

Spartkiade 2017
F# Workshop
Übungen

Voraussetzungen

Windows

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

Erste Schritte

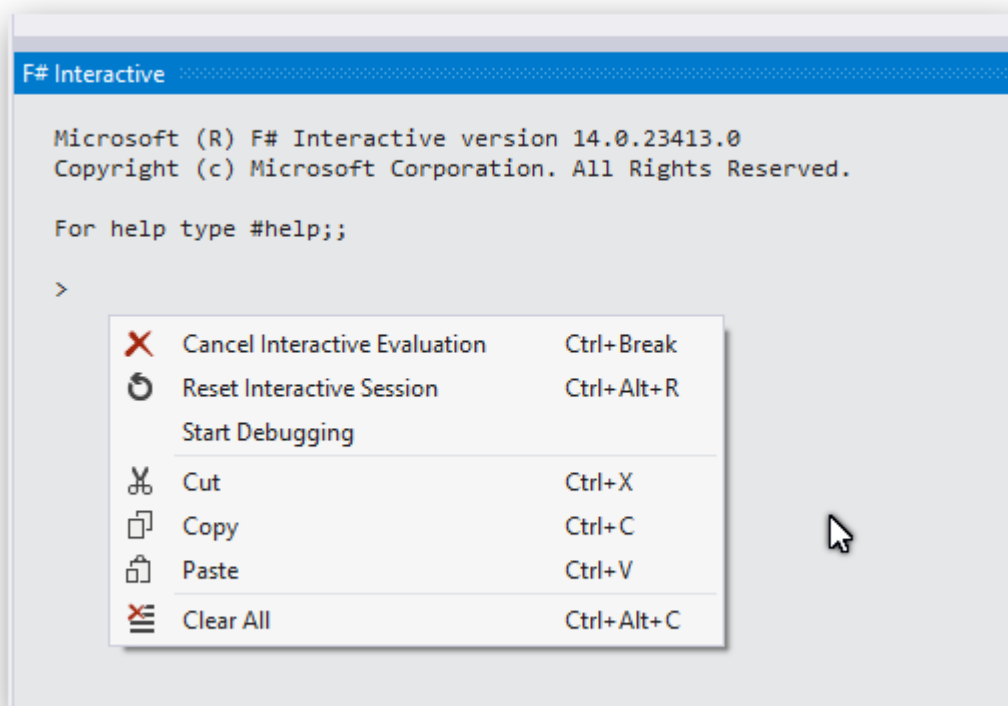
- Lade die Ressourcen für die Übungen auf deinen Rechner
- Öffne das Solution FsharpSpartakiade2017.sln
- Führe die Kompilierung durch
- Akzeptiere die Sicherheitsmeldung

Modul 1

- REPL
- Kommentare
- Ausdrücke
- 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.

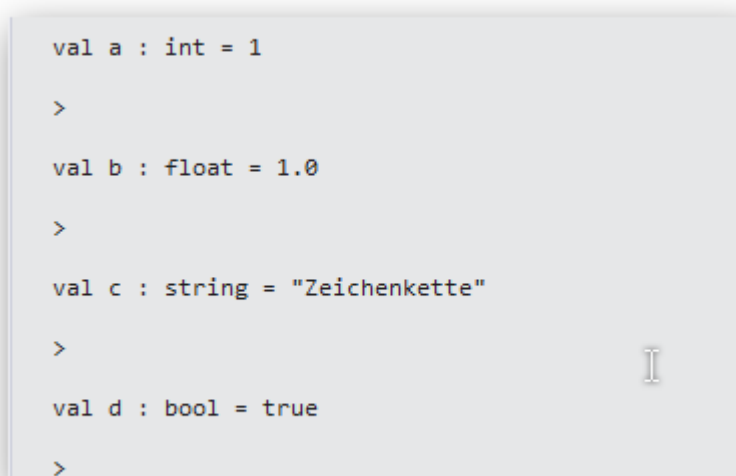
Erzeuge weitere Ausdrücke mit folgenden Typen

- 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 offenen REPL laden.

- Zeile markieren
- Alt+Enter

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



```
> val a : int = 1  
  
> val b : float = 1.0  
  
> val c : string = "Zeichenkette"  
  
> val d : bool = true  
  
>
```

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

Aufgabe: Ausdruck zu erstellen mit Sonderzeichen/Leerzeichen im Namen. In C#/VB.NET geht das ja nicht. In F# schon. Hierfür die Begrenzer `` verwenden.

