

Fl.Functional.Utils

A set of functional-programming extension methods for C# built on top of [LanguageExt](#).

Table of Contents

- [Map](#)
 - [Do](#)
 - [Tee / TeeWhen](#)
 - [MakeOption](#)
 - [MakeEither](#)
 - [OrElse](#)
 - [ForEach](#)
 - [Combine](#)
 - [Using / UsingAsync](#)
 - [Match / MatchAsync](#)
 - [Bind / BindLeftAsync](#)
 - [Tail Recursion](#)
-

Map

Applies a function to any value, enabling fluent transformation pipelines without temporary variables.

```
// Basic transformation
int length = "hello".Map(s => s.Length); // 5

// Chaining transformations
string result = 42
    .Map(n => n * 2)
    .Map(n => $"Value is {n}"); // "Value is 84"

// Async variant
Task<int> asyncResult = "hello".MapAsync(async s =>
{
    await Task.Delay(10);
    return s.Length;
});

// Tuple variant: apply the same function to both elements
(int, int) doubled = (3, 7).SameMap(x => x * 2); // (6, 14)
```

Do

Executes a side-effecting action on a value. Unlike `Tee`, the return type is `void`.

```
// Trigger a side effect (e.g., logging) without breaking a chain
user.Do(u => logger.LogInformation("Processing user {Id}", u.Id));

// Async variant
await user.DoAsync(async u => await auditService.RecordAsync(u.Id));
```

Tee / TeeWhen

Performs a side effect and returns the **original value**, making it easy to inject logging or auditing into a pipeline.

```
// Tee: always execute the side effect
var order = CreateOrder()
    .Tee(o => logger.Log($"Order {o.Id} created"))
    .Tee(o => metrics.Increment("orders.created"));

// TeeWhen with a boolean condition
var processed = order
    .TeeWhen(o => o with { Flagged = true }, order.Amount > 1000);

// TeeWhen with a predicate on the value
var audited = order
    .TeeWhen(o => o with { Audited = true }, o => o.Amount > 500);

// TeeWhen with an Action (side effect only, returns original value)
var logged = order
    .TeeWhen(o => logger.Log("High value order"), () => order.Amount > 1000);

// Async variants
var result = await order
    .TeeWhenAsync(async o => await auditService.RecordAsync(o), o => o.Amount
    > 500);
```

MakeOption

Wraps a value in `Option<T>`, producing `None` for `null` references or when a predicate is satisfied.

```

// Null becomes None automatically
string maybeNull = null;
Option<string> opt = maybeNull.MakeOption(); // None

// Value becomes Some
Option<string> some = "hello".MakeOption(); // Some("hello")

// None when predicate matches
Option<int> opt2 = 0.MakeOption(n => n == 0); // None
Option<int> opt3 = 42.MakeOption(n => n == 0); // Some(42)

// Map and filter in one step
Option<int> length = "hello".MakeOption(s => s.Length, s => s.StartsWith("x"));
// Some(5)

// Async variant
Option<string> asyncOpt = await FetchNameAsync().MakeOptionAsync(s
=> string.IsNullOrEmpty(s));

```

MakeEither

Wraps a value in `Either<TLeft, TRight>`, choosing `Right` for success and `Left` for failure.

```

// Right when value is non-null / predicate is false
Either<string, int> right = 42.MakeEither("value was invalid"); // Right(42)

// Left when predicate holds
Either<string, int> left = (-1).MakeEither(n => n < 0, "negative number"); //
Left("negative number")

// Left value computed lazily via a factory
Either<Error, User> result = user.MakeEither(
    u => u == null,
    () => Error.New("user not found"));

// Map the right side while deciding left
Either<string, int> mapped = "42"
    .MakeEither(s => int.Parse(s), s => !int.TryParse(s, out _), "not a number");

```

OrElse

Extracts the value from an `Option<T>`, falling back to a default if it is `None`.

```
Option<string> name = GetName(); // might be None
```

```
string display = name.OrElse("Anonymous");
```

