DESENVOLVENDO EM C# COM TANGO

PUBLICADO COM GITBOOK

GABRIEL SCHADE

Tabela de conteúdos

Começando

Instalação

Conceitos

Funcional

Operações

Tipos

Módulos

Extensões

Seja bem-vindo à

Destaques

Utilize *pattern matching* com valores opcionais e valores `Either`

```
Option<int> optionalValue = 10;
int value = optionalValue.Match(
        methodWhenSome: number => number,
        methodWhenNone: () => 0);

Either<bool, int> eitherValue = 10;
int value = eitherValue.Match(
        methodWhenRight: number => number,
        methodWhenLeft: boolean => 0);
```

Crie processos contínuos utilizando Then e Catch

```
Continuation<string, int> continuation = 5;

continuation.Then(value => value + 4)
            .Then(value =>
                  {
                    if( value % 2 == 0)
                        return value + 5;
                    else
                        return "ERROR";
                  })
            .Then(value => value + 10)
            .Catch(fail => $"{fail} catched");
```

Crie processos contínuos utilizando pipeline!

```
continuation
    >  (value => value + 4)
    >  (value =>
       {
          if( value % 2 == 0)
              return value + 5;
          else
              return "ERROR";
       })
    >  (value => value + 10)
    >= (fail => $"{fail} catched")
```

Utilize Poderosas Funções de alta ordem!

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
IEnumerable<IEnumerable<int>> result = source.ChunkBySize(3);

//result = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},  {10} }

var (resultEvens, resultOdds) =
    source.Partition(value => value % 2 == 0)

//resultEvens = { 2, 4, 6, 8, 10 }
//resultOdds = { 1, 3, 5, 7, 9 }
```

Utilize operações para redução!

```
int result = source.Scan(10, IntegerOperations.Add);
//result = {11, 13, 16, 20, 25, 31, 38, 46, 55, 65}
```

Utilize Curry e Aplicação Parcial!

```
Func<int, int, int> add =
    (value, value2) => value + value2;

Func<int, Func<int, int>> addCurried = add.Curry();
curriedResult = addCurried(2)(3);

Func<int, int> addPartial = add.PartialApply(2);
int partialResult = addPartial(3);
```

Aproveite o *QuickDelegateCast*!

```
using static Tango.Functional.QuickDelegateCast;

int SampleAdd(int value1, int value2)
    => value1 + value2;

F<int, int, int>(SampleAdd).Curry();
F<int, int, int>(SampleAdd).PartialApply(1);
```

Totalmente Gratuita

A **Tango** é completamente gratuita, tanto a biblioteca quanto sua documentação, além disso, você pode baixar seu PDF para ler offline!

Faça download do livro da documentação :

- Formato PDF
- Formato mobi
- Formato ePub

Índice

Aqui você encontra todos os tópicos disponíveis nesta documentação.

Começando

Instalação

Conceitos

Funcional

Operações

Tipos

Módulos

Extensões

Do desenvolvedor

Eu, Gabriel Schade, trabalho como desenvolvedor de software na plataforma .NET desde 2008, além de atuar como professor no curso de ciência da computação e ser palestrante em eventos de tecnologia.

Mestre em computação aplicada e bacharel em Ciência da Computação. Consumidor e criador de jogos no tempo livre, apaixonado por tecnologia, literatura fantástica e card games.

Investi um tempo para criar esta biblioteca para melhorar a maneira como os desenvolvedores implementam os conceitos de programação funcional em C#.

Gostaria de agradecer meu grande amigo, Rafael Broering de Souza, por ter idealizado e produzido a logomarca da biblioteca, fica aqui o meu mais sincero **obrigado**.

Você pode me ajudar utilizando o □(star) no repositório da **Tango**.

Esteja livre para tirar qualquer dúvida, enviar feedbacks e entrar em contato comigo.

Você pode me encontrar nas redes sociais:

- Github
- Linkedin
- Twitter
- Facebook

Programação Funcional `|>` C# ❤

Obrigado por utilizar a 

Juntos somos mais fortes!

O que é a Tango?

De certa forma, qualquer desenvolvedor C# já utiliza conceitos funcionais em sua aplicação, mesmo que nem sempre seja de forma consciente. Nos acostumamos a utilizar funcionalidades ótimas da linguagem, como por exemplo, as expressões lambda e até as funções de alta ordem providas pela biblioteca System.Linq.

C# é uma linguagem de programação incrível e nós, como desenvolvedores podemos melhorá-la ainda mais. Nesta biblioteca eu implementei uma série de conceitos fundamentais do paradigma de programação funcional, com o objetivo primário de melhorar a experiência de desenvolvedores que desejam incorporar estes conceitos em uma aplicação C#.

Tango é um conjunto de poderosas ferramentas relacionadas à programação funcional para suas aplicações .NET C#. Com esta biblioteca você será capaz de trabalhar com pipelines (através de métodos e operadores), valores opcionais, valores do tipo `Either`.

Além disso, **Tango** também provê extensões para IEnumerables, Options e Either através do Namespace Tango.Module. Com este namespace é possível utilizar métodos populares presentes em outras linguagens como: Map, Map2, Map3, Filter, Reduce, Fold, Scan e assim por diante, sempre respeitando o *lazy load* do tipo `IEnumerable`.

Programação funcional trás uma série de benefícios para sua aplicação e expandir a caixa de ferramentas para soluções melhora muito sua capacidade de resolver problemas.

Entre todas as melhorias desta abordagem, a melhor delas é sem dúvida o fato de que, programação funcional torna a tarefa de desenvolver muito mais **fun**.

O que está esperando para começar?

Pare de brigar com seu código e tire-o para dançar um Tango!

Por onde começar?

A **Tango** é uma biblioteca totalmente voltada para implementações dos conceitos de programação funcional, caso você não esteja familiarizado com implementações deste tipo é fortemente recomendado que você busque fontes externas para compreender este assunto, desta forma, você será capaz de usufruir da **Tango** de forma plena.

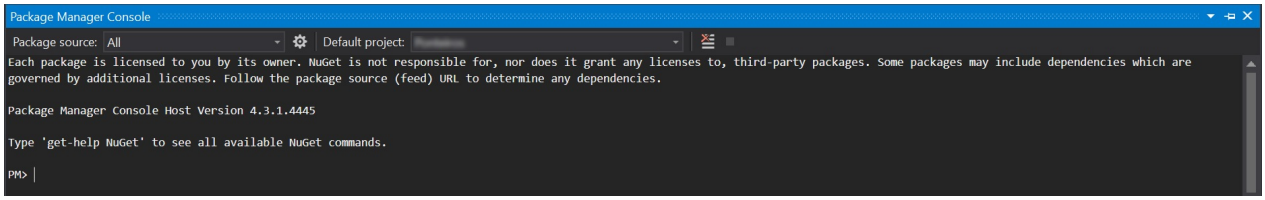
Alguns dos conceitos abordados podem ser encontrados na seção Conceitos.

Com a introdução aos conceitos feita, você está totalmente apto a incorporar a **Tango** em sua aplicação.

Para fazer isso você pode instalá-la via NuGet (fortemente recomendado) ou se preferir realizar a instalação manual.

Instalação via NuGet

A instalação via NuGet é tão simples quanto qualquer outra instalação. Basta utilizar a janela `Package Manager Console` disponível no Visual Studio, conforme imagem abaixo:



E digitar o seguinte comando:

```
PM> Install-Package Tango
```

Com isso a biblioteca **Tango** e suas depêndencias serão instaladas em sua aplicação.

> **Atenção**
>
> Ao instalar a biblioteca através do NuGet todas as dependências serão instaladas automaticamente no projeto.

Dependências

Para a biblioteca funcionar corretamente é necessária a instalação de:

- System.ValueTuple
- Microsoft.Net.Compilers

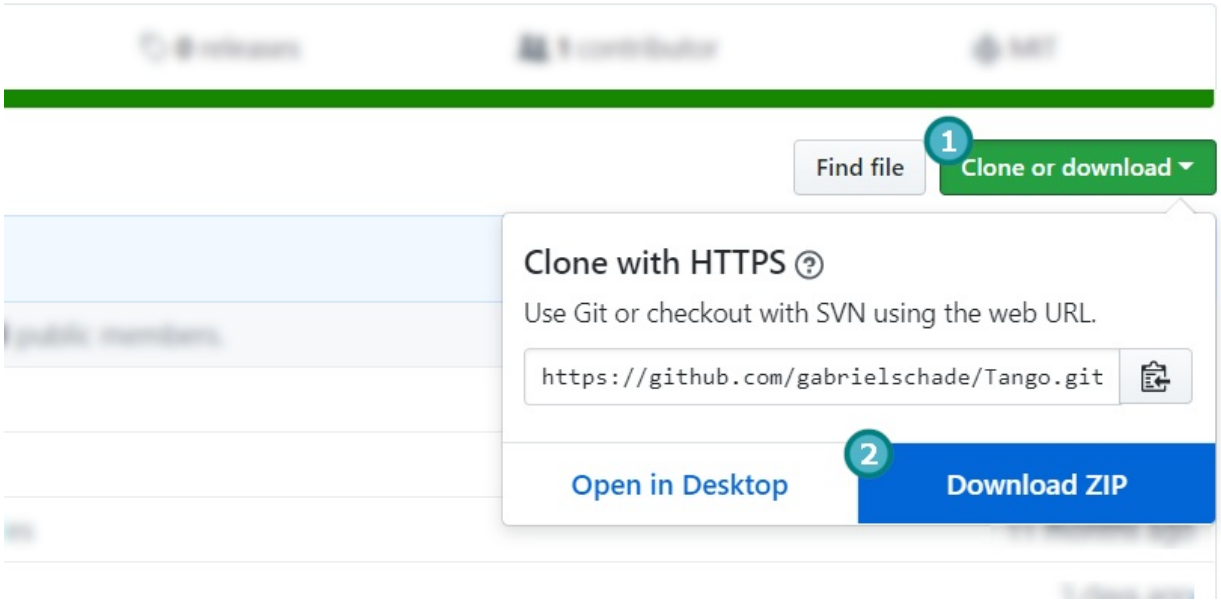
Isso é necessário por conta da biblioteca utilizar a maneira mais moderna de trabalhar-se com tuplas em C#. Tudo isso Para tornar a biblioteca ainda mais confortável de ser utilizada.

Instalação Manual

Apesar de ser mais recomendado realizar a instalação através do NuGet Também é possível realizar a instalação da biblioteca manualmente.

Para fazer isso é necessário acessar o repositório da biblioteca no GitHub.

Depois de acessar o repositório, você precisará pressionar o botão *Clone or download*. Esta ação irá abrir uma pequena janela com diversas informações, pressione o botão *Download ZIP* para realizar o download da biblioteca, conforme imagem a seguir.



Com isso você terá realizado o download de todo o código fonte da biblioteca **Tango**.

Agora você terá de compilá-la na IDE que preferir e adicionar as DLLs geradas na compilação em seu projeto.

> **Atenção**
>
> Você deve adicionar todas as DLLs geradas na compilação ao seu projeto, desta forma você irá garantir que a Tango suas dependências serão carregadas corretamente.

Dependências

Para a biblioteca funcionar corretamente é necessária a instalação de:

- System.ValueTuple
- Microsoft.Net.Compilers

Isso é necessário por conta da biblioteca utilizar a maneira mais moderna de trabalhar-se com tuplas em C#. Tudo isso Para tornar a biblioteca ainda mais confortável de ser utilizada.

Conceitos

Nesta seção você será introduzido à uma série de conceitos relacionados à programação funcional. Estes conceitos geralmente são abstratos, mas possuem implementações concretas dentro da própria biblioteca.

Aqui você encontrará as seguintes subseções:

Utilizando comparação de padrões (Pattern Matching)

Comparação (ou correspondência) de padrões, do inglês *pattern matching* é um dos conceitos fundamentais da programação funcional.

Este conceito pode substituir uma série de desvios de fluxo de código, tais como:

- Comparações If-else
- Switch-cases
- Laços de repetição (for, foreach, while e do)
- Comparação de tipos

Um pattern matching na linguagem F# para substituir uma comparação de If, possui uma sintaxe similar à:

```
match expressaoBooleana with
| true  -> //padrão para o caso true
| false -> //padrão para o caso false
```

Mas podemos realizar comparações entre diferentes padrões, veja um exemplo com um valor inteiro:

```
let valor = 10
match valor with
| 1 | 2 | 3         -> //Quando o valor for 1, 2 ou 3
| n when n % 2 = 0  -> //Quando o valor for par
| 5                 -> //Quando o valor for 5
| _                 -> //Outros casos
```

Existem uma série de benefícios com a utilização do pattern matching, tanto que, a partir da versão 7.0 do C# é possível utilizarmos esta funcionalidade nativamente na linguagem, conforme documentação da Microsoft.

Como implementação na biblioteca **Tango** alguns tipos de objetos possuem um método chamado `Match`.

Este método espera por parâmetros funções (Métodos ou expressões lambdas) para os possíveis padrões de acordo com cada tipo.

Além dele, também há a implementação de um segundo método chamado `Match2`, este método é similar ao primeiro, mas neste caso o padrão é aplicado sobre dois valores diferentes do mesmo tipo ao invés de apenas um valor.

Valores Opcionais

Os valores opcionais são uma maneira de representar dados incertos, nem sempre é possível termos certeza do resultado de alguma operação.

Estes valores geralmente são oferecidos como uma nova proposta ao invés dos tipos anuláveis (*nullables*).

Os valores nulos são problemáticos por se disfarçarem de um valor real, devido ao fluxo de um sistema, ocasionalmente pode ocorrer um acesso à um método ou propriedade de um objeto contendo o valor nulo, isso causa a exceção ArgumentNullException.

Valores opcionais também são uma forma de encapsular um dado, sem permitir que o desenvolvedor acesse esta informação quando ela não existir.

Em algumas linguagens como Haskell, por exemplo, o tipo opcional é representado pelo tipo Maybe, apesar de possuir um nome diferente, o conceito por trás é o mesmo.

Um valor opcional pode estar em um dos dois estados possíveis:

- Contendo alguma coisa;
- Não contendo nada.

E na implementação proposta nesta biblioteca, só é possível acessar o valor de um tipo opcional quando ele encontrar-se no estado: "Contendo alguma coisa".

Como você deve ter notado, a ideia dos tipos anuláveis e dos tipo opcional é bastante similar, mas de forma geral o tipo opcional é mais poderoso, evitando exceções e permitindo criar valores opcionais inclusive para tipos baseado em referência e não somente para tipos primitivos.

As implementações deste conceito são realizadas em duas etapas: através do **tipo** Option e do **módulo** Option.

Valores "Ou um ou Outro" (Either)

Os valores *either*, assim como os valores opcionais são uma maneira de representar dados incertos, mas desta vez o estado "Não contendo nada" é substituído por um outro estado que pode conter mais informações úteis.

Os valores *either* limitam o acesso aos dados que ele contém para que o desenvolvedor tenha acesso somente ao dado correto.

Podemos conter um valor *either* que possa armazenar tanto valores inteiros quanto valores booleans, mas somente um de cada vez.

Um valor *either* pode estar em um dos dois estados possíveis:

- Esquerda - Quando contém um valor do tipo à esquerda;
- Direita - Quand contém um valor do tipo à direita.

Por convenção, utiliza-se o estado à direita para representar dados corretos e o estado à esquerda para representar dados incorretos. Semelhante aos estados de valores opcionais, mas neste caso há um valor em ambos os casos.

Esta implementação foi inspirada no tipo Either da linguagem Haskell.

As implementações deste conceito são realizadas em duas etapas: através do **tipo** Either e do **módulo** Either.

Saindo do void para o Unit

Em algumas linguagens funcionais, **toda** função precisa retornar um tipo de valor. Para os casos onde não há um resultado real utilizamos o tipo `Unit`.

O tipo `Unit` é bastante similar ao tipo `void` em praticamente todos os aspectos. No entanto é possível construir um valor do tipo `Unit`, algo que não temos com o `void`.

Ou seja, é possível criar uma variável do tipo `Unit`, ela não conterá nenhuma informação e nenhum tipo de comportamento, no entanto é algo concreto.

Este valor é útil para permitir a conversão entre uma `Action` e uma `Func`, além disso, permite que operações que não geram retorno sejam colocadas em uma cadeia de execuções.

Delegates `Func` e `Action`

Um delegate é um tipo baseado em referência que por ser utilizado para encapsular um método, seja ele anônimo ou nomeado.

Esta é a forma que o C# utiliza para permitir que sejam criadas funções de alta ordem e para que seja possível tratar os métodos como membros de primeira ordem.

Existem diversos tipos de delegates já inclusos na linguagem, mas entre eles os dois mais importantes para a biblioteca **Tango** são: `Func` e o `Action`.

Os dois podem ser utilizados para representar qualquer método, com uma única, porém importante diferença.

O delegate `Func` só é capaz de encapsular métodos que geram resultados, ou seja, que possuem algum tipo de retorno.

Enquanto o delegate `Action` só é capaz de encapsular métodos que não geram resultados, ou seja, métodos do tipo `void`.

Existem uma série de sobrecargas para os dois delegates contendo diferentes números de *generics*. Estas sobrecargas são utilizadas para identificar os tipos dos parâmetros, que podem variar entre 0 e 16.

Veja um exemplo de encapsulamento de uma função de soma de dois inteiros através do delegate `Func`:

```
Func<int,int,int> Add = (number1, number2) => number1 + number2;
```

Por conta das diferentes sobrecargas é possível utilizar diferentes combinações com o `Func` e o `Action`.

Neste próximo exemplo iremos encapsular um método para verificar se um numero inteiro é um número par:

```
Func<int,bool> IsEven = number => number % 2 == 0;
```

Perceba que no delegate `Func` o último tipo informado no *generics* representa o tipo do retorno da função e os tipos anteriores são respectivamente o tipo de cada parâmetro.

O mesmo ocorre para o caso do Action, mas desta vez não temos um retorno.

```
Action<double> WriteNumber = number => Console.WriteLine(number);
```

A própria biblioteca System.Linq abusa deste conceito, utilizando os delegates `Func` para tornar possível a passagem de expressões lambda como parâmetro para um método como o `Select`, por exemplo.

A biblioteca **Tango** abusa deste conceito para criar funções de alta ordem que melhoram sua usabilidade, tornando o código mais limpo e expressivo.

Utilizando operações encadeadas para processos contínuos

A utilização de operações encadeadas em C# não é nenhuma novidade, utilizamos este recurso em diversas bibliotecas diferentes.

Qualquer método de um objeto que retorne uma instância de si mesmo se torna um método fluente capaz de gerar processos contínuos. Mais uma vez vemos este tipo de implementação na biblioteca `System.Linq`, onde é possível realizar uma filtragem e uma transformação através dos métodos `Where` e `Select`, veja um exemplo:

```
IEnumerable<int> values = Enumerable.Range(0,10)
values.Where(value => value % 2 == 0)
      .Select(value => value * value);
```

Como o exemplo anterior mostra, é possível realizar diversas chamadas consecutivas em uma única linha de comando. Isto é o que chamamos de processo contínuo ou fluente.

A **Tango** implementa uma funcionalidade semelhante em diversos objetos: Option, Either e até mesmo nas próprias coleções como o Linq faz.

Além disso, há um tipo especial de objeto chamado `Continuation` utilizado principalmente para este tipo de processamento fluente, neste caso tratando uma saída de sucesso e uma saída de erro.

Currying e Aplicação Parcial

Currying e aplicação parcial são dois conceitos bastante presentes na programação funcional. Os dois conceitos operam diretamente em funções com o objetivo de alterar seu tipo.

Primeiro vamos entender como exemplificar o tipo de uma função, para isso, vamos utilizar uma sintaxe parecida com a sintaxe utilizada na linguagem F#, também da plataforma .NET.

Em F# o tipo da função é definido por: `parâmetro -> retorno`

Logo, uma função com um parâmetro do tipo `int` e um retorno do tipo `bool` pode ser descrita como: `int -> bool`

Quando houverem múltiplos parâmetros iremos representar aqui (diferente do F#) como: `int, int, int -> bool`

Com isso, estabelecemos uma sintaxe para definir o tipo de uma função.

Agora vamos entender o problema que estas duas técnicas auxiliam na resolução.

A programação funcional é baseada principalmente no conceito de **funções**. Por definição, uma função matemática deve possuir apenas um parâmetro (domínio) e um retorno (alcance).

Logo, isso implicaria que, todas as funções escritas em um código funcional devem conter apenas um parâmetro, como fazer isso? - **Currying**

Entendendo Currying

O processo de *currying* consiste em quebrar funções de N parâmetros em N funções, onde cada função irá receber apenas um parâmetro e retornar uma nova função que espera os parâmetros restantes.

Logo, uma função de soma, que seria representada por: `int, int -> int`

Ao passar pelo processo de currying pode ser representada por: `int -> int -> int`

Vamos realizar um passo-a-passo do processo de currying em uma função simples, que realiza a soma de 2 valores.

Primeiro vamos definir a função como:

```
Func<int, int, int> add =
    (value, value2) => value + value2;
```

Podemos utilizar esta função normalmente, conforme código:

```
int result
Func<int, int, int> add = (value, value2) => value + value2;
result = add(2, 3);
//result = 5
```

Ao realizar o processo de currying na função `add` o retorno será uma função do tipo `Func<int, Func<int,int>>`, ou seja, será uma função que receberá um valor inteiro e retornará um nova função esperando o segundo valor inteiro.

Quando esta segunda função receber o último parâmetro ela irá realizar o processamento proposto por `add`.

Veja como podemos realizar o currying com a **Tango**.

