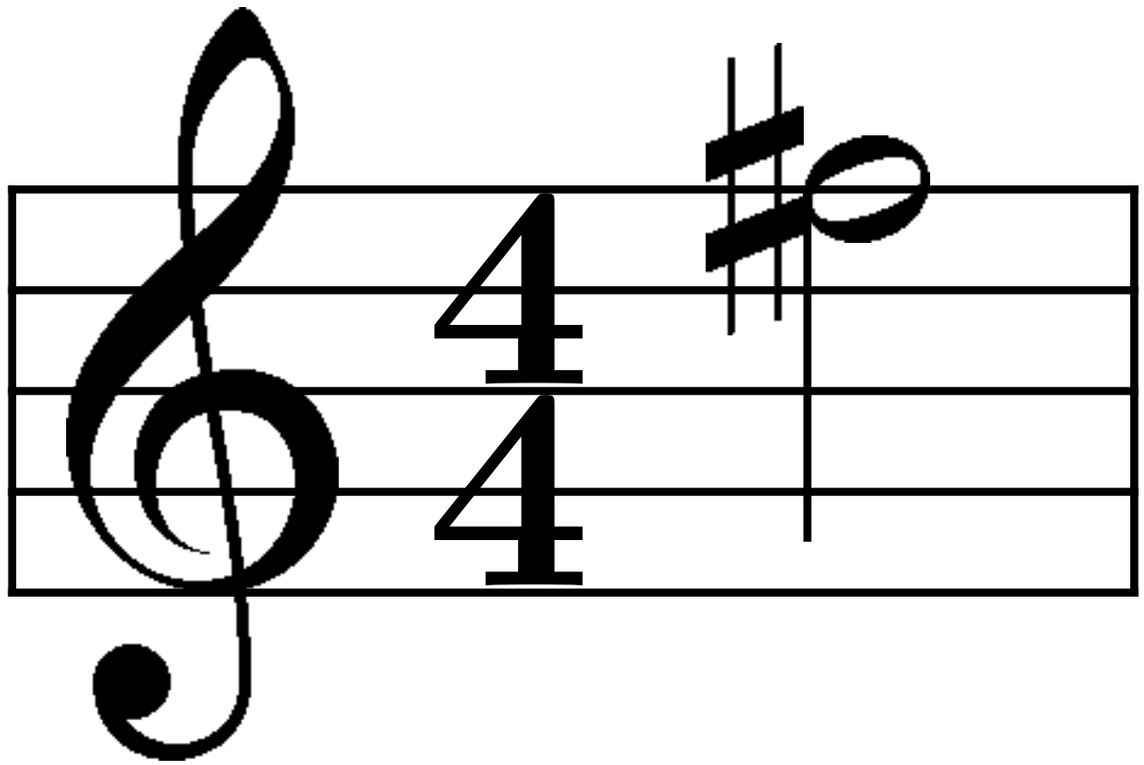
A day of F#



Oliver Sturm
oliver@sturmnet.org
olivers@devexpress.com
http://www.sturmnet.org/blog

Who's that Oliver Sturm guy, anyway?

I am
- Interested in programming languages, databases and a whole bunch of other things
- Microsoft MVP for C#

I do
- Work for DevExpress as Director of Quality
- Blog at http://www.sturmnet.org/blog
- Podcast at http://www.sodthis.com

You should
- Email me at oliver@sturmnet.org
- Follow me on Twitter @olivers

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

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

Avoiding mutable data

- Mutable data requires the developer to doubt, at any point in his code, everything that has happened before

The statement is correct here

Now the above statement is wrong

```
int a = 37;
int b = 42;
int c = a * b;


b = 50;
```

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 will be part of .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

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 being “productized” for Visual Studio 2010

Why is F# interesting?

- Testbed for features we see in C# later
  - past: Generics were introduced to .NET at the initiative of Don Syme
  - past: LINQ has its roots in standard map/reduce/filter approaches
  - .NET 4.0 will have Lazy and Tuple types
- Optimized syntax for Functional Programming allows easier combination of approaches
- Things that go beyond C# - metaprogramming, type inference, ...

For whom is F# interesting?

