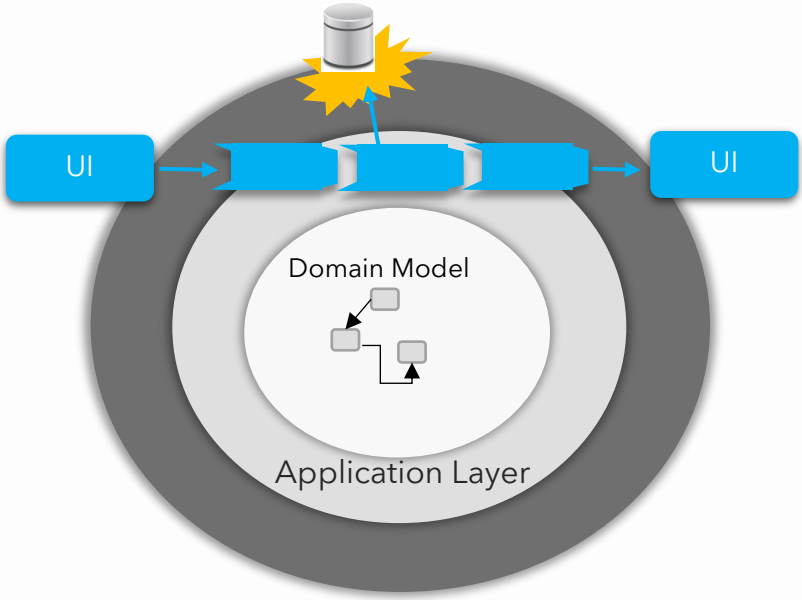
3. functional programmers try to use **pure functions** as much as possible

The general idea is that you write as much of your application as possible in an FP style chain computation using the Task, Option, Either, Validation Monads or their combinations and avoiding side effects and coupling. This would lead to the usual pipeline computation style:

Instead of the chaotic object oriented:



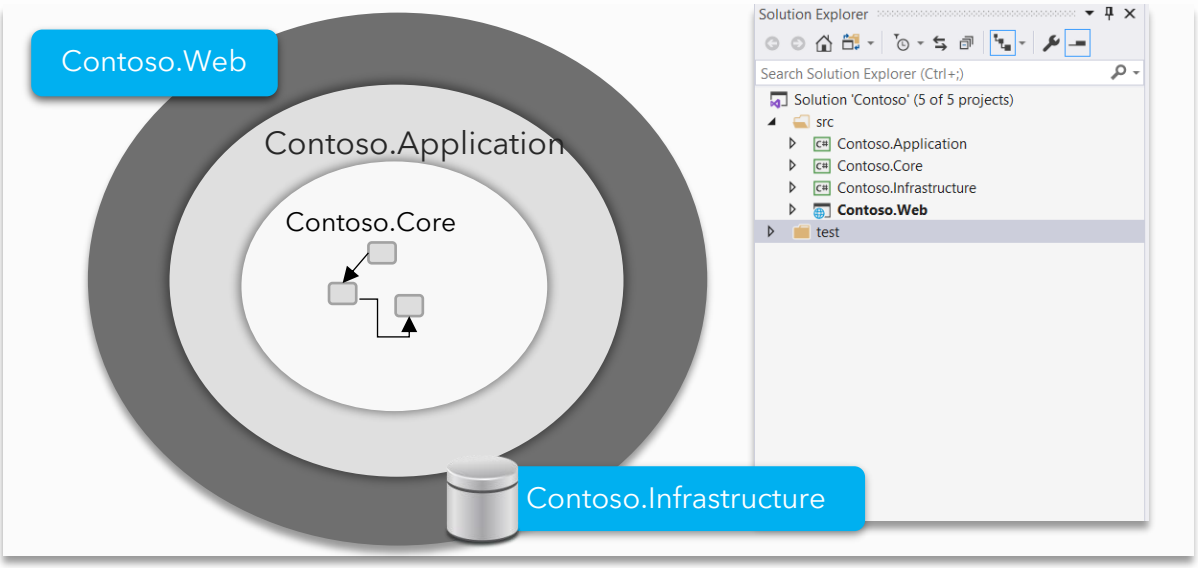
The idea for a functional architecture as laid out by Wlaschin is a pipeline that goes through the layers of the Architecture with minimal coupling and side effects.



8.3 The Contoso Clean Architecture with .NET core and language-ext

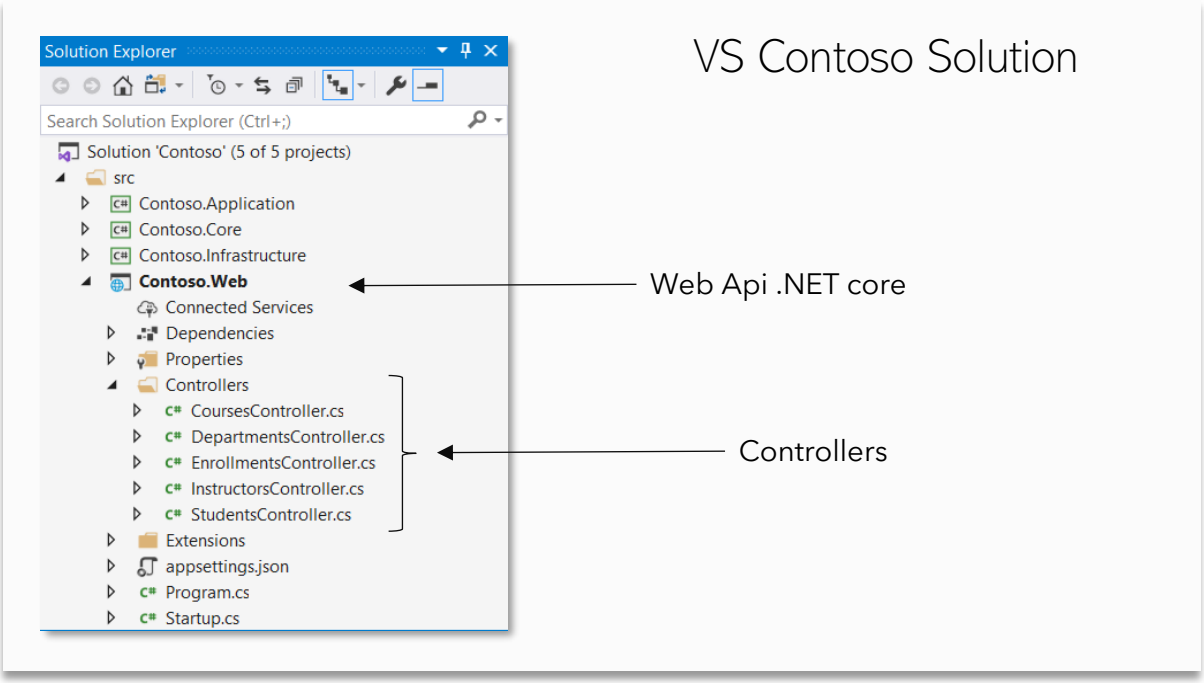
The Contoso application **approximates a Functional architecture Style embedded in a Clean architecture**.

