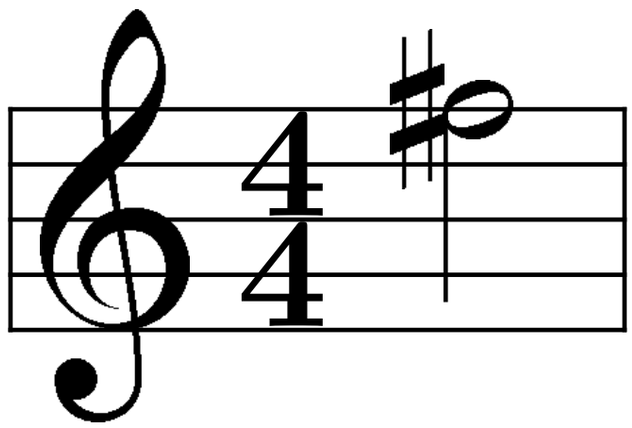
Functional Programming in F#



Oliver Sturm
oliver@oliversturm.com

http://www.oliversturm.com

Oliver Sturm

thinktecture Associate

ineta

MVP Microsoft Most Valuable Professional

Oliver Sturm (@olivers)

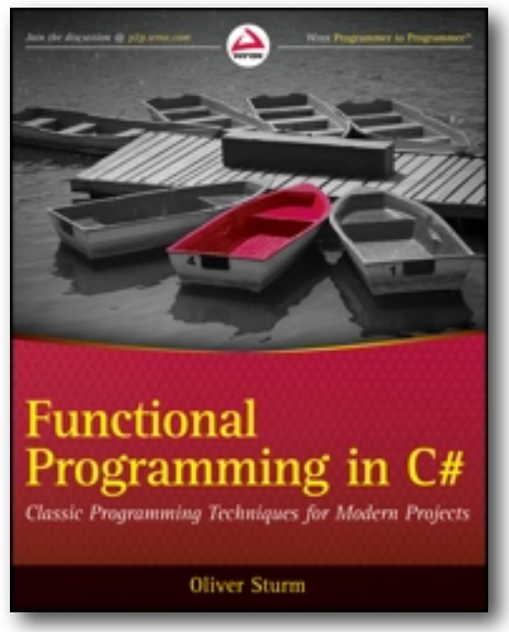
- **Consultant and Trainer, Author**
- **Associate Consultant at thinktecture**

- **.NET Application System Architecture**
  - User Interfaces
  - Data Handling / Data Access Architectures
  - Programming Languages
  - DevExpress Component/Framework Products

