Spartkiade 2017

F# Workshop

Übungen

# Vorausseztungen

## Windows

* Visual Studio 2015 mit F# installiert mit F# Powertools
* Visual Studio Code + F# Ionide Package

# Erste Schritte

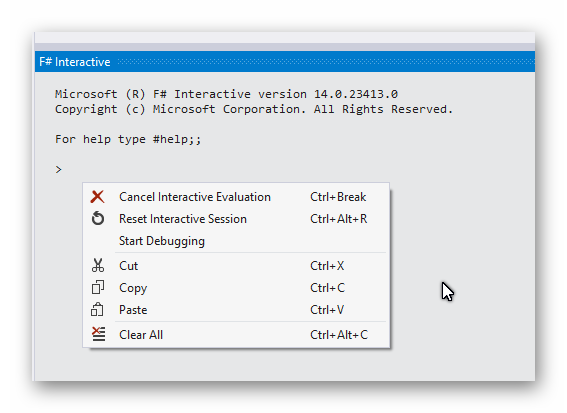
* Lade die Resourcen für die Übungen auf deinen Rechner
* Öffene das Solution FsharpSpartakiade2017.sln
* Führe die Kompilierung durch
* Akzeptiere

# Modul 1

* REPL
* Kommentare
* Ausdrucke
* Funktionen
* Rückgabe
* Einrückung
* Scripte
* Erste Listenfunktionen
* Pattern Matching

## Übung: Interactive

* Öffne das Interaktive Fenster (Ansicht/Andere Fenster)
* Den Inhalt der Sitzung kann man mit rechter Maustaste zurücksetzen und neu beginnen



Folgende Codezeilen im Fenster eingeben (Nicht im Codefenster, sondern direkt im interaktiven Fenster)

Bitte beachten: Bei jeder Eingabe, informiert (PRINT) uns das REPL über das Ergebnis der letzten Auswertung (EVAL).

open System.Windows.Forms;;  
let form = new Form();;  
form.Show();;  
form.Width;;  
form.Width <- 600;;  
let label = new Label();;  
form.Controls.Add(label);;  
label.Text <- "Hallo";;

Bitte beachten: unit() in F# ist das Nicht-Ergebnis Ergebnis. So als hätte void einen Wert und einen Namen.

## Übung: Kommentare

Kommentiere eine Zeile mit //

Kommentiere einen Block: Beginne den Kommentare mit (\* und beende diesen mit \*)

### Übung: Ausdrücke

In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei ErsteSchritte.fs

Definiere den Ausdruck a mit Wert 1

let a = 1

Bewege die Maus auf den Ausdruck a, der Typ wurde vom Compiler ermittelt.

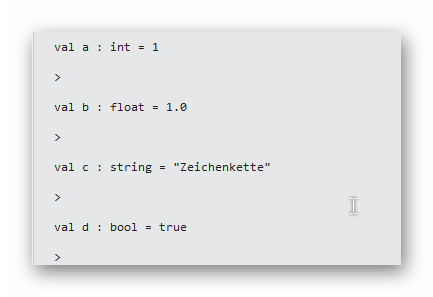
Erzeuge weitere Ausdrücke mit folgenden Typen

* Ausdruck b: float mit dem Wert 1,0
* Ausdruck c: String mit dem Wert „Zeichenkette“
* Ausdruck d: bool mit dem Wert true

Ich kann jetzt die einzelnen deklarierten Ausdrücke im öffnen REPL laden.

* Zeile markieren
* Alt+Enter

Bitte dies für die Ausdrücke a,b,c,d machen und den Inhalt des Interactive kontrollieren



Ich kann den Ausdruck in der Datei ändern und den neuen Wert erneut zum Interactive „Schicken“. Bitte probieren.

## Übung: Funktion

### Erste Funktion

In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei Funktion.fs

Erstelle eine Funktion mit folgenden Eigenschaften

* Name: add1
* Eingabe: ein Parameter x
* Berechnung: Addiere 1 zum Parameter 1

let add1 x = x + 1

Bitte beachten: Ich brauche kein Return. Der letzte Wert wird zurückgegeben.

Funktion zum Interactive schicken und Ausgabe kontrollieren

val add1 : x:int -> int

Wir erweitern die Funktion um einen Zwischen Parameter und führen eine „explizite“ Rückgabe aus.

let add1 x =

let z = x + 1

z

Bitte beachten: keine Klammern sind notwendig um die Funktion einzugrenzen.

Bitte beachten: lediglich die Angabe des Wertes z definiert diesen als Rückgabe Wert.

### Noch eine Funktion

Erstelle eine Funktion mit folgenden Eigenschaften

* Name: isEven
* Eingabe ein Parameter x
* Berechnung: ist x gerade?

Bitte beachten: = ist sowohl Zuweisungsoperator als auch Gleichheitsoperator.

let isEven x = x % 2 = 0

Bitte Schicke die isEven Funktion zum REPL.

Bitte prüfe ob der REPL die Funktion kennt.