The Contoso application is consists of four projects the three of them are class libraries targeting the .NET standard 2.0 framework (Contoso.Application, Contoso.Core, Contoso.Infrastructure) and the "UI" Project named Contoso.Web is a .NET 3.0 Web application project compatible with the rest of the projects



8.4 Web Api

This layer contains the code for the Web API endpoint logic including the Controllers. The API project for the solution will have a single responsibility, and that is only to handle the HTTP requests received by the web server and to return the HTTP responses with either success or failure.



We will handle exceptions and errors that have occurred in the Domain or Data projects to effectively communicate with the consumer of APIs. This communication will use HTTP response codes and any data to be returned located in the HTTP response body.

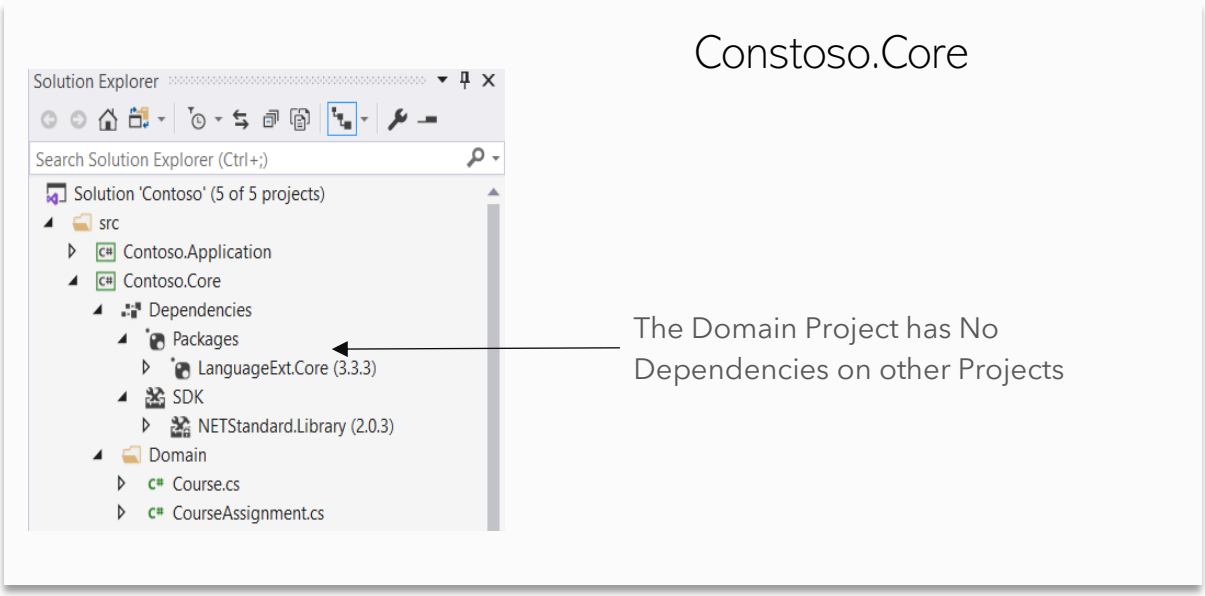
In ASP.NET Core Web API, routing is handled using Attribute Routing. If you need to learn more about Attribute Routing in ASP.NET Core go here. We are also using dependency injection to have the MediatR instance injected into the Controller.

```
[Produces("application/json")]
 [Route("api/[controller]")]
 [ApiController]
 public class CoursesController : ControllerBase
 {
     private readonly IMediator _mediator;
     public CoursesController(IMediator mediator) => _mediator = mediator;

     [HttpGet("{courseId}")]
     public Task<IActionResult> Get(int courseId) =>
         _mediator.Send(new GetCourseById(courseId)).ToActionResult();

     [HttpPost]
     public Task<IActionResult> Create(CreateCourse createCourse) =>
         _mediator.Send(createCourse).ToActionResult();

     [HttpPut]
     public Task<IActionResult> Update(UpdateCourse updateCourse) =>
         _mediator.Send(updateCourse).ToActionResult();

     [HttpDelete]
     public Task<IActionResult> Delete(DeleteCourse deleteCourse) =>
         _mediator.Send(deleteCourse).ToActionResult();
 }
```

Github: source

8.5 Domain Model

The Domain model here resides solely in the **Contoso.Core** project and has no dependencies on any other solution project or tool like Entity Framework for example.



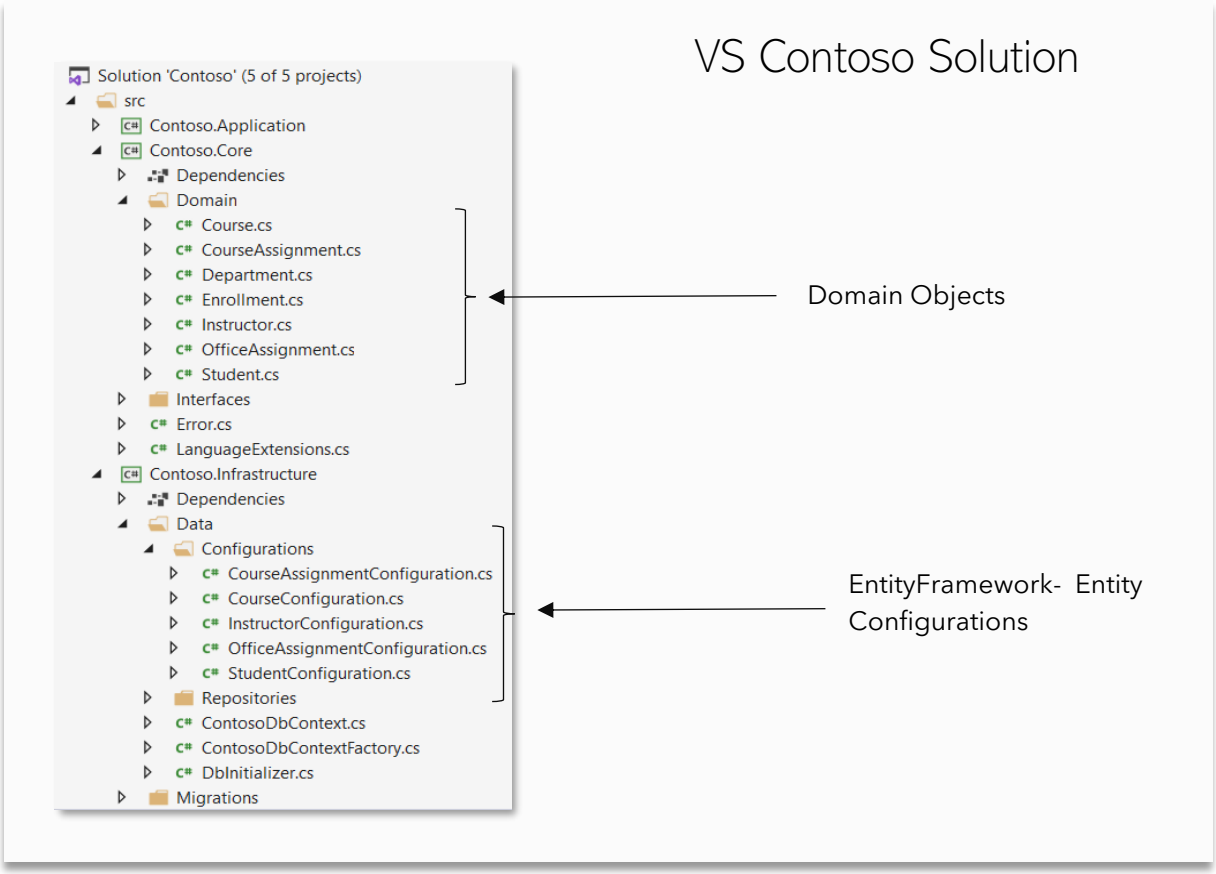
The domain model might seem trivial because in most implementations of the Clean architecture the domain objects are just POCO (Plain old CLR object) objects. For example:

```
public class Course
  {
      public int CourseId { get; set; }
      public string Title { get; set; }
      public int Credits { get; set; }
      public int? DepartmentId { get; set; }

      public Department Department { get; set; }
      public List<Enrollment> Enrollments { get; set; }
      public List<CourseAssignment> CourseAssignments { get; set; }
  }
```