- **Microsoft MVP for C#**
- **INETA Europe Speaker**

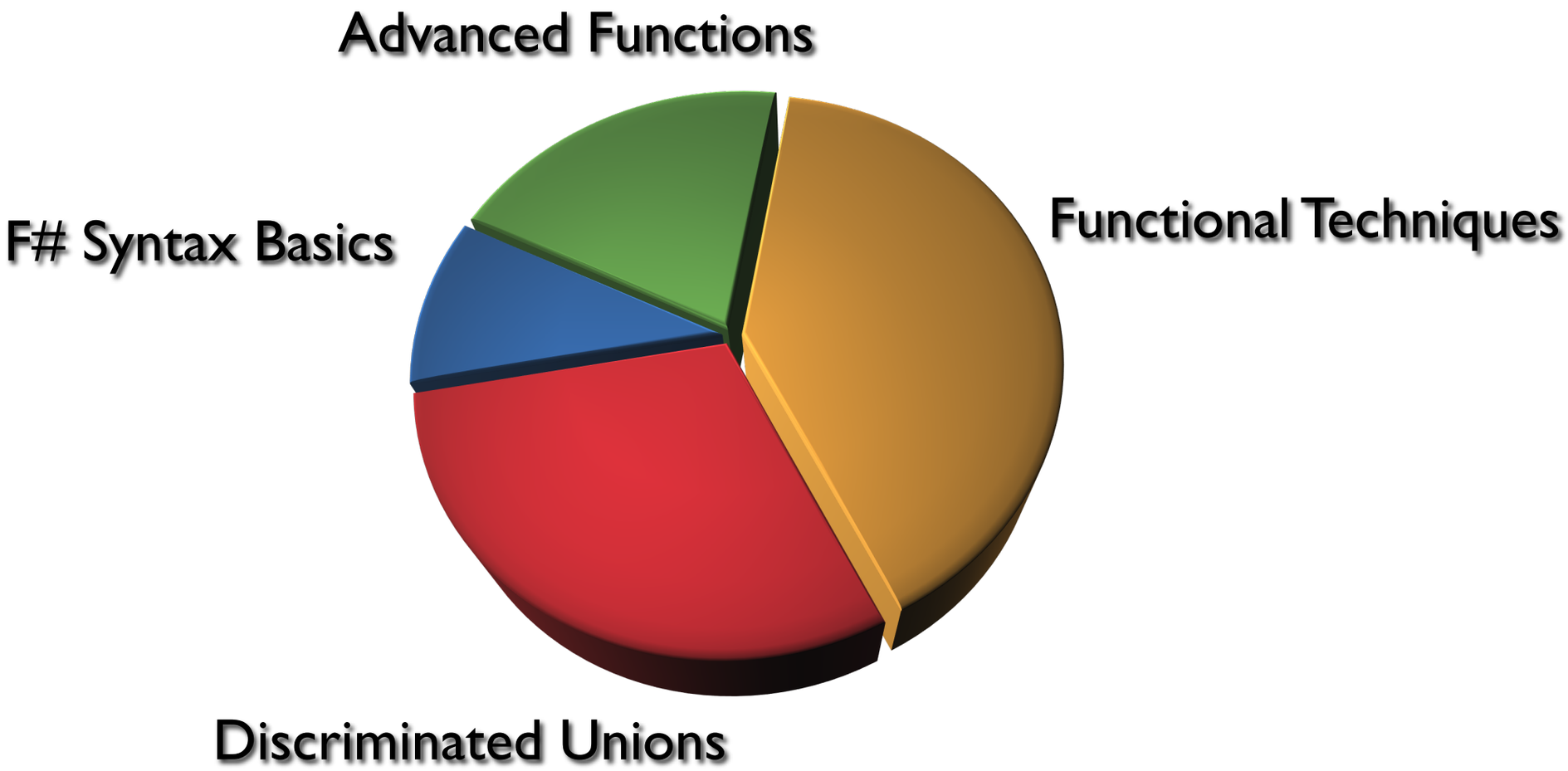
- **Services: http://www.oliversturm.com**
- **Blog: http://www.sturmnet.org/blog**

- **oliver@oliversturm.com**



Agenda

- What is F#?
- What is Functional Programming all about and why is it suddenly interesting?



What is F#?

- A .NET language with Visual Studio integration
- A hybrid language, supporting functional as well as imperative and object oriented paradigms
- Type safe
- Widely cross-compilable with Ocaml
- Scriptable
- F# is part of Visual Studio 2010

What do you need to get started?

- Visual Studio 2010
- Optional: F# PowerPack - source code, VS 2008 and mono support - YMMV

The concept of Functional Programming

- Functional programming is a paradigm
- FP tries to avoid shared state
- Pure functions
  - results calculated only on the basis of input values
  - pure functions don't store or access information outside themselves, avoid side effects
- Functions are first class citizens, enabling higher order functions
- Higher order functions enable currying and partial application
- Functions are building blocks

Avoiding side effects and shared state

- Debugging benefits from this approach
- Testing benefits as well
- Scalability becomes easy
  - no locking of data
  - no manual analysis of code structure to find scalable parts
- Automatic optimization becomes possible
- Maintainability improves drastically

Parallelization is a (the?) trigger

- Writing code without side effects makes parallelization easy
- Parallelization can be fully automatic… not yet in F# though
- Functional approaches offer many solutions
- Functional programming as a paradigm is not language specific, but the more language support there is, the easier it becomes

The reality is that...

- F# is a .NET language that supports several different programming paradigms
- Automatic parallelization does not exist (yet?) on the .NET Framework
  - but Parallel Extensions to the .NET Framework are in .NET 4.0
  - the success of Erlang shows the reality of scalability, stability and maintainability claims, Ericsson reports 99.9999999% uptime (down 1 second in 30 years)
- Automatic optimization does not exist yet either
- A combination of approaches results in the best code for a particular purpose
- The more flexible a programming language is, the easier it is to combine approaches

Demo

OPTIONAL

- F# Syntax Basics

Demo

- Recursive functions
- Lambda expressions
- Nested functions
- Higher order functions

ALMOST OPTIONAL

Recursive functions

- The `rec` keyword is required to denote a function as recursive
  - if `rec` is missing, the function's name is not in scope in its own body
- Tail recursion is applied automatically if the recursive call is the last statement executed in a recursive function

Lambda expressions

- Lambda expressions use the `fun` keyword and the `->` (goes-to) operator
- A lambda expression assigned to a value results in the same function accessible through that value as "let"-style function creation
- Lambda expressions are "anonymous functions"
- Lambda expressions cannot be recursive