```
Func<int, int, int> add =
    (value, value2) => value + value2;

Func<int, Func<int, int>> addCurried = Currying.Curry(add);
curriedResult = addCurried(2)(3);
//curriedResult = 5
```

Veja que a forma de informar os parâmetros na função `addCurried` é um pouco diferente, isso porque ele gera uma série de funções, onde cada uma espera apenas um parâmetro.

A **Tango** também fornece o `Curry` através de métodos de extensão para os delegates `Func`, portanto ao você também poderá realizar a operação descrita anteriormente da seguinte forma:

```
Func<int, int, int> add =
(value, value2) => value + value2;

//Func<int, Func<int, int>> addCurried = Currying.Curry(add);
Func<int, Func<int, int>> addCurried = add.Curry();
curriedResult = addCurried(2)(3);
//curriedResult = 5
```

Entendendo a Aplicação Parcial

A aplicação parcial é um pouco diferente do processo de Currying, mas também envolve a questão dos tipos de uma função.

Através da aplicação parcial é possível realizar a chamada de um método sem informarmos todos os parâmetros. Como resultado desta operação, será retornada uma nova função que espera todos os parâmetros restantes.

Veja as diferenças entre Currying e aplicação parcial em uma função que soma três números inteiros.

Esta seria o tipo da função de soma: `int, int, int -> int`

Ao realizar o Currying nesta função o resultado obtido seria: `int -> int -> int -> int`

Neste ponto a aplicação parcial funciona completamente diferente do processo de Currying, poderíamos inclusive, realizar diferentes aplicações parciais nesta função.

Ao informar apenas um parâmetro o resultado obtido seria: `int, int -> int`

Informando dois parâmetros o resultado obtido seria: `int -> int`

Similar ao processo de Currying, a operação fundamental descrita pela função só é executada quando todos os parâmetros forem informados, independente da quantidade de funções intermediárias geradas.

Veja a implementação descrita, primeiro através de currying:

```
Func<int, int, int, int> add =
    (value, value2) => value + value2;

Func<int, Func<int, Func<int, int>>> addCurried = add.Curry();
curriedResult = addCurried(2)(3)(5);
//curriedResult = 10
```

Utilizando aplicação parcial com apenas um parâmetro:

```
Func<int, int, int, int> add =
    (value, value2) => value + value2;

Func<int, int, int> addPartialApplied = add.PartialApply(2);
partialAppliedResult = addPartialApplied (3,5);
//partialAppliedResult = 10
```

Utilizando aplicação parcial com dois parâmetros:

```
Func<int, int, int, int> add =
    (value, value2) => value + value2;

Func<int, int> addPartialApplied = add.PartialApply(2,3);
partialAppliedResult = addPartialApplied(5);
//partialAppliedResult = 10
```

Todos os métodos disponíveis para aplicação parcial e Currying operam em métodos de até 4 parâmetros, podendo retornar qualquer tipo, inclusive `void`.

Além disso, eles podem ser utilizados como métodos estáticos das classes `Currying` e `PartialApplication` ou como extensões para `Func` e `Action`.

Funcional

> `Tango.Functional`

Este namespace possui as principais implementações voltadas à funções do paradigma funcional. Entre elas estão a extensão que permite as conversões entre os delegates `Action` e `Function`. E a capacidade de realizar os processos de Currying e aplicação parcial em métodos existentes.

É muito importante compreender os conceitos sobre Currying e aplicação parcial para usufruir deste namespace, portanto é fortemente indicado que você leia a seção Currying e Aplicação Parcial caso não esteja familiarizado com o tópico.

Nesta seção você irá encontrar os seguintes tópicos:

- Currying
- Aplicação Parcial
- Extensões
- Cast Rápido para Delegates

Currying

> `Tango.Functional.Currying`

Esta classe *estática* contém diversas sobrecargas para a operação de Currying, onde cada sobrecarga espera em funções com diferentes quantidades de parâmetros, contendo ou não um retorno.

Métodos

Nome	Parâmetros	Retorno	Descrição
Curry	Func<T, T2, TResult> function	Func<T, Func<T2, TResult>>	Cria uma nova função curriada a partir da função informada no parâmetro.
Curry	Func<T, T2, T3, TResult> function	Func<T, Func<T2, Func<T3,TResult>>>	Cria uma nova função curriada a partir da função informada no parâmetro.
Curry	Func<T, T2, T3, T4, TResult> function	Func<T, Func<T2, Func<T3,Func<T4, TResult>>>>	Cria uma nova função curriada a partir da função informada no parâmetro.
Curry	Action<T, T2> action	Func<T, Action<T2>>	Cria uma nova função curriada a partir da função informada no parâmetro.
Curry	Action<T, T2, T3> action	Func<T, Func<T2, Action<T3>>>	Cria uma nova função curriada a partir da função informada no parâmetro.
Curry	Action<T, T2, T3, T4> action	Func<T, Func<T2, Func<T3,Action<T4>>>>	Cria uma nova função curriada a partir da função informada no parâmetro.

Como Usar

As diversas sobrecargas disponíveis podem ser utilizadas para criar novas funções a partir de funções existentes.

Neste exemplo vamos considerar a função `add` como uma função que realiza a soma de dois números:

```
Func<int, int, int> add = (value, value2) => value + value2;
```

Ao aplicarmos a função `Curry` receberemos como retorno uma nova função, esta função irá esperar apenas um dos parâmetros da soma e retornará uma nova função, que por sua vez esperará um último parâmetro e retornará o resultado da soma.

```
Func<int, int, int> add =
    (value, value2) => value + value2;

Func<int, Func<int, int>> addCurried = Currying.Curry(add);
curriedResult = addCurried(2)(3);
```

O conceito para Currying e demais exemplos podem ser encontrados na seção Conceitos > Currying e Aplicação Parcial.

Aplicação Parcial

> `Tango.Functional.PartialApplication`

Esta classe *estática* contém diversas sobrecargas para realizar a aplicação parcial em uma função. Cada uma das sobrecargas espera uma função com diferentes quantidades de parâmetros, contendo ou não um retorno.

Métodos

Nome	Parâmetros	Retorno	Descrição
PartialApply	Func<T, TResult> function T parameter	Func<TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, TResult> function T parameter T2 parameter2	Func<TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, TResult> function T parameter	Func<T2,TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, TResult> function T parameter T2 parameter2 T3 parameter3	Func<TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, TResult> function T parameter T2 parameter2	Func<T3, TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, TResult> function	Func<T2, T3,	Cria uma nova função a partir de uma aplicação parcial à

	T parameter	TResult>	função informada no parâmetro.
PartialApply	Func<T, T2, T3, T4, TResult> function T parameter T2 parameter2 T3 parameter3 T4 parameter4	Func<TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, T4, TResult> function T parameter T2 parameter2 T3 parameter3	Func<T4, TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, T4, TResult> function T parameter T2 parameter2	Func<T3, T4, TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Func<T, T2, T3, T4, TResult> function T parameter	Func<T2, T3, T4, TResult>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T> action T parameter	Action	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2> action T parameter T2 parameter2	Action	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2> action T parameter	Action<T2>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.

			parâmetro.
PartialApply	Action<T, T2, T3> action T parameter T2 parameter2 T3 parameter3	Action	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3> action T parameter T2 parameter2	Action<T3>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3> action T parameter	Action<T2, T3>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3, T4> function T parameter T2 parameter2 T3 parameter3 T4 parameter4	Action	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3, T4> function T parameter T2 parameter2 T3 parameter3	Action<T4>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3, T4> function T parameter T2 parameter2	Action<T3, T4>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.
PartialApply	Action<T, T2, T3, T4> function T parameter	Action<T2, T3, T4>	Cria uma nova função a partir de uma aplicação parcial à função informada no parâmetro.

Como Usar

Assim como no processo de Currying, as diversas sobrecargas disponíveis podem ser utilizadas para criar novas funções a partir de funções existentes.

Neste exemplo vamos considerar a função `add` como uma função que realiza a soma de dois números:

```
Func<int, int, int> add = (value, value2) => value + value2;
```

Ao aplicarmos a função `PartialApply` receberemos como retorno uma nova função, esta função esperará o último parâmetro da soma e retornará o resultado da soma.

```
Func<int, int, int> add =
    (value, value2) => value + value2;

Func<int, int> addPartial = PartialApplication.PartialApply(add, 2);
int partialResult = addPartial(3);
```

Diferente do Currying, ao realizarmos a aplicação parcial precisamos informar um ou mais parâmetros da função e o retorno sempre será uma função que espera todos os parâmetros restantes e retorna o resultado da função principal.

O conceito para Aplicação Parcial e demais exemplos podem ser encontrados na seção Conceitos > Currying e Aplicação Parcial.

Extensões

> `Tango.Functional.FunctionExtensions`

Esta classe *estática* contém diversos métodos que funcionam como extensão para os delegates `Action` e `Func`. Com isso, é possível utilizar todos os métodos disponíveis nas classes `Currying` e `PartialApplication` como extensão para os delegates.

Além disso, esta classe adiciona a possibilidade de converter uma `Action` para um `Func` que retorne um `Unit` e vice-versa.

Métodos

Nome	Parâmetros	Retorno	Descrição
ToFunction	this Action action	Func<Unit>	Converte um delegate Action para um delegate Func, mantendo os mesmos parâmetros e retornando um Unit.
ToFunction	this Action<T> action	Func<T, Unit>	Converte um delegate Action para um delegate Func, mantendo os mesmos parâmetros e retornando um Unit.
ToFunction	this Action<T, T2> action	Func<T, T2, Unit>	Converte um delegate Action para um delegate Func, mantendo os mesmos parâmetros e retornando um Unit.
ToFunction	this Action<T, T2, T3> action	Func<T, T2, T3, Unit>	Converte um delegate Action para um delegate Func, mantendo os mesmos parâmetros e retornando um Unit.
ToFunction	this Action<T, T2, T3, T4> action	Func<T, T2, T3, T4, Unit>	Converte um delegate Action para um delegate Func, mantendo os mesmos parâmetros e retornando um Unit.
ToAction	this Func<Unit> function	Action	Converte um delegate Func que retorne um Unit para um delegate Action, mantendo os mesmos parâmetros.
ToAction	this Func<T, Unit> function	Action<T>	Converte um delegate Func que retorne um Unit para um delegate Action, mantendo os mesmos parâmetros.
ToAction	this Func<T, T2, Unit> function	Action<T, T2>	Converte um delegate Func que retorne um Unit para um delegate Action, mantendo os mesmos parâmetros.
ToAction	this Func<T, T2, T3, Unit> function	Action<T, T2, T3>	Converte um delegate Func que retorne um Unit para um delegate Action, mantendo os mesmos parâmetros.
ToAction	this Func<T, T2, T3, T4, Unit> function	Action<T, T2, T3, T4>	Converte um delegate Func que retorne um Unit para um delegate Action, mantendo os mesmos parâmetros.

> **Atenção**
>
> Esta classe possui mais métodos, mas todos os métodos não listados aqui são apenas sobrecargas para os métodos disponíveis em `Currying` e `PartialApplication`.
>
> Estas sobrecargas simplesmente transformam os métodos das classes acima em extensões para os delegates `Func` e `Action`.

Como Usar

Diferente das outras classes deste namespace, esta classe pode atuar como extensão para os delegates `Func` e `Action`, presentes na linguagem.

Após importar o namespace `Tango.Functional` você será capaz de realizar estas operações como se elas fossem métodos dos próprios delegates!

Veja o exemplo a seguir:

```
Action<int> writeNumber = (number) => Console.WriteLine(number);
Func<int, Unit> functionWriteNumber = writeNumber.ToFunction();
```

Com isso podemos converter um delegate para outro facilmente!

Isso poderá auxiliá-lo na utilização de funções de alta ordem quando o parâmetro necessário for de um delegate diferente do utilizado para encapsular sua função!

As operações de Curry e Aplicação Parcial funcionam de forma bastante similar às descritas nas seções: `Currying` e `PartialApplication`, no entanto, através desta classe os métodos podem ser utilizados como extensão.

Veja a função de soma descrita abaixo:

```
 Func<int, int, int> add = (value, value2) => value + value2;
```

Podemos realizar tanto o processo de currying quanto de aplicação parcial, conforme código:

```
 Func<int, int, int> add =
   (value, value2) => value + value2;

Func<int, Func<int,int>> addCurried = add.Curry();
Func<int,int> addPartialApplied = add.PartialApply(2);
```

Perceba que os métodos agora podem ser utilizados como se fossem métodos do próprio delegate.

```
add.Curry();
//ao invés de:
Currying.Curry(add);
//e
add.PartialApply(2);
//ao invés de:
PartialApplication.PartialApply(add,2);
```

O conceito as técnicas de currying e aplicação parcial, além de mais exemplos podem ser encontrados na seção Conceitos > Currying e Aplicação Parcial.

Cast Rápido para Delegates

> `Tango.Functional.QuickDelegateCast`

Esta classe *estática* contém métodos para transformar uma função nomeada em seu respectivo delegate: `Func` ou `Action`.

Desta forma é possível utilizar as extensões para `Curry` e `Aplicação Parcial` mais facilmente.

Métodos

Nome	Parâmetros	Retorno	Descrição
F	Func<TResult> function	Func<TResult>	Retorna a função enviada por parâmetro.
F	Func<T, TResult> function	Func<T, TResult>	Retorna a função enviada por parâmetro.
F	Func<T, T2, TResult> function	Func<T, T2, TResult>	Retorna a função enviada por parâmetro.
F	Func<T, T2, T3, TResult> function	Func<T, T2, T3, TResult>	Retorna a função enviada por parâmetro.
F	Func<T, T2, TResult> function	Func<T, T2, TResult>	Retorna a função enviada por parâmetro.
A	Action function	Action	Retorna a função enviada por parâmetro.
A	Action<T> function	Action<T>	Retorna a função enviada por parâmetro.
A	Action<T, T2> function	Action<T, T2>	Retorna a função enviada por parâmetro.
A	Action<T, T2, T3> function	Action<T, T2, T3>	Retorna a função enviada por parâmetro.
A	Action<T, T2, T3, T4> function	Action<T, T2, T3, T4>	Retorna a função enviada por parâmetro.

Como Usar

As diversas sobrecargas disponíveis podem ser utilizadas para transformar uma função nomeada em um delegate.

Infelizmente, por conta de uma limitação na linguagem não é possível tratar uma função nomeada diretamente como um delegate, conforme exemplo:

```
int SampleAdd(int value1, int value2)
    => value1 + value2;

SampleAdd.Curry();
SampleAdd.PartialApply(1);
```

A tentativa de executar os métodos `Curry` e `PartialApply` irão causar erro de compilação, notificando que `SampleAdd` é um método e não pode ser utilizado neste contexto.

Para fazer isso precisamos adicionar uma instrução de conversão para o delegate.

```
Func<int, int, int> sampleAddAsDelegate = SampleAdd;

sampleAddAsDelegate.Curry();
sampleAddAsDelegate.PartialApply(1);
```

Através desta classe é possível realizar a conversão *inline*, sem termos de criar instruções adicionais.

Para fazer isso você precisa seguir os seguintes passos:

1. Adicionar uma instrução `using static Tango.Functional.QuickDelegateCast`, para realizar a importação estática do conversor;
2. Utilizar a função correspondente descrita na subseção `Métodos`.

```
using static Tango.Functional.QuickDelegateCast;

int SampleAdd(int value1, int value2)
    => value1 + value2;

F<int, int, int>(SampleAdd).Curry();
F<int, int, int>(SampleAdd).PartialApply(1);
```

Através da sintaxe `F()` e `A()` é possível converter uma função para seu respectivo delegate e com isso, facilitar o processo de `Currying` e `Aplicação Parcial`.

O conceito para Currying e demais exemplos podem ser encontrados na seção Conceitos > Currying e Aplicação Parcial.

Operações

> `Tango.CommonOperations`

Este namespace possui implementações voltadas à disponibilizar operações comuns entre os tipos primitivos no formato de funções. Com isso é possível utilizar apenas o nome das funções como parâmetro para funções de alta ordem.

Estas implementações servem como facilitador para tornar o código mais limpo e organizado, principalmente quando trabalho em conjunto com o módulo de coleções.

Nesta seção você irá encontrar os seguintes tópicos:

- Operações com Booleans
- Operações com Inteiros
- Operações com Decimais
- Operações com Doubles
- Operações com Strings

Operações com Booleans

> `Tango.CommonOperations.BoolOperations`

Esta classe *estática* contém métodos e propriedades com as operações comuns para trabalhar com valores do tipo `bool`. Todos os membros retornam os `delegates` que realizam a operação descrita.

Propriedades

Nome	Tipo	Descrição
Not	Func<bool, bool>	Retorna uma função para representar o operador !.
And	Func<bool, bool, bool>	Retorna uma função para representar o operador &&.
Or	Func<bool, bool, bool>	Retorna uma função para representar o operador ||.

Métodos

Nome	Parâmetros	Retorno	Descrição
AndWith	bool value	Func<bool, bool>	Retorna uma função parcialmente aplicada à função retornada por *And* com o parâmetro informado.
OrWith	bool value	Func<bool, bool>	Retorna uma função parcialmente aplicada à função retornada por *Or* com o parâmetro informado.

Como Usar

Como as propriedades retornam delegates para representar as operações, é possível utilizá-las como métodos.

Operação Not

```
bool value = true;
bool result = BooleanOperations.Not(value);

//result= false
```

Operação Or

```
bool value = true;
bool value2 = false;
bool result= BooleanOperations.Or(value, value2);

//result = true
```

Operação And

```
bool value = true;
bool value2 = false;
bool result= BooleanOperations.And(value, value2);

//result = false
```

Para os métodos temos uma sintaxe um pouco diferente, isso porque é realizada uma aplicação parcial ao método retornado pela própriedade.

Por conta disso, precisamos executar o método com o primeiro parâmetro, para obtermos um novo método que espera o segundo parâmetro:

Operação OrWith

```
bool value = true;
bool value2 = false;
Func<bool,bool> orWithPartial = BooleanOperations.OrWith(value);
bool result = orWithPartial(value2);
//result = true
```

Também podemos realizar a chamada de forma concatenada:

```
bool value = true;
bool value2 = false;
bool result = BooleanOperations.OrWith(value)(value2);

//result = true
```

Operação AndWith

```
bool value = true;
bool value2 = false;
Func<bool,bool> andWithPartial = BooleanOperations.AndWith(value);
bool result = andWithPartial(value2);

//result = false
```

Também podemos realizar a chamada de forma concatenada:

```
bool value = true;
bool value2 = false;
bool result = BooleanOperations.AndWith(value)(value2);

//result = false
```

Operações com Inteiros

> `Tango.CommonOperations.IntegerOperations`

Esta classe *estática* contém métodos e propriedades com as operações comuns para trabalhar com valores do tipo `int`. Todos os membros retornam os `delegates` que realizam a operação descrita.

> **Atenção**
>
> Todas as classes de operações com valores númericos (`int`, `decimal` e `double`) possuem as mesmas propriedades e métodos, alterando apenas o tipo dos parâmetros envolvidos.

Propriedades

Nome	Tipo	Descrição
Add	Func<int, int, int>	Retorna uma função para representar o operador +.
Subtract	Func<int, int, int>	Retorna uma função para representar o operador -.
Multiply	Func<int, int, int>	Retorna uma função para representar o operador *.
Divide	Func<int, int, int>	Retorna uma função para representar o operador /.
Add3	Func<int, int, int, int>	Retorna uma função para representar o operador + entre três valores (valor1 + valor2 + valor3).
Subtract3	Func<int, int, int, int>	Retorna uma função para representar o operador - entre três valores (valor1 - valor2 - valor3).
Multiply3	Func<int, int, int, int>	Retorna uma função para representar o operador * entre três valores (valor1 * valor2 * valor3).
Divide3	Func<int, int, int int>	Retorna uma função para representar o operador / entre três valores (valor1 / valor2 / valor3).

Métodos

Nome	Parâmetros	Retorno	Descrição
AddWith	int value	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Add* com o parâmetro informado.
SubtractWith	int value	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Subtract* com o parâmetro informado.
MultiplyWith	int value	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Multiply* com o parâmetro informado.
DivideWith	int value	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Divide* com o parâmetro informado.
Add3With	int value	Func<int, int, int>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com o parâmetro informado.
Add3With	int value int value2	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com os dois parâmetros informados.
Subtract3With	int value	Func<int, int, int>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com o parâmetro informado.
Subtract3With	int value int value2	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com os dois parâmetros informados.
Multiply3With	int value	Func<int, int, int>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com o parâmetro informado.
Multiply3With	int value int value2	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com os dois parâmetros informados.
Divide3With	int value	Func<int, int, int>	Retorna uma função parcialmente aplicada à função retornada por *Divide3* com o parâmetro informado.
Divide3With	int value int value2	Func<int, int>	Retorna uma função parcialmente aplicada à função retornada por *Divide3* com os dois parâmetros informados.

Como Usar

Como as propriedades retornam delegates para representar as operações, é possível utilizá-las como métodos.

Operação Add

```
int value = 10;
int value2 = 5;
int result = IntegerOperations.Add(value, value2);

//result= 15
```

Operação Subtract

```
int value = 10;
int value2 = 5;
int result = IntegerOperations.Subtract(value, value2);

//result= 5
```

Operação Multiply

```
int value = 10;
int value2 = 5;
int result = IntegerOperations.Multiply(value, value2);

//result= 50
```

Operação Divide

```
int value = 10;
int value2 = 5;
int result = IntegerOperations.Divide(value, value2);

//result= 2
```

Para os métodos temos uma sintaxe um pouco diferente, isso porque é realizada uma aplicação parcial ao método retornado pela própriedade.

Por conta disso, precisamos executar o método com os primeiros parâmetros, para obtermos um novo método que espera os parâmetros restantes:

Operação AddWith

```
int value = 10;
int value2 = 5;
Func<int,int> addWith= IntegerOperations.AddWith(value);
int result = addWith(value2);

//result= 15
```

Também podemos realizar a chamada de forma concatenada:

```
int value = 10;
int value2 = 5;
int result = IntegerOperations.AddWith(value)(value2);

//result= 15
```

As operações ~*With* existem para as quatro operações descritas, todas elas seguem a mesma característica do exemplo anterior.

Os métodos ~*With* também são aplicáveis as funções que utilizam três parâmetros, nestes casos você pode utilizar a aplicação parcial com um ou dois parâmetros, de acordo com a necessidade.

Operação Add3With

```
int value = 10;
int value2 = 5;
int value3 = 5
Func<int, int, int> addWith2= IntegerOperations.Add3With(value);
int result = addWith2(value2, value3);

//result= 20
```

Também podemos realizar a chamada de forma concatenada:

```
int value = 10;
int value2 = 5;
int value3 = 5
int result = IntegerOperations.Add3With(value)(value2, value3);

//result= 20
```

E por fim, podemos informar os dois parâmetros na primeira sobrecarga, em qualquer um dos modos:

```
int value = 10;
int value2 = 5;
int value3 = 5
Func<int, int> addWith= IntegerOperations.Add3With(value, value2);
int result = addWith(value3);

//result= 20
```

```
int value = 10;
int value2 = 5;
int value3 = 5
int result = IntegerOperations.Add3With(value,value2)(value3);