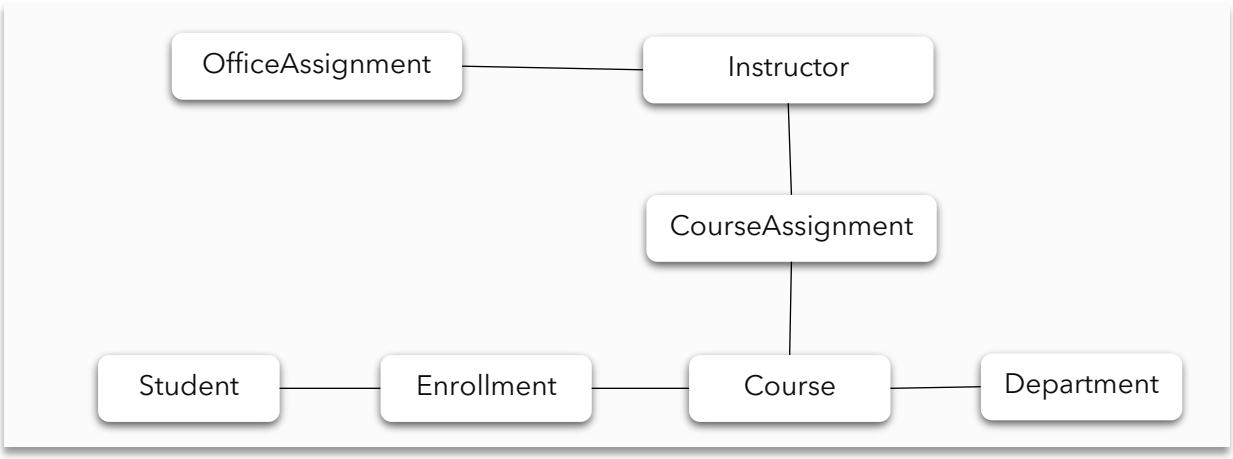
Github: source

This is often seen as an anti-pattern in the Object oriented world and especially in DDD world where it has a special name: anemic domain model. Others see this as a form of proper design practice.

In small scale applications this is simply fine. What we are going to focus here is not where we are going to place the business logic but the use of the **Functional Types in the Domain Model**



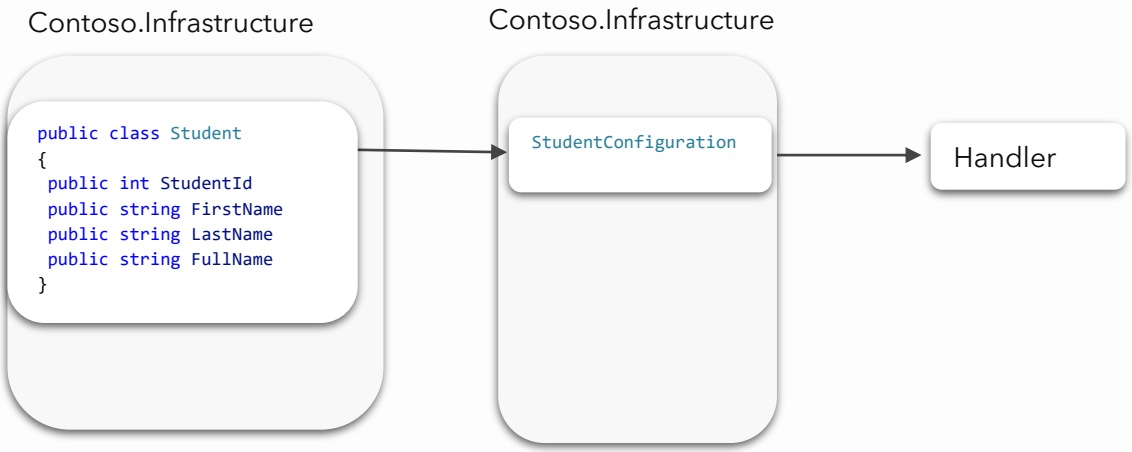
The complete Domain model of this application can be represented in a UML diagram



Github: source

Code First

The projects follow the Code first Entity Framework paradigm which means that there is no annotations or dependencies of the Domain Model to the Entity Framework. Instead the correlation between OOP entities and database Entities is achieved using configuration files




```
class StudentConfiguration :
   IEntityTypeConfiguration<Student>
{
    public void Configure(EntityTypeBuilder<Student> builder)
    {
        builder.Property(b => b.FirstName)
            .HasMaxLength(50);
        builder.Property(b => b.LastName)
            .HasMaxLength(50);
    }
}
```

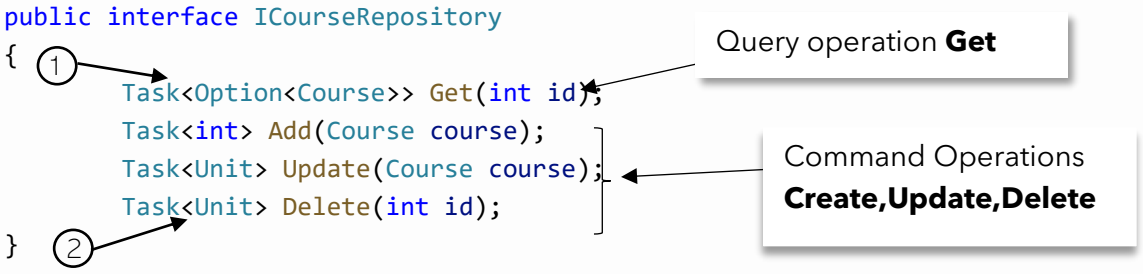
Github: source

8.5.1 Repositories

The repositories are the contact point of the Domain with the outer layers. And in this point is where we can use the functional structures of `Option` and `Either`. As you will see in most of the Repositories the `Get` operations have a similar signature:

```
Task<Option<T>> Get(int id)
```

For the methods that we want to retrieve a single Element by its Id we could also use the Either in the case that we care exposing any Errors in the Database.