- FP in general is good at calculating things – academics, financial sector, business logic, ...
- MS wants to drive VS adoption in technical computing markets and gain academic mindshare
- Big banks have been working on F# projects for a long time, some were waiting for the productization commitment
- Systems like Erlang show benefits of concurrency strategies that FP promotes
- F# as a multi-paradigm language can be interesting to everybody as a more flexible alternative to other .NET languages

What do you need to get started?

- F# CTP for VS 2008 available: http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/
- The latest release is the May 2009 CTP
- VS 2010 beta 1 contains a corresponding state, but not the F# powerpack
- Expect changes at least until Visual Studio 2010 is finalized
- Packages to run F# in mono are also available

Agenda

- Some background on Functional Programming (0:30)
- **Basics of F# syntax** (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

Demo

- Immutable values, data types and type inference
- Tuples, lists and option types
- Match expressions
- Mutable values and reference cells
- Functions
- Code structure and formatting
- Flow control using “if” and “for”

Basic types

- Basic F# types are: (s)byte, (u)int16, (u)int32, (u)int64, string, single, double, char, (u)nativeint, bool, decimal, bigint and unit
- Literals can have postfixes denoting their types:
  - y (byte), s (int16), l (int32), n (nativeint), L (int64) for integer types, combine these with u for an unsigned value (uy, us, …)
  - f (float), I (bigint), m (decimal)
- Integer types can also use the prefixes 0x, 0o and 0b to represent numbers in hexadecimal, octal and binary format

Tuples

- Tuples combine other types into a new type, e.g. `int*string*int` to combine an `int`, a `string` and another `int`
- Tuple Literals in code use parentheses: `(10, “text”, 40)`
- Tuples allow the construction of combined “record” types on the fly, without prior declaration
- Tuples don’t add behavior, they are “just” a convenient combined data type

Working with .NET data types

- All .NET Framework data types are available for use in F#
- Instantiating them through their constructors requires a call like this:

```
let sb = StringBuilder()
```

- Convention: if the type implements IDisposable, use an explicit “new” keyword:

```
let form = new Form()
```

Option Types

- Option types represent the notion that values might not "have a value"
- A value of type `Option<T>` can be assigned either `Some(T)` or `None`
- Similar concept to the idea of "null" in many imperative languages

Lists

- Lists in F# are immutable data structures, implemented as singly-linked lists
- Literals in code use brackets: `[1, 2, 3]`
- Lists are typed, so the types of contained elements must be compatible
- The empty list is represented as `[]`
- The operators `::` and `@` are used to prepend an element to a list and concatenate two lists, respectively
  - both these operators return a new list object – lists are immutable

Match Expressions

- Match expressions apply one or more patterns to a value
- Checks can test for literals or other values
- Complex tests can test for `Some()` vs. `None` to support option types, empty lists and `hd :: tl` constructs to support lists
- Decomposing of values in match expressions is also possible

Mutability

- Values are immutable by default
- Mutable values are supported through the `mutable` keyword and the "ref cells" technique
- Only ref cells can be used in closures
- Assignment operators differ depending on the "kind" of mutability used
  - assignments for "mutable" types use `<-`
  - with ref cells, assignments use `:=`
- Immutability can be shallow
  - the built-in array type is not immutable itself

Functions

- Functions are values themselves
- Functions are in curried format automatically
- Calls to functions separate parameters by spaces
- Indentation defines the structure of a function's body
- The result of the last statement in the body of a function is also the result of the function

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- **Advanced functions and functional techniques** (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

Demo

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

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

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- **Discriminated unions** (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

Simple discriminated unions

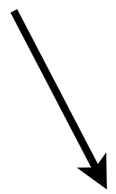
- Discriminated unions are data types
- In simple cases discriminated unions behave like enums
- Their uses cases are similar to those of enums
  - case/kind distinction

```
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

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- **Exception handling and other loose ends** (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

Exception Handling

- F# needs to be able to handle exceptions to be compatible with .NET
- try/with and try/finally are available, same as try/catch and try/finally in C#
- Exception handling blocks can be used in expressions
- Pattern matching applied to exceptions makes filtering very flexible
- Use the exception keyword to create custom exceptions

Handling IDisposable

- “using” is a higher order function that disposes at the end of the function it gets passed
- “use" is a keyword that disposes at the end of the current code block