//result= 20
```

Perceba que as versões *~With* passam pelo processo de **aplicação parcial**, não de currying.

Operações com Decimais

> `Tango.CommonOperations.DecimalOperations`

Esta classe *estática* contém métodos e propriedades com as operações comuns para trabalhar com valores do tipo `decimal`. Todos os membros retornam os `delegates` que realizam a operação descrita.

> **Atenção**
>
> Todas as classes de operações com valores númericos (`int`, `decimal` e `double`) possuem as mesmas propriedades e métodos, alterando apenas o tipo dos parâmetros envolvidos.

Propriedades

Nome	Tipo	Descrição
Add	Func<decimal, decimal, decimal>	Retorna uma função para representar o operador +.
Subtract	Func<decimal, decimal, decimal>	Retorna uma função para representar o operador -.
Multiply	Func<decimal, decimal, decimal>	Retorna uma função para representar o operador *.
Divide	Func<decimal, decimal, decimal>	Retorna uma função para representar o operador /.
Add3	Func<decimal, decimal, decimal, decimal>	Retorna uma função para representar o operador + entre três valores (valor1 + valor2 + valor3).
Subtract3	Func<decimal, decimal, decimal, decimal>	Retorna uma função para representar o operador - entre três valores (valor1 - valor2 - valor3).
Multiply3	Func<decimal, decimal, decimal, decimal>	Retorna uma função para representar o operador * entre três valores (valor1 * valor2 * valor3).
Divide3	Func<decimal, decimal, decimal, decimal>	Retorna uma função para representar o operador / entre três valores (valor1 / valor2 / valor3).

Métodos

Nome	Parâmetros	Retorno	Descrição
AddWith	decimal value	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Add* com o parâmetro informado.
SubtractWith	decimal value	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Subtract* com o parâmetro informado.
MultiplyWith	decimal value	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Multiply* com o parâmetro informado.
DivideWith	decimal value	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Divide* com o parâmetro informado.
Add3With	decimal value	Func<decimal, decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com o parâmetro informado.
Add3With	decimal value decimal value2	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com os dois parâmetros informados.
Subtract3With	decimal value	Func<decimal, decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com o parâmetro informado.
Subtract3With	decimal value decimal value2	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com os dois parâmetros informados.
Multiply3With	decimal value	Func<decimal, decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com o parâmetro informado.
Multiply3With	decimal value decimal value2	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com os dois parâmetros informados.
Divide3With	decimal value	Func<decimal, decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Divide3* com o parâmetro informado.
	decimal		Retorna uma função parcialmente

Divide3With	value decimal value2	Func<decimal, decimal>	Retorna uma função parcialmente aplicada à função retornada por *Divide3* com os dois parâmetros informados.

Como Usar

Como as propriedades retornam delegates para representar as operações, é possível utilizá-las como métodos.

Operação Add

```
decimal value = 10;
decimal value2 = 5;
decimal result = DecimalOperations.Add(value, value2);

//result= 15
```

Operação Subtract

```
decimal value = 10;
decimal value2 = 5;
decimal result = DecimalOperations.Subtract(value, value2);

//result= 5
```

Operação Multiply

```
decimal value = 10;
decimal value2 = 5;
decimal result = DecimalOperations.Multiply(value, value2);

//result= 50
```

Operação Divide

```
decimal value = 10;
decimal value2 = 5;
decimal result = DecimalOperations.Divide(value, value2);

//result= 2
```

Para os métodos temos uma sintaxe um pouco diferente, isso porque é realizada uma aplicação parcial ao método retornado pela própriedade.

Por conta disso, precisamos executar o método com os primeiros parâmetros, para obtermos um novo método que espera os parâmetros restantes:

Operação AddWith

```
decimal value = 10;
decimal value2 = 5;
Func<decimal ,decimal> addWith= DecimalOperations.AddWith(value);
decimal result = addWith(value2);

//result= 15
```

Também podemos realizar a chamada de forma concatenada:

```
decimal value = 10;
decimal value2 = 5;
decimal result = DecimalOperations.AddWith(value)(value2);

//result= 15
```

As operações *~With* existem para as quatro operações descritas, todas elas seguem a mesma característica do exemplo anterior.

Os métodos *~With* também são aplicáveis as funções que utilizam três parâmetros, nestes casos você pode utilizar a aplicação parcial com um ou dois parâmetros, de acordo com a necessidade.

Operação Add3With

```
decimal value = 10;
decimal value2 = 5;
decimal value3 = 5
Func<decimal, decimal, decimal> addWith2 = DecimalOperations.Add3With(value);
decimal result = addWith2(value2, value3);

//result= 20
```

Também podemos realizar a chamada de forma concatenada:

```
decimal value = 10;
decimal value2 = 5;
decimal value3 = 5
decimal result = DecimalOperations.Add3With(value)(value2, value3);

//result= 20
```

E por fim, podemos informar os dois parâmetros na primeira sobrecarga, em qualquer um dos modos:

```
decimal value = 10;
decimal value2 = 5;
decimal value3 = 5
Func<decimal, decimal> addWith= DecimalOperations.Add3With(value, value2);
decimal result = addWith(value3);

//result= 20
```

```
decimal value = 10;
decimal value2 = 5;
decimal value3 = 5
decimal result = DecimalOperations.Add3With(value,value2)(value3);

//result= 20
```

Perceba que as versões ~*With* passam pelo processo de **aplicação parcial**, não de currying.

Operações com Doubles

> `Tango.CommonOperations.DoubleOperations`

Esta classe *estática* contém métodos e propriedades com as operações comuns para trabalhar com valores do tipo `double`. Todos os membros retornam os `delegates` que realizam a operação descrita.

> **Atenção**
>
> Todas as classes de operações com valores númericos (`int` , `decimal` e `double`) possuem as mesmas propriedades e métodos, alterando apenas o tipo dos parâmetros envolvidos.

Propriedades

Nome	Tipo	Descrição
Add	Func<double, double, double>	Retorna uma função para representar o operador +.
Subtract	Func<double, double, double>	Retorna uma função para representar o operador -.
Multiply	Func<double, double, double>	Retorna uma função para representar o operador *.
Divide	Func<double, double, double>	Retorna uma função para representar o operador /.
Add3	Func<double, double, double, double>	Retorna uma função para representar o operador + entre três valores (valor1 + valor2 + valor3).
Subtract3	Func<double, double, double, double>	Retorna uma função para representar o operador - entre três valores (valor1 - valor2 - valor3).
Multiply3	Func<double, double, double, double>	Retorna uma função para representar o operador * entre três valores (valor1 * valor2 * valor3).
Divide3	Func<double, double, double, double>	Retorna uma função para representar o operador / entre três valores (valor1 / valor2 / valor3).

Métodos

Nome	Parâmetros	Retorno	Descrição
AddWith	doublevalue	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Add* com o parâmetro informado.
SubtractWith	double value	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Subtract* com o parâmetro informado.
MultiplyWith	double value	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Multiply* com o parâmetro informado.
DivideWith	double value	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Divide* com o parâmetro informado.
Add3With	double value	Func<double, double, double>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com o parâmetro informado.
Add3With	double value double value2	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Add3* com os dois parâmetros informados.
Subtract3With	double value	Func<double, double, double>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com o parâmetro informado.
Subtract3With	double value double value2	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Subtract3* com os dois parâmetros informados.
Multiply3With	double value	Func<double, double, double>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com o parâmetro informado.
Multiply3With	double value double value2	Func<double, double>	Retorna uma função parcialmente aplicada à função retornada por *Multiply3* com os dois parâmetros informados.
Divide3With	double value	Func<double, double, double>	Retorna uma função parcialmente aplicada à função retornada por *Divide3* com o parâmetro informado.
	double value		Retorna uma função parcialmente

Divide3With	double value2	double>	*Divide3* com os dois parâmetros informados.

Como Usar

Como as propriedades retornam delegates para representar as operações, é possível utilizá-las como métodos.

Operação Add

```
double value = 10;
double value2 = 5;
double result = DoubleOperations.Add(value, value2);

//result= 15
```

Operação Subtract

```
double value = 10;
double value2 = 5;
double result = DoubleOperations.Subtract(value, value2);

//result= 5
```

Operação Multiply

```
double value = 10;
double value2 = 5;
double result = DoubleOperations.Multiply(value, value2);

//result= 50
```

Operação Divide

```
double value = 10;
double value2 = 5;
double result = DoubleOperations.Divide(value, value2);

//result= 2
```

Para os métodos temos uma sintaxe um pouco diferente, isso porque é realizada uma aplicação parcial ao método retornado pela própriedade.

Por conta disso, precisamos executar o método com os primeiros parâmetros, para obtermos um novo método que espera os parâmetros restantes:

Operação AddWith

```
double value = 10;
double value2 = 5;
Func<double ,double> addWith= DoubleOperations.AddWith(value);
double result = addWith(value2);

//result= 15
```

Também podemos realizar a chamada de forma concatenada:

```
double value = 10;
double value2 = 5;
double result = DoubleOperations.AddWith(value)(value2);

//result= 15
```

As operações ~*With* existem para as quatro operações descritas, todas elas seguem a mesma característica do exemplo anterior.

Os métodos ~*With* também são aplicáveis as funções que utilizam três parâmetros, nestes casos você pode utilizar a aplicação parcial com um ou dois parâmetros, de acordo com a necessidade.

Operação Add3With

```
double value = 10;
double value2 = 5;
double value3 = 5
Func<double, double, double> addWith2 = DoubleOperations.Add3With(value);
double result = addWith2(value2, value3);

//result= 20
```

Também podemos realizar a chamada de forma concatenada:

```
double value = 10;
double value2 = 5;
double value3 = 5
double result = DoubleOperations.Add3With(value)(value2, value3);

//result= 20
```

E por fim, podemos informar os dois parâmetros na primeira sobrecarga, em qualquer um dos modos:

```
double value = 10;
double value2 = 5;
double value3 = 5
Func<double, double> addWith= DoubleOperations.Add3With(value, value2);
double result = addWith(value3);

//result= 20
```

```
double value = 10;
double value2 = 5;
double value3 = 5
double result = DoubleOperations.Add3With(value,value2)(value3);

//result= 20
```

Perceba que as versões ~*With* passam pelo processo de **aplicação parcial**, não de currying.

Operações com Strings

> `Tango.CommonOperations.StringOperations`

Esta classe *estática* contém métodos e propriedades com as operações comuns para trabalhar com valores do tipo `string`. Todos os membros retornam os `delegates` que realizam a operação descrita.

Propriedades

Nome	Tipo	Descrição
Concat	Func<string, string>	Retorna uma função para representar a operação de concatenação entre duas strings (string.Concat).
Concat3	Func<string, string, string>	Retorna uma função para representar a operação de concatenação entre três strings (string.Concat).

Métodos

Nome	Parâmetros	Retorno	Descrição
ConcatWith	string value	Func<string, string>	Retorna uma função parcialmente aplicada à função retornada por *Concat* com o parâmetro informado.
Concat3With	string value	Func<string, string, string>	Retorna uma função parcialmente aplicada à função retornada por *Concat3* com o parâmetro informado.
Concat3With	string value string value2	Func<string, string>	Retorna uma função parcialmente aplicada à função retornada por *Concat3* com os parâmetros informados.

Como Usar

Como as propriedades retornam delegates para representar as operações, é possível utilizá-las como métodos.

Operação Concat

```
string value = "Hello";
string value2 = " World";
string result = StringOperations.Concat(value, value2);

//result= "Hello World"
```

Operação Concat3

```
string value = "Hello";
string value2 = " my ";
string value3 = "World";
string result = StringOperations.Concat3(value, value2, value3);

//result= "Hello my World"
```

Para os métodos temos uma sintaxe um pouco diferente, isso porque é realizada uma aplicação parcial ao método retornado pela própriedade.

Por conta disso, precisamos executar o método com os primeiros parâmetros, para obtermos um novo método que espera os parâmetros restantes:

Operação ConcatWith

```
string value = "Hello";
string value2 = " World";
Func<string, string> concatWith = StringOperations.ConcatWith(value);
string result = concatWith(value2);

//result= "Hello World"
```

Também podemos realizar a chamada de forma concatenada:

```
string value = "Hello";
string value2 = " World";
string result = StringOperations.ConcatWith(value)(value2);

//result= "Hello World"
```

O método *~With* também é aplicável ao *Concat3*, nestes casos você pode utilizar a aplicação parcial com um ou dois parâmetros, de acordo com a necessidade.

Operação Concat3With

```
string value = "Hello";
string value2 = " my ";
string value3 = "World";
Func<string, string, string> concat3With= StringOperations.Concat3With(value);
string result = concat3With(value2, value3);

//result= "Hello my World"
```

Também podemos realizar a chamada de forma concatenada:

```
string value = "Hello";
string value2 = " my ";
string value3 = "World";
string result = StringOperations.Concat3With(value)(value2, value3);

//result= "Hello my World"
```

E por fim, podemos informar os dois parâmetros na primeira sobrecarga, em qualquer um dos modos:

```
string value = "Hello";
string value2 = " my ";
string value3 = "World";
Func<string, string> concat3With= StringOperations.Concat3With(value, value2);
string result = concat3With(value3);

//result= "Hello my World"
```

```
string value = "Hello";
string value2 = " my ";
string value3 = "World";
string result = StringOperations.Concat3With(value, value2)(value3);

//result= "Hello my World"
```

Tipos

> `Tango.Types`

Este é um dos namespaces mais importantes da **Tango**, nele encontram-se as implementações para atender os principais conceitos da programação funcional. Através dele você poderá utilizar os tipos que definem valores opcionais, eithers e até utilizar pipelines com o Continuation.

Nesta seção você irá encontrar os seguintes tópicos:

- Unit
- `Option<T>`
- `Either<TLeft, TRight>`
- `Continuation<TFail, TSuccess>`

Unit

> `Tango.Types.Unit`

Esta classe representa a falta de um valor, similar ao tipo `void`.

O conceito deste tipo de valor pode ser encontrado na seção Conceitos > Saindo do void para o Unit.

Como Usar

Você pode criar um valor `Unit` normalmente como toda struct.

```
Unit unit = new Unit();
```

Esta estrutura não contém nenhuma propriedade e nenhum método além das operações existentes em qualquer estrutura: `Equals`, `GetHashCode`, `GetType` e `ToString`.

Conversão entre os delegates `Action` e `Func`

Algumas funções de alta ordem solicitam um delegate do tipo `Func` por parâmetro. É possível utilizar a função de extensão `ToFunction` para que seja possível utilizar um delegate `Action` para estas funções.

Como originalmente o delegate `Action` representa funções que retornam `void` não seria possível convertê-las para `Func`. Neste ponto a **Tango** utiliza o tipo `Unit` para que a função continue não retornando nenhum valor, mas que possa ser representada pelo delegate `Func`.

O conceito dos delegates `Func` e `Action` pode ser encontrado na seção Conceitos > Delegates Func e Action.

`Option<T>`

> `Tango.Types.Option<T>`

Esta classe representa valores opcionais.

Instâncias de `Option<T>` encapsulam um valor que pode ou não existir. Este valor só pode ser acessado através dos métodos `Match` e `Match2`.

Os valores opcionais são considerados no estado de `IsNone` (não contendo nada) quando o valor armazenado nele é igual à `null` ou ao valor `default` de seu respectivo tipo.

Propriedades

Nome	Tipo	Descrição
IsSome	bool	Retorna true quando o valor contido no tipo opcional não é nulo nem igual ao seu valor default. Caso contrário retorna false. Representa o estado "contendo alguma coisa".
IsNone	bool	Retorna true quando o valor contido no tipo opcional é nulo ou igual ao seu valor default. Caso contrário retorna false. Representa o estado "não contendo nada".

Construtores

Parâmetros	Retorno	Descrição
T value	Option<T>	Inicializa uma nova instância de um valor opcional de acordo com o parâmetro recebido.

Métodos

Nome	Parâmetros	Retorno	Descrição
Some	T value	Option<T>	Inicializa uma nova instância de um valor opcional com o estado IsSome, caso o parâmetro recebido entre nas condições do estado IsNone é gerado um opcional neste estado.
None		Option<T>	Inicializa uma nova instância de um valor opcional com o estado IsNone.
			Permite uma maneira de aplicar um método à um valor opcional sem

Match	Func<T, TResult> methodWhenSome Func<TResult> methodWhenNone	TResult	necessidade de checar o estado do valor. Para isso são passados dois métodos contendo o mesmo tipo de retorno por parâmetro, um que será chamado no caso do estado IsSome e outro que será chamado no caso do estado IsNone.
Match2	Option<T2> option2 Func<T, T2, TResult> methodWhenSome Func<TResult> methodWhenNone	TResult	Permite uma maneira de aplicar um método à dois valores opcionais sem necessidade de checar os estados de cada valor. Para isso são passados dois métodos contendo o mesmo tipo de retorno por parâmetro, um que será chamado no caso dos dois valores opcionais estarem no estado IsSome e outro que será chamado caso algum dos valores esteja no estado IsNone.
Match	Action<T> actionWhenSome Action actionWhenNone	void	Permite uma maneira de aplicar um método à um valor opcional sem necessidade de checar o estado do valor. Para isso são passados dois métodos que não retornam nada (void) por parâmetro, um que será chamado no caso do estado IsSome e outro que será chamado no caso do estado IsNone.
Match2	Option<T2> option2 Action<T, T2> actionWhenSome Action actionWhenNone	void	Permite uma maneira de aplicar um método à dois valores opcionais sem necessidade de checar os estados de cada valor. Para isso são passados dois métodos que não retornam nada (void) por parâmetro, um que será chamado no caso dos dois valores opcionais estarem no estado IsSome e outro que será chamado caso algum dos valores esteja no estado IsNone.

Sobrecargas de operadores

Operador	Parâmetros	Descrição
Cast implícito	T value	Permite a realização de um cast implícito de qualquer valor T para seu respectivo valor opcional. Internamente realiza a chamada do método Some.

Como Usar

Você pode criar um valor opcional de várias formas diferentes.

Criando valores opcionais

Você pode utilizar o construtor para criar valores contendo o estado `IsSome` ou `IsNone` de acordo com o parâmetro, conforme código:

Utilizando construtores

```
Option<int> valueWithSome = new Option<int>(10); //-> IsSome
Option<int> valueWithNone = new Option<int>(0);  //-> IsNone
```

Você pode utilizar os métodos estáticos `Some` e `None` para especificar o estado desejado para o tipo opcional.

Neste caso deve-se tomar cuidado com a utilização do `Some`, pois caso o valor não atenda os requisitos, mesmo utilizando este método você irá gerar um valor com o estado `IsNone`.

Utilizando os métodos *estáticos* Some e None

```
Option<int> valueWithSome = Option<int>.Some(10); //-> IsSome
Option<int> valueWithNone = Option<int>.Some(0);  //-> IsNone
Option<int> value2WithNone = Option<int>.None();  //-> IsNone
```

Por último temos a versão mais simples de todas, através do cast implícito, com isso você não precisa se preocupar com nenhum tipo de sintaxe, basta criar o valor e a linguagem fará todo o trabalho.

Utilizando cast implícito

```
Option<int> valueWithSome = 10; //-> IsSome
Option<int> valueWithNone = 0;  //-> IsNone
```

Através do cast implícito implementado no tipo Option, você poderá gerar novos métodos em sua aplicação sem alterar nada no corpo especial da função, apenas indicando que a função retorna um valor opcional.

Veja este exemplo:

```
private Option<string> GetTextIfEven(int value)
{
  if (value % 2 == 0)
    return "Even";
  else
    return null;
}
```

Podemos reescrevê-lo de forma mais elegante:

```
private Option<string> GetTextIfEven(int value)
  => value % 2 == 0 ?
      "Even"
      : null;
```

E perceba que apenas sinalizamos que o retorno é um valor opcional. Nas instruções `return` estamos retornando uma string comum.

Neste método quando o valor informado for um número par, será retornado um valor opcional no estado `IsSome` contendo a string `"Even"`, caso seja um valor impar, será criado um valor opcional com o estado `IsNone`, desta forma não será preciso tratar o valor nulo fora deste método.

Apesar deste método funcionar corretamente, é mais indicado utilizar o método `None` ao invés de `null`, conforme exemplo:

```
private Option<string> GetTextIfEven(int value)
=> value % 2 == 0 ?
"Even"
: Option<string>.None();
```

Obtendo informação de um valor opcional

Para obter os valores armazenados em um valor opcional é necessário utilizar os métodos `Match` ou `Match2`, não há nenhuma outra forma de obter os valores.

Com isso, o programar fica impossibilitado de obter o valor armazenado em um tipo opcional sem tratar adequadamente as duas possibilidades.

Os métodos `Match` esperam dois métodos por parâmetro, estes métodos podem realizar transformações no valor opcional, ou apenas retorná-los, conforme exemplos:

Utilizando Match com parâmetros nomeados

```
Option<int> optionalValue = 10;
int value = optionalValue.Match(
        methodWhenSome: number => number,
        methodWhenNone: () => 0);
```

Perceba que o primeiro parâmetro é o método que será executado quando o estado for `IsSome`, por conta disso, este método recebe um parâmetro (number). No exemplo, o método apenas retorna este valor.

O segundo parâmetro é o método que será executado quando o estado for `IsNone`, ou seja, quando não haver valor no tipo opcional. Por conta disso, este segundo método não recebe nenhum parâmetro e precisa retornar um valor do mesmo, garantindo que o programa não irá gerar erros.

Você pode omitir os nomes dos parâmetros e utilizá-los normalmente como qualquer método C#:

Utilizando Match

```
Option<int> optionalValue = 10;
int value = optionalValue.Match(
        number => number,
        () => 0);
//value = 10
```

Além disso, você também pode aplicar algum tipo de transformação no valor no momento de obtê-lo, como por exemplo, elevá-lo ao quadrado:

```
Option<int> optionalValue = 10;
int value = optionalValue.Match(
        number => number * number,
        () => 0);
//value = 100
```

Você também pode retornar o valor que precisar para o caso do estado ser `IsNone`, nos exemplos anteriores foi utilizado o valor zero, mas não há nenhuma obrigatoriedade nisso.

Nos casos onde é necessário realizar uma comparação com dois valores opcionais de cada vez é necessário utilizar o método `Match2`, este método funciona de maneira similar aos exemplos anteriores, mas com dois valores opcionais ao invés de somente um.

Utilizando Match2

```
Option<int> optionalValue = 10;
Option<int> optionalValue2 = 15;

int value = optionalValue.Match2(
            optionalValue2,
            (number1, number2) => number1 + number2,
            () => 2);

//value = 25
```

Neste exemplo está sendo realizada a soma de dois valores opcionais diferentes, no caso, ambos com o estado `IsSome`, fazendo com que o primeiro método seja executado:

`(number1, number2) => number1 + number2`

Neste método anônimo o parâmetro `number1` contém o valor armazenado pela variável `optinalValue` (10) e o parâmetro `number2` da variável `optionalValue2` (15).

Neste exemplo a variável `value` receberá o valor 25.

Caso um dos valores optionais estivesse no estado `IsNone` o segundo método seria disparado e o valor armazenado em `value` seria 2, conforme código a seguir.

Utilizando Match2