Github: source

Also, the result is wrapped in a `Task` because in most cases the Persistance using Entity Framework 6 can be made Asynchronous using `Async` methods for Save and Query .You can take a look at the Repositories implementations where the `SaveChangesAsync` and `SingleOrDefaultAsync` are used.

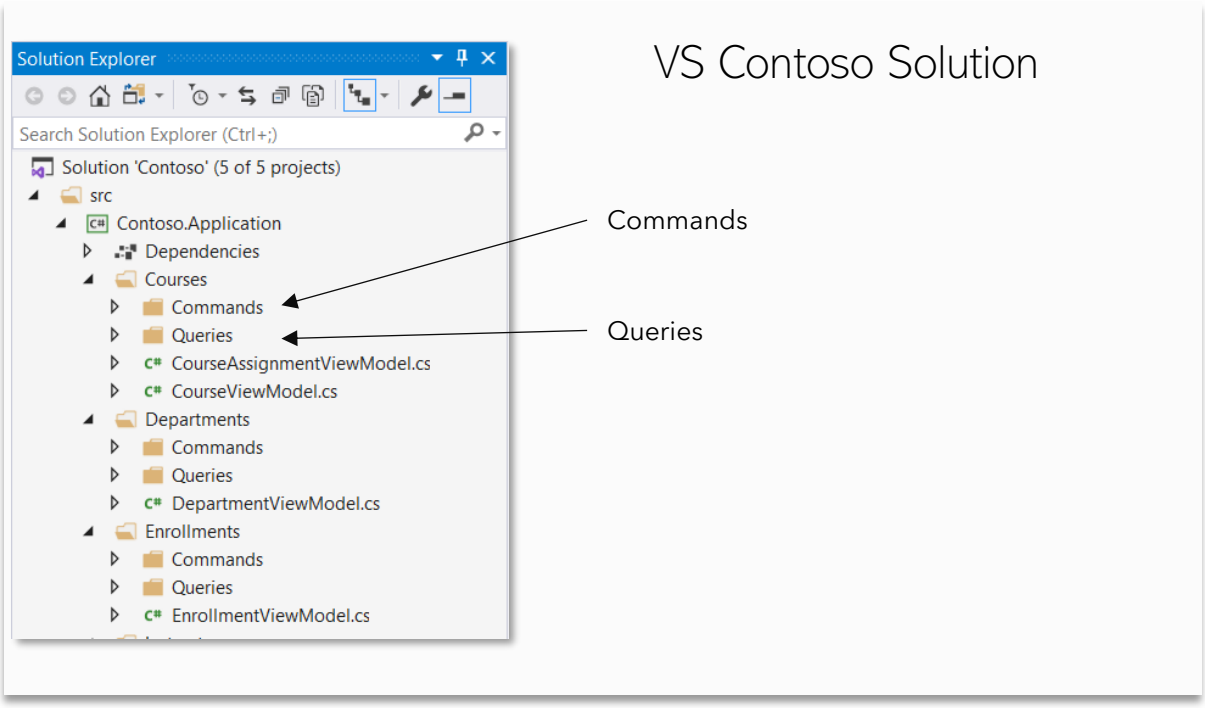
8.6 CQRS

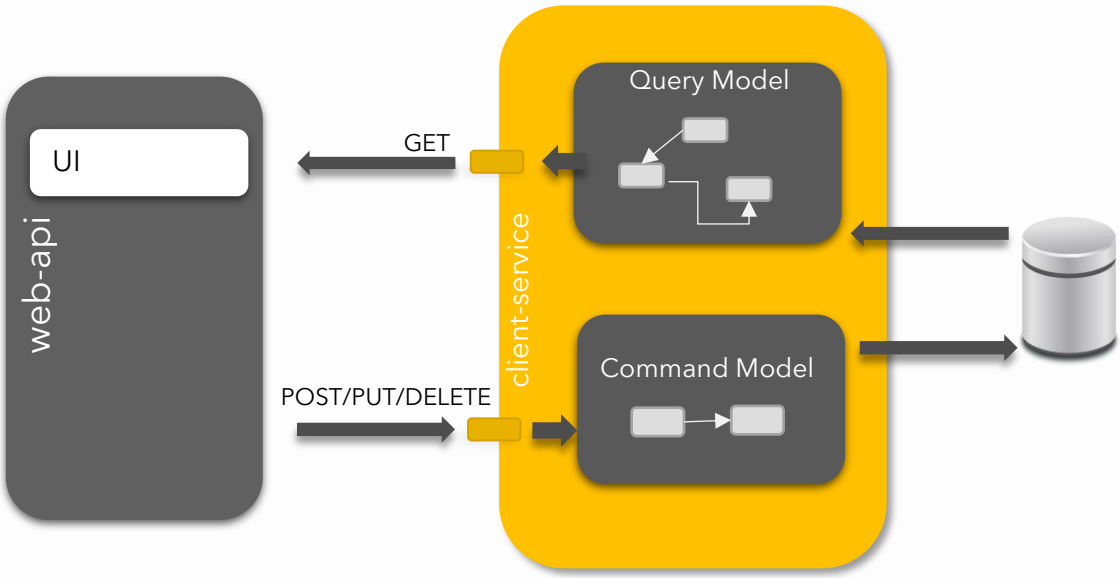
The project uses the CQRS architectural pattern to reduce coupling. The term CQRS, (Command Query Responsibility Segregation) is based on the famous CQS idea. It is the closest architecture available to split these responsibilities to accomplish more and simplify the design. It's all about separating things, especially if they're very different conceptually when it comes to reading and updating models.

The pattern basically divides into two categories:

1. ***Queries**:* get the information only, and never change anything within the model.

2. ***Commands**:* that's when you perform the writes, updates, and deletes (along with all the logic's complexities inherent to the business rules).