Übung: Funktion

Erste Funktion

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

Erstelle eine Funktion mit folgenden Eigenschaften

- Name: add
- Eingabe: die Parameter x und y
- Berechnung: addiere x und y

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

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

Funktion zum Interactive schicken und Ausgabe kontrollieren

```
val add : x:int -> y:int -> int
```

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

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

Bitte beachten: Die Parameter der Funktion sind nur durch Leerzeichen getrennt.

Bitte beachten: keine Klammern sind notwendig um die Funktion einzugrenzen. Durch das Einrücken erkennt der Compiler die Begrenzungen der Funktion.

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.

Typ Erkennung

Erstelle folgende Funktionen

- Name: addOneToInt
- Eingabe: ein Parameter x
- Berechnung: Addiere 1 (int) zum Parameter x

- Name: addOneToFloat
- Eingabe: ein Parameter x
- Berechnung: Addiere 1 (float) zum Parameter x

- Name: addMoreToString
- Eingabe: ein Parameter x
- Berechnung: Verkette „More“ mit x

```
let addOneToInt x = x + 1
let addOneToFloat x = x + 1.0
let addMoreToString x = x + "More"
```

Erstelle diese Funktionen und schicke sie dann zum Interactive. Prüfe die Signaturen der jeweiligen Funktionen.

Erstelle eine Funktion mit folgenden Eigenschaften

- Name: length
- Eingabe: ein Parameter x
- Berechnung: Länge der Zeichenkette x

Bitte beachten: Compiler kann den Typ nicht erkennen. Eine Explizite Deklaration des Typs ist notwendig.

Kopiere das folgende in die Datei Function.fs

```
let apply f x :int = f x
```

Kopiere Schicke diese Funktion zum REPL. Untersuche die Signatur.

Funktion apply nimmt eine Funktion f und einen Parameter x entgegen.

Funktion f wird gegen x ausgeführt.

Das Ergebnis ist das Ergebnis der Funktion und ist vom Typ int.

Bitte beachten: Ich kann Funktionen als Parameter in einer Funktion verwenden. In C# kann ich dies mit Func<> auch machen. In F# ist lassen sich solche Deklarationen einfacher schreiben.

Übung: Script und REPL

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

Erstelle eine Funktion mit folgenden Eigenschaften:

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

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. Oder durch die Eingabe von #quit;; im interaktiven Fenster.

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: Verketteten

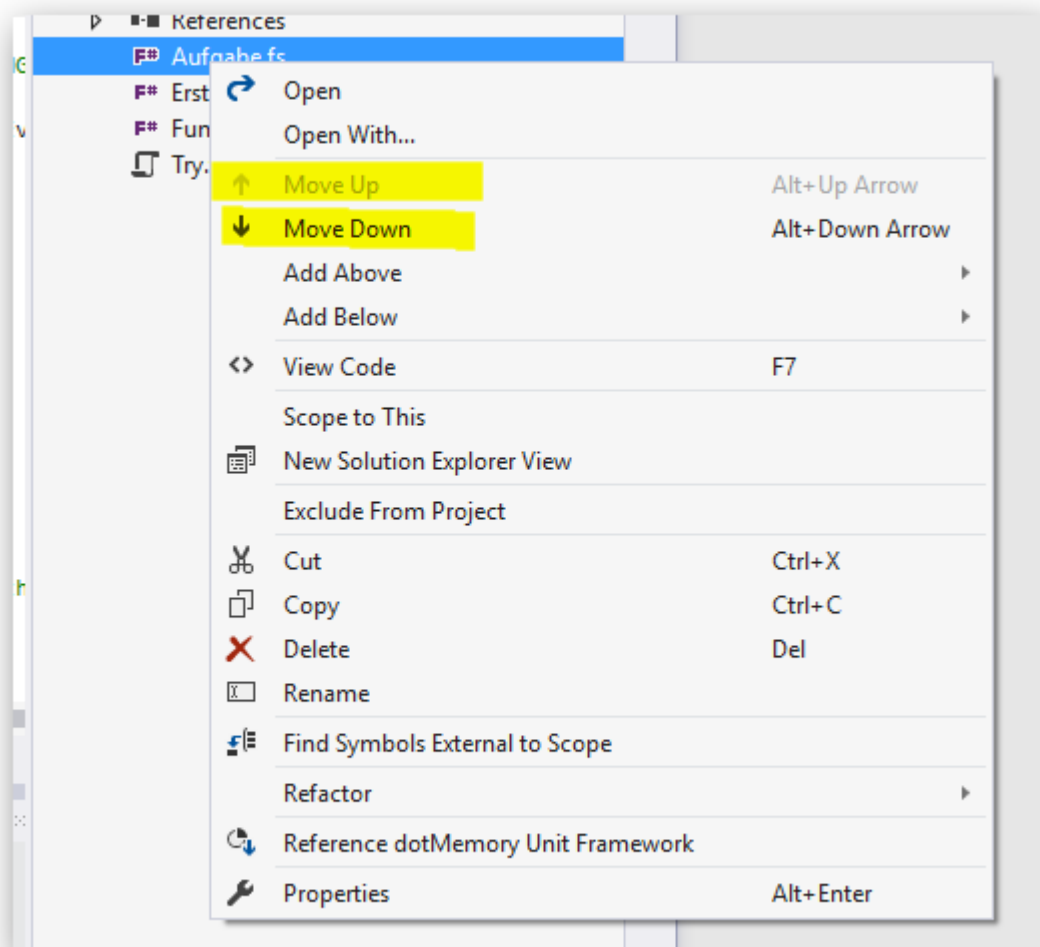
In Visual Studio, gehe zur Anwendung Modul1 und öffne die Datei Aufgabe.fs. Darin ist die Beschreibung der Aufgabe.

