

F#

...ein kurzer Überblick

Was ist F#?

- ▶ Entwickelt seit 2005 von Microsoft Research
- ▶ Funktionale Sprache mit imperativen und objektorientierten Elementen
- ▶ Verwandtschaft mit Objective CAML
- ▶ Sowohl kompilier- als auch interpretierbar
- ▶ Entwicklung in F# wird voll unterstützt durch Visual Studio 2008, ab VS 2010 im Standardlieferumfang
- ▶ Kompilat wird im .NET-Framework ausgeführt und kann auf alle .NET-Standardbibliotheken zugreifen

Wozu denn **noch eine** funktionale Sprache?

- ▶ **Zwei große Vorteile von F#:**
 - ▶ Durch Unterstützung in Visual Studio steht eine mächtige und komfortable IDE zur Verfügung, mit der auch größere Projekte umsetzbar sind
 - ▶ Durch das .NET-Framework können die Kompilate auf einem stabilen Runtime-System mit mächtigen Standardbibliotheken aufbauen und leicht selbst als Bibliotheken in C# / C++-Projekte integriert werden
- ▶ **...und leider auch ein Nachteil:**
 - ▶ Die Verzahnung mit dem .NET-Framework erschwert die Ausführung von F#-Programmen auf nicht-Windows-Betriebssystemen
 - ▶ Möglichkeiten bestehen aber, siehe Mono-Projekt

Das Obligatorische zuerst: Hello World!

```
printfn "Hello World"
```

- ▶ „printfn“ funktioniert ähnlich printf in C und gibt einen formatierten String aus
- ▶ Unspektakulär – aber erste Erkenntnis: Funktionsaufrufe brauchen keine Klammern!

Wertzuweisungen

```
let x = 2
let y = 3
let z = x + y
```

- ▶ „let“ weist Variablen Werte zu
- ▶ Die Variablen sind dabei gar nicht „variabel“, sondern unveränderbar!
- ▶ „Variable Variablen“ müssen mit „mutable“ deklariert werden

Typen

```
let x = 2.2
let (y :float) = (float 3)
let z = x + y
printfn "%f + %f = %f" x y z
```

- ▶ Die Variablentypen werden, wann immer möglich, automatisch abgeleitet; Standardtyp ist int32
- ▶ Explizite Typzuweisung ist möglich, ebenso Casting
- ▶ F# ist trotz Typableitung statisch und stark typisiert

Funktionen

```
let rec factorial n =  
    if n=0 then 1 else n * factorial (n-1)
```

- ▶ Auch Funktionen werden mit „let“ deklariert und sind „First-Class Citizens“
- ▶ „rec“ kennzeichnet rekursive Funktionen
- ▶ Funktionen haben beliebig viele Argumente und genau einen Rückgabewert, dessen Typ automatisch abgeleitet wird
- ▶ F# unterstützt Closures!

Tupel

```
let swap (a,b) = (b,a)  
swap („Web“, 2.0)
```

- ▶ (n)-Tupel sind schnell und einfach erstellbare Wertesammlungen
- ▶ Tupel gelten als ein Argument -> Funktionen können mit ihnen mehrere Werte zurückliefern

Listen

```
let numbers1 = [1; 2; 3; 4]
let numbers2 = [1..4]
let numbers3 = 1 :: 2 :: 3 :: 4 :: []
```

- ▶ Alle 3 Befehle erzeugen dieselbe Liste
- ▶ Listen können Werte beliebiger Typen enthalten, jedoch müssen alle Werte einer Liste vom selben Typ sein

Sequenzen

```
let numbers = seq { 1 .. 1000000000 }
```

- ▶ Im Gegensatz zu Listen werden die Elemente in Sequenzen „lazy“ evaluiert, d.h. erst erzeugt, wenn auf sie zugegriffen wird
- ▶ Sequenzen können daher beliebig viele Elemente „enthalten“

Currying

```
let add x y = x + y  
let increment x = add x 1
```

```
let var = 2  
let var2 = increment var
```

- ▶ Durch Currying können Funktionen mit mehreren Argumenten zu Funktionen mit weniger Argumenten gemacht werden

Pipelining

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

```
let square (x :int) = x * x
```

```
let negate x = -x
```

```
let complexFunction x =
```

```
    x |> square |> add 5 |> negate
```

- ▶ Pipelining ermöglicht einfach lesbare Verknüpfung mehrerer Funktionen

Pattern Matching

```
let rec factorial2 (n :int64) =  
  match n with  
  | 0L -> 1L  
  | _ -> n * factorial2 (n - (int64 1))
```

- ▶ Pattern Matching stellt eine Art „switch“-Konstrukt dar
- ▶ Das Default-Pattern „_“ trifft immer zu

Pattern Matching 2

```
let rec factorial3 = function
  | 0 -> 1
  | n -> n * factorial3 (n - 1)
```

- ▶ Das Schlüsselwort „function“ ist eine Kurzform zur Pattern-Deklaration
- ▶ Neben festen Werten können auch Variablen als Pattern genutzt werden; ihnen wird der Eingabewert zugewiesen

Pattern Matching 3

```
let rec factorial4 = function
  | 0 -> 1
  | n when n < 0 -> 0
  | n when n > 0 -> n * factorial4 (n - 1)
  | _ -> 0
```

- ▶ Zuweisungen können optional auch mit Bedingungen verknüpft werden

Klassen

```
type BankAccount = class
  val owner : string
  val mutable balance : int

  new (owner, balance) =
    {   owner = owner
        balance = balance }

  member x.Deposit(value) = x.balance <- x.balance + value
  member x.Withdraw(value) = x.balance <- x.balance - value
end
```

Noch mehr...!

- ▶ **F# kann noch weit mehr:**
 - ▶ Sets und Maps
 - ▶ Arrays
 - ▶ Option Types
 - ▶ Vererbung und Interfaces
 - ▶ Module und Namespaces
 - ▶ Operator-Überladung
 - ▶ Active Patterns

- ▶ ...doch das sprengt den Rahmen dieser Übersicht ;-)