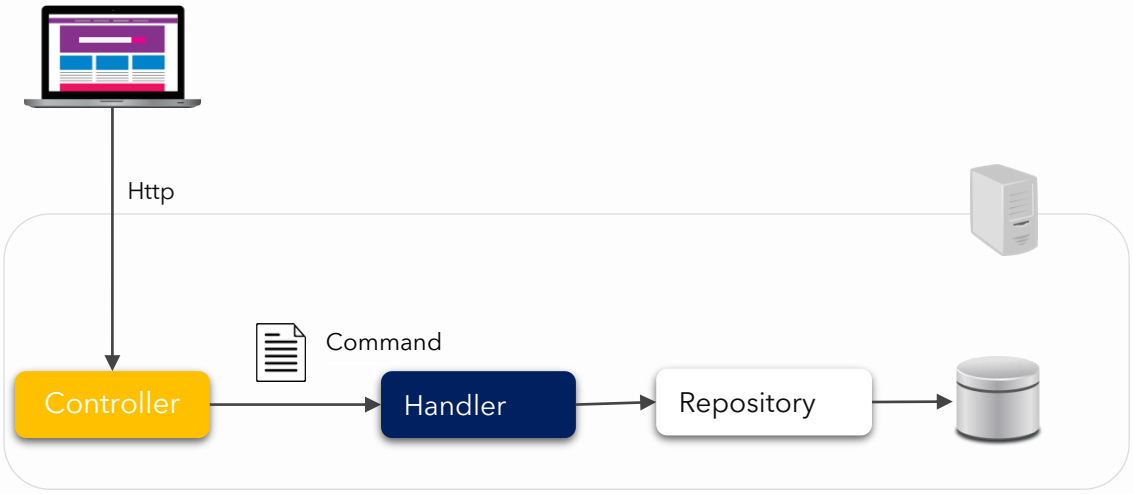
From the point of view of a web application HTTP web API the Queries would be the GET methods and the Commands would be POST/PUT/DELETE methods

1. ***Queries*** - GET methods
2. ***Commands*** - POST/PUT/DELETE methods

Microsoft has the CQRS pattern listed as a Part of the Cloud design patterns and it because of its rising usage you can find reference in many of the Architecture guidance guides for

example in the .NET Microservices: Architecture for Containerized .NET Applications in the Apply simplified CQRS and DDD patterns chapter.

Of course CQRS can get complex inside large Domain Driven Architectures and Microsoft has provides some Guides in the past like *Exploring CQRS and Event Sourcing* that are archived but maintain their value and worth exploring.



8.7 CQRS and MediatR

8.7.1 The Command Pattern with a Mediator

In the context of CQRS Commands and Queries are just DTOs (Data transfer objects) that encapsulate all the information needed to perform an action. For example:

```
public class CreateCourse : Record<CreateCourse>, IRequest<Either<Error, int>>
{
    public CreateCourse(string title, int credits, int departmentId)
    {
        Title = title;
        Credits = credits;
        DepartmentId = departmentId;
    }
```

```
        public string Title { get; }
        public int Credits { get; }
        public int DepartmentId { get; }
    }
```

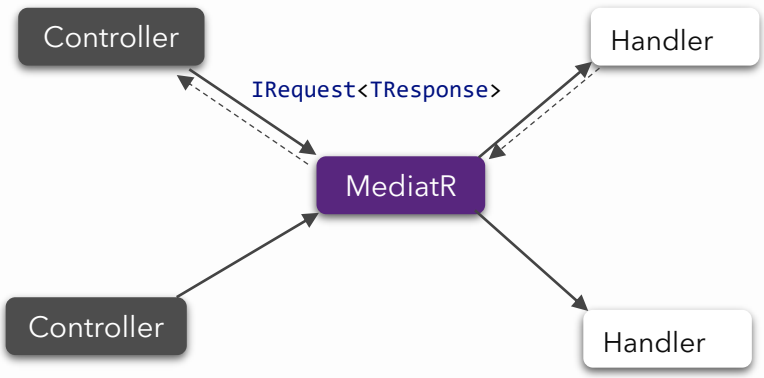
Github: source

In this project the `CreateCourse` DTO extends to Types:

1. is extending the `Record<CreateCourse>` class that belong to the language extension library (Record<A>) and just provides Equals, IEquatable.Equals, IComparer.CompareTo, GetHashCode, operator==, operator!=, operator >, operator >=, operator <, and operator <= implementations for the class. This is not needed in this project but it is demonstrated as a good practice using the library.

2. Implements the `IRequest<Either<Error, int>>` interface that comes from the MediatR and allows the MediatR to find which object to use as a message. You can see the basic request/response pattern that MediatR uses here.

A powerful improvement can be made to the approach by decoupling the execution from the command state via the mediator pattern. In this pattern all commands, each of which consist of simple serializable state, are dispatched to the mediator and the mediator determines which handlers will execute the command:



Because the command is decoupled from execution it is possible to associate multiple handlers with a command and moving a handler to a different process space (for example splitting a Web API into multiple Web APIs) is simply a matter of changing the configuration of the mediator. And because the command is simple state we can easily serialize it into, for example, an event store (the example below illustrates a mediator that is able to serialize commands to stores directly, but this can also be accomplished through additional handlers:

Since commands are imperatives, they are typically named with a verb in the imperative mood (for example, "create" or "update"), and they might include the aggregate type, such

as CreateOrderCommand. Unlike an event, a command is not a fact from the past; it is only a request, and thus may be refused.

It's perhaps worth also briefly touching on the CQRS pattern – while the Command Message pattern may facilitate a CQRS approach it neither requires it or has any particular opinion on it.

With that out the way let's look at how this pattern impacts our applications architecture.

Now inside each controller the Command message is send immediately to the medietR to be dispatched

```
[Produces("application/json")]
    [Route("api/[controller]")]
    [ApiController]
    public class CoursesController : ControllerBase
    {
        private readonly IMediator _mediator;
        public CoursesController(IMediator mediator) => _mediator = mediator;

        [HttpGet("{courseId}")]
        public Task<IActionResult> Get(int courseId) =>
            _mediator.Send(new GetCourseById(courseId)).ToActionResult();
   …

}
```

The MediatR send has this signature and just takes a `IRequest<TResponse>` request message

```
Task<TResponse> Send<TResponse>(IRequest<TResponse> request, CancellationToken can
cellationToken = default)
```

This is hooked to the receiving end to an `IRequestHandler` that has to implement a `Handle(IRequest<TResponse> request)` method


```
public class CreateCourseAssignmentHandler : IRequestHandler<CreateCourseAssignmen
t, Validation<Error, int>>
{
        private readonly IInstructorRepository _instructorRepository;
        private readonly ICourseRepository _courseRepository;
        private readonly ICourseAssignmentRepository _courseAssignmentRepository;

public CreateCourseAssignmentHandler(IInstructorRepository instructorRepository,
            ICourseRepository courseRepository,
            ICourseAssignmentRepository courseAssignmentRepository)
      {
```

```
            _instructorRepository = instructorRepository;
            _courseRepository = courseRepository;
            _courseAssignmentRepository = courseAssignmentRepository;
        }

        public Task<Validation<Error, int>> Handle(CreateCourseAssignment request,
                    CancellationToken cancellationToken) {… }

    }
```

Github: source

We are going now to see in more detail the two basic paths Commands and Queries and how the language-ext functional style fits in.