ForEach

Iterates over an `IEnumerable<T>` safely, returning `Option<Unit>` (`None` when the source is `null`).

```
IEnumerable<int> numbers = GetNumbers(); // might be null
```

```
numbers
    .ForEach(n => Console.WriteLine(n))
    .Match(
        _ => Console.WriteLine("Done"),
        () => Console.WriteLine("No items"));
```

```
// Null-safe: no NullReferenceException
```

```
((IEnumerable<int>)null).ForEach(Console.WriteLine); // None, nothing executed
```

Combine

Merges two `Action<T>` delegates. If the first action is `null`, the second is used as the fallback.

```
Action<List<string>> addItem = list => list.AddRange(new[] { "a", "b" });
Action<List<string>> addFooter = list => list.Add("end");
```

```
// Both actions run in sequence
```

```
Action<List<string>> combined = addItem.Combine(addFooter);
combined(myList); // myList = ["a", "b", "end"]
```

```
// Null-safe: when the first action is null, only the default runs
```

```
Action<List<string>> safe = ((Action<List<string>>)null).Combine(addFooter);
safe(myList); // myList = ["end"]
```

Using / UsingAsync

Safely manages `IDisposable` resources while returning a value, enabling them inside expression pipelines.

```

// Action overload - returns Unit
Unit _ = Functional.Using(
    new FileStream("data.bin", FileMode.Open),
    stream => ProcessStream(stream));

// Func overload - returns a value
byte[] bytes = Functional.Using(
    new FileStream("data.bin", FileMode.Open),
    stream => ReadAllBytes(stream));

// Chained disposables
string result = Functional.Using(
    new SqlConnection(connectionString),
    conn => new SqlCommand("SELECT ...", conn),
    (conn, cmd) => (string)cmd.ExecuteScalar());

// Async variants
string content = await Functional.UsingAsync(
    new HttpClient(),
    client => client.GetStringAsync("https://example.com"));

```

Match / MatchAsync

Pattern-matches an `Either` or `Task<Either>` into a single output value.

```

Either<string, int> result = Parse(input);

// Synchronous match
string message = result.Match(
    Right: value => $"Parsed: {value}",
    Left:  error => $"Error: {error}");

// Async match on Task<Either>
Task<Either<string, int>> asyncResult = ParseAsync(input);

string msg = await asyncResult.MatchAsync(
    onRightAsync: async value => await FormatAsync(value),
    onLeftAsync:  async error => await LogAndFormatAsync(error));

// Mixed sync/async
string msg2 = await asyncResult.MatchAsync(
    onRight: value => $"OK: {value}",
    onLeftAsync: async error => await LogAndFormatAsync(error));

```

Bind / BindLeftAsync

Chains `Either`-returning functions, handling the left (error) side asynchronously.

```
Task<Either<string, int>> step1 = ParseAsync(input);

// Remap the Left side when the Task resolves
Task<Either<ErrorCode, int>> step2 = step1.BindLeftAsync(
    errorMessage => ErrorCode.FromMessage(errorMessage));

// Or chain inside a pipeline
var pipeline = await GetUserAsync(id)
    .BindLeftAsync(msg => new NotFoundError(msg));
```

Tail Recursion

Executes recursive algorithms iteratively via a trampoline, avoiding stack overflows.

```
using Fl.Functional.Utls.Recursion;

// Compute factorial with safe tail recursion
async Task<int> Factorial(int n)
{
    return await TailRecursion.ExecuteAsync(Step(n, 1));

    Func<Task<RecursionResult<int>>> Step(int remaining, int accumulator) =>
        () => remaining <= 1
            ? RecursionResult<int>.CreateLastAsync(accumulator,
Step(0, accumulator))
            : RecursionResult<int>.CreateNextAsync(accumulator, Step(remaining - 1,
remaining * accumulator));
}

int result = await Factorial(10); // 3628800
```

Namespace Fl.Functional.Utils

Classes

[Functional](#)