```
Option<int> optionalValue = 10;
Option<int> optionalValue2 = Option<int>.None();

int value = optionalValue.Match2(
            optionalValue2,
            (number1, number2) => number1 + number2,
            () => 2);
// value = 2
```

Tanto para o método `Match` quanto para o método `Match2` há uma sobrecarga onde os métodos não precisam retornar nenhum tipo de valor (`void`).

O conceito dos valores opcionais pode ser encontrado na seção Conceitos > Valores Opcionais.

`Either<TLeft, TRight>`

> `Tango.Types.Either<TLeft, TRight>`

Esta classe representa valores "ou um ou outro".

Instâncias de `Either<TLeft, TRight>` encapsulam um valor que pode pertencer a um dos dois possíveis tipos: `TLeft` ou `TRight`. Este valor só pode ser acessado através dos métodos `Match` e `Match2`.

Uma utilização comum deste tipo é uma alternativa ao tipo `Option<T>` para lidar com possíveis erros, mas neste caso o estado "Não contendo nada" é substituído por um valor de um tipo diferente.

Por convenção o tipo `TLeft` representa o tipo problemático enquanto o `TRight` representa o tipo resultante no caminho feliz.

Os valores either possuem dois estados:

- Esquerda (`IsLeft`) - Quando contém um valor do tipo `TLeft` ;
- Direita (`IsRight`) - Quando contém um valor do tipo `TRight` .

Propriedades

Nome	Tipo	Descrição
IsLeft	bool	Retorna true quando o valor contido no tipo either é do tipo definido por TLeft.
IsRight	bool	Retorna true quando o valor contido no tipo either é do tipo definido por TRight.

Construtores

Parâmetros	Retorno	Descrição
TLeft left	Either<TLeft, TRight>	Inicializa uma nova instância de um valor Either com o estado IsLeft encapsulando o valor informado no parâmetro.
TRight right	Either<TLeft, TRight>	Inicializa uma nova instância de um valor Either com o estado IsRight encapsulando o valor informado no parâmetro.

Métodos

Nome	Parâmetros	Retorno	Descrição
Match	Func<TRight,TResult> methodWhenRight Func<TLeft, TResult> methodWhenLeft	TResult	Permite uma maneira de aplicar um método à um valor either sem necessidade de checar o estado do valor. Para isso são passados dois métodos por parâmetro. Estes métodos precisam retornar o mesmo tipo e cada um deles deve esperar um parâmetro diferente, dos tipos TLeft e TRight. Eles serão chamados de acordo com o estado do valor Either (IsLeft ou IsRight).
Match2	Either<TLeft2, TRight2> either2 Func<TRight, TRight2, T> methodWhenBothRight Func<TRight, TLeft2, T> methodWhenRightLeft Func<TLeft, TRight2, T> methodWhenLeftRight Func<TLeft, TLeft2, T> methodWhenBothLeft	T	Permite uma maneira de aplicar um método à dois valores either sem necessidade de checar os estados de cada valor. Para isso são passados quatro métodos contendo o mesmo tipo de retorno por parâmetro, onde cada um deles será chamado de acordo com o estado dos dois valores either.
Match	Action<TRight> methodWhenRight Action<TLeft> methodWhenLeft	Unit	Permite uma maneira de aplicar um método à um valor either sem necessidade de checar o estado do valor. Para isso são passados dois métodos por parâmetro. Estes métodos precisam não conter retorno (void) e cada um deles deve esperar um parâmetro diferente, dos tipos TLeft e TRight. Eles serão chamados de acordo com o estado do valor Either (IsLeft ou IsRight).
	Either<TLeft2, TRight2> either2 Action<TRight, TRight2> methodWhenBothRight		Permite uma maneira de aplicar um método à dois valores either sem necessidade de checar os estados de cada valor.

Match2	Action<TRight, TLeft2> methodWhenRightLeft Action<TLeft, TRight2> methodWhenLeftRight Action<TLeft, TLeft2> methodWhenBothLeft	T	Para isso são passados quatro métodos por parâmetro, onde eles precisam não retornar nenhum valor (void) e cada um será chamado de acordo com o estado dos dois valores either.

Sobrecargas de operadores

Operador	Parâmetros	Retorno	Descrição
Cast implícito	TLeft left	Either<TLeft, TRight>	Permite a criação de um valor Either através de um cast implícito de qualquer valor TLeft para seu respectivo valor Either.
Cast implícito	TRight right	Either<TLeft, TRight>	Permite a criação de um valor Either através de um cast implícito de qualquer valor TRight para seu respectivo valor Either.
Cast implícito	Either<TLeft, TRight>	Option<TLeft>	Permite a criação de um valor Option<TLeft> através de um cast implícito de um valor Either. Caso o valor Either esteja no estado IsRight será criado um valor opcional no estado IsNone.
Cast implícito	Either<TLeft, TRight>	Option<TRight>	Permite a criação de um valor Option<TRight> através de um cast implícito de um valor Either. Caso o valor Either esteja no estado IsLeft será criado um valor opcional no estado IsNone.
Cast implícito	Continuation<TLeft, TRight>	Either<TLeft, TRight>	Permite a criação de um valor Either<TLeft, TRight> através de um cast implícito de um valor Continuation. Caso o valor Continuation esteja no estado IsFail será criado um valor Either no estado IsLeft, caso contrário, no estado IsRight.

Como Usar

Você pode criar um valor either de várias formas diferentes, inclusive para atribuir mais informações à um valor `Option<T>`.

Criando valores Either

Você pode utilizar o construtor para criar valores contendo o estado `IsLeft` ou `IsRight` de acordo com o parâmetro, conforme código:

Utilizando o construtor

```
Either<bool, int> valueWithRight = new Either<bool, int>(10); //-> IsRight
Either<bool, int> valueWithLeft = new Either<bool, int>(false); //-> IsLeft
```

Devido às sobrecargas de cast implícito você não precisa se preocupar com nenhum tipo de sintaxe e nem utilizar o construtor, basta criar o valor de um dos dois tipos `TLeft` ou `TRight` declarado e a linguagem fará todo o trabalho.

Utilizando cast implícito

```
Either<bool, int> valueWithRight = 10;   //-> IsRight
Either<bool, int> valueWithLeft = false; //-> IsLeft
```

Através deste cast implícito, você poderá gerar novos métodos para retornar o tipo Either em sua aplicação sem alterar nada no corpo especial da função, apenas indicando que a função retorna um valor Either e informando duas instruções de return com tipos **diferentes**.

Veja este exemplo:

```
private Either<string, int> GetSquareIfEven(int value)
{
  if (value % 2 == 0)
    return value * value;
  else
    return "Odd";
}
```

Pode parecer estranho ter duas instruções de retorno com valores de tipos diferentes, mas na verdade ambos retornam um valor do tipo `Either`, mas estamos deixando esta responsabilidade com o C#.

Obtendo informação de um valor Either

Assim como nos valores opcionais é necessário utilizar os métodos `Match` ou `Match2` para obter a informação de um valor Either, no entanto, existe a possibilidade de extrair a informação de um valor Either realizando um cast implícito para um valor opcional.

Neste caso o valor opcional terá que identificar um dos dois tipos `TLeft` ou `TRight`. Caso o tipo identificado pelo `Option` seja o tipo referente ao estado atual do valor `Either` será gerado um valor opcional no estado `IsSome`, caso contrário será gerado no estado `IsNone`.

Com isso, é necessário tratar adequadamente todas as possibilidades para obter um valor de um `Either`.

Utilizando o cast implícito para Option

```
Either<bool, int> eitherValue = 10;
Option<int> optionValue = eitherValue;

//optionValue.IsSome = true
```

```
Either<bool, int> eitherValue = 10;
Option<bool> optionValue = eitherValue;

//optionValue.IsSome = false
//optionValue.IsNone = true
```

> **Atenção**
>
> Mesmo realizando o cast implícito para o tipo correto, podem haver casos onde o valor opcional esteja no estado IsNone.
>
> Esta situação ocorre quando o valor armazenado no tipo Either é igual ao valor `default` ou igual à `null`.

```
Either<bool, int> eitherValue = 0;
Option<int> optionValue = eitherValue;

//optionValue.IsSome = false
//optionValue.IsNone = true
```

Os métodos `Match` esperam dois métodos por parâmetro, estes métodos podem realizar transformações no valor Either, ou apenas retorná-los, conforme exemplos:

Utilizando Match com parâmetros nomeados

```
Either<bool, int> eitherValue = 10;
int value = eitherValue.Match(
        methodWhenRight: number => number,
        methodWhenLeft: boolean => 0);

//value = 10
```

O primeiro método será executado apenas se o valor either estiver no estado `IsRight`, logo, este método recebe um valor do tipo à direita por parâmetro, `int`, no exemplo.

O segundo método recebe um valor do tipo à esquerda, `bool`, mas note que ambos precisam retornar valores do mesmo tipo. Portanto, foi utilizado o valor zero (0) para seguir o fluxo da aplicação caso o Either esteja no estado `IsLeft`.

Não há necessidade de nomear os métodos, basta utilizá-los na ordem correta.

Utilizando Match

```
Either<bool, int> eitherValue = 10;
int value = eitherValue.Match(
        number => number,
        boolean => 0);

//value = 10
```

Além disso, você também pode aplicar algum tipo de transformação no valor no momento de obtê-lo, como por exemplo, elevá-lo ao quadrado.

```
Either<bool, int> eitherValue = 10;
int value = eitherValue.Match(
        number => number * number,
        boolean => 0);

//value = 100
```

Você também pode retornar o valor que precisar para o caso do estado ser `IsLeft`, nos exemplos anteriores foi utilizado o valor zero, mas não há nenhuma obrigatoriedade nisso.

O método Match também não precisa retornar nenhum dos dois tipos do valor Either.

Utilizando Match para retornar um novo valor

```
Either<bool, int> eitherValue = 10;
string value = eitherValue.Match(
        number => number.ToString(),
        boolean => boolean.ToString());

//value = "10"
```

```
Either<bool, int> eitherValue = false;
string value = eitherValue.Match(
        number => number.ToString(),
        boolean => boolean.ToString());

//value = "false"
```

Nos casos onde é necessário realizar uma comparação com dois valores Either de cada vez é necessário utilizar o método `Match2`, este método é consideravelmente mais complexo, pois você terá de lidar com todas as possíveis combinações de resultado.

Utilizando Match2 com dois Either no estado IsRight

```
Either<bool, int> either = 15;
Either<bool, int> either2 = 10;
int result =
    either.Match2(
                  either2,
                  (value1, value2) => value1 + value2,
                  (value1, value2) => value1,
                  (value1, value2) => value2,
                  (value1, value2) => 0
                );
//result = 25
```

Neste exemplo está sendo realizada a soma de dois valores Either diferentes, no caso, ambos com o estado `IsRight`, fazendo com que o primeiro método seja executado:

`(value1, value2) => value1 + value2`

Neste método anônimo o parâmetro `value1` contém o valor armazenado pela variável `either` (10) e o parâmetro `value2` da variável `either2` (15).

Neste exemplo a variável `result` receberá o valor 25.

Para este tipo de comparação existem outros três resultados possíveis dependendo do estado de cada um dos valores Either:

Utilizando Match2 com os estados IsRight e IsLeft, respectivamente

```
Either<bool, int> either = 15;
Either<bool, int> either2 = true;
int result =
    either.Match2(
                  either2,
                  (value1, value2) => value1 + value2,
                  (value1, value2) => value1, // -> Selecionado
                  (value1, value2) => value2,
                  (value1, value2) => 0
                 );
//result = 15
```

Utilizando Match2 com os estados IsLeft e IsRight, respectivamente

```
Either<bool, int> either = false;
Either<bool, int> either2 = 10;
int result =
    either.Match2(
                  either2,
                  (value1, value2) => value1 + value2,
                  (value1, value2) => value1,
                  (value1, value2) => value2, // -> Selecionado
                  (value1, value2) => 0
                 );
//result = 10
```

Utilizando Match2 com dois Either no estado IsLeft

```
Either<bool, int> either = false;
Either<bool, int> either2 = true;
int result =
    either.Match2(
                  either2,
                  (value1, value2) => value1 + value2,
                  (value1, value2) => value1,
                  (value1, value2) => value2,
                  (value1, value2) => 0 // -> Selecionado
                 );
//result = 0
```

Tanto para o método `Match` quanto para o método `Match2` há uma sobrecarga onde os métodos informados não precisam retornar nenhum tipo de valor (`void`).

O conceito dos valores "Ou um ou outro" pode ser encontrado na seção Conceitos > Valores "Ou um ou outro".

`Continuation<TFail, TSuccess>`

> `Tango.Types.Continuation<TFail, TSuccess>`

Esta classe representa valores que permitem encadeamento através de pipeline.

Instâncias de `Continuation<TFail, TSuccess>` encapsulam um valor que pode pertencer a um dos dois possíveis tipos: `TFail` ou `TSuccess`.

Neste ponto o tipo `Continuation` é muito semelhante ao tipo `Either`, mas as semelhanças acabam por aí. O propósito dos dois tipos é bastante distinto.

Enquanto o tipo `Either` encarrega-se de armazenar um dos dois valores possíveis, o tipo `Continuation`, além de também fazer isso, expõe uma série de métodos e operadores para criar um fluxo idiomático, sofisticado e poderoso para composição de seus métodos.

Devido à natureza deste tipo, não há uma demanda para compará-lo em conjunto com outro `Continuation`, por conta disso, este tipo não possui nenhuma implementação para o método `Match2`.

Ao invés disso, existem os métodos `Then` e `Catch`, que devem ser utilizados respectivamente para dar continuídade à composição quando a operação anterior foi bem sucedida e mal sucedida.

Através destes dois métodos a estrutura `Continuation` gera um pipeline entre duas ou mais funções, conectando sempre a saída de uma função como parâmetro da função seguinte.

Caso você tenha uma função f definida por `A -> B` e uma função g definida por `B -> C`, será possível criar uma nova função fg `A -> C` conectando o retorno de f como parâmetro de entrada para a função g.

Os valores `Continuation` possuem dois estados:

- Sucesso - Quando contém um valor do tipo `TSuccess`;
- Falha - Quando contém um valor do tipo `TFail`.

Propriedades

Nome	Tipo	Descrição
IsSuccess	bool	Retorna true quando o valor contido no tipo Continuation é do tipo definido por TSuccess. Representa o estado "Sucesso".
IsFail	bool	Retorna true quando o valor contido no tipo Continuation é do tipo definido por TFail. Representa o estado "Falha".

Construtores

Parâmetros	Retorno	Descrição
TSuccess success	Continuation<TFail, TSuccess>	Inicializa uma nova instância de um valor Continuation com o estado IsSuccess encapsulando o valor informado no parâmetro.
TFail fail	Continuation<TFail, TSuccess>	Inicializa uma nova instância de um valor Continuation com o estado IsFail encapsulando o valor informado no parâmetro.

Métodos

Nome	Parâmetros	Retorno	D
Return	TSuccess success	Continuation<TFail, TSuccess>	Inicializ instânc valor C com o IsSucc encaps valor ir parâme
Return	TFail fail	Continuation<TFail, TSuccess>	Inicializ instânc valor C com o encaps valor ir parâme
Match	Func<TSuccess,TResult> methodWhenSuccess Func<TFail, TResult> methodWhenFail	TSuccess	Permit maneir um mé valor C sem ne de che do valc Para is passac métodc parâme métodc retorna tipo e c deles c um par diferen TFail e Eles se

			chama acordo estado Continu ou IsSu
Match	Action<TSuccess> methodWhenSuccess Action<TFail> methodWhenFail	Unit	Permite maneir um mé valor C sem ne de che do valo Para is passad método parâme método não co (void) e deles d um par diferen TFail e Eles se chama acordo estado Either IsSucc
Then	Func<TSuccess, Continuation<TFail, TSuccess>> thenMethod	Continuation<TFail, TSuccess>	Permite maneir e pode compo através diferen Quand armaze Continu no esta IsSucc informa parâme executa utilizan armaze Continu parâme contrár apenas até um

			Catch s encont
Then	Func<TSuccess, Continuation<TFail, TNewSuccess>> thenMethod	Continuation<TFail, TNewSuccess>	Permite maneir e pode compo através diferen Quand armaze Contin no esta IsSucc informa parâme execut utilizan armaze Contin parâme contrár apenas até um Catch s encont
Then	Func<TParameter, TSuccess, Continuation<TFail, TNewSuccess>> thenMethod TParameter parameter	Continuation<TFail, TNewSuccess>	Permite maneir e pode compo que co de dife funçõe valor a em Co estiver IsSucc informa parâme execut utilizan armaze Contin valor ir parame dois pa Caso o ocorre bypass método encont
			Permite

Catch	Func<TFail, Continuation<TFail, TSuccess>> catchMethod	Continuation<TFail, TSuccess>	maneir e pode compo que co de dife funçõe valor a em Co estiver IsFail c informa parâme executa utilizan armaze Continu contrár apenas até um Then s encont
Catch	Func<TFail, Continuation<TNewFail, TSuccess>> catchMethod	Continuation<TNewFail, TSuccess>	Permite maneir e pode compo que co de dife funçõe valor a em Co estiver IsFail c informa parâme executa utilizan armaze Continu contrár apenas até um Then s encont
Finally	Action finallyMethod	Continuation<TFail, TSuccess>	Permite maneir código executa Continu TSucce indepe seu es isso, e

			duplica método Catch.
Finally	Action< Either<TFail,TSuccess> > finallyMethod	Continuation<TFail, TSuccess>	Permite maneir código execut Continu TSucce indepe seu es isso, ev duplica método Catch.
Merge	Func<TSuccess, Continuation<TNewFail,TNewSuccess>> mergeMethod	Continuation<(Option<TFail>, Option<TNewFail>), (TSuccess, TNewSuccess)>	Permite maneir diferen Continu um úni valores agrupa tuplas: Continu TNewF (TSucc TNewS

Sobrecargas de operadores

Operador	Parâmetros	Retorno	Descrição
Cast implícito	TSuccess success	Continuation<TFail, TSuccess>	Inicializa uma nova instância de um valor Continuation com o estado IsSuccess encapsulando o valor informado no parâmetro.
Cast implícito	TFail fail	Continuation<TFail, TSuccess>	Inicializa uma nova instância de um valor Continuation com o estado IsFail encapsulando o valor informado no parâmetro.
Cast			Permite a criação de um valor Option<TFail> através de um cast implícito de um valor Continuation.

implícito		Option<TFail>	Caso o valor Continuation esteja no estado IsFail será criado um valor opcional no estado IsSome.
Cast implícito	Continuation<TFail, TSuccess>	Option<TSuccess>	Permite a criação de um valor Option<TSuccess> através de um cast implícito de um valor Continuation. Caso o valor Continuation esteja no estado IsSuccess será criado um valor opcional no estado IsSome.
Cast implícito	Either<TFail, TSuccess>	Continuation<TFail, TSuccess>	Permite a criação de um valor Continuation<TRight> através de um cast implícito de um valor Either. Caso o valor Either esteja no estado IsLeft será criado um valor Continuation no estado IsFail, caso contrário, no estado IsSuccess.
Operador maior (>)	Continuation<TFail, TSuccess> Func<TSuccess, Continuation<TFail, TSuccess>> thenMethod	Continuation<TFail, TSuccess>	Executa o método Then para realizar o pipeline através do operador.
Operador maior ou igual (>=)	Continuation<TFail, TSuccess> Func<TSuccess, Continuation<TFail, TSuccess>> catchMethod	Continuation<TFail, TSuccess>	Executa o método Catch para realizar o pipeline através do operador.

> **Atenção**
>
> Por conta de obrigações da linguagem este tipo também implementa os operadores menor (`<`) e menor ou igual (`<=`). Mas caso qualquer um dos dois seja utilizado, a biblioteca irá lançar a exceção `NotSupportedException` .

Como Usar

Você pode criar um valor do tipo `Continuation` de várias formas diferentes, geralmente a criação de um `Continuation` indica que uma série de funções serão executadas a seguir.

Os modos de criação de um `Continuation` são bastante semelhantes à criação de um valor `Either` :

Criando valores Continuation

Você pode utilizar o construtor para criar valores contendo o estado `IsFail` ou `IsSuccess` de acordo com o parâmetro, conforme código:

Utilizando o construtor

```
Continuation<bool, int> valueWithRight = new Continuation<bool, int>(10); //-> IsSucce
ss
Continuation<bool, int> valueWithLeft = new Continuation<bool, int>(false); //-> IsFail
```

Você também pode utilizar o método *estático* `Return` .

Utilizando o método *estático* Return

```
Continuation<bool, int> valueWithRight = Continuation<bool, int>.Return(10); //-> IsSu
ccess
Continuation<bool, int> valueWithLeft = Continuation<bool, int>.Return(false); //-> Is
Fail
```

E como os tipos anteriores descritos nesta seção, também há as sobrecargas de cast implícito onde não é necessário se preocupar com nenhum tipo de sintaxe, basta criar o valor de um dos dois tipos `TFail` ou `TSuccess` declarado e a linguagem fará todo o trabalho.

Utilizando cast implícito

```
Continuation<bool, int> valueWithRight = 10;   //-> IsSuccess
Continuation<bool, int> valueWithLeft = false; //-> IsFail
```

Através deste cast implícito, você poderá gerar novos métodos para retornar o tipo `Continuation` em sua aplicação sem alterar nada no corpo especial da função, apenas indicando que a função retorna um valor deste tipo. Assim como no `Either` é possível informar duas instruções de return com tipos **diferentes**.

Veja este exemplo:

```
private Continuation<string, int> GetSquareIfEven(int value)
{
  if (value % 2 == 0)
    return value * value;
  else
    return "Odd";
}
```

Obtendo informação de um valor Continuation

Assim como nos valores `Either` e `Option`, a informação armazenada em um `Continuation` está encapsulada. Para obter a informação de um `Continuation` é necessário realizar um cast implícito para um valor opcional para `TFail` ou `TSuccess` ou utilizarmos o método `Match`.

Caso o tipo identificado pelo `Option` seja o tipo referente ao estado atual do valor `Continuation` será gerado um valor opcional no estado `IsSome`, caso contrário será gerado no estado `IsNone`.

Utilizando o cast implícito para Option

```
Continuation<bool, int> continuationValue = 10;
Option<int> optionValue = continuationValue;

//optionValue.IsSome = true
//optionValue.IsNone = false
```

```
Continuation<bool, int> continuationValue = 10;
Option<bool> optionValue = eitherValue;

//optionValue.IsSome = false
//optionValue.IsNone = true
```

> **Atenção**
>
> Mesmo realizando o cast implícito para o tipo correto, podem haver casos onde o valor opcional esteja no estado IsNone.
>
> Esta situação ocorre quando o valor armazenado no tipo Continuation é igual ao valor `default` ou igual à `null`.