8.7.2 Query Workflow

In this section we are going to see step by step the workflow of some of the Query paths. We will look at the `GetCourseById` case. We assume that we had a http get request from the client. This means that inside the `CoursesController` the `Get(int courseId)` action is called with the parameter coursed that was used in the http request:

```
public class CoursesController : ControllerBase
 {
    [HttpGet("{courseId}")]
       public Task<IActionResult> Get(int courseId) =>
           _mediator.Send(new GetCourseById(courseId)).ToActionResult();
     …
 }
```

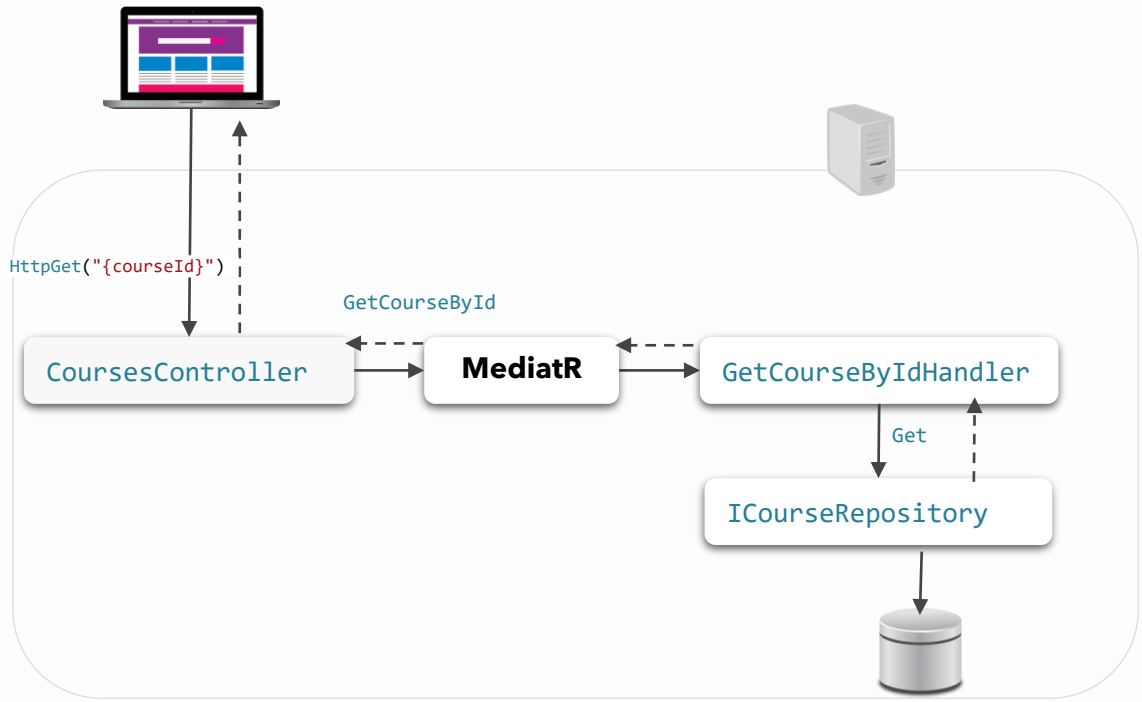
Github: source

As you can see a message is immediately sent to the mediator. The message DTO(data tranfer object) that is used here is `GetCourseById`

```
public class GetCourseById : Record<GetCourseById>, IRequest<Option<CourseViewMode
l>>
 {
        public GetCourseById(int courseId) => CourseId = courseId;
        public int CourseId { get; }
 }
```

Github: source

This DTO only has only the `CourseId` that we want to retrieve.



The Mediator the dispatches the message and pass it to the appropriate handler which is the `GetCourseByIdHandler` which the `MediatR` identifies because it implements the `IRequestHandler<GetCourseById, _ >` interface.

```
GetCourseByIdHandler : IRequestHandler<GetCourseById, Option<CourseViewModel>>
```

This is the Type of the **Request** Message

This is the Type of the **Response** Message

At this point the MediatR invokes the Handle method of the Handler passing inside the request DTO message `GetCourseById`

```
public class GetCourseByIdHandler:IRequestHandler<GetCourseById, Option<CourseView
Model>>
{

public Task<Option<CourseViewModel>> Handle(GetCourseById request,
                    CancellationToken cancellationToken) =>
            Fetch(request.CourseId)
```

The MediatR calls this method

```
                .MapT(Project);
}
```

Github: source

Now at that point the main transaction is finally executed

```
Fetch(request.CourseId).MapT(Project);
```

where Fetch is

```
private Task<Option<Course>> Fetch(int id) => _courseRepository.Get(id);
```

we use the repository that was injected into the handler in order to get the Course with the specific Id from the database. This returns a `Task<Option<Course>>` now we want to convert the `Course` into a `CourseViewModel` which is a new DTO following the View Model pattern:

```
public class CourseViewModel : Record<CourseViewModel>
{
   public int CourseId { get; }
   public string Title { get; }
   public int Credits { get; }
   public int? DepartmentId { get; }
}
```

Here in order to do the Mapping we have to use the `MapT` in order to Bypass the Task and apply the function onto the `Option<Course>` which applies the following transformation to the course if its not `Option.None`

