Functional Programming Techniques in C#

{Code PaLOUsa}

Made Possible By These Amazing Sponsors

amazon alexa

Amazon, Echo, Alexa, and the related logo are trademarks of Amazon.com, Inc. or its affiliates.

SOFTWARE GUILD
A Wiley Brand

Andromeda
Galactic Solutions

modis

Who Am I?

My name is **Darren Hale**.

I create software.

I'm easy to find online:

- Twitter: @darrenhale
- LinkedIn: https://linkedin.com/in/darrenhale
- GitHub: https://github.com/haled
- Email: darren.e.hale@gmail.com

The Challenge (or more about me)

- Constantly improve coding technique.
- Test Drive new code creation.
- Keep tests straightforward and easy to understand. (Tests should define what the code does.)
- Keep code as simple as possible. (Hard things should be hard and easy things should not be hard.)
- Object-oriented thinking is ingrained in my being.

Intro to Functional Programming

- Jim Weirich did a live coding exercise to explain functional programming. Magic!

https://www.infoq.com/presentations/Y-Combinator/

- Pointed to higher-level approach to code.

What Is Functional Programming?

Object Oriented Programming

- Central construct is a class.

Functional Programming

- Central construct is a function or expression.

C#

- Central construct is a class.
- Functions are first-class citizens.

Characteristics of Functional Programs

Purity

- Same inputs **ALWAYS** produces same output.
- Is easy to test.

Pure Code Examples

```
public int Sample1()
{
    return 42;
}
```

```
public int Sample2(int a, int b)
{
    return a + b;
}
```

```
public int Sample3(int a, int b)
{
    int z = a + b;
    Console.WriteLine(z);
    return z;
}
```

Impure Code Examples #1

```
public int CurrentSeconds()
{
    return DateTime.Now.Second;
}


public string GetGreeting(string name)
{
    return "Hello " + name + " the current time is " + DateTime.Now;
}
```

Impure Code Examples #2

```
public Sum GetSum(int a, int b)
{
    if(a < 0 || b < 0)
    {
        return null;
    }
    else
    {
        return new Sum(a + b);
    }
}
```

```
public int GetSum2(int a, int b)
{
    if(a < 0 || b < 0)
    {
        throw new Exception("I don't do negatives.");
    }
    else
    {
        return a + b;
    }
}
```

Characteristics of Functional Programs

Purity

Easier to Reason About

- Create functions that work on similar items.
- Allows for a fluent call style.
- Allows code to read like prose.
- (Appropriate for series of steps triggered by an event.)

Easier to Read Example

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHttpsRedirection()
        .UseAuthentication()
        .UseCors()
        .UseMvc();
}
```

Characteristics of Functional Programs

Purity

Easier to Reason About

Immutability

- Things don’t change.
- Stateless

Immutability

```
// Mutation
public MathProblem Addition(MathProblem prob)
{
    prob.Answer = prob.X + prob.Y;
    return prob;
}
```

```
// Immutable
public MathProblem Subtraction(MathProblem prob)
{
    MathProblem result = new MathProblem(prob.X, prob.Y);
    result.Answer = prob.X + prob.Y;
    return result;
}
```

Characteristics of Functional Programs

Purity

Easier to Reason About

Immutability

Consistent Behavior

- Can codify some “boilerplate” behavior.
- Codify team conventions and practices.
- <Code in future slides.>

Characteristics of Functional Programs

Purity

Easier to Reason About

Immutability

Consistent Behavior

Sophisticated Re-use

- Composing functions from other functions.
- Higher-order abstractions.
- <Code in future slides.>

C# and Functional Programming

Functions as First-Class Citizens

- Define functions as data.
- Pass functions as data.
- Functions passed as parameters to other functions.
- Functions as return types from other functions.

Functions as First-Class Citizens

```
private delegate decimal MathCalculator(int val);  // Define function as data

public decimal Square(int x)
{
    return (decimal) x * x;
}

// Pass function as data
// Functions passed as parameters to other functions
public decimal CalcCircleArea(Func<int, decimal> rSquared, int radius)
{
    return 3.14159m * rSquared(radius);
}

// Function as a return type
public Func<decimal> CalcCircleAreaFunction(int radius)
{
    return () => 3.14159m * Square(radius);
}
```

Functional Constructs in C#

Delegates

Action

Functions

Lambdas

Extension Methods

Delegates

A delegate is a *type* that represents a reference to a method.