```
Continuation<bool, int> continuationValue = 0;
Option<int> optionValue = eitherValue;

//optionValue.IsSome = false
//optionValue.IsNone = true
```

O método `Match` disponível nesta estrutura funciona da mesma maneira que os métodos `Match` disponíveis em `Option` e `Either`.

Os métodos `Match` esperam dois métodos por parâmetro, estes métodos podem realizar transformações no valor `Continuation`, ou apenas retorná-los, conforme exemplos a seguir.

Utilizando Match com parâmetros nomeados

```
Continuation<bool, int> continuationValue = 10;
int value = continuationValue.Match(
        methodWhenSuccess: success => success,
        methodWhenFail: fail => 0);

//value = 10
```

O primeiro método será executado apenas se o valor `Continuation` estiver no estado `IsSuccess`, logo, este método recebe um valor do tipo que representa sucesso por parâmetro, `int`, no exemplo.

O segundo método recebe um valor do tipo que representa uma falha, `bool`, mas note que ambos precisam retornar valores do mesmo tipo. Portanto, foi utilizado o valor zero (0) para seguir o fluxo da aplicação caso o `Continuation` esteja no estado `IsFail`.

Não há necessidade de nomear os métodos, basta utilizá-los na ordem correta.

Utilizando Match

```
Continuation<bool, int> continuationValue = 10;
int value = continuationValue.Match(
         success => success,
         fail => 0);

//value = 10
```

Além disso, você também pode aplicar algum tipo de transformação no valor no momento de obtê-lo, como por exemplo, elevá-lo ao quadrado.

```
Continuation<bool, int> continuationValue = 10;
int value = continuationValue.Match(
        success => success * success,
        fail => 0);

//value = 100
```

Você também pode retornar o valor que precisar para o caso do estado ser `IsFail`, nos exemplos anteriores foi utilizado o valor zero, mas não há nenhuma obrigatoriedade nisso.

O método Match também não precisa retornar nenhum dos dois tipos do `Continuation`.

Utilizando Match para retornar um novo valor

```
Continuation<bool, int> continuationValue = 10;
string value = continuationValue.Match(
        success => success.ToString(),
        fail    => fail.ToString());

//value = "10"
```

```
Continuation<bool, int> continuationValue = true;
string value = continuationValue.Match(
        success => success.ToString(),
        fail    => fail.ToString());

//value = "true"
```

Não há a possibilidade de comparar dois valores `Continuation` ao mesmo tempo, este tipo não implementa o método `Match2` por não fazer parte de seu contexto.

Criando Pipelines de Execução

O real valor do tipo `Continuation` está em sua capacidade de gerar métodos sofisticados e limpos através de operações em pipeline com os métodos `Then` e `Catch`.

Neste primeiro exemplo iremos apenas alterar um valor inteiro somando-o com cinco e depois com dez.

Utilizando o Then para alterar o valor

```
Continuation<bool, int> continuation = 5;
Option<int> optionResult =
    continuation.Then(value => value + 5)
                .Then(value => value + 10);

//optionResult.IsSome = true
//optionResult.Some = 20
```

O objeto `continuation` inicialmente continha o valor 5, após executarmos o primeiro método `Then` o processamento é feito e o valor armazenado (5) é passado como parâmetro para a função anônima: `value => value + 5` .

Por fim, o resultado desta operação (10) é passado como parâmetro para a segunda função anônima: `value => value + 10` , produzindo o resultado final (20).

Quando o Then não é executado

Caso o `continuation` esteja no estado `IsFail` nenhum dos métodos `Then` é executado. Ocorre apenas um bypass do valor.

```
Continuation<bool, int> continuation = true;
Option<int> optionResult =
    continuation.Then(value => value + 5)
                .Then(value => value + 10);

//optionResult.IsSome = false
//optionResult.IsNone = true
```

Neste caso, podemos obter o resultado utilizando um `Option<bool` para mapear a falha.

Utilizando cast implícito para obter a falha

```
Continuation<bool, int> continuation = true;
Option<bool> optionResult =
    continuation.Then(value => value + 5)
                .Then(value => value + 10);

//optionResult.IsSome = true
//optionResult.Some = true
```

Um ponto importante a ser ressaltado é a possibilidade de uma das funções utilizadas em pipeline retornar o tipo referente à falha. Caso isso ocorra todos os métodos `Then` após ela serão ignorados.

Quando a falha ocorre no meio do caminho

```
Continuation<bool, int> continuation = 5;
Option<int> optionResult =
    continuation.Then(value => value + 4)
                .Then(value =>
                      {
                        if( value % 2 == 0)
                            return value + 5;
                        else
                            return false;
                      })
                .Then(value => value + 10);

//optionResult.IsSome = false
```

No exemplo anterior, a falha ocorre somente no segundo `Then`, isso significa que o `Then` que o antecede executa normalmente, mas o `Then` que vem depois não será executado.

Para criar métodos que lidem com os erros, é necessário utilizar o método `Catch`.

Utilizando o Catch para tratar erros

Quando um `Continuation` está no valor `IsFail` os métodos `Then` não serão executados, eles apenas passarão o valor para o próximo método, até encontrar um método `Catch`.

```
Continuation<string, int> continuation = 5;
Option<string> optionResult =
    continuation.Then(value => value + 4)
                .Then(value =>
                      {
                        if( value % 2 == 0)
                            return value + 5;
                        else
                            return "ERROR";
                      })
                .Then(value => value + 10)
                .Catch(fail => $"{fail} catched");

//optionResult.IsSome = true
//optionResult.Some = "ERROR catched"
```

Assim como o método `Then` também é possível executar diversos métodos `Catch` de forma encadeada.

Catches encadeados

```
Continuation<string, int> continuation = 5;
Option<string> optionResult =
    continuation.Then(value => value + 4)
                .Then(value =>
                      {
                        if( value % 2 == 0)
                            return value + 5;
                        else
                            return "ERROR";
                      })
                .Then(value => value + 10)
                .Catch(fail => $"{fail} catched")
                .Catch(message => $"{message} again")
                .Catch(message => $"{message} and again");

//optionResult.IsSome = true
//optionResult.Some = "ERROR catched again and again"
```

Evite repetição de código com o Finally

Em alguns casos é necessário executar uma determinada ação independente do resultado contido no `Continuation`. Esta é a função principal do método `Finally`, funciona de maneira similar ao `Then` e ao `Catch`, mas neste caso, a função sempre é executada.

Outra diferença clara entre o `Finally` e os dois anteriores é a função recebida por parâmetro. Neste caso, você pode utilizar uma função que não recebe nenhum parâmetro.

Ela será executada e depois disso, o `Continuation` será retornado novamente para manter o fluxo contínuo.

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();

ContinuationModule.Resolve(5)
                  .Then(value => value + 4)
                  .Then(value => value + 10)
                  .Catch(fail => $"{fail} catched")
                  .Finally(() => stopwatch.Stop());
```

Apesar de não ser obrigatório, o `Finally` geralmente é a última chamada do fluxo de um `Continuation`, além disso, é comum este método causar algum efeito colateral, portanto, seja cuidadoso.

Outra particularidade deste método é o fato de que ele não é capaz de modificar o valor armazenado no `Continuation`. Na verdade, é possível **acessar** o valor armazenado usando a sobrecarga que recebe um `Action<Either<TFail, TSuccess>>` como parâmetro.

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();

ContinuationModule.Resolve<string, int>(5)
                  .Then(value => value + 4)
                  .Then(value => value + 10)
                  .Catch(fail => $"{fail} catched")
                  .Finally(values => Console.WriteLine(
                      values.Match(
                        number => number.ToString(),
                        text => text)));
```

Assim como o `Then` e o `Catch`, o método `Finally` também permite encadeamentos.

Merging two pipelines in a single one

This isn't a common operation, but sometimes you need to merge two different `Continuation` pipelines to complete your use case. In order to make it easy, you can use the `Merge` method.

The argument function of this method receive the entire `Continuation<TFail, TSuccess>` itself as argument and need to return an brand new `Continuation<TNewFail, TNewSuccess>`. The `Merge` return is a new grouped `Continuation<(TFail, TNewFail), (TSuccess, TNewSuccess)>`.

This new `Continuation` will be in Success state only when the two previous ones are also in this state, otherwise it will be in Fail state.

You can continue your pipeline, but now, you need to access the `Item` properties, since the values were grouped.

```
Continuation<bool, double> continuation2 = 28.5;
ContinuationModule.Resolve<string, int>(10)
                  .Then(value => value + 2)
                  .Merge(value => continuation2)
                  .Then(values => values.Item1 + values.Item2)
                  .Match(value => value, _ => 0);
```

Encadeando métodos com o operador de pipeline

A linguagem funcional da plataforma .NET, o F#, possui um operador para realizar pipelines, este operador é definido por: `|>` para pipe-foward e `<|` para reverse pipe ou pipe-backward.

Com eles podemos realizar operações na linguagem F# como estas:

```
let append1 string1 = string1 + ".append1"
let append2 string1 = string1 + ".append2"

let result1 = "abc" |> append1
printfn "\"abc\" |> append1 gives %A" result1

let result2 = "abc"
              |> append1
              |> append2
printfn "result2: %A" result2

[1; 2; 3]
|> List.map (fun elem -> elem * 100)
|> List.rev
|> List.iter (fun elem -> printf "%d " elem)
printfn ""
```

Infelizmente não há como criar novos operadores no C# até a versão atual. No entanto, é possível sobrescrever os operadores existentes.

Pensando nisso, realizei a sobrescrita dos operadores `>` e `>=` para funcionarem de forma similar ao pipe-foward do F#.

Ao invés de realizarem as comparações de maior e maior ou igual, os operadores atuam recebendo como parâmetro um delegate `Func` idêntico aos utilizados nos métodos `Then` e `Catch`.

Por tanto é possível realizar as operações em pipeline substituindo as chamadas ao método `Then` pelo operador `>` e as chamadas ao método `Catch` pelo operador `>=`.

Utilizando o operador `>` para Then

```
Continuation<bool, int> continuation = 5;
Option<int> optionResult =
    continuation
    > (value => value + 5)
    > (value => value + 10)
    > (value => value + 10)


//optionResult.IsSome = true
//optionResult.Some = 30
```

Utilizando os operadores `>` e `>=` para Then e Catch

```
Continuation<string, int> continuation = 5;
Option<string> optionResult =
    continuation
    >  (value => value + 4)
    >  (value =>
       {
          if( value % 2 == 0)
              return value + 5;
          else
              return "ERROR";
       })
    >  (value => value + 10)
    >= (fail => $"{fail} catched")
    >= (message => $"{message} again")
    >= (message => $"{message} and again");

//optionResult.IsSome = true
//optionResult.Some = "ERROR catched again and again"
```

> **Atenção**
>
> Há uma limitação na utilização dos operadores.
>
> 1. Não há uma versão possível do pipe-backward
> 2. É possível utilizar o operador somente nos métodos que retornam um valor do mesmo tipo que seu parâmetro, diferente dos métodos `Then` e `Catch` não há como sobrecarregar os parâmetros com *generics*.

Quando o tipo do valor é alterado durante os métodos

Os dois métodos para realizar pipelines possuem sobrecargas para alterar o tipo do valor armazenado no `Continuation`, desta forma é possível transformar o valor ao longo da execução.

Alterando o tipo do Continuation durante as execuções

```
Continuation<object, int> continuation = 10;
Option<string> optionResult =
    continuation
    .Then<bool>(value => value % 2 == 0)
    .Then<string>(value => value ? "Even" : "Odd");

//optionResult.IsSome = true
//optionResult.Some = "Even"
```

Note que para alterar o tipo não é necessário utilizar a notação de *generics* `Then<tipoDestino>`, mas você pode explicitá-lo se quiser. O mesmo ocorre com o método `Catch`.

Alterando diversas vezes os tipos

```
Continuation<string, int> continuation = 1;

Option<double> optionResult =
    continuation
    .Then<bool>(value =>
    {
        if (value % 2 == 0)
            return true;
        else
            return "Life, the Universe and Everything";
    })
    .Catch<double>(message => 42.0);

//optionResult.IsSome = true
//optionResult.Some = 42.0
```

Quando é necessário unir os resultados em um único tipo

Quando é necessário unificar as duas possibilidades de valores em um `Continuation` é sugerido utilizar o método `Match` após todas as execuções.

```
Continuation<bool, int> continuation = 5;
int integerResult =
    continuation.Then(value => value + 4)
                .Then(value =>
                      {
                        if( value % 2 == 0)
                            return value + 5;
                        else
                            return false;
                      })
                .Then(value => value + 10)
                .Match(success => success,
                       fail    => 0);

//integerResult  = 0
```

Assim como nos outros tipos você pode utilizar o `Match` para retornar qualquer tipo, desde que ambos os métodos o retornem.

Módulos

> `Tango.Modules`

Em conjunto com os tipos, este é um dos namespaces mais importantes da **Tango**. Neste namespace encontram-se classes estáticas funcionando como módulos em programação funcional para prover implementações de funções bastante populares, como `Filter`, `Map`, `Reduce`, `Fold`, `Scan` e muitos outros.

Através destes módulos você poderá tirar o máximo proveito dos tipos `Either` e `Option`, além de extender os métodos disponíveis para coleções do tipo `IEnumerable`.

Todos os métodos que utilizam uma instância de um dos objetos citados acima também são implementados como métodos de extensão no namespace `Tango.Linq`, permitindo o uso através do módulo ou do próprio objeto.

A documentação dos módulos segue um formato um pouco diferente, por se tratar apenas de métodos estáticos, cada um deles será explicado individualmente.

Nesta seção você irá encontrar os seguintes tópicos:

- Option
- Either
- Collection

`OptionModule`

> `Tango.Modules.OptionModule`
>
> `Tango.Modules.Option.Linq`

Este módulo possui as implementações para utilizar em conjunto com o tipo `Option<T>`.

Quando possível, os exemplos utilizarão o método de extensão, mas em todos os casos ele pode ser substituído pelo método do módulo.

> **Atenção**
>
> Em alguns casos a ordem dos parâmetros é alterada para o método de extensão. Isso ocorre porque os métodos presentes no módulo são pensados para aplicação parcial, enquanto os métodos de extensão são pensados para se parecerem mais com os métodos da `System.Linq`.

Métodos

Apply

Cria um novo valor opcional onde o valor encapsulado é o resultado da função `applying` sobre o valor opcional atual, quando a função e o valor estiverem no estado `IsSome`, caso contrário retorna um novo valor opcional no estado `IsNone`.

Parâmetros	Retorno
Option<Func<T, TResult>> applying Option<T> option	Option<TResult>

Como usar

Esta função é comumente utilizada para alterar um valor opcional através de uma função encapsulada em um segundo valor opcional.

Funciona como uma alternativa ao `Map` para situações onde tanto o valor quanto à função estiverem sob o contexto opcional.

É possível utilizar a função `Apply` através de duas sintaxes diferentes:

1. Utilizando `Apply<T, TResult>(function)` com a `function` pondendo ou não ser um `Option`;
2. Utilizando `Apply(optionFunction)`, dispensando a necessidade de *generics*. Mas este caso só é possível se a função estiver sob um contexto opcional.

Explicitando a função opcional

```
Func<int, int> multiply2 = value => value * 2;
Option<Func<int, int>> optionFunction = multiply2;

Option<int> optionValue = 4;
Option<int> result = optionValue.Apply(optionFunction);

//result.IsSome = true
//result.Some = 8
```

Utilizando a função fora do contexto opcional

```
Func<int, int> multiply2 = value => value * 2;

Option<int> optionValue = 4;
Option<int> result = optionValue.Apply<int,int>(multiply2);

//result.IsSome = true
//result.Some = 8
```

Quando a função estiver no estado IsNone

```
Option<Func<int, int>> optionFunction =
    Option<Func<int, int>>.None();

Option<int> optionValue = 4;
Option<int> result = optionValue.Apply(optionFunction);

//result.IsSome = false
//result.IsNone = true
```

Quando o valor estiver no estado IsNone

```
Func<int, int> multiply2 = value => value * 2;

Option<int> optionValue = Option<int>.None();
Option<int> result = optionValue.Apply<int,int>(multiply2);

//result.IsSome = false
//result.IsNone = true
```

AsEnumerable

Converte um valor opcional para um IEnumerable de tamanho 0 ou 1.

Parâmetros	Retorno
Option<T> option	IEnumerable<T>

Como usar

Caso o valor opcional esteja no estado `IsSome` é gerado um `IEnumerable<T>` contendo-o, caso contrário é gerado um `IEnumerable<T>` vazio.

Quando o valor opcional está no estado IsSome

```
Option<int> optionValue = 42;
IEnumerable<int> result = optionValue.AsEnumerable();

//result = { 42 }
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
IEnumerable<int> result = optionValue.AsEnumerable();

//result = { }
```

Bind

Cria um novo valor opcional onde o valor encapsulado é o resultado da função `binder` sobre o valor opcional atual, quando o valor estiver no estado `IsSome`, caso contrário retorna um novo valor opcional no estado `IsNone`.

Parâmetros	Retorno
Func<T, Option<TResult>> binder Option<T> option	Option<TResult>

Como usar

Esta função é comumente utilizada para alterar um valor opcional através de uma função que espera como parâmetro um valor comum e retorna um valor opcional.

Esta função é similar ao `Map`, mas neste caso a função aplicada já retorna um valor opcional.

Quando o valor opcional está no estado IsSome e a função retorna um valor no estado IsSome

```
Option<int> SquareWhenEven(int value)
{
    if(element % 2 == 0)
        return value * value;
    else
        return Option<int>.None();
}

Option<int> optionValue = 4;
Option<int> result = optionValue.Bind(SquareWhenEven);

//result.IsSome = true
//result.Some = 8
```

Quando o valor opcional está no estado IsSome e a função retorna um valor no estado IsNone

```
Option<int> SquareWhenEven(int value)
{
    if(element % 2 == 0)
        return value * value;
    else
        return Option<int>.None();
}

Option<int> optionValue = 5;
Option<int> result = optionValue.Bind(SquareWhenEven);

//result.IsSome = false
//result.IsNone = true
```

Quando o valor opcional está no estado IsNone

```
Option<int> SquareWhenEven(int value)
{
    if(element % 2 == 0)
        return value * value;
    else
        return Option<int>.None();
}

Option<int> optionValue = Option<int>.None();
Option<int> result = optionValue.Bind(SquareWhenEven);

//result.IsSome = false
//result.IsNone = true
```

Count

Avalia o valor opcional e retorna 1 para o estado `IsSome` ou 0 para o estado `IsNone`.

Parâmetros	Retorno
Option<T> option	int

Como usar

Caso o valor opcional esteja no estado `IsSome` é retornado o valor 1, caso contrário 0.

Quando o valor opcional está no estado IsSome

```
Option<double> optionValue = 42.0;
int count = optionValue.Count();

//count = 1
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
int count = optionValue.Count();

//count = 0
```

Exists

Retorna `true` se o valor opcional está no estado `IsSome` e se a função `predicate` retornar `true`, caso contrário retorna `false`.

Parâmetros	Retorno
Func<T, bool> predicate Option<T> option	bool

Como usar

Caso o valor opcional esteja no estado `IsNone` sempre será retornado o valor `false`. Caso o valor opcional esteja no estado `IsSome` a função `predicate` é aplicada sobre o valor encapsulado e seu retorno será o retorno do método.

Quando o valor opcional está no estado IsSome e a função retorna true

```
Option<int> optionValue = 4;
bool result = optionValue.Exists(value => value % 2 == 0);

//result = true
```

Quando o valor opcional está no estado IsSome e a função retorna false

```
Option<int> optionValue = 3;
bool result = optionValue.Exists(value => value % 2 == 0);

//result = false
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
bool result =
    optionValue.Exists(value => value % 2 == 0);

//result = false
```

Filter

Retorna o valor opcional com o estado `IsSome` quando o valor encapsulado está no estado `IsSome` e se a função `predicate` retornar `true` ao ser aplicada ao valor, caso contrário retorna valor opcional com o estado `IsNone`.

Parâmetros	Retorno
Func<T, bool> predicate Option<T> option	Option<T>

Como usar

Caso o valor opcional esteja no estado `IsNone` ele simplesmente será retornado novamente.

Caso o valor opcional esteja no estado `IsSome` a função `predicate` é aplicada sobre o valor encapsulado e caso a `predicate` retornar `true` o valor opcional é retornado novamente, caso contrário é retornado um valor opcional no estado `IsNone`.

Quando o valor opcional está no estado IsSome e a função retorna true

```
Option<int> optionValue = 4;
Option<int> result = optionValue.Filter(value => value % 2 == 0);

//result.IsSome = true
//result.Some = 4
```

Quando o valor opcional está no estado IsSome e a função retorna false

```
Option<int> optionValue = 3;
Option<int> result = optionValue.Filter(value => value % 2 == 0);

//result.IsSome = false
//result.IsNone = true
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
Option<int> result = optionValue.Filter(value => value % 2 == 0);

//result.IsSome = false
//result.IsNone = true
```

Fold

Cria um novo valor do tipo `TState` aplicando a função `folder` ao valor opcional e à um valor `state` informado por parâmetro. Caso o valor opcional esteja no estado `IsNone`, o valor `state` é retornado.

Parâmetros	Retorno
Func<TState, T, TState> folder TState state Option<T> option	TState

Como usar

Esta função realiza uma transformação de um `Option<T>` para um `TState` ao aplicar a função `folder`. Caso o valor opcional esteja no estado `IsNone` a função `folder` não é executada e o parâmetro `state` é retornado.

Quando o valor opcional está no estado IsSome

```
string state = "The number is: "
Option<int> optionValue = 10;
string result = optionValue.Fold(
                    state,
                    (_state, value) => string.Concat(_state, value) );

//result = "The number is: 10"
```

Quando o valor opcional está no estado IsNone

```
string state = "The number is: "
Option<int> optionValue = Option<int>.None();
string result = optionValue.Fold(
                    state,
                    (_state, value) => string.Concat(_state, value) );

//result = "The number is: "
```

Quando o valor opcional está no estado IsSome e o state também é um valor inteiro

```
int state = 30
Option<int> optionValue = 10;
int result = optionValue.Fold(
                 state,
                 (_state, value) => _state + value );

//result = 40
```

Quando o valor opcional está no estado IsNone e o state também é um valor inteiro

```
int state = 30
Option<int> optionValue = Option<int>.None();
int result = optionValue.Fold(
                 state,
                 (_state, value) => _state + value );

//result = 30
```

FoldBack

Cria um novo valor do tipo TState aplicando a função `folder` ao valor opcional e à um valor `state` informado por parâmetro. Caso o valor opcional esteja no estado `IsNone`, o valor state é retornado.

Parâmetros	Retorno
Func<T, TState, TState> folder Option<T> option TState state	TState

Como usar

Esta função realiza uma transformação de um `Option<T>` para um `TState` ao aplicar a função `folder`, semelhante ao `Fold`. A única diferença entre este método e o método `Fold` é sua ordem de parâmetros e a ordem de parâmetros de sua função `folder`.

Quando o valor opcional está no estado IsSome