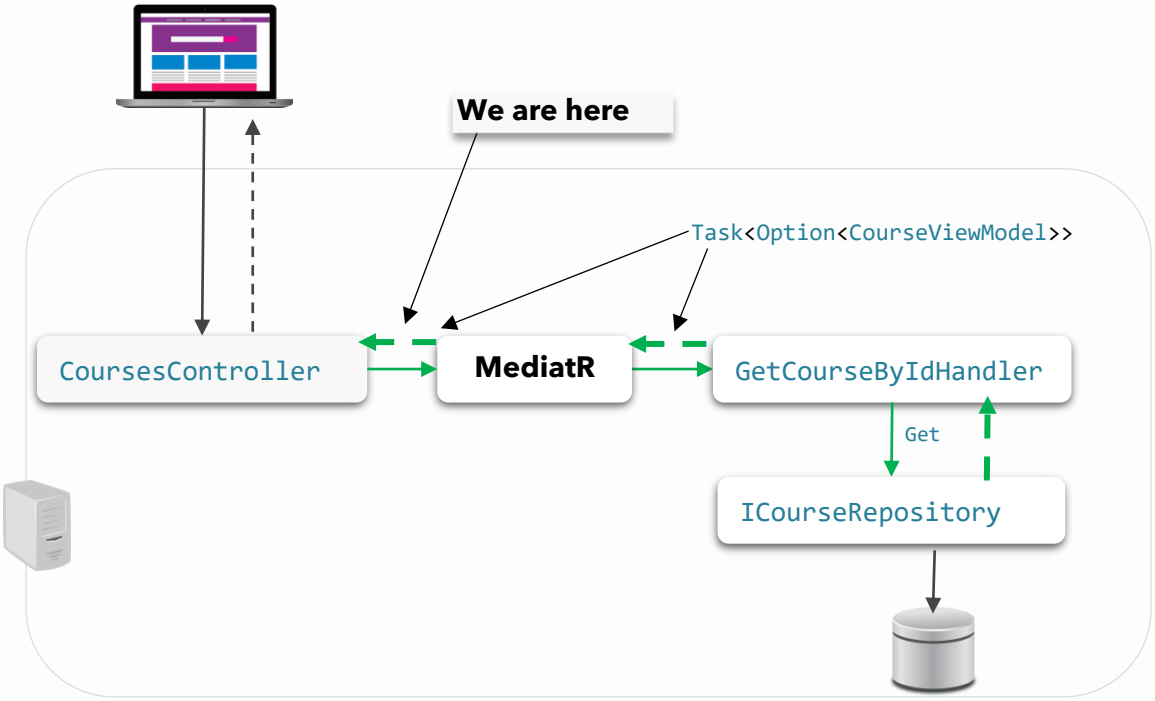
```
private static CourseViewModel Project(Course course) =>

            new CourseViewModel(course.CourseId,
                                course.Title,
                                course.Credits,
                                course.DepartmentId);
```

Github: source

after the `.MapT(Project)` we have a `Task<Option<CourseViewModel>>` as a result. This is what the mediator returns back to the controller as a response.Ok , now we are reached again inside the `CoursesController` at the point

```
_mediator.Send(new GetCourseById(courseId)).ToActionResult();
```

Github: source



The `_mediator.Send(new GetCourseById(courseId))` has finished and return a result on with we apply the `.ToActionResult()`. This is an extension method that will convert the option to an appropriate `IActionResult` that will be the response of the Web Api Controller

```
public static IActionResult ToActionResult<T>(this Option<T> option) =>
            option.Match<IActionResult>(
                Some: t => new OkObjectResult(t),
                None: () => new NotFoundResult());
```

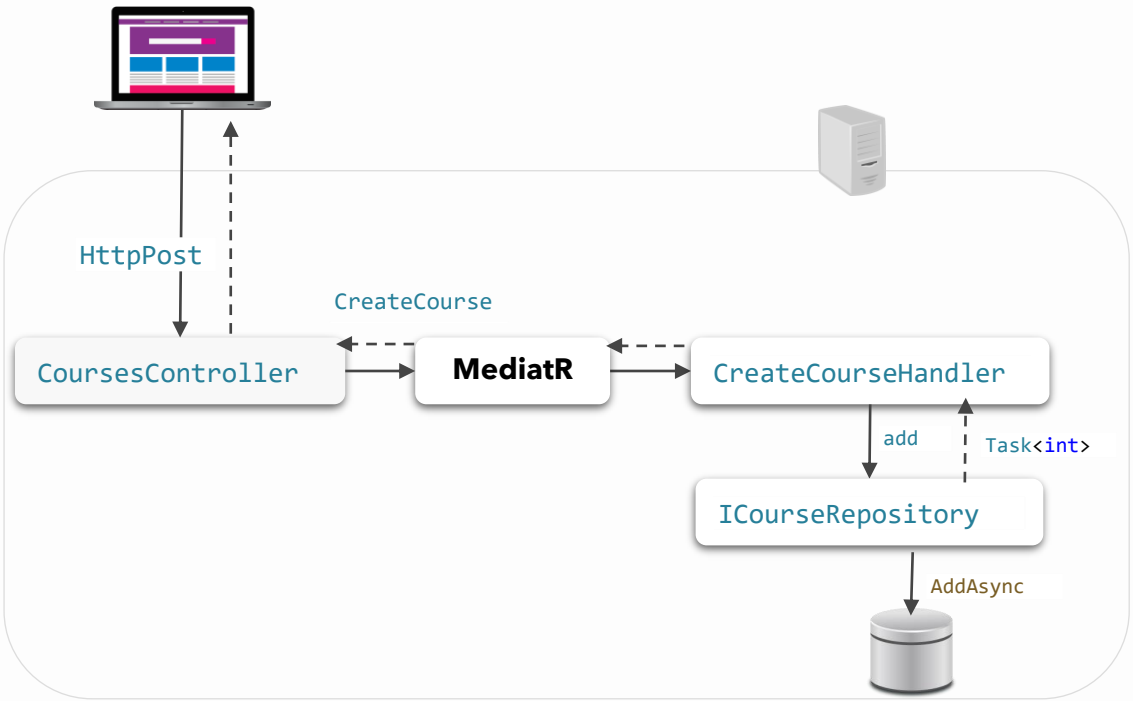
GitHub source

You see here that we pattern match on the Option and in the case where is None we return a `NotFoundResult` that corresponds into a `Not Found (404)` response. At this point obviously we could send a more custom response to the client side. This concludes the lifecycle of the request.

8.7.3 Command Workflow

In this section we are going to see some of the Command paths. We will first examine the `CreateCourse` case, and then we will see some of the other cases to cover all the functional patterns in the Contoso Sample. Ok, again we assume that we had a http POST request this time from the client. This means that inside the `CoursesController` the `Create(CreateCourse createCourse)` action is called, and as an argument we have the deserialized `CreateCourse` object that was used in the http ajax request at the client side:

```
[HttpPost]
    public Task<IActionResult> Create(CreateCourse createCourse) =>
        _mediator.Send(createCourse).ToActionResult();
```



The message `CreateCourse` is immediately sent to the mediator. The DTO message that is used here is :

```
public class CreateCourse : Record<CreateCourse>, IRequest<Either<Error, int>>
{
        public string Title { get; }
        public int Credits { get; }
        public int DepartmentId { get; }
}
```

GitHub source

This DTO only has all the data that we want to store to the DB.

The Mediator the dispatches the message and pass it to the `CreateCourseHandler` which the `MediatR` identifies because it implements the `IRequestHandler<CreateCourse , _ >` interface. As usual The MediatR invokes the Handle method of the Handler passing inside the request DTO message `CreateCourse`

```
public class CreateCourseHandler : IRequestHandler<CreateCourse, Either<Error, in
t>>
{
  public Task<Either<Error, int>> Handle(CreateCourse request,
  CancellationToken cancellationToken) =>
            Validate(request)
                .MapT(PersistCourse)
                .Bind(v => v.ToEitherAsync());
    …
}
```

The MediatR calls the `Handle` method

GitHub source

1. Now at tha point the main transaction is finally executed

```
Validate(request)
    .MapT(PersistCourse)
    .Bind(v => v.ToEitherAsync());
```

This is a composition of three functions first we validate the request. There was no need to do this for the Query workflow, so this is a new step.