Nested functions

- Functions do not have to be on “top level” scope
- Values assigned inside functions can also be functions
- The scope of nested functions is the same as for any other value assigned on the same “level”

Higher order functions

- Higher order functions take other functions as parameters or return them as return values
- Any parameter passed to a function in F# can be a function itself
- Functions can be return values of functions
- Lambda expressions can be used to pass functions "in-line"
- Functions referred to by the values they have been assigned to can also be used to pass as parameters

Nested calls and chaining (piping)

- Calls to functions delimit parameters with spaces
- Use parentheses to resolve ambiguity as well as readability issues:
```
mult (add 10 30) 40
```
- Pipes append the result of one function to the parameter list of another:
```
add 10 30 |> mult 40
```
- Pipes result in a more natural order of calls in complex nested statements

Composition

Assuming B = f1(A), C = f2(B)

→ C = f2(f1(A))

```
let square x = x * x

let triple x = 3 * x
...
let a = 10
let b = square a
let c = triple b
...
let c = triple (square a)
```

Getting from a to c step by step →

Getting from a to c by nesting calls →

Demo

- Nested calls, chaining
- Composition

Closures

- Closures capture values that are used by functions, when these functions leave their scope

```
let createCalculation val =
    let calc x = val * x

    calc
```

The function calc needs the value val to perform its calculation

The function calc leaves the scope of the value val

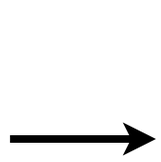
Demo

- Closures

Currying

- Currying is the process of transforming a function with multiple parameters into a chain of functions that each take one parameter and return the next function, until on the deepest level the calculation can be performed with all parameters.

```
let add x y = x + y
```



```
let add x =
    (fun y -> x + y)
```

Demo

- Curried format functions
- Partial application
- Functional precomputation
- Memoization

Curried format functions

- In curried format, functions always take exactly one parameter
- Functions might return other functions to gather additional parameters
- The last function in the “chain” can perform the calculation using all parameter values
- Closures are used to store parameter values

Partial application

- Applying a function partially means passing in some, but not all, parameters needed by the function
- Due to the curried format, partial application of a function means that another function is returned
- Partial application is one approach in the area of “function construction”, i.e. creating new functions out of existing ones

Functional precomputation

- Precomputation is an approach where expensive calculations are performed in advance of an algorithm run
- Precomputed values are stored for later use
- Functional precomputation means using a function, or a closure, as a storage location
  - no "external" storage is therefore needed
- Since curried functions are automatic in F#, the approach is very elegant in this language

Memoization

- Memoization is a caching pattern that stores values which have been calculated once for later reuse
- It is possible to memoize as a wrapper function
- “Deep” memoization, i.e. memoization of a chain of curried functions, requires Reflection
- Memoizing via a wrapper functions is meant not to change the algorithm
  - when the function is recursive, memoization “from the outside” is typically not possible

Simple discriminated unions

- Discriminated unions are data types
- In simple cases discriminated unions behave like enums
- Their uses cases are similar to those of enums
  - case/kind distinction

```
type MemberKind =
     | Method = 1
     | Property = 2
     | Field = 3
```

Elements that carry data

- Elements of discriminated unions can carry data
- In this case, the compiler generates classes automatically

```
type Product =
      | OwnProduct of string
      | RemoteReference of int
```

```
class Product { ... }
class OwnProduct : Product { ... }
class RemoteReference : Product { ... }
```

Data can be complex

- The data carried by elements may be of any valid F# type
- Use of simple tuples and other discriminated unions results in powerful data structures

```
type Product =
     | OwnProduct of string
     | RemoteReference of int

type StoreBooking =
     | Incoming of Product * Count
     | Outgoing of Product * Count
```

Definitions can be recursive

- Type definitions for discriminated unions can refer to themselves
- Complex hierarchies can be created using recursive discriminated unions

```
type Control =
      | Button of Caption
      | CheckButton of Caption * Checked
      | Edit of Caption * StrContent
      | Container of Control list
```

Demo

- Discriminated unions
  - simple unions / enums
  - data-carrying members
  - members with complex types
  - recursive unions
  - using match expressions to analyze union hierarchies
  - implementation of a linked list using discriminated unions

Summary

- F# - great new option for .NET development
- Complete feature set spanning functional as well as imperative/object-oriented programming
- I hope it was fun!

Thank you

Please feel free to contact me about the content anytime.

oliver@oliversturm.com