```
public delegate int AddTwoFunction(int val);

AddTwoFunction adder = AddTwo;
```

Functions

A function is a delegate that has 0 or more input parameters and a defined output.

Func<T,TResult>

The above definition is for a function that takes T and returns TResult.

```
Func<int, long> AddTwoFunction;

public static long AddTwo(int x)
{
    return (long) x + 2;
}
```

Action

An Action is a delegate with no return value and up to 16 parameters.

```
public Action<string> PrintMessage;

public static void MessagePrinter(string value)
{
    Console.WriteLine(value);
}

PrintMessage = MessagePrinter;

PrintMessage = Console.WriteLine;
```

Lambdas

A lambda expression is an expression that has an expression as its body or a statement block as its body.

```
int[] nums = {0, 1, 2, 3, 4, 5};
var doubles = nums.Select(x => x * 2);

services.AddApiVersioning(p =>
{
   p.AssumeDefaultVersionWhenUnspecified = true;
   p.ReportApiVersions = true;
});
```

Lambdas (cont'd.)

```
public static decimal CalcCircleArea(Func<int,long> rSquared, int radius)
{
    decimal pi = 3.14159m;

    return pi * rSquared(radius);
}

Func<int, long> Square = x => x * x;
var area = CalcCircleArea(Square, 5);

var otherArea = CalcCircleArea(x => (long) x * x, 7);
```

```
public static void RecordMessage(Func<string> generator, string message)
{
    Console.WriteLine(generator + message);
}

RecordMessage(() => "Parse string.".Substring(2, 4), " do you see the parsing?");
```

More Examples

```
public static long Triple(int value)
{
    return value * 3;
}

public delegate long Multiplier(int value);

static Multiplier tripler = Triple;

public static void PrintMath(Func<int, long> calculator, int suppliedValue)
{
    var result = calculator(suppliedValue);
    Console.WriteLine("The math operation on " + suppliedValue + " resulted in " + result);
}

public static void PrintMathTyped(Multiplier calculator, int suppliedValue)
{
    var result = calculator(suppliedValue);
    Console.WriteLine("The math operation on " + suppliedValue + " resulted in " + result);
}

public static void Main(string[] args)
{
    PrintMath(Triple, 4);
    PrintMath(x => x * 4, 4);

    PrintMathTyped(tripler, 5);  // Func and delegate are different types
}
```

Why Use It in OO Programs

Re-use Algorithms

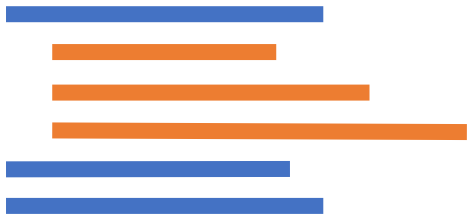
Higher-Order Abstractions

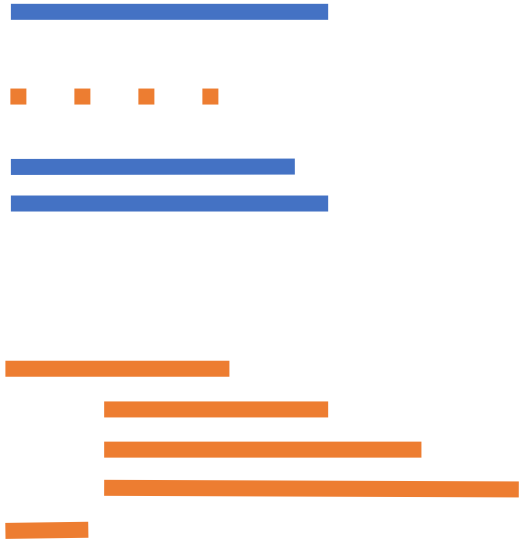
Performance Improvements

Simplify Testing

Real-World Application

Refactoring





Refactoring - Normal

