♣ Virtuální metody
Metody objektů nejsou automaticky virtuální, tj. správná metoda se vybírá podle typu proměnné, na které je vyvolána. Pro vytvoření virtuální metody použijte `abstract member jméno : typ`. Konkrétní implementace potom uvádějte slovem `override`:

```
type A() =
  abstract member f : int -> int
  override this.f x = x
type B() =
  inherit A()
  override this.f x = x+1
```

Místo `override` lze použít i klíčové slovo `default`, je úplně totožné:

```
type A() =
  abstract member f : int -> int
  default this.f x = x
```

Třída musí implementovat všechny slíbené virtuální metody. Pokud to nedělá, lze ji označit za abstraktní třídu a nelze vytvářet její instance:

```
[<AbstractClass>]
type A =
  abstract interface System.IFormattable
  abstract member f : int -> int
```

Rozhraní čili interfaces

Stejně jako v .NETu existují kromě objektů ještě rozhraní. Rozhraní je sada metod. Díky tomu není problém s vícenásobnou dědičností –– rozhraní může vzniknout sloučením několika rozhraní a třída může implementovat libovolný počet rozhraní.

Definice nového rozhraní:

```
type IMoje =
  //inherit na rozhraní, od nichž "dědím"
  inherit System.IComparable
  inherit System.IFormattable
  //abstraktní metody rozhraní
  abstract member F : int -> int
```

Implementovat rozhraní jde jenom v definici třídy (nebo object expression) a udělá se to jako:

```
type A() =
  interface IMoje with
    member this.CompareTo(_) = 0
    member this.ToString(_,_) = ""
    member this.F(_) = 0
```

Object expression

Při vytváření objektů lze použít i tzv. object expression –– to je metoda vytvoření instance anynymního objektu založeném na už existujícím objektu či rozhraní.

Například pro vytvoření instance na základě objektu lze udělat

```
let o = { new obj() with member this.ToString() = "Hello world!" }
```

Statically resolved types

```
let add x y = x + y
add 4.4 5.5
add 4 5
```
Nejde, předchozí řádka otypuje add jako `add : float -> float -> float`

Pomocí `inline` způsobíme, že se kód vkládá na každé místo volání, a může v každém takovém místě mít jiný typ:

```
let inline add x y = x + y
add 4.4 5.5
add 4 5
(*val inline add : ^a ->  ^b ->  ^c
                  when (^a or ^b) : (static member (+) :  ^a *  ^b ->  ^c) *)
```

Typ `^a` je takzvaně statically resolved type –– konkrétní typ se určuje při kompilaci, a to na každém místě použití zvlášť. Takové typy jdou použít jenom v `inline` funkcích. Tyto typy mohou specifikovat tzv. member constraints, tj. mohou říkat "libovolný typ `a`, který má následující metody a properties":

```
let inline increment (x:^a) = (^a: (static member Increment : ^a -> unit) x)
let inline increment (x:^a) = (^a: (member Increment : unit -> unit) x)
let inline increment (x:^a) = (^a: (member Increment : int -> unit) (x,4))
```

Můžeme vytvořit sum, který sečte libovolný seznam:

```
let inline sum (xs : ^a list) = List.fold (+) (LanguagePrimitives.GenericZero :
^a) xs
```

Tato funkce má typ:

```
val inline sum : ^a list ->  ^a
                 when ^a : (static member (+) : ^a * ^a -> ^a)
                 and  ^a : (static member Zero : ^a)
```

List.sum má ve skutečnosti přesně tento typ.

```
val inline average : list:^T list -> ^T
                     when ^T : (static member ( + ) : ^T * ^T -> ^T)
                     and  ^T : (static member DivideByInt : ^T*int -> ^T)
                     and  ^T : (static member Zero : ^T)
```

Delegáti, Eventy a IEvent

V .NET jazycích je třeba předávat funkci jako objekt –– k tomu slouží `delegate`.
Tento typ můžeme používat i vytvářet v F#pu.

```
type action = delegate of unit -> unit
let callDelegate (a : action) = a.Invoke()
```

Víceargumentové delegáty je třeba vytvářet pomocí n–tic, tj. jako

```
type adder = delegate of int * int -> int
```

