Objekty

Každý objekt dědí právě od jednoho jiného objektu, na vrcholku hierarchie je `System.Object`, který se v F#pu jmenuje *`obj`*, který nedědí od nikoho, a má metody:

```
abstract this.Equals : obj -> bool
abstract this.GetHashCode : unit -> int
abstract this.ToString : unit -> string
member   this.GetType : unit -> System.Type
protected member   this.MemberwiseClone : unit -> obj
protected abstract this.Finalize : unit -> unit
static member Equals : obj * obj -> bool
static member ReferenceEquals : obj * obj -> bool
```

Každý objekt jde "zabalit" do *`obj`* pomocí *`box : 'a -> obj`*. Zpátky jde použít *`unbox<'a> : obj -> 'a`*. Ještě se hodí test, zda je v *`obj`* zabaleno něco zajímavého typu −− k tomu slouží operátor `:?`. Čtěte to jako "je typu?", stejne se jako `:` čte jako "je typu". Dá se použít jako samostatný výraz nebo v pattern matchingu:

```
let tryUnboxInt a =                    let tryUnboxInt = function
  if a :> int then Some (unbox<int> a)   | :? int as a -> Some (unbox<int> a)
              else None                  | _            -> None
```

Ve skutečnosti jde samozřejmě přetypovávat na libovolné předky a potomky. Na předky (a implementovaná rozhraní) jde přetypovat pomocí `:>`, na potomky je možné testovat pomocí `:?` a přetypovávat pomocí `:?>`. Tedy *`box`* a *`unbox`* jsou jenom:

```
let box x = x :> obj
let unbox<'subtype> (x:obj) : 'subtype = x :?> 'subtype
```

Samozřejmě `:?>` a *`unbox<'a>`* mohou vyhodit výjimku, pokud v předkovi není přetypovaný potomek.

Vlastní třídy se definují jako

```
type Jméno[<typové argumenty>]([argumenty[) =
  //inherit předek([argumenty konstruktoru])
  //let-definice
  //do-definice
  //member-definice
```

```
type Dict<'key, 'value when 'key : equality>(s : seq<'key * 'value>) =
  let mutable data = s |> List.ofSeq

  member this.Exists key = List.exists (fst >> (=) key) data
  member self.DelKey key = data <- List.filter (fst >> (<>) key) data
  member d.AddKey (key, value) = d.DelKey key
                                 data <- (key, value) :: data
  new() = Dict<'key, 'value>(Seq.empty)
  member this.Count = List.length data
  member this.Item with get key = let is_key(k,v)=if k=key then Some v else None
                                  List.pick is_key data
  member this.Item with set key value = this.AddKey (key, value)
```

Jednotlivé položky z definice třídy

♣ **`inherit`**: syntaxe je **`inherit`** `rodič([argumenty_konstruktoru])`. Není-l iuvedeno, dědí se od *`obj`*.

♣ **`let`**−definice: privátní **`let`** definice, mohou být i **`mutable`**. Jsou viditelné jenom uvnitř třídy.
Smí používat argumenty hlavního konstruktoru třídy. Inicializace se provede při volání hlavního konstruktoru.

- **`static let`**: hodnota, která je sdílena všemi instancemi třídy. Inicializace při volání statického konstruktoru.

```
type Cls() =
  static let mutable instances = 0
  do instances <- instances + 1
  static member Instances = instances
```

♣ **`do`**−definice: příkazy typu *`unit`* uvozené klíčkovým slovem **`do`**. Provedou se při volání hlavního konstruktoru.

- **`static do`**: příkazy, které se provedou při volání statického konstruktoru.

♣ **`member`** definice:
Definice viditelných metod, properties a dalších konstruktorů. Je možné používat funkce `a -> b -> c`, pro vnější svět se vždy zkompilují jako `a * b -> c`, ale uvnitř F#pu záleží na tom, jak je napíšete.

- další konstruktory: v F#pu je jeden konstruktor hlavní −− ten, jehož argumenty jsou uvedeny za jménem typu. Mohou existovat i další konstruktory, které ale nakonec musí volat hlavní konstruktor. Definují se jako
  `new` `(argumenty) = zde_konstrukce_objetu [`**`then`** `výraz]`
  Nepovinný dodatek **`then`** `výraz` má smysl, když chcete vykonat nějakou práci po konstrukci objektu.
- funkce: klasické členské funkce, definují se pomocí
  `member` `this.funkce argumenty = ...`
  Slovo `this` slouží k pojmenování instance a může být libovolné −− `self`, `x`, `v`, ...
  - **`static member`** jsou statické instance, neuvádějí jméno instance, tj. **`static member`** `funkce`.

- properties: vlastnosti, které mohou mít getter a setter. Getter má typ `unit -> 'a`, setter `'a -> unit`.
  Je možné je definovat najednou nebo zvlášť.
  ```
  member this.Length with get() = ...  member this.Length with get() = ...
  member this.Length with set l = ...                      and  set l = ...
  ```
  Pokud definujeme pomocí **member** funkci bez parametrů, je to automaticky getter.
- indexer: speciální property jménem `Item` s jedním argumentem navíc.
  Na její getter se převádí volání `a.[5]`, na její setter se převádí `a.[5] <- 6`
  Může mít jako argument n−tici, pak se jednotlivé položky oddělují čárkami, tj. `a.[4, 5, 6]`.

