

# C# : Functional Programming

The Power of Tiny Abstractions



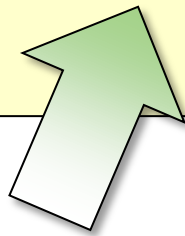
# Overview

- **Why Functions?**
- **Examples**
  - **Laziness**
  - **Parallel programming**
  - **Validation**
  - **Asynchrony**

# Why Functional?

- **Functions are abstractions, too!**
  - Pass functions as parameters
  - Treat functions as data
  - Easy to compose and combine
- **Functional programming excels at separating concerns**
  - Programming in the small versus programming in the large

```
var numbers = new[] {1, 2, 3, 6, 7, 8, 10};  
var odd = numbers.Where(n => n % 2 == 1);
```



**Knows how to filter**



**Knows what to filter**

# Functional Advantages

- Small, reusable abstractions
- Lazy
- Immutability => parallelism

Reusable

```
private static IEnumerable<T> Filter<T>(IEnumerable<T> numbers,  
                                         Predicate<T> predicate)  
{  
    foreach (var number in numbers)  
    {  
        if (predicate(number)) yield return number;  
    }  
}
```

Lazy

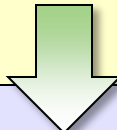
```
var numbers = new[] {1.0, 2, 3, 6, 7, 8, 10};  
var evenNumbers = numbers.AsParallel().Where(n => n%2 == 0);
```

Easy To Parallel

# Removing Loops

- Functional programming often hides the looping construct
  - Results in less imperative code

```
private static IEnumerable<T> Filter<T>(IEnumerable<T> numbers,  
                                         Predicate<T> predicate)  
{  
    foreach (var number in numbers)  
    {  
        if (predicate(number)) yield return number;  
    }  
}
```



```
private static IEnumerable<T> Filter<T>(IEnumerable<T> numbers,  
                                         Predicate<T> predicate)  
{  
    return numbers.Where(number => predicate(number));  
}
```

# Map / Filter / Reduce

- Common functional parlance
- With LINQ it's Select / Where / Aggregate

```
public double TotalSalaryForManagers(IList<Employee> employees)
{
    return employees.Where(e => e.IsManager) // filter
                      .Select(e => e.Salary)  // map
                      .Sum();                 // reduce
}
```

# Declarative Programming

- Less imperative == more declarative
- Why should validation rules branch the execution path?

```
var errors = new List<ValidationError>();  
if(string.IsNullOrEmpty(movie.Title))  
{  
    errors.Add(new ValidationError() {Message = "..."});  
}  
if(movie.Duration < 30)  
{  
    errors.Add(new ValidationError() {Message = "..."});  
}  
else if(movie.Duration > 360)  
{  
    errors.Add(new ValidationError() {Message = "..."});  
}
```

# A Declarative Approach

- Treat validation code as data to evaluate

```
public class ValidationRule<T>
{
    public Func<T, bool> Rule { get; set; }
    public ValidationError Error { get; set; }
}
```

```
new ValidationRule<Movie>
{
    Rule = m => string.IsNullOrEmpty(m.Title),
    Error = new ValidationError() {Message = "..."}
};
```

```
public IEnumerable<ValidationError> Validate (Movie movie)
{
    Func<ValidationRule<Movie>, bool> theRuleFails = r => r.Rule(movie);
    return GetRules().Where(theRuleFails).Select(r => r.Error);
}
```





# Continuations

- Passing functions to describe what to do next
  - Makes asynchronous invocation easy

```
DoAsync(Work,  
    () => Console.WriteLine("Success!"),  
    ()=> Console.WriteLine("Error!"));
```

```
static void DoAsync(Func<bool> func, Action onSuccess, Action onError)  
{  
    func.BeginInvoke((asyncResult) =>  
    {  
        bool result = func.EndInvoke(asyncResult);  
        if (result)  
        {  
            onSuccess();  
        }  
        else  
        {  
            onError();  
        }  
    }, null);  
}
```

# Functions as Data

- Pass them as parameters
- Return them from methods

```
private static Action MakeGreetingFunction()  
{  
    if(DateTime.Now.Hour < 12)  
    {  
        return () => Console.WriteLine("Good morning!");  
    }  
    return () => Console.WriteLine("Good evening!");  
}
```

```
var f = MakeGreetingFunction();  
f();
```

# Closures

- Binding functions to state
  - What happens to the name parameter?

```
private static Action MakeGreetingFunction(string name)
{
    if(DateTime.Now.Hour < 12)
    {
        return () => Console.WriteLine("Good morning, {0}", name);
    }
    return () => Console.WriteLine("Good evening, {0}", name);
}
```

# Functions as Parameters

- Passing anonymous methods around makes LINQ work
- Also useful to abstract away what should happen around method invocation
  - Caching, memoization, automatic retries

```
Action fetchMovies = () =>
{
    // ... call web service
};

fetchMovies.WithRetry();
```

```
static void WithRetry(this Action action)
{
    int retryCount = 0;
    bool succesful = false;
    do
    {
        try
        {
            action();
            succesful = true;
        }
        catch (NetworkException e)
        {
            retryCount++;
        }
    } while (retryCount < 3 && !succesful);
}
```

# Summary

- Functions are perfect abstractions for programming in the small
- Treat functions as data
- Also see: “LINQ – Beyond Queries” at [pluralsight.com](http://pluralsight.com)