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"
```

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. In diesem Fall gibt „Sonst“ zurück.

Pattern Matching unterstützt auch OR, AND, sowie beliebige Operationen

```
let matchIntAndOr x =
  match x with
  | 0 | 1 | 2 -> "0 oder 1 oder 2"
  | x when x > 2 && x < 100 -> "Between 2 and 100" // when ist ein Guard
  | _ -> "sonst"
```

Übung: Listen

Übung: Listen erstellen

In visual Studio, gehe zur Anwendugn Modul1 und öffne die Datei Lisl. Erstelle folgende Listen:

```
module List

let leer = []

let list0 = 3::2::1::leer

let list1 = [6;5;4]

let list2 = list1 @ list0
```

Bitte beachten: das Operator :: fügt einen Element am Ende einer Liste. @ hingegen fügt zwei Listen zu einer Liste. Dies geschieht zu einem Preis. Welchen?

Frage: Warum kann ich folgendes Code nicht schreiben?

```
let list3 = list0 :: list1
```

Übung: Funktionen für Listen

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: Pattern matching für Listen

Pattern Matching für Listen in F# hilft dabei die Liste zu zerlegen. Pattern Matching hilft mir eine Liste folgendermaßen zu zerlegen

- Liste ist leer
- Liste hat nur ein Element
- Zerlege Liste in Head (erster Element) und Tail (Liste bestehend aus den restlichen Elementen)

Kopiere folgenden Code Abschnitt in die Datei List.fs in Modul1

```
let f l =
    match l with
    | [] -> "Leere Liste"
    | h::t ->
        sprintf "Erster Element ist %A und Liste hat noch %i Elemente" h (List.length
t)
```

Versuche Diverse Liste zu erstellen und die Funktion zu testen.

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 Gleichheit 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.

Für Tuples mit 2 Werten gibt es die speziellen Funktionen

```
let doubleIntTuple = 1,2
let first0 = fst doubleIntTuple
let sound0 = snd doubleIntTuple
```

Aufgabe: Erstelle eine third und fourth.

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-hoc 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
```

Bitte beachten: In diesem Fall wird der Tuple zerlegt beim pattern matching. Der erste Wert wird geprüft (true). Der zweite Wert, value, steht der Funktion zur Verfügung und kann angesprochen werden.

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 Wert unvollständig zu erstellen.

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
let { Real = real1; } = complexNumber
let real2 = complexNumber.Real
```

Bitte beachten: Ich muss die Ausdrücke real0 oder imaginary nicht vorher definieren!

Bitte beachten: Die Zerlegung ist auch für einzelne Werte möglich

Bitte beachten: Eine Zerlegung in Punkt-Stil, wie in C# ist ebenfalls möglich

```
let complexNumber0 = { complexNumber with Imaginary = 3.0 }
```

Bitte beachten: Schlüsselwort with erlaubt es beim Klonen eines Records einzelne Werte neu zu belegen. In diesem Fall hat Imaginary in complexNumber0 einen anderen Wert als in complexNumber

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"
```

Übung: DU Deklaration und Konstruktion

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

```
type Shape =
| Rectangle of width : float * length : float
| Circle of radius : float
| Square of side : float
```

Bitte beachten: Rectangle ist als Tuple deklariert. Das hätte auch ein Record sein können, nur dann muss dieser vorab deklariert werden.