```
let writeFile() =
    using (File.CreateText("file.txt"))
        (fun file ->
            file.WriteLine("Stuff in the file"))

let writeFile2() =
    use file = File.CreateText("file2.txt")
    file.WriteLine("Stuff in other file")
```

Demo

- Catching exceptions
- Creating custom exceptions
- Raising and rethrowing exception
- try/finally
- use
- using

Catching exceptions

- Exceptions can be caught using try/with blocks
- These blocks can appear on their own or in expressions

```
try
    let foo = 10 / 0
    printfn "%d" foo
with
    | :? InvalidOperationException as e ->
        printf "InvalidOperationException: %s" e.Message
    | e -> printfn "Other exception: %s" e.Message

let result =
    try
        Some(10 / 0)
    with
        | _ -> None
```

Creating custom exceptions

- The “exception” keyword is used to create custom exceptions
- Implementation of the exception is automatic, only the type of the data it carries needs to be specified

```
exception MyOwnException of string * int

try
    raise (MyOwnException("An error has occurred", 42))
with
    | MyOwnException(strVal, intVal) ->
        printfn "Got exception with str='%s', int=%d"
          strVal intVal
```

Demo

- Casting up
- Casting down
- Using match expressions for type checking
- Converting numeric data types

Lists

- Lists in F# are immutable single-linked lists
- The implementation is based on discriminated unions
- List comprehension syntax
  - [ ] is the empty list
  - `[ b .. s .. e ]` initializes a list with values starting with b, stepping by s and ending at e
  - `[ for … -> … ]` initializes a list with a custom calculation
- The module Microsoft.FSharp.Collections.List makes further helper functionality available
- `hd :: tl` syntax is available in match expressions

Sequences

- F# sequences are based on `IEnumerable<T>`
- Evaluation is lazy, like an iterator in C#, or more generally, a continuation
- List comprehensions similar to lists are available, but using curly braces { ... }
- The `seq { ... }` workflow can be used for more complex value creation
  - the workflow uses the `yield` and `yield!` keywords to "pass back" data to the outer context
- The module Microsoft.FSharp.Collections.Seq has more functionality

