Progress Telerik

Functional Programming With C# 7.1

CHEAT SHEET

Functional programming is a style that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

Immutable Types

An object whose state cannot be modified after it is created, lowering the risk of side-effects.
https://dotnetfiddle.net/K928pP

Mutable

```
public class Rectangle
{
     public int Length { get; set; }
     public int Height { get; set; }

     public void Grow(int length, int height)
     {
            Length += length;
            Height += height;
     }
}

Rectangle r = new Rectangle();
r.Length = 5;
r.Height = 10;
r.Grow(10, 10);

// r.Length is 15, r.Height is 20,
// same instance of r
```

Immutable

```
public class ImmutableRectangle
{
     int Length { get; }
     int Height { get; }

     public ImmutableRectangle(int length,
                               int height)
     {
         Length = length;
         Height = height;
     }

     public ImmutableRectangle Grow(int length,
       int height) =>
     new ImmutableRectangle(Length + length,
                            Height + height);
}

ImmutableRectangle r = new
ImmutableRectangle(5, 10);

r = r.Grow(10, 10);
// r.Length is 15, r.Height is 20,
// is a new instance of r
```

Expressions Instead of Statements

Statements define an action and are executed for their side-effect.
Expressions produce a result without mutating state.

https://dotnetfiddle.net/ozZIL3

Example

Both of the following code examples produce the same results. The expression produces a result without mutations.

Statement

```
public static string GetSalutation(int hour) {
     string salutation; // placeholder value
          if (hour < 12)
               salutation = "Good Morning";
          else
               salutation = "Good Afternoon";
     return salutation; // mutated variable
}
```

Expression

```
public static string GetSalutation(int hour) =>
 hour < 12 ? "Good Morning" : "Good Afternoon";
```

ValueTuples

Tuple is a more efficient and more productive lightweight syntax to define a data structure that carries more than one value. **Requires NuGet Package System.ValueTuple**

- Represent data without DTO classes
- Lower memory footprint than a class
- Return multiple values from methods without the need for out variables

Example

```
(double lat, double lng) GetCoordinates(string
                                        query)
{
     //DO search query ...
     return (lat: 47.6450905056185,
             lng: 122.130835641356);
}

var pos = GetCoordinates("15700 NE 39th St,
Redmond, WA");

pos.lat; //47.6450905056185
pos.lng; //122.130835641356
```

Func Delegates

Func Delegates encapsulate a method. When declaring a Func, input and output parameters are specified as T1-T16, and TResult.

https://dotnetfiddle.net/EyGLvp

- **Func<TResult> –** matches a method that takes no arguments, and returns value of type **TResult**.
- **Func<T, TResult> –** matches a method that takes an argument of type T, and returns value of type **TResult**.
- **Func<T1, T2, TResult> –** matches a method that takes arguments of type T1 and T2, and returns value of type **TResult**.
- **Func<T1, T2, ..., TResult> –** and so on up to 16 arguments, and returns value of type **TResult**.

Example

Both of the following code examples produce the same results. The expression produces a result without mutations.

```
Func<int, int> addOne = n => n +1;
Func<int, int, int> addNums = (x,y) => x + y;
Func<int, bool> isZero = n => n == 0;

Console.WriteLine(addOne(5));
// 6
Console.WriteLine(isZero(addNums(-5,5)));
// True

int[] a = {0,1,0,3,4,0};
Console.WriteLine(a.Count(isZero));
// 3
```

Higher Order Functions / Functions as Data

A function that accepts another function as a parameter, or returns another function.

https://dotnetfiddle.net/jhn5BZ

Example

method signature

```
int IEnumerable.Count<T>(Func<T, Bool>
predicate)
```

Source code for Count()

```
int count = 0;
    foreach (TSource element in source)
    {
        checked // overflow exception check
        {
              if (predicate(element))
              // func<T,Bool> invoked
            {
                count++;
            }
        }
    }
return count;
```

usage

```
bool[] bools = { false, true, false, false };

int f = bools.Count(bln => bln == false);
// out = 3
int t = bools.Count(bln => bln == true);
// out = 1
```

Method Chaining (~Pipelines)

Since C# lacks a Pipeline syntax, pipelines in C# are created with design patterns that allow for methods to chain. The result of the method chain should produce the desired value and type.

http://demos.telerik.com/aspnet-mvc/grid

Example

Both of the following code examples produce the same results. The expression produces a result without mutations.

```
string str = new StringBuilder()
     .Append("Hello ")
     .Append("World ")
     .ToString()
     .TrimEnd()
     .ToUpper();
// HELLO WORLD
```

Example, Telerik Grid HTML Helper

```
Html.Kendo()
.Grid(Model)
.Name("grid")
.Columns(columns =>
{
columns.Bound(product => product.ProductID);
columns.Bound(product => product.ProductName);
columns.Bound(product => product.UnitsInStock);
}) // Render HTML Data Grid
```

Extension Methods

Extension methods are a great way to extend method chains and add functionality to a class.

Note: Telerik UI for ASP.NET MVC's HTML Helpers are built using extension methods.