## Übung: Script und REPL

In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei try.fsx

* Führe einen Reset in Interactive
* Referenziere die anderen Code Dateien mit Hilfe der Anweisung #load

#load "ErsteSchritte.fs"

#load "Function.fs"

#load "Aufgabe.fs"

* Lade die anderen Code Dateien mit Hilfe der Anweisung open

open ErsteSchritte

open Function

open Aufgabe

* Schicke den Inhalt der Datei zum Interactive. Jetzt können die Ausdrücke aus den Dateien verwendet werden.
* Zeige die Ausdrücke aus ErsteSchritte.fs im Interactive. Hierfür den Namen des Ausdrucks eingeben gefolgt von zwei Semikolons.
* Definiere einen Ausdruck aPlus1. Ausdruck führt die Funktion add1 mit a (aus ErsteSchritte.fs) als Parameter.

let aPlus1 = add1 a

Schicke aPlus1 zum REPL. Kontrolliere das Ergebnis:

val aPlus1 : int = 2

Bitte Beachten: Ich kann das Interactive Fenster immer mit der Reset Funktion neu laden.

## Übung: Script und FSI

In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei try.fsx. Füge folgende Zeile ein

printfn "print"

In Explorer, öffne die Datei Run\_ErsteSchritte.cmd und trage folgende Zeilen ein

fsi Try.fsx  
pause

In Die Ausgabe Funktion printfn funktioniert auch im Script.

## Übung: Listen Funktionen

Diese sind in .net auch vorhanden! Um die Sigantur einer Funktion erhalten gibt es zwei Wege

* Funktion in einer Code Datei oder Script schreiben und F12 Taste klicken. So gelange ich in eine Deklarationsdatei wo die Signatur und das Kommentar sind
* Funktion in Interactive schreiben, mit ;; ausführen

Bitte beachten: Das Einschließen in Klammern zeigt dass es sich um zusammenhängende Werte/Funktionen handelt.

So kann ich z.B. sehen dass die Funktion filter in Modul List folgende Signatur hat:

val filter : predicate:('T -> bool) -> list:'T list -> 'T list

val map : mapping:('T -> 'U) -> list:'T list -> 'U list

Bitte beachten: List hat viel mehr Funktion wir kommen noch dazu.

## Übung: Verketten

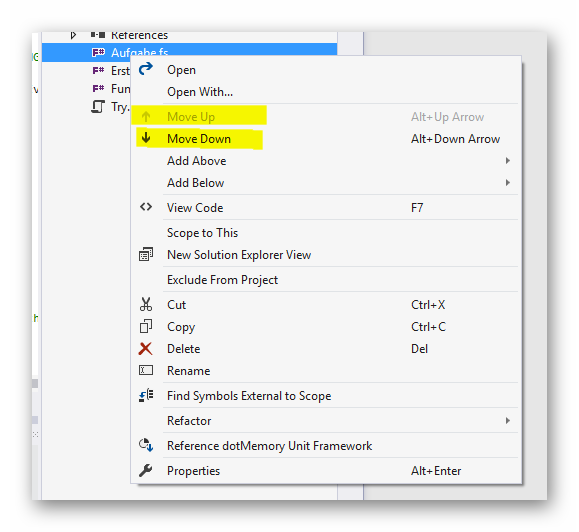
In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei Aufgabe.fs. Darin ist die Beschreibung der Aufgabe.

Erstelle einen Ausdruck der den gewünschten Wert beinhaltet:

* Analysiere die notwendigen Referenzen
* Positioniere die Datei Aufgabe so dass diese an die notwendigen Referenzen herankommen kann
* Referenziere die notwendigen Code Dateien

Die Lösung

* Die Funktionen die ich brauche sind add1 und isEven
* Beide befinden sich in der Datei Function.fs
* Von daher muss Aufgabe.fs „unter“ Function.fs stehen
* In VS, Aufgabe.fs wählen und die rechte Maustaste klicken.



* Bewege die Datei Aufgabe.fs bis diese die notwendigen Funktionen erreichen kann
* Importiere die Funktionen mit Hilfe des Schlüssel Wortes open
* Definiere einen Ausdruck summeOhneVerkettung der die Werte ermittelt ohne Verwendung des Verkettungsoperator
* Definiere einen Ausdruck summe der die Werte ermittelt mit Hilfe des Verkettungsoperator
* Teste den Wert aus dem Script Try.fsx heraus.

## Übung: Pattern Matching

In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei Patternmatching.fs.

Erstelle eine Funktion mit folgenden Eigenschaften

* Name: matchInt
* Eingabe ein Parameter x
* Berechnung:
  + Wenn x den Wert 0 hat, gibt „Null“ zurück
  + Wenn x den Wert 1 hat, gib „Eins“ zurück

let matchInt x =

match x with

| 0 -> "Null"

| 1 -> "Eins"

| \_ -> "Sonst"

Bitte beachten: Der Compiler gibt eine Fehlermeldung, wenn nicht alle möglichen Fälle behandelt sind.

Definiere bitte den allgemeinen Fall (wildcard) der dann ausgeführt wird, wenn x weder 0 ist noch 1.