Demo

- Lists
- List handling
  - head/tail
  - cons, append
- Sequences

Agenda

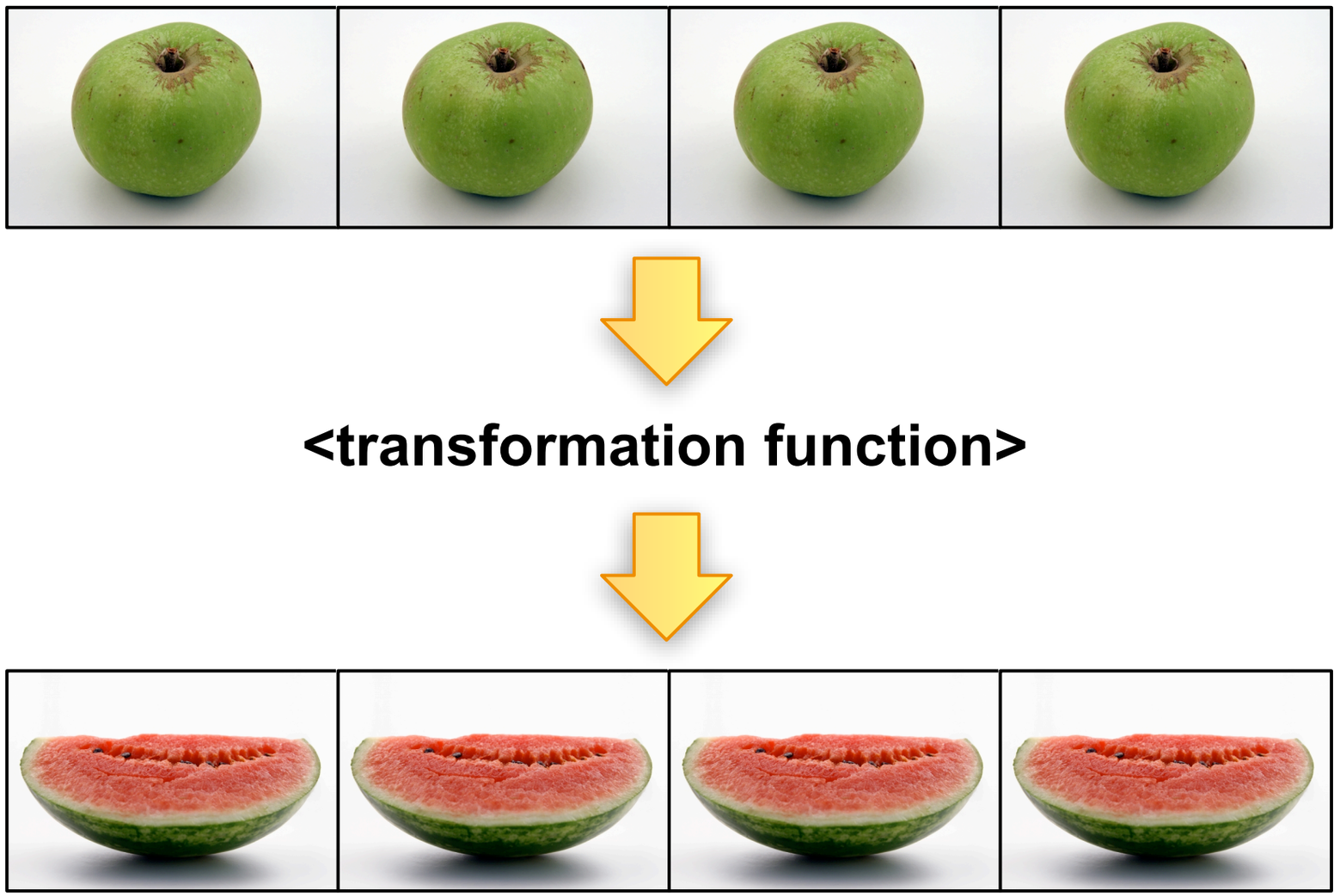
- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- **Map, Filter and Fold** (0:30)
- Object Oriented Programming (1:00)
- A quick look at some add-ons

Map, Filter and Fold

- Available in almost all programming languages/ runtimes, sometimes with varying names
- Available in LINQ as Select, Filter and Aggregate
  - functionality provided by map, filter and fold is similar to database querying functionality
- F# implementations List.XXX are for lists, Seq.XXX are for sequences
- Using function construction, these functions form the basis of new functionality

Map

- Map applies a function to each element in a list, returning a list of results



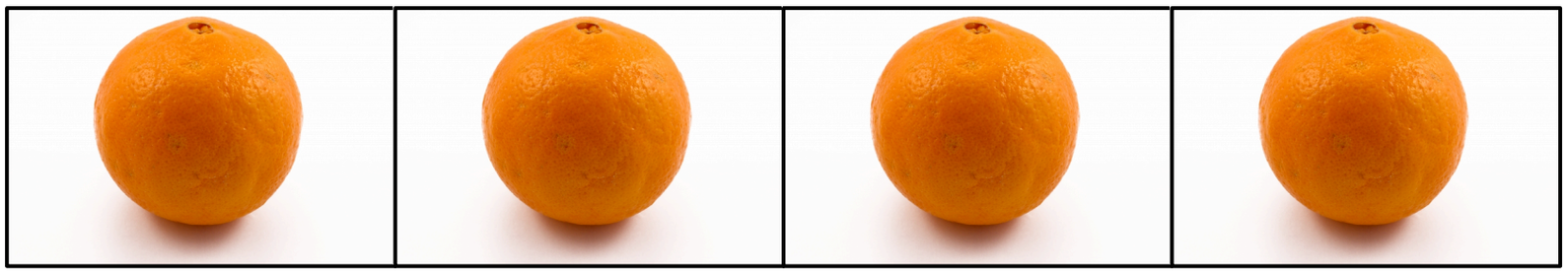
Filter

- Filter applies a predicate function to each element in a list, returning the elements that fit the predicate

<filter function>

Fold

- Fold aggregates values in a list using an aggregation function



<aggregation function>

4

Google MapReduce

- Google MapReduce is a parallel computation framework based on map and fold (reduce)
- The combined application of these functions targets a surprisingly broad range of algorithms
- Google have used MapReduce to recreate their index and it is now being used to keep the index up to date
- Part of Amazon's AWS package is a MapReduce implementation based on Apache Hadoop

Demo

- Applying map, filter and fold to work with data
- Function construction using partial application
- Advanced applications of fold
  - loop replacement
  - map and filter as applications of fold

Standard higher order functions w/ partial application

- Partially applying map, filter and fold easily creates complex new functions
- These functions are generic, so beware of "value restriction"
  - when the partially applied parameter(s) restrict the types in any way, the F# compiler will show a Value Restriction error
  - either specify an explicit return type for the partially applied function, or construct a syntactic function

Value restriction →

Solution: specific return type →

Value restriction →

Solution: syntactic function →