```
int state = 30
Option<int> optionValue = 10;
int result = optionValue.FoldBack(
                (value, _state) => value + _state,
                state);

//result = 40
```

Iterate

Aplica uma função ao valor opcional quando ele estiver no estado `IsSome`.

Parâmetros	Retorno
Action<T> action Option<T> option	void

Como usar

Esta função é uma alternativa ao `Map` para funções que não produzem nenhum resultado. Por não retornar um valor, esta função bloqueia encadeamentos após sua execução.

Quando o valor opcional está no estado IsSome

```
Option<string> optionValue = "Hello Dev";
optionValue.Iterate(value => Console.WriteLine(value));

//"Hello Dev"
```

Map

Cria um novo valor opcional onde o valor encapsulado é o resultado da função `mapping` sobre o valor opcional atual, quando o valor estiver no estado `IsSome`, caso contrário retorna um novo valor opcional no estado `IsNone`.

Parâmetros	Retorno
Func<T, TResult> mapping Option<T> option	Option<TResult>

Como usar

Esta função é comumente utilizada para alterar um valor opcional através de uma função comum. Com o `Map` é possível aplicar uma função que espera um valor do tipo `int` à um `Option<int>`, por exemplo.

Esta função utiliza o `Match` para extrair o valor do tipo `Option<T>`, aplica a função `mapping` e encapsula o resultado em um novo valor opcional.

Esta função é similar ao `Select` do namespace `System.Linq`, mas para valores opcionais.

Quando o valor opcional está no estado IsSome

```
Option<int> optionValue = 4;
Option<int> result = optionValue.Map(value => value * 2);

//result.IsSome = true
//result.Some = 8
```

Quando o valor opcional está no estado IsSome e o resultado é de outro tipo: int -> string

```
Option<int> optionValue = 4;
Option<string> result = optionValue.Map(value => value.ToString());

//result.IsSome = true
//result.Some = "4"
```

Quando o valor opcional está no estado IsSome e é utilizada uma função nomeada

```
int SquareAndDouble(int value)
{
    return value * value * 2;
}

Option<int> optionValue = 4;
Option<int> result = optionValue.Map(SquareAndDouble);

//result.IsSome = true
//result.Some = 32
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
Option<int> result = optionValue.Map(value => value * 2);

//result.IsSome = false
//result.IsNone = true
```

OfNullable

Cria um valor opcional a partir de um valor anulável.

Parâmetros	Retorno
T? nullableValue	Option<T>

> **Atenção**
>
> Este método não está disponível na versão através de métodos de extensão.

Como usar

É gerado um `Option<T>` no estado `IsSome`. Ou `IsNone` quando o valor anulável conter `null` ou o valor padrão de seu tipo.

Quando o valor anulável é null

```
int? value = null;
Option<int> optionValue =
      OptionModule.OfNullable(value);

//optionValue.IsNone = true
```

Quando o valor anulável contém valor

```
int? value = 42;
Option<int> optionValue =
      OptionModule.OfNullable(value);

//optionValue.IsSome = true
//optionValue.Some = 42
```

ToArray

Converte um valor opcional para um array de tamanho 0 ou 1.

Parâmetros	Retorno
Option<T> option	T [ ]

Como usar

Caso o valor opcional esteja no estado `IsSome` é gerado um array contendo-o, caso contrário é gerado um array vazio.

Quando o valor opcional está no estado IsSome

```
Option<int> optionValue = 42;
int[] result = optionValue.ToArray();

//result = [| 42 |]
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
int[] result = optionValue.ToArray();

//result = [||]
```

ToList

Converte um valor opcional para uma lista de tamanho 0 ou 1.

Parâmetros	Retorno
Option<T> option	List<T>

Como usar

Caso o valor opcional esteja no estado `IsSome` é gerada uma lista contendo-o, caso contrário é gerada uma lista vazio.

Quando o valor opcional está no estado IsSome

```
Option<int> optionValue = 42;
List<int> result = optionValue.ToList();

//result = [ 42 ]
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
List<int> result = optionValue.ToList();

//result = []
```

ToNullable

Cria um valor anulável a partir de um valor opcional.

Parâmetros	Retorno
Option<T> option	T?

Como usar

É gerado um valor anulável `T?` a partir de um valor `Option<T>`. Caso o valor opcional esteja no estado `IsSome` o valor anulável receberá o valor encapsulado. Caso o estado seja `IsNone`, o valor anulável receberá `null`.

Quando o valor opcional está no estado IsSome

```
Option<int> optionValue = 42;
int? result = optionValue.ToNullable();

//result = 42
```

Quando o valor opcional está no estado IsNone

```
Option<int> optionValue = Option<int>.None();
int? result = optionValue.ToNullable();

//result = null
```

`EitherModule`

> `Tango.Modules.EitherModule`
>
> `Tango.Modules.Either.Linq`

Este módulo possui as implementações para utilizar em conjunto com o tipo `Either<TLeft, TRight>`.

Diversos métodos deste módulo possuem uma versão `Left` e `Right`, utilizadas para aplicar em apenas um dos lados. Estas versões serão menciondas no tópico da função para os dois tipos.

Quando possível, os exemplos utilizarão o método de extensão, mas em todos os casos ele pode ser substituído pelo método do módulo.

> **Atenção**
>
> Em alguns casos a ordem dos parâmetros é alterada para o método de extensão. Isso ocorre porque os métodos presentes no módulo são pensados para aplicação parcial, enquanto os métodos de extensão são pensados para se parecerem mais com os métodos da `System.Linq`.

Métodos

- Exists
- Iterate
- Fold
- FoldBack
- Map
- Swap
- ToTuple

Exists

Retorna o valor gerado pela função `predicateWhenRight` ou `predicateWhenRight` de acordo com o estado do valor `either`.

Parâmetros	Retorno
Func<TRight, bool> predicateWhenRight Func<TLeft, bool> predicateWhenLeft Either<TLeft, TRight> either	bool

Como usar

Caso o valor `Either` esteja no estado `IsLeft`, o resultado será o valor gerado pela função `predicateWhenLeft`, caso contrário será o valor gerado pela função `predicateWhenRight`.

Quando o valor Either está no estado IsRight e a função retorna true

```
Either<string, int> either = 20;
bool result =
    either.Exists(
        right => right == 20,
        left => left == "Hello World");

//result = true
```

Quando o valor Either está no estado IsLeft e a função retorna true

```
Either<string, int> either = "Hello World";
bool result =
    either.Exists(
        right => right == 20,
        left => left == "Hello World");

//result = true
```

Quando o valor Either está no estado IsRight e a função retorna false

```
Either<string, int> either = 15;
bool result = either.Exists(
right => right == 20,
left => left == "Hello World");
//result = false
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `ExistsLeft` e `ExistsRight` para obter o mesmo resultado, mas desta vez aplicando a função somente em um dos valores.

Sempre que estas funções forem aplicados para valores `Either` que não estão do mesmo tipo que a função de avaliação será retornado `false`.

ExistsRight Quando o valor Either está no estado IsRight

```
Either<string, int> either = 20;
bool result = either.ExistsRight(right => right == 20);

//result = true
```

ExistsLeft Quando o valor Either está no estado IsRight

```
Either<string, int> either = 20;
bool result = either.ExistsLeft(left => left == "Hello World");

//result = false
```

Iterate

Aplica uma função ao valor `Either<TLeft, TRight>` de acordo com eu estado.

Parâmetros	Retorno
Action<TRight> actionWhenRight Action<TRight> actionWhenLeft Either<TLeft, TRight> either	void

Como usar

Esta função é uma alternativa ao `Map` para funções que não produzem nenhum resultado. Por não retornar um valor, esta função bloqueia encadeamentos após sua execução.

Quando o valor Either está no estado IsRight

```
Either<string, int> either = 10;
either.Iterate(
    right => Console.WriteLine(right.ToString()),
    left => Console.WriteLine($"Hello {left}"));

//"10"
```

Quando o valor Either está no estado IsLeft

```
Either<string, int> either = " World";
either.Iterate(
    right => Console.WriteLine(right.ToString()),
    left => Console.WriteLine($"Hello {left}"));

//"Hello World"
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `IterateLeft` e `IterateRight` para obter o mesmo resultado, mas desta vez aplicando a função em somente um dos valores.

Sempre que estas funções forem aplicados para valores `Either` que não estão do mesmo tipo que a função de avaliação será realizada uma instrução para gerar um novo `Unit` (não produzir resultados).

IterateLeft Quando o valor Either está no estado IsRight

```
Either<string, int> either = 10;
either.IterateLeft(left => Console.WriteLine($"Hello {left}"));
//
```

IterateLeft Quando o valor Either está no estado IsLeft

```
Either<string, int> either = " World";
either.IterateLeft(
left => Console.WriteLine($"Hello {left}"));
//"Hello World"
```

Fold

Cria um novo valor do tipo `TState` aplicando a função `folder` ao valor `either` de acordo com seu estado.

Parâmetros	Retorno
Func<TState, TRight, TState> folderWhenRight Func<TState, TLeft, TState> folderWhenLeft TState state Either<TLeft, TRight> either	TState

Como usar

Esta função realiza uma transformação de um `Either<TLeft, TRight>` para um `TState` ao aplicar a função `folder` respectiva ao seu estado.

Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result = eitherValue.Fold(
    state,
    (_state, right) => right + _state,
    (_state, left)  => _state + 10);

//result = 42
```

Quando o valor Either está no estado IsLeft

```
int state = 20
Either<string, int> eitherValue = "ERROR";
int result = eitherValue.Fold(
    state,
    (_state, right) => right + _state,
    (_state, left)  => _state + 10);

//result = 30
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `FoldLeft` e `FoldRight` para obter o mesmo resultado, mas desta vez aplicando a função em somente um dos valores.

Sempre que estas funções forem aplicados para valores `Either` que não estão do mesmo tipo que a função de avaliação será retornado o valor `state` informado por parâmetro.

FoldRight Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result = eitherValue.FoldRight(
    state,
    (_state, right) => right + _state);

//result = 42
```

FoldRight Quando o valor Either está no estado IsLeft

```
int state = 20
Either<string, int> eitherValue = "ERROR";
int result = eitherValue.FoldRight(
    state,
    (_state, right) => right + _state);

//result = 20
```

FoldLeft Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result = eitherValue.FoldLeft(
    state,
    (_state, left) => _state + 10);

//result = 20
```

FoldBack

Cria um novo valor do tipo `TState` aplicando a função folder ao valor `either` de acordo com seu estado.

Parâmetros	Retorno
Func<TRight, TState, TState> folderWhenRight Func<TLeft, TState, TState> folderWhenLeft Either<TLeft, TRight> either TState state	TState

Como usar

Esta função realiza uma transformação de um `Either<TLeft, TRight>` para um TState ao aplicar a função folder, semelhante ao `Fold`. A única diferença entre este método e o método `Fold` é sua ordem de parâmetros e a ordem de parâmetros de suas funções `folderWhen`.

Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result =
    eitherValue.FoldBack(
        (right, _state) => right + _state,
        (left, _state) => _state + 10,
        state);

//result = 42
```

Quando o valor Either está no estado IsLeft

```
int state = 20
Either<string, int> eitherValue = 22;
int result =
    eitherValue.FoldBack(
        (right, _state) => right + _state,
        (left, _state) => _state + 10,
        state);

//result = 30
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `FoldBackLeft` e `FoldBackRight` para obter o mesmo resultado, mas desta vez aplicando a função em somente um dos valores.

Sempre que estas funções forem aplicados para valores `Either` que não estão do mesmo tipo que a função de avaliação será retornado o valor `state` informado por parâmetro.

FoldBackRight Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result =
    eitherValue.FoldBackRight(
        (right, _state) => right + _state,
        state);

//result = 42
```

FoldBackRight Quando o valor Either está no estado IsLeft

```
int state = 20
Either<string, int> eitherValue = "ERROR";
int result = eitherValue.FoldBackRight(
(right, _state) => right + _state,
state);
//result = 20
```

FoldBackLeft Quando o valor Either está no estado IsRight

```
int state = 20
Either<string, int> eitherValue = 22;
int result = eitherValue.FoldBackLeft(
(left, _state) => _state + 10,
state);
//result = 20
```

Map

Cria um novo valor `Either<TLeft, TRight>`, onde o valor encapsulado é o resultado da função `mapping` sobre o valor `either`, de acordo com seu estado.

Parâmetros	Retorno
Func<TRight, TRightResult> mappingWhenRight Func<TLeft, TLeftResult> mappingWhenLeft Either<TLeft, TRight> either	Either<TLeftResult, TRightResult>

Como usar

Esta função é comumente utilizada para alterar um valor `Either` através de uma função, semelhante ao `Map` dos valores opcionais.

Esta função utiliza o `Match` para extrair o valor do tipo `Either<TLeft, TRight>`, aplica a função `mappingWhenRight` ou `mappingWhenLeft` de acordo com o estado do valor.

Quando o valor Either está no estado IsRight e os tipos são transformados

```
Either<string, int> either = 10;
Either<int, bool> eitherResult =
    either.Map(
        right => right % 2 == 0,
        left => Convert.ToInt32(left));

//eitherResult.IsRight = true
//eitherResult.Right = true
```

Quando o valor Either está no estado IsLeft e os tipos são transformados

```
Either<string, int> either = "25";
Either<int, bool> eitherResult =
    either.Map(
        right => right % 2 == 0,
        left => Convert.ToInt32(left));

//eitherResult.IsLeft = true
//eitherResult.Left = 25
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `MapLeft` e `MapRight` para obter o mesmo resultado, mas desta vez aplicando a função somente em um dos valores.

Sempre que estas funções forem aplicados para valores `Either` que não estão do mesmo tipo que a função de avaliação será retornado o próprio valor `Either`.

MapRight Quando o valor Either está no estado IsRight

```
Either<string, int> either = 10;
Either<string, int> eitherResult =
    either.MapRight(right => right * 2);

//eitherResult.IsRight = true
//eitherResult.Right = 20
```

MapRight Quando o valor Either está no estado IsLeft

```
Either<string, int> either = "Hello World";
Either<string, int> eitherResult = either.MapRight(
right => right * 2);

//eitherResult.IsLeft = true
//eitherResult.Left = "Hello World"
```

Swap

Troca o tipo do `Either<TLeft, TRight>` alterando os valores `Right` e `Left`.

Parâmetros	Retorno
Either<TLeft, TRight> either	Either<TRight, TLeft>

Como usar

Este método deve ser utilizado quando for necessário alterar a ordem entre os valores TLeft e TRight.

Quando o valor Either está no estado IsRight

```
Either<string, int> either = 42;
Either<int, string> eitherResult = either.Swap();

//eitherResult.IsLeft = true
//eitherResult.Left = 42
```

Quando o valor Either está no estado IsLeft

```
Either<string, int> either = "Hello";
Either<int, string> eitherResult = either.Swap();

//eitherResult.IsRight = true
//eitherResult.Right = "Hello"
```

ToTuple

Transforma um valor `Either<TLeft, TRight>` em uma tupla de valores opcionais, onde um deles terá o estado `IsSome` e o outro `IsNone`, de acordo com o valor encapsulado no `either`.

Parâmetros	Retorno
Either<TLeft, TRight> either	(Option<TLeft> Left, Option<TRight> Right)

Como usar

Este método deve ser utilizado quando for necessário extrair as informações dos dois lados de um `Either`.

Quando o valor Either está no estado IsLeft

```
Either<string, int> either = "Hello";
(Option<string> Left, Option<int> Right) tupleResult =
either.ToTuple();

//tupleResult.Left.IsSome = true
//tupleResult.Left.Some = "Hello"

//tupleResult.Right.IsSome = false
//tupleResult.Right.IsNone = true
```

Abordagens para apenas um dos lados

Você pode utilizar os métodos `ToOptionLeft` e `ToOptionRight` para obter o mesmo resultado, mas desta vez obtendo somente um dos valores.

ToOptionLeft Quando o valor Either está no estado IsLeft

```
Either<string, int> either = "Hello";
Option<string> leftResult = either.ToOptionLeft();

//leftResult.IsSome = true
//leftResult.Some = "Hello"
```

ToOptionRight Quando o valor Either está no estado IsLeft

```
Either<string, int> either = "Hello";
Option<int> rightResult = either.ToOptionRight();
//rightResult.IsSome = false
//rightResult.IsNone = true
```

`ContinuationModule`

> `Tango.Modules.ContinuationModule`
>
> `Tango.Modules.Continuation.Linq`

Este módulo possui as implementações para utilizar em conjunto com o tipo `Continuation<TFail, TSuccess>`.

> **Atenção**
>
> O método AsContinuation existe somente como extension method, mas internamente ele executa os métodos `ContinuationModule.Resolve` ou `ContinuationModule.Reject` disponíveis no módulo.

Métodos

- AsContinuation
- Resolve
- Reject
- All

AsContinuation

Transforma um valor `TSuccess` ou `TFail` em um `Continuation<TFail, TSuccess>`.

Parâmetros	Retorno
TSuccess value	Continuation<TFail, TSuccess>
TFail value	Continuation<TFail, TSuccess>

Como Usar

Este método comumente é utilizado para iniciar um novo pipeline de `Continuation` nos estados `Success` ou `Fail`.

Quando o valor é um TSuccess

```
string result = 10.AsContinuation<string, int>()
                  .Then(value => value + 10)
                  .Catch(fail => $"Error: {fail}")
                  .Match(number => number.ToString(),
                         error => error);
//result = "20"
```

Quando o valor é um TFail

```
string result = "test".AsContinuation<string, int>()
                      .Then(value => value + 10)
                      .Catch(fail => $"Error: {fail}")
                      .Match(number => number.ToString(),
                             error => error);
//result = "Error: test"
```

Resolve

Inicializa uma nova instância de `Continuation<TFail,TSuccess>` com o parâmetro `TSuccess`.

Parâmetros	Returno
TSuccess success	Continuation<Unit, TSuccess>
TSuccess success	Continuation<TFail, TSuccess>

Como usar

Este método deve ser utilizado para iniciar um novo pipeline de `Continuation` no estado `Success`. Você pode especificar o tipo `TFail`, caso contrário ele será um `Unit` value.

Sem especificar o tipo TFail

```
Continuation<Unit, int> continuation = ContinuationModule.Resolve(10);

//continuation.IsSuccess = true
//continuation.Success = 10
```

Especificando o tipo TFail

```
Continuation<string, int> continuation =
     ContinuationModule.Resolve<string,int>(10);

//continuation.IsSuccess = true
//continuation.Success = 10
```

Reject

Inicializa uma nova instância de `Continuation<TFail,TSuccess>` com o parâmetro `TFail`.

Parâmetros	Retorno
TFail fail	Continuation<TFail, Unit>
TFail fail	Continuation<TFail, TSuccess>

Como Usar

Este método deve ser utilizado para iniciar um novo pipeline de `Continuation` no estado `Fail`. Você pode especificar o tipo `TSuccess`, caso contrário ele será um `Unit` value.

Sem especificar o tipo TSuccess

```
Continuation<string, Unit> continuation =
   ContinuationModule.Reject("error");

//continuation.IsFail = true
//continuation.Fail = "error"
```

Especificando o tipo TSuccess

```
Continuation<string, int> continuation =
   ContinuationModule.Reject<string, int>("error");

//continuation.IsFail = true
//continuation.Fail = "error"
```

All

Inicializa uma nova instância de `Continuation<(TFail1, TFail2, TFailN),(TSuccess1, TSuccess2, TSuccessN)>` de acordo com os continuations informado nos parâmetros.

O novo `Continuation<TFail, TSuccess>` criado estará no estado `Success` somente quando todos os parâmetros também estiverem neste estado. Caso contrário ele será criado no estado `Fail`.

Parâmetros	Retorno
Continuation<TFail1, TSuccess1> continuation1 Continuation<TFail2, TSuccess2> continuation2	Continuation<(Option<TFail1>, Option<TFail2>), (TSuccess1, TSuccess2)>
Continuation<TFail1, TSuccess1> continuation1 Continuation<TFail2, TSuccess2> continuation2 Continuation<TFail3, TSuccess3> continuation3	Continuation<(Option<TFail1>, Option<TFail2>, Option<TFail3>), (TSuccess1, TSuccess2, TSuccess3)>
Continuation<TFail1, TSuccess1> continuation1 Continuation<TFail2, TSuccess2> continuation2 Continuation<TFail3, TSuccess3> continuation3 Continuation<TFail4, TSuccess4> continuation4	Continuation<(Option<TFail1>, Option<TFail2>, Option<TFail3>, Option<TFail4>), (TSuccess1, TSuccess2, TSuccess3,TSuccess4)>

Como usar

Este método deve ser utilizado para unir diferentes valores `Continuation` em um único pipeline. É possível unir até quatro pipelines diferentes e todos os resultados serão agrupados em tuplas.

Unindo dois Continuations com sucesso

```
var continuation1 =
    ContinuationModule.Resolve<string, int>(10);

var continuation2 =
    ContinuationModule.Resolve<float, string>("number");

var result = ContinuationModule
                .All(continuation1, continuation2)
                .Then(values => $"{values.Item2} is {values.Item1}")

//result.IsSuccess = true
//result.Success = "number is 10"
```

Unindo Continuations quando um deles está no estado `Fail`

```
var continuation1 =
    ContinuationModule.Resolve<string, int>(10);

var continuation2 =
    ContinuationModule.Resolve<float, string>("number");

var continuation3 =
    ContinuationModule.Reject<string, bool>("error");



var result = ContinuationModule
                .All(continuation1, continuation2)
                .Then(values => $"{values.Item2} is {values.Item1}")

//result.IsFail= true
//result.Fail= (
//    Option.IsNone = true,
//    Option.IsNone = true,
//    Option.IsSome = true | Option.Some = "error"
//)
```

`CollectionModule`

> `Tango.Modules.CollectionModule`
>
> `Tango.Modules.Collection.Linq`

Este módulo possui as implementações para utilizar em conjunto com o tipo `IEnumerable<T>`.

Quando possível, os exemplos utilizarão o método de extensão, mas em todos os casos ele pode ser substituído pelo método do módulo.

> **Atenção**
>
> 1. O tipo `IEnumerable<T>` funciona sob o conceito de *lazy load*, este módulo respeita este comportamento, mas em alguns métodos é necessário realizar a avaliação do `IEnumerable`. Os métodos que realizam esta operação são sinalizados.
> 2. Nos métodos que envolvem duas ou mais coleções a função será aplicada somente até a quantidade de elementos da **menor** coleção.
> 3. Em alguns casos a ordem dos parâmetros é alterada para o método de extensão. Isso ocorre porque os métodos presentes no módulo são pensados para aplicação parcial, enquanto os métodos de extensão são pensados para se parecerem mais com os métodos da `System.Linq`.

Métodos

- Append
- Choose
- ChunkBySize
- Collect
- CompareWith
- CountBy
- Concat
- Distinct
- Empty
- Exists
- Exists2
- Filter
- FindIndex

- Fold
- Fold2
- FoldBack
- FoldBack2
- ForAll
- ForAll2
- ForAll3
- Head
- HeadAndTailEnd
- Range
- Generate
- Initialize
- Iterate
- Iterate2
- IterateIndexed
- IterateIndexed2
- Map
- Map2
- Map3
- MapIndexed
- MapIndexed2
- MapIndexed3
- Partition
- Permute
- Pick
- Reduce
- ReduceBack
- Replicate
- Scan
- Scan2
- ScanBack
- ScanBack2
- Tail
- TryFind
- Unzip
- Unzip3
- Zip
- Zip3

Append

Retorna uma nova coleção contendo os elementos de `source1` e `source2`.

Parâmetros	Retorno
IEnumerable<T> source1 IEnumerable<T> source2	IEnumerable<T>

Como usar

É gerada uma nova coleção unindo todos os elementos de `source2` em `source1` e mantendo a ordem dos elementos.

Unindo duas coleções

```
//IEnumerable<int> first = { 1, 2, 3, 4, 5 }
//IEnumerable<int> second = { 6, 7, 8, 9, 10 }

IEnumerable<int> result = first.Append(second)

//result = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

```
//IEnumerable<int> first = { 6, 7, 8, 9, 10 }
//IEnumerable<int> second = { 1, 2, 3, 4, 5 }

IEnumerable<int> result = first.Append(second)

//result = { 6, 7, 8, 9, 10, 1, 2, 3, 4, 5 }
```

Choose

É gerada uma nova coleção aplicando a função `chooser` em cada um dos elementos. Os elementos cuja função `chooser` retorne um valor opcional o estado `IsNone` são filtrados.