Můžeme vytvořit i delegát vracející funkci :)

```
type incrementer = delegate of int -> (int -> int)
let increment (del : incrementer) x = del.Invoke x
```

Vytvořit delegát lze přímo z F# funkce:

```
let my_incrementer = incrementer (fun i x -> x + i)
```

Delegát může obsahovat několik funkcí, které se vyvolají všechny. C# dělá syntaktický cukřík, můžeme si ho přidělat sami:

```
type action = delegate of unit -> unit
let inline (++) (x : ^a) (y : ^a) : ^a = System.Delegate.Combine(x, y) :?> ^a
let twice = let d = action (fun _ -> printfn "Fire!") in d ++ d
```

V .NETu jsou dále Eventy, ty jsou založené na delegátech. Jsou velmi podobné jako `MulticastDelegate`, ale mohou být v interfacech a lze je vyvolat jenom z třídy, kde jsou definovány.

I v F#pu existují `Event`, ale fungují trochu jinak než v C#pu.

Samotné používání eventů je stejné:

```
open System.Windows.Forms
let f = new Form(Visible = true)
f.Click.Add(fun evArgs -> printfn "Clicked")
```

Kromě metody `Add` lze používat `AddHandler` a `RemoveHandler`, které pracují s delegátem správného typu:

```
let h = System.EventHandler (fun sender evArgs -> printfn "Clicked from")
f.Click.AddHandler h
f.Click.RemoveHandler h
```

Samotný `Event` je v F#pu objekt jako každý jiný:

```
type Cls() =
  let event1 = Event<int>()

  [<CLIEvent>]
  member this.Event1 = event1.Publish

  member this.Fire(x) = event1.Trigger(x)

let c = Cls()
c.Event1.Add(fun i -> printfn "Event1 with %d" i)
```

S eventy je možné pracovat jako s proudem událostí pomocí metod v modulu `Event@:*)`

```
add       : ('T -> unit) -> IEvent<'Del,'T> -> unit
choose    : ('T -> 'U option) -> IEvent<'Del,'T> -> IEvent<'U>
filter    : ('T -> bool) -> IEvent<'Del,'T> -> IEvent<'T>
map       : ('T -> 'U) -> IEvent<'Del, 'T> -> IEvent<'U>
merge     : IEvent<'Del1,'T> -> IEvent<'Del2,'T> -> IEvent<'T>
pairwise  : IEvent<'Del,'T> -> IEvent<'T * 'T>
partition : ('T -> bool) -> IEvent<'Del,'T> -> IEvent<'T> * IEvent<'T>
scan      : ('U -> 'T -> 'U) -> 'U -> IEvent<'Del,'T> -> IEvent<'U>
split     : ('T -> Choice<'U1,'U2>) -> IEvent<'Del,'T> ->IEvent<'U1>*IEvent<'U2>

f.MouseMove
  |> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)
  |> Event.add ( fun evArgs -> f.BackColor <-
                           System.Drawing.Color.FromArgb(evArgs.X, evArgs.Y, 0))
```

Implementace je založena na observer design pattern, typy jsou i v .NETu:

```
type System.IObservable<'t> = interface
  abstract Subscribe : observer:IObserver<'t> -> IDisposable
end

type System.IObserver<'t> = interface
  abstract OnCompleted : unit -> unit
  abstract OnError : error:Exception -> unit
  abstract OnNext : value:'t -> unit
end
```

V F#pu lze s těmito hodnotami také pracovat jako s proudem událostí pomocí modulu `Observer`:

```
add       : ('T -> unit) -> IObservable<'T> -> unit
choose    : ('T -> 'U option) -> IObservable<'T> -> IObservable<'U>
filter    : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
map       : ('T -> 'U) -> IObservable<'T> -> IObservable<'U>
merge     : IObservable<'T> -> IObservable<'T> -> IObservable<'T>
pairwise  : IObservable<'T> -> IObservable<'T * 'T>
partition : ('T -> bool) -> IObservable<'T> -> IObservable<'T> * IObservable<'T>
scan      : ('U -> 'T -> 'U) -> 'U -> IObservable<'T> -> IObservable<'T>
split    : ('T->Choice<'U,'V>)->IObservable<'T>->IObservable<'U>*IObservable<'V>
subscribe : ('T -> unit) -> IObservable<'T> -> IDisposable
```