```
let intSequenceAdder = Seq.fold (+) 0
let intSequenceAdder : seq<int>->int =
  Seq.fold (+) 0

let squareMapper = Seq.map
  (fun x -> x * x)
let squareMapper l = Seq.map
  (fun x -> x * x) l
```

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- **Object Oriented Programming** (1:00)
- A quick look at some add-ons

Immutable classes (records)

- F# supports immutable classes (records)
  - supports the functional ideal of having immutable data
- There is a cloning syntax that creates "modified clones"
  - this is used instead of making changes to data stored in class instances

```
type Point =
    { X: float; Y: float }
    member x.Shift(dx, dy) =
        { x with X = x.X + dx; Y = x.Y + dy }
```

Record types and constructed types

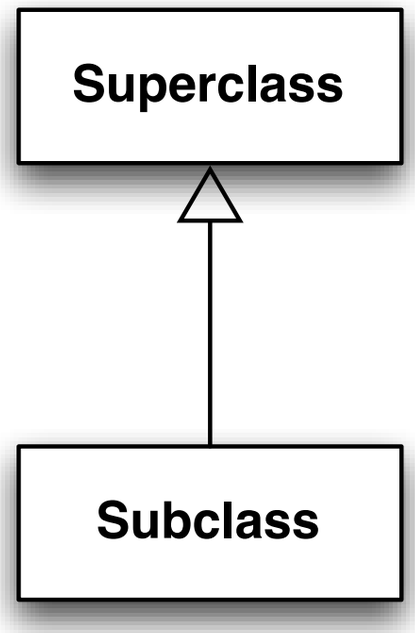
- Constructed types have constructors, like classes in C#
- Record types do not have constructors
- Record types are instantiated through type inference

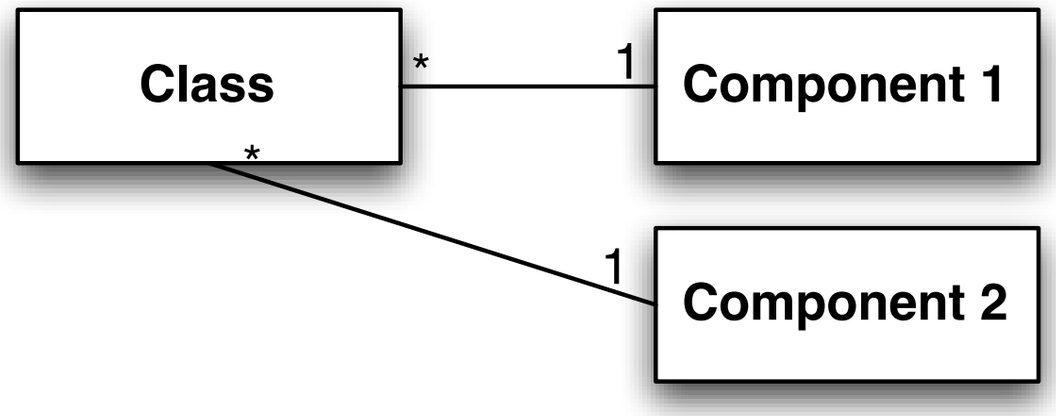
```
type Point =
    { X: float; Y: float }
    member x.Shift(dx, dy) =
        { x with X = x.X + dx; Y = x.Y + dy }

let point1 = { X = 10.3; Y = 11.4 }
```

Composition vs. Inheritance

- Gang of Four: favor composition over inheritance
- F# has syntactic and structural support for deriving classes from other classes by means of composition
- Implementation inheritance is equally well supported





Demo

- F# specific OO approaches
  - immutable classes
  - record types and constructed types
  - modified cloning expressions
  - object creation expressions
  - interface instantiation
  - composition instead of inheritance
  - augmenting types
  - declaring mutually dependent types and functions
- Standard class elements

Interface instantiation

- Interfaces are types with only abstract members
- Object creation expressions can be used to "instantiate interfaces"

An interface type →

Object creation expression instantiates the interface →