```
public decimal CalcPayment(int amount, int length)
{
    decimal payment = 0.0m;

    if(length != 0)
    {
        var rawPayment = (decimal) amount / length;
        payment = rawPayment + (rawPayment * 0.03m);
    }

    return payment;
}
```

```
public decimal CalcPaymentRefactored(int amount, int
length)
{
    decimal payment = 0.0m;

    if (length != 0)
    {
        payment = DeterminePayment(amount, length);
    }

    return payment;
}
```

```
public decimal DeterminePayment(int amt, int len)
{
    var rawPayment = (decimal) amt / len;
    return (rawPayment + (rawPayment * 0.03m));
}
```

Real-World Application

"Outside In" Refactoring

Refactoring – “Outside In”

```
public decimal CalcPayment(int amt, int len)
{
    decimal payment = 0.0m;

    if (len != 0)
    {
        var rawPayment = (decimal)amt / len;
        payment = rawPayment + (rawPayment * 0.03m);
    }

    return payment;
}
```

```
public bool IsZero(int value)
{
    return value == 0;
}

public decimal CalcPaymentRefactored(int amt, int len)
{
    decimal payment = 0.0m;

    var rawPayment = IsZero(len) ? 0.0m : (decimal)amt / len;
    payment = rawPayment + (rawPayment * 0.03m);

    return payment;
}
```

Try/Catch

```
public static string Calculate(int value)
{
    decimal result = 0.0m;
    var message = "Pending";

    try
    {
        result = (decimal) 1 / value;
        message = "Success";
    }
    catch (System.Exception e)
    {
        message = "Failure due to: " + e.Message;
    }

    return "The result is " + result
           + " with a message of " + message;
}
```

```
public static IResult TryCatch(Func<decimal> workload)
{
    try
    {
        var result = new SuccessResult();
        result.ResultValue = workload();
        result.Message = "Success";
        return result;
    }
    catch (System.Exception e)
    {
        var result = new FailedResult();
        result.ResultValue = 0.0m;
        result.Message = "Failure due to: " + e.Message;
        return result;
    }
}


var result = TryCatch(() => 1 / someValue);
```

Function Composition

```
public static long MultiplierWithTiming(int a, int b)
{
    Console.WriteLine("Starting at " + DateTime.Now);

    var result = a * b;

    Console.WriteLine("Ending at " + DateTime.Now);

    return result;
}

public static long AddWithTiming(int a, int b)
{
    Console.WriteLine("Starting at " + DateTime.Now);

    var result = a + b;

    Console.WriteLine("Ending at " + DateTime.Now);

    return result;
}
```

Function Composition

```
public static Func<dynamic> Timer(Func<dynamic> workload)
{
    return () =>
    {
        Console.WriteLine("Start time: " + DateTime.Now);
        var result = workload();
        Console.WriteLine("End time: " + DateTime.Now);
        return result;
    };
}

public static int Multiply(int x, int y)
{
    return x * y;
}

public static void Main(string[] args)
{
    Func<dynamic> Multiplier = Timer(() => Multiply(4, 5));
    Console.WriteLine("Timed result is " + Multiplier());
}
```

Fun with Function Composition

```
public static void Main(string[] args)
{
    var nums = new List<int> { 1, 2, 3, 4, 5, 6 };
    Func<dynamic> ListMultiplication = () =>
    {
        return nums.Aggregate((x, y) => Multiply(x, y));
    };
    Func<dynamic> TimedMultiplication = Timer(ListMultiplication);
    var result = TimedMultiplication();
    Console.WriteLine("The result is -> " + result);
    Console.ReadLine();
}
```

Summary

What is functional programming?

Functional constructs in C#

Taste of How to Apply Functional Programming Concepts

Thank You!

My name is **Darren Hale**.

I create software.

I'm easy to find online:

- Twitter: @darrenhale
- LinkedIn: @darrenhale
- GitHub: https://github.com/haled
- Email: darren.e.hale@gmail.com

References:

- Jim Weirich Talk: https://www.infoq.com/presentations/Y-Combinator/
- Kathleen Dollard Talk: https://www.ustream.tv/recorded/114916291