# Modul 2: Datentypen

### Übung: Tuple Construction

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

type IntTuple = int \* int

let intTuple = 1,1

Bitte beachten: Typ Namen fangen mit einem Großbuchstaben an.  
Bitte beachten: im Englischen spricht man von *every int ‚times‘ every int*.

Weiteres Beispiel

type IntStringTuple = int \* string // every int 'times' every string

let intStringTuple = 1,"string"

Weiteres Beispiel

type TripleIntTuple = int \* int \* int

let tripleIntTuple = 1,2,3

Weiteres Beispiel Probiere weitere Werte. Schicke die Ausdrücke zum REPL.

Bitte beachten: Tuples haben keine Namen. Deshalb ist jedes Tuple aus *int times int* deklarationsgleich mit jedem anderen *int times int* Tuple

### Übung: Tuple Composition

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

type Complex = float \* float

let complex = 1.0,1.0

type Composition = IntStringTuple \* Complex

let composition = intStringTuple,complex

Bitte beachten: Ich kann komplexe Typen zusammenfügen zu neuen Typen.

### Übung: Tuple Deconstruction/Zerlegung

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

let cFirst,cSecond = complex

let kFirst,kSecond = composition

Bitte die Zeilen zum REPL schicken. und auswerten.

### Übung: Tuple Gleichheit

F# implementiert die Strukturelle Gleicheit mit.

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

let complexZweite = 1.0,1.0

let complexDritte = 2.0,1.0

Im REPL prüfe die Gleichheit dieser Ausdrücke. In F# dient = als Gleichheitsoperator.

### Übung: Tuple Pattern Matching

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

let matchTuple c =

match c with

| 0.0,0.0 -> "0.0,0.0"

| 1.0,1.0 -> "0.0,0.0"

| 1.0,2.0 -> "1.0,1.0"

| 1.0,\_ -> "1.0,Irgendwas"

| \_,\_ -> "sonst"

Bitte beachten: Ich kann hier auch die wildcard verwenden.

Bitte beachten: Wenn ich einen Tuple ad-hoch deklariere in einem Aufruf, benötigt man Klammern um den Wert einzugrenzen.

Bitte schicke diese Funktion zum REPL und teste unterschiedliche Werte.

### Übung: Tuple - .net API

Manche Funktionen im .net Framework haben eine eigne Implementierung in F# die die Möglichkeiten des Tuples nutzen.

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Tuple.fs. Trage folgende Zeilen ein:

open System

let showParseResult result =

match result with

| true,value -> sprintf "Geparster Wert ist %s" (value.ToString())

| false,\_ -> "Wert könnte nicht geparst werden"

let tryParseResult = Int32.TryParse "Irgendwas" |> showParseResult

let tryParseResult' = Int32.TryParse "1" |> showParseResult

In diesem Fall, hat Int32.TryParse folgende Signatur

string -> bool \* int

### Übung: Record Deklaration und Construction

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Record.fs. Trage folgende Zeilen ein:

type ComplexNumber = { Real: float; Imaginary: float; }

type GeoCoord = { Lat: float; Long: float; }

let complexNumber = { Real = 1.0; Imaginary = 1.0; }

let hamburg = { Lat = 53.55326; Long = 9.99300; }

Bitte beachten: Typ Namen fangen mit einem Großbuchstaben an.  
Bitte beachten: nach der Eingabe des Namen des ersten Bestandteils, erkennt der Compiler dass es sich um diesen Typ handelt.

Es ist jedoch möglich den Namen des Typs anzugeben bei der Konstruktion:

let complexNumber0 = { ComplexNumber.Real = 1.0; Imaginary = 1.0; }

Bitte beachten: Dies kann manchmal notwendig sein, wenn der Compiler den Typ nicht eindeutig finden kann.

Definiere eigne Records, erzeuge Werte, und prüfe diese im REPL.   
Versuche einen Record Ausdrück unvollständig zu deklarieren.  
Erzeuge zwei Record Typen mit den gleichen Bestandteilen und unterschiedliche Namen. Versuche Werte davon zu erstellen ohne den Namen des Record Typen zu erwähnen.

### Übung: Record Zerlegung und Klonen

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Record.fs. Trage folgende Zeilen ein:

let { Real = real0; Imaginary = imaginary; } = complexNumber // Alle Member

let { Real = real1; } = complexNumber // Einzelne

let real2 = complexNumber.Real // Point Style ist auch möglich für einen Wert

Bitte beachten: Ich muss die Ausdrücke nicht vorher definieren!

let complexNumber0 = { complexNumber with Imaginary = 3.0 }

Bitte beachten: Schlüsselwort with erlaubt es die einen Record zu klonen und dabei einzelne Werte neu zu belegen.

### Übung: Record Pattern Matching

In Visual Studio, gehe zur Anwendung Modul2 und öffne die Datei Record.fs. Trage folgende Zeilen ein:

let matchRecord c =

match c with

| { Real = 1.0; Imaginary = 1.0; } -> "Real = 1.0 & Imaginary = 1.0"

| { Real = 1.0 } -> "Real = 1.0"

| { Imaginary = 2.0 } -> "Imaginary = 2.0"

| \_ -> "sonst"