```
type IInOut =
    abstract Input: unit -> string
    abstract Output: obj -> unit

let implementer = { new IInOut with
        member x.Input() = Console.ReadLine()
        member x.Output(o) = printfn "%A" o }
```

Type augmentation

- Types can be “opened” and extended
- This is especially useful for basic types and discriminated unions

```
type Int32 with
    member x.SpecialOutput() = printfn "Value is: %d" x

type Thing =
    | OneThing of string
    | OtherThing of int

type Thing with
    member x.OutputThing() =
        match x with
        | OneThing(s) -> printfn "Thing with a string: %s" s
        | OtherThing(i) -> printfn "An int thing: %d" i
```

Mutually dependent types and functions

- Everything that is needed in the scope of a type or function to create the element itself, must exist before the scope is entered
- Types and functions that are mutually dependent must be declared in one block using the “and” keyword

```
type TypeA() =
    let b = TypeB()

and TypeB() =
    let a = TypeA()

let rec functionA() = functionB()
and functionB() = functionA()
```

Mutable classes

- F# can create all standard class elements on the .NET platform
- Implementation inheritance works just like C#
- Accessibility does not know “protected”
- Overloading members requires OverloadIDAttribute
- Constructors and methods with optional arguments are supported

Demo

- Mutable classes with standard .NET OO elements
  - properties
  - indexer properties
  - overloaded methods
  - accessibility modifiers
  - implementation inheritance

Properties

- Properties can have get/set accessors
- Backing stores can use both kinds of mutability, or any other storage mechanism

```
type Person(firstName: string, lastName: string) =
    let mutable firstName = firstName
    let mutable lastName = lastName

    member p.FirstName with get() = firstName
                       and set(n) = firstName <- n
    member p.LastName with get() = lastName
                      and set(n) = lastName <- n
```

Method overloading

- Methods can be overloaded on their parameter list
- Methods can also be overloaded on their return types
- Overloads with conflicting numbers of parameters require the application of the attribute OverloadID

```
type FlexibleWalker() =
    member x.Walk(a, b) =
        printfn "Walking with %d and %d" a b
    member x.Walk(a) =
        printfn "Walking with %d" a
    [<OverloadID("WalkWithString")>]
    member x.Walk(s) =
        printfn "Walking with the string '%s'" s
```

Accessibility modifiers

- Modifiers are “public”, “private” and “internal”
- Modifiers can be specified with let-bindings, modules, types, members, constructors, get/set accessors and record type members
- Default accessibility is “public”
  - lexical scope sometimes restricts, e.g. when accessing let-bindings in classes
- All non-public entities in F# are “internal” in the compiled .NET assembly

Agenda

- Some background on Functional Programming (0:30)
- Basics of F# syntax (0:45)
- Advanced functions and functional techniques (1:25)
- Discriminated unions (0:40)
- Exception handling and other loose ends (1:00)
- Map, Filter and Fold (0:30)
- Object Oriented Programming (1:00)
- **A quick look at some add-ons**

Additional functionality

- LINQ
- Math (matrix, complex, others)
- Plotting
- Async
- Reflection add-ons
- FsLex/FsYacc
- Collections
- OCaml compatibility
- Some of the extensions are part of the F# PowerPack

Optional demo, in case we're bored now

- ADO.NET and LINQ to SQL data access

Summary

- F# - great new option for .NET development
- Complete feature set spanning functional as well as imperative/object-oriented programming
- Lots of things we haven't seen: active patterns, workflows, ... and of course lots of applications of functional programming
- I hope it was fun!

Thank you

Please feel free to contact me about the content anytime.

oliver@sturmnet.org