Argumenty **member** členů mají víc vlastností, než ty u **let** definice:

- pojmenované argumenty: při volání je možné specifikovat argumenty pořadím, nebo pomocí jejich jména:
  `dict.Add(value=1, key=1)`
  Přesné pravidlo je, že nejprve jsou argumenty zadané pořadím a nakonec následují pojmenované argumenty v libovolném pořadí.
- nepovinné argumenty: některé argumenty mohou být označené jako nepovinné:
  ```
  member this.Add(key, value, ?throwIfPresent)
  ```
  Takové argumenty nemusí být při volání specifikovány a chovají se jako *option*<'a>.
  Při jejich použití se hodí funkce `defaultArg : option<'a> -> 'a -> 'a`, která dostane option a pokud nic neobsahuje, vrátí svůj druhý argument, tj. default:
  ```
  member this.Add(key, value, ?throwIfPresent) =
    let throw = defaultArg throwIfPresent false
  ```
- **byref**, argumenty předávané odkazem: je možné předat argument odkazem pomocí
  ```
  type Incrementator(z) =
    member this.Increment(i : int byref) =
      i <- i + z

  let mutable x = 0
  Incrementator(4).Increment(&x)
  ```
- `ParamArray`, funkce s proměnlivým počtem argumentů: můžete vytvářet funkce s proměnlivým počtem argumentů. Jako poslední argument uveďte argument typu *obj*`[]` s atributem `ParamArray`:
  ```
  type Summator() =
    member this.SumAll([<ParamArray>] args : obj[]) =
      args |> Seq.sumBy (unbox<int>)

  Summator().SumAll(1, 2, 3, 4, 5)
  ```
- přetížené operátory: pro vlastní definované typy můžete přetěžovat operátory. Stačí je uvést jako statické member operátory. tj:
  ```
  type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y
    static member ( + ) (a:Vector, b:Vector) =
      Vector(a.X + b.X, a.Y + b.Y)
    static member ( * ) (a:Vector, c) =
      Vector(c * a.X, c * a.Y)
    static member ( * ) (c, a:Vector) =
      Vector(c * a.X, c * a.Y)
    static member ( * ) (a:Vector, b:Vector) =
      a.X * b.X + a.Y * b.Y
    static member (~- ) (a:Vector) =
      Vector(- a.X, - a.Y)
  ```

♣ Rovnost a porovnávání: F# má dvě typová omezení:

- `'a when 'a : equality` pro objekt, který umí testovat na rovnost. Lze s ním dělat:
  ```
  (=)  : 'T -> 'T -> bool when 'T : equality
  (<>) : 'T -> 'T -> bool when 'T : equality
  hash : 'T -> int when 'T : equality
  ```
- `'a when 'a : comparison` pro objekt, který umí porovnávat větší menší. Lze s ním dělat:
  ```
  (<)  : 'T -> 'T -> bool when 'T : comparison
  (<=) : 'T -> 'T -> bool when 'T : compariso@
  (>)  : 'T -> 'T -> bool when 'T : comparison
  (>=) : 'T -> 'T -> bool when 'T : comparison
  compare : 'T -> 'T -> int when 'T : comparison
  min     : 'T -> 'T -> 'T when 'T : comparison
  max     : 'T -> 'T -> 'T when 'T : comparison
  ```

Nové typy automaticky implementují tyto operátory, když je to vhodné:

- výčtové typy, recordy a algebraické datové typy implementují strukturální rovnost a porovnávání
- třídy implementují referenční rovnost a žádné porovnávání

Při vytváření typu je možné toto chování řídit následujícími atributy:
`StructuralEquality; StructuralComparison` −− rovnost i porovnávání se implemenují podle obsahu
`ReferenceEquality` −− rovnost podle referenční (fyzické) shodnosti
`NoEquality; NoComparison` −− žádná rovnost a porovnávání
`CustomEquality` −− vlastní rovnost, je třeba přetížit metody `Equals` a `GetHashCode`
`CustomComparison` −− vlastní rovnost, je třeba implementovat `IComparable` nebo `IComparable<'a>`
Jen některé z těchto kombinací dávají smysl (kompilátor si kdyžtak bude stěžovat).

♣ Vytváření objektů
Volání konstruktoru lze provést buďpomocí `System.String [|'a'|]` nebo kompletně pomocí
`new System.String([|'a'|])`.

Navíc v parametrech konstruktoru mohou být uvedeny i libovolné vlastnosti (properties) objektu, které jsou po vytvoření objektu nastavené na požadovanou hodnotu, tj. místo

```
let f = new System.Windows.Forms.Form()
f.Width <- 300
f.Visible <- true
```

lze psát jenom

```
let f = new System.Windows.Forms.Form(Width = 300, Visible = true)
```