Functional Programming in F#

Quantitative Strategies
Zbigniew Fiałkiewicz
Artur Tadrała

CREDIT SUISSE

F# Types: Classes

```
type MyType(x : int) =
        // primary constructor
        let y = x + 1
        do printfn "constructing MyType object %d" x
        //mutable state
        let mutable myState = 0
        //end of primary constructor
        new() = MyType(3) // secondary constructor
        // read-only property
        member this.X = x
        // equivalent read-only property
        member this.Y with get() = x
        // writable property
        member this.State
             with get() = myState
             and set(newValue) = myState <- newValue
        // method
        member this.MyMethod() = printfn "My method"
```

F# Types: Classes

```
//create instances(objects of type MyType)
let b1= MyType(0)
```
constructing MyType object 0
val b1 : MyType

```
let b2= MyType()
```
constructing MyType object 3
val b2 : MyType

```
//execute method
b1.MyMethod()
```
My method executed
val it : unit = ()

Inheritance

```
type BaseType() =
      member  x.BaseMethod() = printfn "Base method"

type DerivedType() =
     inherit BaseType()
       member x.DerivedMethod() = printfn "Derived method"
       member x.CallBase() =
            base.BaseMethod()
            printfn "CallBase done"
```

```
let d = DerivedType()
d.BaseMethod()
```
Base method
```
d.DerivedMethod()
```
Derived method
```
d.CallBase()
```
Base method
CallBase done

Interfaces

```
type MyInterface =
    abstract Calculate : int -> int -> int
    abstract GiveName : unit -> string
```

- a collection of abstract functions (no implementation)
- notice there is no constructor after type name
- provide a contract that all implementers need to adhere to
- often used on boundaries between components, to document clear expectations

Interface implementation

```
type MyImpl() =
    interface MyInterface with
        member this.Calculate x1 x2 = x1 + x2
        member this.GiveName () = "I am MyImpl"


    let z = MyImpl()

    //z.GiveName() won't work

    (z :> MyInterface).GiveName()
```

"I am MyImpl"

Object Expression

```
let k =
      { new MyInterface with
          member this.Calculate x1 x2 = x1 - x2
          member this.GiveName () = "I am Anonymous"
      }
```

k.GiveName()

"I am Anonymous"

Exceptions

```
exception FSharpEx of string

 // throwing exceptions
 let divThrows x y =
     match y with
     | 0 -> failwith "y should not be zero" //throws
     | _ -> x/y

 let divThrows2 x y =
     match y with
     | 0 -> raise (FSharpEx "y should not be zero")
     | _ -> x/y

 let divThrows3 x y =
     match y with
     | 0 -> raise (new ArgumentOutOfRangeException("y", "y != 0"))
     | _ -> x/y
```

Exceptions

```
// catching exceptions

 type DivResult =
    | Result of int
    | Error of string

 let divSafe x y =
     try
         divThrows3 x y |> Result
     with
     | FSharpEx str -> Error str
     | Failure str -> Error str
     | :? System.ArgumentException as ex -> Error ex.Message
```

Exceptions

```
// try finally
 let divfinally x y =
     try
         x / y
     finally
         printf "Dividing %d %d\n" x y

// to combine try-with and try-finally you must nest one in
another
```

printf / printfn

```
// based on Ansi-c prinf format instead of c#/.net standard
string.Format i.e. %d %f instead of {0} {1}

printf "This should print integer %d, and this string %s,
this %f float" 10 "test" System.Math.PI
```

This should print integer 10, and this string test, this 3.141593 floatval it : unit = ()

```
// statically typed
printf "expecting float %f" 10
#ERROR

// proper F# function
let f : string -> float -> bool -> unit =
        printf "%s %f %b“
val f : (string -> float -> bool -> unit)
```

Printf/printfn

```
// handles F# types
type printRecord = { a: string option; b: DivResult}
let recordValue = {a = Some "test"; b = Error "error" }
printfn "%A" recordValue
```

val recordValue : printRecord = {a = Some "test";
b = Error "error";}

```
// indentation
  for (i,s) in [ (1,"a"); (-22,"bb"); (333,"ccc"); (-
4444,"dddd") ] do
     printfn "|%*i|%-5s|" 20 i s
```

| 1|a |
| -22|bb |
| 333|ccc |
| -4444|dddd |

Printf/printfn

```
let petabyte = pown 2.0 50
printfn "float: %f exponent: %e compact: %g" petabyte
petabyte petabyte
```

float: 1125899906842620.000000 exponent: 1.125900e+015 compact: 1.1259e+15

```
// precision
printfn "2 digits precision: %.2f. 4 digits precision:
%.4f." 123.456789 123.456789
```

2 digits precision: 123.46. 4 digits precision: 123.4568.