Bitte folgende Code Zeilen eingeben

```
let recatngle = Rectangle(1.0,1.0)
let circle = Circle(2.0)
let square = Square(3.0)
```

Bitte beachten: Ich kann keinen Wert vom Typ Shape definieren. Man kann die union cases als Konstruktor Funktionen sehen, die als Ergebnis einen Wert vom Typ des union case zurückgeben. Z.B. kann ich den Konstruktor in einer List.map Funktion aufrufen.

Bitte folgende Codezeilen eingeben:

```
let rectangle1 = Rectangle(width = 1.3, length = 10.0)
```

Bitte beachten: Ich kann beim Erzeugen eines DU Wertes auch die Namen der Werte angeben.

Bitte folgende Codezeilen eingeben:

```
open Record
```

```

type DuExample =
| Empty
| Complex of ComplexNumber
| Coordinate of GeoCoord

```

```
let empty = Empty
```

Bitte beachten: in diesem Beispiel habe ich einen „leeren“ union case definiert. Diesen lässt sich konstruieren und verwenden.

Übung: DU – Single Case

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

```

type Longitude = Longitude of float
type Latitude = LatitudeConstructorFunction of float

```

```

let longitude = Longitude(9.993009)
let latitude = LatitudeConstructorFunction(53.553260)

```

Bitte beachten: in der ersten Zeile, Erster Longitude ist der Name des Typen, Zweiter ist der name des Constructors. Diese können auch unterschiedlich sein, ist aber bei single case union nicht üblich.

Aufgabe: Versuche bitte eine Longitude mit einer LATitude zu vergleichen, beide sind ja float. Geht das?

Übung: DU – Pattern Matching und Zerlegung

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

```

let area s =
    match s with
    | Rectangle (w,l) -> w*l
    | Circle(r) -> System.Math.PI*(r ** 2.0)
    | Square(s) -> (s ** 2.0)
let circleArea = area (Circle (5.0))
let rectangleArea = Rectangle(length = 5.0, width = 5.0) |> area

```

Bitte beachten: Pattern Matching ist ausschöpfend: ich muss alle Fälle berücksichtigen.

Bitte beachten: Die Funktion area gibt mir einen float als Ergebnis zurück. Ich transformiere damit einen Wert vom Typ Shape in eine Fläche, egal welcher Ausprägung dieser hat.

Übung: Option – Konstruktion

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

```

open DU

let s = Some "string"
let i = Some 1
let f = Some 1.0

let c = Some (Circle(1.0))

```

Bitte beachten: Ich kann einen Option mit records (oder auch DUs) erstellen. Das ist mit Reference Werten (Klassen in c#) und Nullable nicht möglich.

Bitte beachten: Option ist eine DU, daher TypName.UnionCaseName ist eine Funktion. In diesem Fall Some.

Option wird oft als Ergebnis eines Pattern Matching gegeben.

```
let matchForOption s =  
  match s with  
  | Some t -> t  
  | _ -> "Keine Option"  
  
let some = matchForOption (Some "Sure")  
let none = matchForOption None
```

Bitte beachten: das Pattern matching zerlegt den option (falls kein None) und bietet den darin enthaltenen Wert für die Pattern matching Funktion.

Übung: Option – Verwendung

Aufgabe: Erstelle eine Funktion mit folgenden Eigenschaften

- Name: parseInt
- Eingabe ein Parameter x
- Berechnung:
 - Wenn x sich als int parsen lässt dann eine option int
 - Wenn x sich nicht parsen lässt dann none

Kopiere das nachfolgende Code in die Datei Option.fs

```
let ``add ten using module functions`` numberAsText =  
  let parsed = parseInt numberAsText  
  let even = Option.filter (fun x -> x % 2 = 0) parsed  
  Option, otherwise None  
  let added = Option.bind (fun x -> x + 10 |> Some ) even  
  add 10, otherwise None  
  added
```

Modul 2: Funktionen und praktisches

Übung: Partial Application

In Visual Studio gehe zur Anwendung Modul3 und öffne die Datei PartialApplication.fs.

Erstelle eine Funktion mit folgenden Eigenschaften

- Name: add
- Eingabe: zwei Parameter x und y
- Berechnung: ermittle die Summe von x + y

Bitte beachte die Signatur der Funktion.

Versuche eine neue Funktion add5 zu erstellen mit Hilfe der Funktion add.

Übung: Komposition von Funktionen

In Visual Studio gehe zur Anwendung Modul3 und öffne die Datei PartialApplication.fs.

Gebe das nachfolgende Code ein

```
let compose f g x = f (g x)  
let times n x = n * x  
let multiply2 = times 2  
let add1Multiply2 = add5 >> multiply2
```