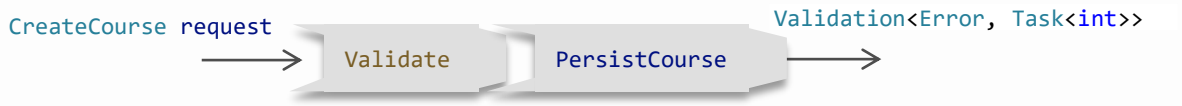
```
private async Task<Validation<Error, Course>> Validate(CreateCourse create) =>
          (await DepartmentMustExist(create), ValidateTitle(create))
              .Apply((id, title) => new Course { Title = title, DepartmentId = i
d, Credits = create.Credits });
```

GitHub source

This uses the validation monad to do some validations and then use the apply `.Apply` to collect the results and recreate the object `new Course { }`. If one or more of the validations failed the result of the Apply is a `Validation` that contains all the errors. In this second case the computation stops there and the subsequent step of `PersistCourse` does not happen.

2. The next step is the `.MapT(PersistCourse)` bypass the Task and applies the function `PersistCourse` on the content of the `Validation` (which is a `Course` object) from the previous step using the `Validation.Map` method.

   At this point the Type that we have at our hands after the `PersistCourse` is `Validation<Error, Task<int>>`



[You can see the types at any given state by placing the cursor over the variable

]

3. we want to convert this `Validation<Error,Task<int>>` into a `Task<Either<Error,int>>` that follows the usual pattern of responses so here is implemented an extension method for this purpose just to transfer the `Task` outside of the `Validation` and also convert the Validation to an Either using `validation.ToEither()`

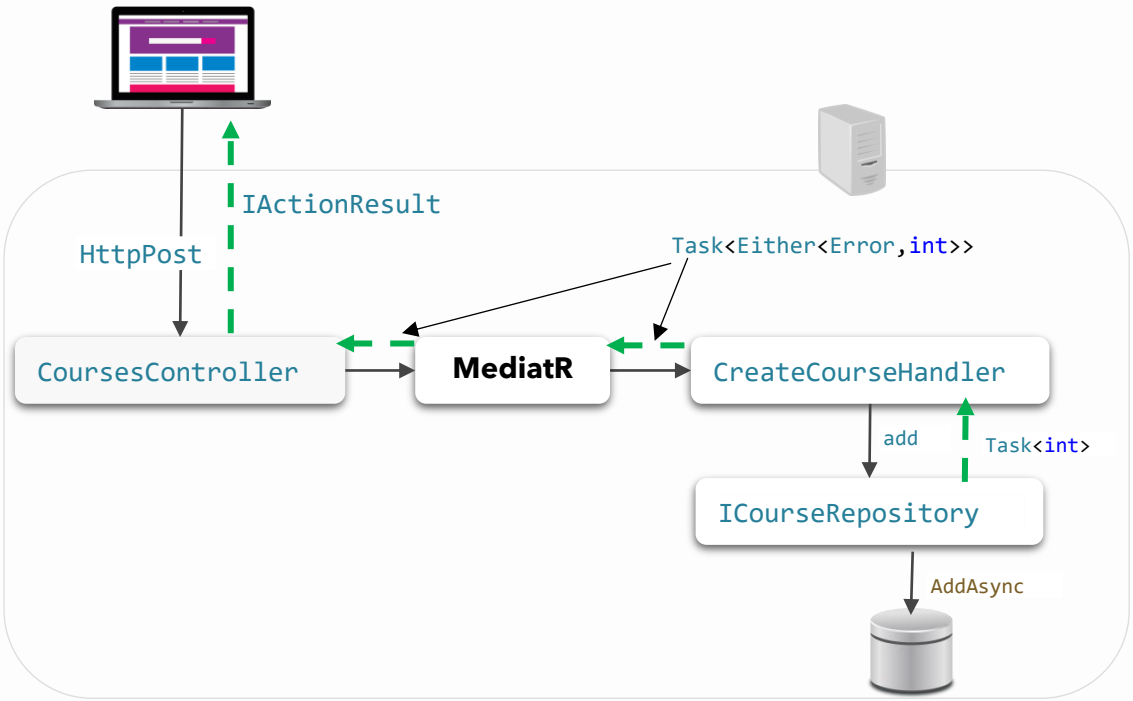
```
public static Task<Either<Error, R>> ToEitherAsync<R>
                        (this Validation<Error, Task<R>> validation) =>
                                        validation.ToEither()
                                            .MapLeft(errors => errors.Join())
                                            .MapAsync<Error, Task<R>, R>(e =>
                                    e);
```

   And is applied through the `.Bind(v => v.ToEitherAsync());`

   All the difficulty of this functional approach is concentrated on how to properly convert the types using the Map, and Bind and the various Transformation methods.

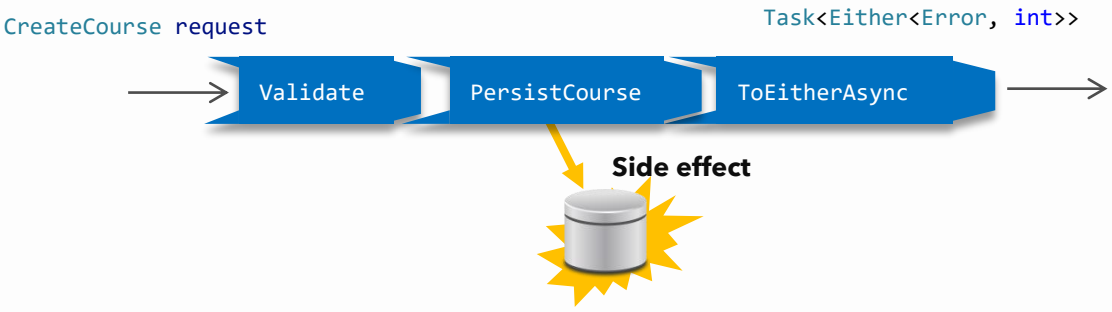
4. Finally, this is send back to controller where the method `ToActionResult` `Task<IActionResult> ToActionResult<L, R>(this Task<Either<L, R>> either)` Is used to convert the Either to an Appropriate `IActionResult`

The whole workflow is the following



Usually we try to avoid side effects, but side effects are unavoidable in the eal world

In the above examples the access to the DB is a side effect because it goes outside of the boundaries of the function and the Workflow and changes the environment of the program.


CreateCourse request
Task<Either<Error, int>>
Validate
PersistCourse
ToEitherAsync
Side effect

Contact

Feel free to contact me at:

https://leanpub.com/u/dimitrispapadim

https://medium.com/@dimpapadim3

https://github.com/dimitris-papadimitriou-chr

dimitrispapadim@live.com