Esta função atua de forma semelhante à um `Map` e `Filter` em conjunto.

Parâmetros	Retorno
Func<T, Option<TResult>> chooser IEnumerable<T> source	IEnumerable<TResult>

Como usar

Escolhendo valores pares em uma coleção com uma função nomeada

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Option<string> NumberToStringWhenEven(int value)
{
    if(value % 2 == 0)
        return value.ToString();
    else
        return Option<string>.None();
}

IEnumerable<int> result = source.Choose(NumberToStringWhenEven);

//result = { "2", "4", "6", "8", "10" }
```

Escolhendo o quadrado dos valores ímpares em uma coleção através de uma função anônima

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

IEnumerable<int> result =
    source.Choose(value =>
            {
                if(value % 2 == 1)
                    return value * value;
                else
                    return Option<int>.None();
            });

//result = { 1, 9, 25, 49, 81 }
```

ChunkBySize

Divide uma coleção em vários pedaços de acordo com os parâmetros.

Parâmetros	Retorno
int chunkSize IEnumerable<T> source	IEnumerable<IEnumerable<T>>

Exceções

Tipo	Situação
ArgumentException	Quando o parâmetro chunkSize não for um valor positivo.

Como usar

É gerada uma nova coleção onde cada elemento também é um pedaço da coleção com o tamanho definido por `chunkSize`. Nos casos onde a coleção não for divisível pelo `chunkSize` será gerado um último pedaço com os elementos restantes.

Dividindo uma coleção em pedaços iguais

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
IEnumerable<IEnumerable<int>> result = source.ChunkBySize(2);

//result = { {1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10} }
```

Dividindo uma coleção em pedaços de tamanho não divisível pela quantidade de elementos

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
IEnumerable<IEnumerable<int>> result = source.ChunkBySize(3);

//result = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},  {10} }
```

Collect

Transforma cada elemento da coleção em um novo `IEnumerable<TResult>`, depois disso, todos os resultados são unidos em uma única coleção.

Parâmetros	Retorno
Func<T, IEnumerable<TResult>> mapping IEnumerable<T> source	IEnumerable<TResult>

Como usar

É gerada uma nova coleção aplicando a função `mapping` em cada um dos elementos. Depois disso, todas as coleções resultantes são unidas.

Esta função atua de forma semelhante à um `SelectMany`.

Gerando uma nova coleção

```
//IEnumerable<int> source = { 1, 2, 3 }

IEnumerable<int> GenerateNumbers(int value)
{
    for(int index = 1; index <= 3; index++)
        yield return value * 10;
}

IEnumerable<int> result = source.Collect(GenerateNumbers);

//result = = { 10, 20, 30, 20, 40, 60, 30, 60, 90 }
```

CompareWith

Compara duas coleções utilizando uma função para compará-las.

Esta função retorna o primeiro resultado diferente de zero da comparação entre os elementos. A função retorna o resultado de `comparer` sobre os dois elementos.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Func<T, T, int > comparer IEnumerable<T> source1 IEnumerable<T> source2	int

Como usar

A função `comparer` deve receber um elemento de cada coleção e retornar um valor inteiro em relação à comparação dos dois elementos.

Esta função foi inspirada na função `compareWith` disponível no módulo List do F#.

Comparando duas coleções

```
//IEnumerable<int> source = { 1, 2, 4 }
//IEnumerable<int> source2 = { 1, 2, 3 }

int result = source.CompareWith(source2,
      (element1, element2) => element1 > element2 ? 1
                            : element1 < element2 ? -1
                            : 0 );

// result = 1
```

Comparando duas coleções obtendo a diferença

```
//IEnumerable<int> source = { 1, 42, 3 }
//IEnumerable<int> source2 = { 1, 2, 3 }

int result = source.CompareWith(source2,
      (element1, element2) => element1 - element2);

// result = 42
```

Note que no exemplo anterior, o resultado é `42` por conta da primeira subtração retornar `0`.

CountBy

Este método aplica uma função para gerar uma chave à cada elemento e retorna uma nova coleção onde cada elemento é uma tupla entre a chave e a quantidade de elementos com esta chave na coleção.

Parâmetros	Retorno
Func<T, TKey> projection IEnumerable<T> source	IEnumerable<(TKey Key, int Count)>

Como usar

Para utilizar este método basta informar a função `projection` para projetar a chave de cada elemento.

Contando números pares e ímpares em uma coleção

```
//IEnumerable<int> source = { 1 .. 100 }

IEnumerable<(bool, int)> result = source.CountBy(value => value % 2 == 0);

// result = { (false, 50), (true, 50) }
```

Contado produtos de acordo com a faixa de valor

```
class Product {
    int Id {get; set;}
    string Name {get; set;}
    double Price {get; set;}
}

// products:
// |    Id    |    Name    |    Price    |
// |    1     |  Notebook  |     800     |
// |    2     |    Mouse   |      20     |
// |    3     |   Wallet   |      40     |
// |    4     |    Book    |      10     |
// |    5     | Smartphone |     400     |

IEnumerable<(string, Product)> result =
    products.CountBy(
        product =>
            product.Price > 200 ? "Expensive" : "Cheap")

// result = { ("Expensive", 2), ("Cheap", 3) }
```

Concat

Retorna uma nova coleção contendo os elementos de todas as coleções informadas no parâmetro.

Parâmetros	Retorno
IEnumerable<IEnumerable<T>> sources	IEnumerable<T>
params IEnumerable<T>[ ] sources	IEnumerable<T>

Como usar

As duas sobrecargas podem ser utilizadas de forma similar. A primeira exige que a coleção de coleções seja do tipo `IEnumerable` e a segundo provê parâmetros em forma de array, permitindo o envio de várias coleções separadas.

Unindo uma coleção de coleções

```
//IEnumerable<IEnumerable<int>> sources =
//    { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 } }

IEnumerable<int> result = sources.Concat()

//result = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

Unindo coleções distintas

```
//IEnumerable<int> first = { 6, 7, 8, 9, 10 }
//IEnumerable<int> second = { 1, 2, 3, 4, 5 }
//IEnumerable<int> third = { 2, 4, 6, 8, 10 }

IEnumerable<int> result = first.Concat(second, third);

//result = { 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 2, 4, 6, 8, 10 }
```

Distinct

Retorna a coleção sem elementos repetidos de acordo com os métodos `comparer` e `hashCodeGetter`.

Internamente este método utiliza o objeto `EqualityComparerBuilder<T>`.

Parâmetros	Retorno
Func<T, T, bool> comparer Func<T, int> hashCodeGetter IEnumerable<T> source	int

Como usar

Removendo elementos duplicados

```
class Product {
    int Id {get; set;}
    string Name {get; set;}
    double Price {get; set;}
}

// products:
// |    Id    |    Name    |    Price    |
// |    1     |  Notebook  |     800     |
// |    2     |    Mouse   |      20     |
// |    3     |   Wallet   |      40     |
// |    4     |    Book    |      10     |
// |    5     | Smartphone |     400     |
// |    1     |  Notebook  |     800     |
// |    1     |  Notebook  |     800     |
// |    4     |    Book    |      10     |

IEnumerable<Product> result = products.Distinct(
    (product1, product2) => product1.Id == product2 == Id,
     product => product.Id.GetHashCode()
    );

// result:
// |    Id    |    Name    |    Price    |
// |    1     |  Notebook  |     800     |
// |    2     |    Mouse   |      20     |
// |    3     |   Wallet   |      40     |
// |    4     |    Book    |      10     |
// |    5     | Smartphone |     400     |
```

Empty

Retorna uma coleção sem elementos.

> **Atenção**
>
> Este método não possui uma versão como método de extensão.

Parâmetros	Retorno
	IEnumerable<T>

Como usar

Criando uma coleção vazia

```
IEnumerable<int> result = CollectionModule.Empty<int>();
```

Exists

Testa se pelo menos um dos elementos de uma coleção satisfaz a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro elemento que satisfaça a condição informada.

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	bool

Como usar

Verificando a existência de um valor par na coleção

```
//IEnumerable<int> source = { 1, 3, 5, 7, 8, 9, 11 }
bool result = source.Exists(value => value % 2 == 0);

//result = true
```

Exists2

Testa se pelo menos um par de elementos na mesma posição nas duas coleções satisfaz a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro par de elementos que satisfaça a condição informada.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Func<T, T, bool> predicate IEnumerable<T> source1 IEnumerable<T> source2	bool

Como usar

Verificando a existência de um valor par e outro ímpar em duas coleções

```
//IEnumerable<int> first = { 1, 3, 5, 7, 9, 9, 2 }
//IEnumerable<int> second = {1, 3, 1, 9, 2, 3, 5 }

bool result = first.Exists2(
        second,
        (value1, value2) =>    value1 % 2 == 0
                            && value2 % 2 == 1);

//result = true when 2 && 5
```

Filter

Retorna uma coleção contendo cada elemento que, ao ser aplicado à função `predicate` retorne `true` .

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	IEnumerable<T>

Como usar

Filtrando para obter os valores pares em uma coleção com uma função nomeada

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

bool NumberIsEven(int value)
{
    return value % 2 == 0;
}

IEnumerable<int> result = source.Filter(NumberIsEven);

//result = { 2, 4, 6, 8, 10 }
```

Filtrando para obter os valores ímpares em uma coleção com uma função anônima

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

IEnumerable<int> result =
    source.Filter(value => value % 2 == 1);

//result = { 1, 3, 5, 7, 9 }
```

FindIndex

Testa os elementos de uma coleção até um satisfazer a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro elemento que satisfaça a condição informada, retornando o índice do elemento.

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	int

Exceções

Tipo	Situação
InvalidOperationException	Quando o predicate retorna false para todos os elementos.

Como usar

Obtendo o primeiro elemento par de uma coleção

```
//IEnumerable<int> source = { 1, 3, 5, 7, 8, 9, 10 }
bool result = source.FindIndex(value => value % 2 == 0);

//result = 4
```

Fold

Aplica a função `folder` em cada elemento da coleção, acumulando o resultado enquanto a percorre.

Este método considera o valor em `state` como valor inicial e o resultado acumulado ao longo da coleção é retornado como resultado do método.

Este método é semelhante ao `Reduce`, mas neste caso é considerado um `state` inicial.

Parâmetros	Retorno
Func<TState, T, TState> folder TState state IEnumerable<T> source	TState

Como usar

Acumulando uma quantidade em cada elemento através de uma coleção

```
//IEnumerable<Animal> source =
//    { ("Cats",4), ("Dogs",5), ("Mice",3), ("Elephants",2) }

int result = source.Fold( 6, (_state, element) => _state + element.Item2);

//result = 20
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
int result = source.Fold(15, IntegerOperations.Add);

//result = 15 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
//result = 70
```

Fold2

Aplica a função `folder` em cada um par de elementos na mesma posição nas duas coleções, acumulando o resultado enquanto a percorre.

Este método considera o valor em `state` como valor inicial e o resultado acumulado enquanto percorre as duas coleções. Este acumulador é retornado como resultado do método.

Parâmetros	Retorno
Func<TState, T, T2, TState> folder TState state IEnumerable<T> source IEnumerable<T2> source2	TState

Como usar

Acumulando o maior valor na mesma posição de duas coleções

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<int> source2 = { 3, 2, 1 }

int result =
    source.Fold2(
        source2,
        12,
        (_state, element1, element2) =>
             _state + Math.Max(element1, element2) );

//result = 20
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 2, 3, 5, 0 }
//IEnumerable<int> source2 = { 3, 2, 0, 5 }

source.Fold2(source2, 30, IntegerOperations.Add3);

//result = 50
```

FoldBack

Aplica a função `folder` em cada elemento da coleção, acumulando o resultado enquanto a percorre, neste caso, a coleção é percorrida do último índice para o primeiro.

Este método considera o valor em `state` como valor inicial e o resultado acumulado ao longo da coleção é retornado como resultado do método.

Este método é semelhante ao `ReduceBack`, mas neste caso é considerado um `state` inicial.

Parâmetros	Retorno
Func<T, TState, TState> folder IEnumerable<T> source TState state	TState

Como usar

Acumulando uma quantidade em cada elemento através de uma coleção

```
//IEnumerable<Animal> source =
//    { ("Cats",4), ("Dogs",5), ("Mice",3), ("Elephants",2) }

int result = source.FoldBack( (_state, element) => _state + element.Item2, 6);

//result = 20
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
int result = source.FoldBack(IntegerOperations.Add, 15);

//result = 15 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1
//result = 70
```

FoldBack2

Aplica a função `folder` em cada um par de elementos na mesma posição nas duas coleções, acumulando o resultado enquanto a percorre, neste caso, as coleções são percorridas do último índice para o primeiro.

Este método considera o valor em `state` como valor inicial e o resultado acumulado enquanto percorre as duas coleções. Este acumulador é retornado como resultado do método.

Parâmetros	Retorno
Func<T, T2, TState, TState> folder IEnumerable<T> source IEnumerable<T2> source2 TState state	TState

Como usar

Acumulando o maior valor na mesma posição de duas coleções

```
//IEnumerable<Animal> source = { 1, 2, 3 }
//IEnumerable<Animal> source2 = { 3, 2, 1 }

int result =
    source.FoldBack2(
        source2,
        (_state, element1, element2) =>
             _state + Math.Max(element1, element2),
        12);

//result = 20
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 2, 3, 5, 0 }
//IEnumerable<int> source2 = { 3, 2, 0, 5 }

source.FoldBack2(source2, IntegerOperations.Add3, 30);

//result = 50
```

ForAll

Testa se todos os elementos de uma coleção satisfaz a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro elemento que não satisfaça a condição informada.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	bool

Como usar

Verificando se todos os valores da coleção são números pares

```
//IEnumerable<int> source = { 2, 4, 6, 8, 10 }
bool result = source.ForAll(value => value % 2 == 0);

//result = true
```

ForAll2

Testa se cada par de elementos na mesma posição nas duas coleções satisfaz a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro par de elementos que não satisfaça a condição informada.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Func<T, T2, bool> predicate IEnumerable<T> source IEnumerable<T2> source2	bool

Como usar

Verificando se todos os valores das duas coleções são números pares

```
//IEnumerable<int> source =  { 2, 4, 6, 8, 10 }
//IEnumerable<int> source2 = { 4, 2 }

bool result =
    source.ForAll2(
        source2,
        (element1, element2) => element1 % 2 == 0
                             && element2 % 2 == 0);

//result = true
```

```
//IEnumerable<int> source =  { 2, 4, 6, 8, 10 }
//IEnumerable<int> source2 = { 4, 2, 10, 12, 16 }

bool result =
    source.ForAll2(
        source2,
        (element1, element2) => element1 % 2 == 0
                             && element2 % 2 == 0);

//result = true
```

Resultado true mesmo contendo um valor ímpar

É necessário tomar cuidado, por conta dos tamanhos diferentes enter as duas coleções, pode ocorrer este tipo de situação:

```
//IEnumerable<int> source =  { 2, 4, 6, 8, 10 }
//IEnumerable<int> source2 = { 4, 2, 10, 12, 16, 1 }

bool result =
    source.ForAll2(
        source2,
        (element1, element2) => element1 % 2 == 0
                             && element2 % 2 == 0);

//result = true
```

O método retornará `true` por conta da diferença de tamanho entre as duas coleções, com isso o último elemento de `source2` não foi avaliado.

ForAll3

Testa se cada trio de elementos na mesma posição nas três coleções satisfaz a condição definida por `predicate`.

Esta função é interrompida assim que encontrar o primeiro trio de elementos que não satisfaça a condição informada.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Func<T, T2, T3, bool> predicate IEnumerable<T> source IEnumerable<T2> source2 IEnumerable<T3> source3	bool

Como usar

Verificando se todos os valores das três coleções são iguais

```
//IEnumerable<int> source =  { 4, 2, 6, 8, 10 }
//IEnumerable<int> source2 = { 4, 2 }
//IEnumerable<int> source3 = { 4, 2 }

bool result =
    source.ForAll3(
        source2,
        source3,
        (element1, element2,element3) => element1 == element2
                                      && element2 == element3);

//result = true
```

```
//IEnumerable<int> source =  { 4, 2, 6, 8, 10 }
//IEnumerable<int> source2 = { 4, 2, 5 }
//IEnumerable<int> source3 = { 4, 2, 5 }

bool result =
    source.ForAll3(
        source2,
        source3,
        (element1, element2,element3) => element1 == element2
                                      && element2 == element3);

//result = false
```

Head

Retorna o primeiro elemento de uma coleção.

Parâmetros	Retorno
IEnumerable<T> source	T

Exceções

Tipo	Situação
InvalidOperationException	Quando a coleção está vazia.

Como usar

Obtendo o primeiro elemento de uma coleção

```
//IEnumerable<int> source = { 5, 8, 9, 10 }
int result = source.Head();

//result = 5
```

HeadAndTailEnd

Retorna o primeiro e o último elemento de uma coleção em forma de tupla.

Parâmetros	Retorno
IEnumerable<T> source	(T Head, T TailEnd)

Exceções

Tipo	Situação
InvalidOperationException	Quando a coleção está vazia.

Como usar

Obtendo o primeiro elemento e o último de uma coleção

```
//IEnumerable<int> source = { 5, 8, 9, 10 }
int result = source.HeadAndTailEnd();

//result = (5, 10)
```

Obtendo o primeiro elemento e o último de uma coleção que contém apenas um valor

```
//IEnumerable<int> source = { 42 }
int result = source.HeadAndTailEnd();

//result = (42, 42)
```

Range

Cria uma coleção de valores inteiros contendo todos os valores no intervalo fechado entre os parâmetros `first` e `second`.

> **Atenção**
>
> Este método não possui uma versão como método de extensão.

Parâmetros	Retorno
int first int second	IEnumerable<int>

Como usar

Criando uma coleção de números inteiros em ordem crescente

```
IEnumerable<int> result = CollectionModule.Range(1, 10);

//result = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

Criando uma coleção de números inteiros em ordem decrescente

```
IEnumerable<int> result = CollectionModule.Range(10, 1);

//result = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Criando uma coleção de números inteiros contendo apenas um valor

```
IEnumerable<int> result = CollectionModule.Range(1, 1);

//result = { 1 }
```

Generate

Cria uma coleção de valores de acordo com os parâmetros.

> **Atenção**
>
> Este método não possui uma versão como método de extensão.

Parâmetros	Retorno
params T[ ] values	IEnumerable<T>

Como usar

Criando uma coleção de números inteiros

```
IEnumerable<int> result = CollectionModule.Generate(1, 5, 3, 2);

//result = { 1, 5, 3, 2 }
```

Criando uma coleção de strings

```
IEnumerable<int> result =
    CollectionModule.Generate("Hello", " ", "World", " and ", "Tango");

//result = { "Helo", " ", "World", " and ", "Tango" }
```

Initialize

Cria uma coleção de valores de acordo com o resultado da função `initializer` sobre cada índice.

> **Atenção**
>
> Este método não possui uma versão como método de extensão.

Parâmetros	Retorno
int length Func<int, T> initializer	IEnumerable<T>

Como usar

Criando uma coleção de números inteiros

```
IEnumerable<int> result =
    CollectionModule.Initialize(5, index => index * 2);

//result = { 0, 2, 4, 6, 8 }
```

Iterate

Aplica a função `action` em cada elemento da coleção.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Action<T> action IEnumerable<T> source	void

Como usar

Escrevendo todos os elementos no console

```
//IEnumerable<string> source = { "Hello", " ", "Tango" }

source.Iterate( value => Console.Write(value) );

//"Hello Tango"
```

Iterate2

Aplica a função `action` em cada par de elementos das coleções.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Action<T, T2> action IEnumerable<T> source IEnumerable<T2> source2	void

Como usar

Escrevendo todos os elementos no console

```
//IEnumerable<string> source = { "Hello", " favorite" }
//IEnumerable<string> source2 = { " my", " library" }


source.Iterate2(source2,
    (element1, element2) => Console.Write($" 1:{element1}, 2:{element2}.") );

//" 1:Hello, 2: my. 1:favorite, 2: library".
```

IterateIndexed

Aplica a função `action` em cada elemento da coleção. Neste caso a função `action` além de receber o elemento, recebe seu índice.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Action<int, T> action IEnumerable<T> source	void

Como usar

Escrevendo todos os elementos no console

```
//IEnumerable<string> source = { "Hello", " ", "Tango" }

source.IterateIndexed( (index, value) => Console.Write($"{index} - {value} ") );

//"1 - Hello 2 - Tango"
```

IterateIndexed2

Aplica a função `action` em cada par de elementos da coleção. Neste caso a função `action` além de receber os elementos, recebe seu índice.