Example

```
// Extends the StringBuilder class to accept a
predicate

public static StringBuilder AppendWhen(
     this StringBuilder sb, string value,
     bool predicate) =>
          predicate ? sb.Append(value) : sb;

Usage

string htmlButton = new StringBuilder()
     .Append("<button")
     .AppendWhen(" disabled", isDisabled)
     .Append(">Click me</button>")
     .ToString();
```

Tip

Add the [*DebuggerNonUserCodeAttribute*] attribute to utility extension methods for easier debugging.

You can read more about this attribute at davefancher.com:

https://davefancher.com/2016/01/28/functional-c-debugging-method-chains/

Extension Methods

Extension methods are a great way to extend method chains and add functionality to a class.

Note: Telerik UI for ASP.NET MVC's HTML Helpers are built using extension methods.

Example

```
// Extends the StringBuilder class to accept a predicate

public static StringBuilder AppendWhen(
     this StringBuilder sb, string value,
     bool predicate) =>
          predicate ? sb.Append(value) : sb;
```

Usage

```
string htmlButton = new StringBuilder()
     .Append("<button")
     .AppendWhen(" disabled", isDisabled)
     .Append(">Click me</button>")
     .ToString();
```

Tip

Add the [*DebuggerNonUserCodeAttribute*] attribute to utility extension methods for easier debugging.

You can read more about this attribute at davefancher.com:

https://davefancher.com/2016/01/28/functional-c-debugging-method-chains/

Yield

Using **yield** to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration.

You consume an iterator method by using a foreach statement or LINQ query.

Yield is the basis for many LINQ methods.

https://dotnetfiddle.net/D4tgdG

Example

Without Yield

```
public static IEnumerable<int>
GreaterThan(int[] arr, int gt) {
     List<int> temp = new List<int>();
          foreach (int n in arr) {
               if (n > gt) temp.Add(n);
       }
     return temp;
}
```

With Yield

```
public static IEnumerable<int>
GreaterThan(int[] arr, int gt) {
     foreach (int n in arr) {
          if (n > gt) yield return n;
     }
}
```

LINQ

The gateway to functional programming in C#. LINQ makes short work of most imperative programming routines that work on arrays and collections.

Methods by Category

Quantify
All, Any, Contains

Filter
Where, OfType

Project/Transform
Select, SelectMany, Zip

Criteria/Set
Distinct, Except, Intersect, Union

Sorting
OrderBy, OrderByDecending, ThenBy, ThenByDecending, Reverse

Aggregation
Aggregate, Average, Count, LonCount, Max, Min, Sum

Partition/Join
Skip, SkipWhile, Take, TakeWhile, Join, GroupJoin

Grouping
GroupBy, ToLookup

Thread-Safe Collections

Since Functional programming promotes thread safety via immutability, these Thread-Safe Collections important to know.

The .NET Framework 4 introduces the **System.Collections.Concurrent** namespace, which includes several collection classes that are both thread-safe and scalable. Multiple threads can safely and efficiently add or remove items from these collections, without requiring additional synchronization in user code.

Thread-Safe Collections

Blocking Collection<T>
Provides bounding and blocking functionality for any type that implements IProducerConsumerCollection<T>.

IProducerConsumerCollection<T>
The interface that a type must implement to be used in a BlockingCollection.

Concurrent Queue<T>
Thread-safe implementation of a FIFO (first-in, first-out) queue.

Concurrent Dictionary<TKey, TValue>
Thread-safe implementation of a dictionary of key-value pairs.

Concurrent Stack<T>
Thread-safe implementation of a LIFO (last-in, first-out) stack.

Concurrent Bag<T>
Thread-safe implementation of an unordered collection of elements.

Resources

Functional Programming Self Guided Workshop
Functional Programming vs. Imperative Programming (C#)
Refactoring Data Grids with C# Extension Methods
Better Code with Functional Programming
Functionally Similar – Comparing Underscore.js to LINQ
Giving Clarity to LINQ Queries by Extending Expressions

This resource is brought to you by Telerik and Kendo UI.

These convenient bundles include a wide-range of UI, reporting and productivity tools for both .NET and JavaScript technologies and support that's got your back in every step of your project.

Thanks to our intuitive APIs, alongside thousands of demos with source code availability, comprehensive documentation and a full assortment of VS templates, you will get up and running with our tools in no time and fully embrace your inner warrior (kendoka/ninja).

By leveraging the broad array of themes, skins, styling and customization options, your application will awe even the best front end designers.

Learn more

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying mission-critical business applications. Progress empowers enterprises and ISVs to build and deliver cognitive-first applications that harness big data to derive business insights and competitive advantage. Progress offers leading technologies for easily building powerful user interfaces across any type of device, a reliable, scalable and secure backend platform to deploy modern applications, leading data connectivity to all sources, and award-winning predictive analytics that brings the power of machine learning to any organization. Over 1,700 independent software vendors, 80,000 enterprise customers, and 2 million developers rely on Progress to power their applications. Learn about Progress at **www.progress.com** or +1-800-477-6473.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw
For regional international office locations and contact information,
please go to www.progress.com/worldwide

Rev 2017/09 | 170914-0021

Progress®