> **Atenção**
>
> Este método causa a avaliação do `IEnumerable<T>`.

Parâmetros	Retorno
Action<int, T, T2> action IEnumerable<T> source IEnumerable<T2> source2	void

Como usar

Escrevendo todos os elementos no console

```
//IEnumerable<string> source = { "Hello", " favorite" }
//IEnumerable<string> source2 = { " my", " library" }


source.IterateIndexed2(source2,
    (index, element1, element2) => Console.Write($" {index}:{element1}, {element2}.")
);

//" 1:Hello, my. 2:favorite, library".
```

Map

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada um dos elementos.

Semelhante ao `Select` de `System.Linq`.

Parâmetros	Retorno
Func<T, TResult> mapping IEnumerable<T> source	IEnumerable<TResult>

Como usar

Dobrando os valores em uma coleção

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

IEnumerable<int> result = source.Map(value => value * 2);

//result = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 }
```

Map2

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada par de elementos das duas coleções.

Parâmetros	Retorno
Func<T, T2, TResult> mapping IEnumerable<T> source IEnumerable<T2> source2	IEnumerable<TResult>

Como usar

Dobrando a soma dos valores em duas coleções

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5 }
//IEnumerable<int> source2 = { 5, 3, 3, 3, 0 }

IEnumerable<int> result =
    source.Map2(
        source2,
        (value1, value2) => (value1 + value2) * 2);

//result = { 12, 10, 12, 14, 0}
```

Map3

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada trio de elementos das três coleções.

Parâmetros	Retorno
Func<T, T2, T3, TResult> mapping IEnumerable<T> source IEnumerable<T2> source2 IEnumerable<T3> source3	IEnumerable<TResult>

Como usar

Concatenando valores de três coleções

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5 }
//IEnumerable<int> source2 = { 5, 3, 3, 3, 0 }
//IEnumerable<string> source3 = {"Value", "Value2" }

IEnumerable<string> result =
    source.Map3(
        source2,
        source3,
        (value1, value2, value3) =>
            $"{value3}: {value1 + value2}");

//result = { "Value: 6", "Value2: 5" }
```

MapIndexed

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada um dos elementos. Neste caso a função `mapping` além de receber o elemento, recebe seu índice.

Parâmetros	Retorno
Func<int, T, TResult> mapping IEnumerable<T> source	IEnumerable<TResult>

Como usar

Multiplicando os valores em uma coleção pelo valor respectivo de seu índice

```
//IEnumerable<int> source = { 1, 2, 3 }

IEnumerable<int> result =
    source.MapIndexed(
        (index, value) => value * index);

//result = { 0, 2, 6 }
```

Retornando o índice e o elemento através de uma tupla

```
//IEnumerable<int> source = { 1, 2, 3 }

IEnumerable<int> result =
    source.MapIndexed(
        (index, value) => (index, value) );

//result = { (0,1), (1,2), (2,3) }
```

MapIndexed2

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada par de elementos das duas coleções. Neste caso a função `mapping` além de receber os elementos, recebe seu índice.

Parâmetros	Retorno
Func<int, T, T2, TResult> mapping IEnumerable<T> source IEnumerable<T2> source2	IEnumerable<TResult>

Como usar

Multiplicando a soma dos valores em duas coleções por seus respectivos índices

```
//IEnumerable<int> source =  { 1, 2, 3 }
//IEnumerable<int> source2 = { 4, 5, 6 }

IEnumerable<int> result =
    source.MapIndexed2(
        source2,
        (index, value1, value2) => (value1 + value2) * index);

//result = { 0, 7, 18}
```

MapIndexed3

É gerada uma nova coleção com o resultado da função `mapping` aplicado em cada trio de elementos das três coleções. Neste caso a função `mapping` além de receber os elementos, recebe seu índice.

Parâmetros	Retorno
Func<int, T, T2, T3, TResult> mapping IEnumerable<T> source IEnumerable<T2> source2 IEnumerable<T3> source3	IEnumerable<TResult>

Como usar

Concatenando valores e índice de três coleções

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5 }
//IEnumerable<int> source2 = { 5, 3, 3, 3, 0 }
//IEnumerable<string> source3 = {"Value", "Val", "V" }

IEnumerable<string> result =
    source.Map3(
        source2,
        source3,
        (index, value1, value2, value3) =>
            $"{value3} {index}: {value1 + value2}");

//result = { "Value 0: 6", "Val 1: 5", "V 2: 6" }
```

Partition

Divide uma coleção em duas aplicando uma função `predicate`. Uma coleção contém os valores para qual a função retornou `true` e a outra para os valores `false`.

A ordem dos elementos é preservada nas duas coleções.

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	(IEnumerable<T> Trues, IEnumerable<T> Falses)

Como usar

Filtrando valores pares e ímpares

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

var (resultEvens, resultOdds) =
    source.Partition(value => value % 2 == 0)

//resultEvens = { 2, 4, 6, 8, 10 }
//resultOdds = { 1, 3, 5, 7, 9 }
```

Permute

Retorna uma nova coleção com todos os elementos permutados de acordo com a permutação especificada em `indexMap`.

Parâmetros	Retorno
Func<int, int> indexMap IEnumerable<T> source	IEnumerable<T>

Como usar

Permutando valores em uma coleção

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5 }

IEnumerable<int> result =
    source.Permute(index => (index + 1) % 5)

//result = { 5, 1, 2, 3, 4 }
```

Permutando valores em uma coleção

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5 }

IEnumerable<int> result =
    source.Permute(index => (index + 2) % 5)

//result = { 4, 5, 1, 2, 3 }
```

Pick

Aplica a função `chooser` em cada um dos elementos, a função interrompe assim que um resultado for um opcional no estado `IsSome`. Depois disso, este valor é retirado do contexto opcional e retornado como resultado final da operação.

Semelhante ao `Choose`, mas retornando apenas o primeiro elemento.

Parâmetros	Retorno
Func<T, Option<T2>> chooser IEnumerable<T> source	T2

Exceções

Tipo	Situação
InvalidOperationException	Quando a função Chooser não retornar nenhum elemento no estado IsSome.

Como usar

Obtendo o dobro do primeiro valor ímpar em uma coleção através de uma função anônima

```
//IEnumerable<int> source = { 2, 2, 4, 4, 6, 6, 7, 8, 9 }

int result = source.Pick(value =>
             {
                 if(value % 2 == 1)
                     return value * 2;
                 else
                     return Option<int>.None();
             });

//result = 14
```

Reduce

Aplica a função `reduction` em cada elemento da coleção, acumulando o resultado enquanto a percorre.

Parâmetros	Retorno
Func<T, T, T> reduction IEnumerable<T> source	TState

Como usar

Acumulando a soma dos elementos em uma coleção

```
//IEnumerable<int> source =
//    { 4, 5, 6, 7 }

int result = source.Reduce(
        (accumulator, element) => accumulator + element);

//result = 22
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
int result = source.Fold(15, IntegerOperations.Add);

//result = 15 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
//result = 70
```

ReduceBack

Aplica a função `reduction` em cada elemento da coleção, acumulando o resultado enquanto a percorre. neste caso, a coleção é percorrida do último índice para o primeiro.

Parâmetros	Retorno
Func<T, T, T> reduction IEnumerable<T> source	TState

Como usar

Acumulando uma subtração de cada elemento de uma coleção

```
//IEnumerable<int> source =
//    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

int result = source.ReduceBack(
       (accumulator, element) => element - accumulator

//result = 5
```

Replicate

Cria uma coleção de valores replicando um valor inicial.

> **Atenção**
>
> Este método não possui uma versão como método de extensão.

Parâmetros	Retorno
int count T initial	IEnumerable<T>

Como usar

Criando uma coleção de strings

```
IEnumerable<string> result =
    CollectionModule.Replicate(5, "Hello");

//result = { "Hello", "Hello", "Hello", "Hello", "Hello" }
```

Scan

Aplica a função `folder` em cada elemento da coleção, acumulando o resultado enquanto a percorre.

Este método considera o valor em `state` como valor inicial e o resultado acumulado ao longo da coleção.

Este método é semelhante ao `Fold`, mas neste caso os resultados intermediários também são retornados.

Parâmetros	Retorno
Func<TState, T, TState> folder TState state IEnumerable<T> source	TState

Como usar

Acumulando uma quantidade em cada elemento através de uma coleção

```
//IEnumerable<Animal> source =
//    { ("Cats",4), ("Dogs",5), ("Mice",3), ("Elephants",2) }

int result = source.Scan( 6, (_state, element) => _state + element.Item2);

//result = { 10, 15, 18, 20}
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
int result = source.Scan(10, IntegerOperations.Add);

//result = {11, 13, 16, 20, 25, 31, 38, 46, 55, 65}
```

Scan2

Aplica a função `folder` em cada um par de elementos na mesma posição nas duas coleções, acumulando o resultado enquanto a percorre.

Este método considera o valor em `state` como valor inicial e o resultado acumulado enquanto percorre as duas coleções.

Este método é semelhante ao `Fold2`, mas neste caso os resultados intermediários também são retornados.

Parâmetros	Retorno
Func<TState, T, T2, TState> folder TState state IEnumerable<T> source IEnumerable<T2> source2	TState

Como usar

Acumulando o maior valor na mesma posição de duas coleções

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<int> source2 = { 3, 2, 1 }

int result =
source.Scan2(
        source2,
        12,
        (_state, element1, element2) =>
        _state + Math.Max(element1, element2) );

//result = {15, 17, 20 }
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 2, 3, 5, 0 }
//IEnumerable<int> source2 = { 3, 2, 0, 5 }

source.Scan2(source2, 30, IntegerOperations.Add3);

//result = { 35, 40, 45, 50}
```

ScanBack

Aplica a função `folder` em cada elemento da coleção, acumulando o resultado enquanto a percorre, neste caso, a coleção é percorrida do último índice para o primeiro.

Este método considera o valor em `state` como valor inicial e o resultado acumulado ao longo da coleção.

Este método é semelhante ao `FoldBack`, mas neste caso os resultados intermediários também são retornados.

Parâmetros	Retorno
Func<T, TState, TState> folder IEnumerable<T> source TState state	TState

Como usar

Acumulando uma quantidade em cada elemento através de uma coleção

```
//IEnumerable<Animal> source =
//    { ("Cats",4), ("Dogs",5), ("Mice",3), ("Elephants",2) }

int result = source.FoldBack( (_state, element) => _state + element.Item2, 6);

//result = {20, 16, 11, 8}
```

Para operações comuns entre valores `int`, `decimal`, `double`, `string` e `bool` você pode utilizar as operações como `folder`.

Utilizando uma operação como folder

```
//IEnumerable<int> source = { 0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
int result = source.FoldBack(IntegerOperations.Add, 15);

//result = {65, 65, 64, 62, 59, 55, 50, 44, 37, 29, 20}
```

ScanBack2

Aplica a função `folder` em cada um par de elementos na mesma posição nas duas coleções, acumulando o resultado enquanto a percorre, neste caso, as coleções são percorridas do último índice para o primeiro.

Este método considera o valor em `state` como valor inicial e o resultado acumulado enquanto percorre as duas coleções.

Este método é semelhante ao `FoldBack2`, mas neste caso os resultados intermediários também são retornados.

Parâmetros	Retorno
Func<T, T2, TState, TState> folder IEnumerable<T> source IEnumerable<T2> source2 TState state	TState

Como usar

Acumulando o maior valor na mesma posição de duas coleções

```
//IEnumerable<int> source =  { 0, 1, 2, 3, 4, 5 }
//IEnumerable<int> source2 = { 5, 4, 3, 2, 1, 0 }

int result =
    source.ScanBack2(
        source2,
        (_state, element1, element2) =>
             _state + Math.Max(element1, element2),
        10);

//result = { 40, 35, 30, 25, 20, 15 }
```

Tail

Retorna a coleção sem o elemento `Head`.

Parâmetros	Retorno
IEnumerable<T> source	IEnumerable<T>

Como usar

Obtendo o Tail de uma coleção

```
//IEnumerable<int> source = { 5, 8, 9, 10 }
IEnumerable<int> result = source.Tail();

//result = { 8, 9, 10 }
```

TryFind

Retorna o primeiro elemento que ao ser aplicado à função `predicate` retorne `true`.

Caso não encontre nenhum elemento, é retornado um `Option<T>` no estado `IsNone`.

Parâmetros	Retorno
Func<T, bool> predicate IEnumerable<T> source	Option<T>

Como usar

Obtendo um elemento na coleção

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Option<int> result = source.TryFind(value => value == 5);

//result.IsSome = true
//result.Some = 5
```

Quando não houver elemento que satisfaça a condição

```
//IEnumerable<int> source = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Option<int> result = source.TryFind(value => value == 15);

//result.IsSome = false
//result.IsNone = true
```

TryPick

Aplica a função `chooser` em cada um dos elementos, a função interrompe assim que um resultado for um opcional no estado `IsSome`.

Semelhante ao `Pick`, mas retornando o valor ainda sob o contexto opcional e sem levantar nenhuma exceção.

Caso não encontre um valor que satisfaça a função, é retornado um valor `Option<T>` no estado `IsNone`.

Parâmetros	Retorno
Func<T, Option<T2>> chooser IEnumerable<T> source	T2

Como usar

Obtendo o dobro do primeiro valor ímpar em uma coleção através de uma função anônima

```
//IEnumerable<int> source = { 2, 2, 4, 4, 6, 6, 7, 8, 9 }

Option<int> result = source.TryPick(value =>
            {
                if(value % 2 == 1)
                    return value * 2;
                else
                    return Option<int>.None();
            });

//result.IsSome = true
//result.Some = 14
```

Quando não houver elemento que satisfaça a condição

```
//IEnumerable<int> source = { 2, 2, 4, 4, 6, 6, 6, 8, 8 }

Option<int> result = source.TryPick(value =>
            {
                if(value % 2 == 1)
                    return value * 2;
                else
                    return Option<int>.None();
            });

//result.IsSome = false
//result.IsNone = true
```

Unzip

Separa uma coleção de pares (tuplas) em uma tupla com as duas coleções.

Parâmetros	Retorno
IEnumerable<(T, T2)> source	(IEnumerable<T>, IEnumerable<T2>)

Como usar

Com coleções do mesmo tipo

```
//IEnumerable<int, int> source = { (1, 2), (3, 4) }

var (resultLeft, resultRight) = source.Unzip();

//resultLeft =  { 1, 3 }
//resultRight = { 2, 4 }
```

Com coleções de tipo diferentes

```
//IEnumerable<int, bool> source = { (1, true), (2, false)}

var (resultLeft, resultRight) = source.Unzip();

//resultLeft =  { 1, 2 }
//resultRight = { true, false }
```

Unzip3

Separa uma coleção de trios (tuplas) em uma tupla com as três coleções.

Parâmetros	Retorno
IEnumerable<(T, T2, T3)> source	(IEnumerable<T>, IEnumerable<T2>, IEnumerable<T3>)

Como usar

Com coleções do mesmo tipo

```
//IEnumerable<int, int> source = { (1, 2, 3), (4, 5, 6) }

var (resultLeft, resultCenter, resultRight) = source.Unzip3();

//resultLeft =   { 1, 4 }
//resultCenter = { 2, 5 }
//resultRight =  { 3, 6 }
```

Com coleções de tipo diferentes

```
//IEnumerable<int, bool, string> source =
//    { (1, true, "Hello"), (2, false, "World")}

var (resultLeft, resultCenter, resultRight) = source.Unzip3();

//resultLeft =  { 1, 2 }
//resultCenter = { true, false }
//resultRight = { "Hello", "World" }
```

Zip

Combina duas coleções diferentes em uma coleção de pares (tuplas).

Parâmetros	Retorno
IEnumerable<T> source IEnumerable<T2> source2	IEnumerable<(T, T2)>

Como usar

Com coleções do mesmo tipo e tamanho

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<int> source2 = { -1, -2, -3 }

IEnumerable<(int, int)> result = source.Zip(source2);

//result = { (1, -1), (2, -2), (3, -3)}
```

Com coleções de tipo diferentes

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<bool> source2 = { false, true, true }

IEnumerable<(int, bool)> result = source.Zip(source2);

//result = { (1, false), (2, true), (3, true)}
```

Com coleções de tamanhos diferentes

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<string> source2 = { "One", "Two" }

IEnumerable<(int, string)> result = source.Zip(source2);

//result = { (1, "One"), (2, "Two")}
```

Zip

Combina três coleções diferentes em uma coleção de trios (tuplas).

Parâmetros	Retorno
IEnumerable<T> source IEnumerable<T2> source2 IEnumerable<T3> source3	IEnumerable<(T, T2, T3)>

Como usar

Com coleções do mesmo tipo e tamanho

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<int> source2 = { -1, -2, -3 }
//IEnumerable<int> source3 = { 0, 1, 0 }


IEnumerable<(int, int)> result = source.Zip3(source2, source3);

//result = { (1, -1, 0), (2, -2, 1), (3, -3, 0)}
```

Com coleções de tipo diferentes

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<bool> source2 = { false, true, true }
//IEnumerable<int> source3 = { 0, 1, 0 }

IEnumerable<(int, bool, int)> result = source.Zip3(source2, source3);

//result = { (1, false, 0), (2, true, 1), (3, true, 0)}
```

Com coleções de tamanhos diferentes

```
//IEnumerable<int> source = { 1, 2, 3 }
//IEnumerable<string> source2 = { "One", "Two" }
//IEnumerable<int> source3 = { 0, 1, 0, 1 }

IEnumerable<(int, string, int)> result = source.Zip(source2);

//result = { (1, "One", 0), (2, "Two", 1)}
```

Extensões

> `Tango.Linq`

Este namespace possui implementações em diversos locais do projeto. Todas as implementações são voltadas à disponibilizar uma extensão para trabalhar em conjunto com o namespace System.Linq.

Além disso, este namespace também habilita os módulos: Option, Either e Collection como métodos de extensão ao invés de classes estáticas.

Nesta seção você irá encontrar os seguintes tópicos:

- Extensões para Enum
- Construtor de EqualityComparer
- Módulos como extensão

Extensões para Enums

> `Tango.Linq.EnumExtensions`

Esta classe contém dois métodos capazes de transformar um `enum` em uma coleção do tipo `IEnumerable<T>`, onde T é o tipo do `enum`.

Métodos

Nome	Parâmetros	Retorno	Descrição
AsEnumerable		IEnumerable<T>	Converte um enum para uma coleção do tipo IEnumerable, onde cada elemento representa um valor do enum.
AsEnumerableSkipZero		IEnumerable<T>	Converte um enum para uma coleção do tipo IEnumerable, onde cada elemento representa um valor do enum, ignorando o valor zero, comumente utilizado para opções como: Nenhum, não existente e etc.

Como Usar

Para transformar um respectivo enum do tipo `T` em um `IEnumerable<T>` basta realizar uma chamada aos métodos, informando o tipo do enum como parâmetro através do *generics*.

Veja o exemplo a seguir:

```
enum Options
{
  None = 0,
  FirstOption = 1,
  SecondOption = 2,
  ThirdOption = 3
}

IEnumerable<Options> result = EnumExtensions.AsEnumerable<Options>();

// result = [None, FirstOption, SecondOption, ThirdOption]
```

Você também pode ignorar o valor zero do enum, utilizando o método `AsEnumerableSkipZero`.

```
enum Options
{
  None = 0,
  FirstOption = 1,
  SecondOption = 2,
  ThirdOption = 3
}

IEnumerable<Options> result = EnumExtensions.AsEnumerable<Options>();

// result = [FirstOption, SecondOption, ThirdOption]
```

Construtor de EqualityComparer

> `Tango.Linq.EqualityComparerBuilder<T>`

Esta classe implementa a interface `IEqualityComparer<T>` e tem o objetivo principal prover um método de criação para objetos concretos que implementam esta interface de comparação.

Com isso é possível utilizar métodos como o `Distinct` do namespace `System.Linq` de forma dinâmica e realizando qualquer tipo de comparação, mesmo entre objetos do mesmo tipo.

Caso a interface fosse implementada diretamente na classe que será comparada, haveria apenas uma de forma de realizar uma comparação entre objetos desta classe.

Com esta implementação torna-se possível implementar comparações genéricas.

Propriedades

Nome	Tipo	Descrição
Comparer	Func<T, T, bool>	Propriedade para armazenar o método que será utilizado para realizar a comparação entre os objetos.
HashCodeGetter	Func<T, int>	Propriedade para armazenar o método que será utilizado para realizar a operação para obter o *HashCode* do objeto.

Métodos

Nome	Parâmetros	Retorno	Descrição
Create	Func<T, T, bool> comparer Func<T, int> hashCodeGetter	EqualityComparerBuilder	Cria uma nova instância deste objeto encapsulando as funções informadas no parâmetro para executar os métodos da interface *IEqualityComparer<T>*
Equals	T x T y	bool	Método da interface *IEqualityComparer<T>*
GetHashCode	T obj	int	Método da interface *IEqualityComparer<T>*

Como Usar

Para criar uma nova instância de um comparador basta utilizar o método *estático* `Create` informando as funções que devem ser utilizadas na comparação do objeto.

Veja o exemplo a seguir:

```
class Product
{
   public int Id { get; set; }
   public string Description { get; set; }
}

Func<Product, Product, bool> compareProducts =
      (product1, product2) => product1.Id == product2.Id;

Func<Product, int> getHashCodeProduct =
      product => product.Id.GetHashCode();

var comparer =
    EqualityComparerBuilder<Product>.Create(
        compareProducts,
        getHashCodeProduct);

IEnumerable<Product> products = GetProducts();
products.Distinct(comparer);
```

No exemplo anterior, o método `Distinct` não levará mais em conta a refêrencia do objeto `Product`, mas sim as comparações feitas através do objeto `comparer`.

Você também pode informar os métodos anônimos diretamente no parâmetro do método `Create`:

```
class Product
{
   public int Id { get; set; }
   public string Description { get; set; }
}
IEnumerable<Product> products = GetProducts();
products.Distinct(
    EqualityComparerBuilder<Product>.Create(
            (product1, product2) => product1.Id == product2.Id,
            product => product.Id.GetHashCode());
);
```

Módulos como extensão

> `Tango.Linq.CollectionLinqExtensions`
>
> `Tango.Linq.EitherLinqExtensions`
>
> `Tango.Linq.OptionLinqExtensions`

Os três principais módulos disponíveis na **Tango** podem ser utilizados como métodos de extensão através da importação do namespace `Tango.Linq`.

Com esta importação é possível utilizar praticamente todos os métodos disponíveis nos módulos: Option, Either e Collection.

Apenas os métodos que não operam sobre um objeto já criado que não estão disponíveis como extensão.

Você pode encontrar os métodos originais em:

- CollectionModule
- OptionModule
- EitherModule