Object-oriented Programming in F#

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/˜blr/`

F# as an Object-oriented Programming Language

F# has an object-oriented part

Important to know something about this – accessing .NET libraries and services is done in an object-oriented fashion

F# has these kinds of *object types*:

- Concrete object types
- Object interface types

Concrete Object Types

F# objects can be both *mutable* and *immutable*

Methods are called *members*

The simplest case: extending a conventional F# type with *member declarations*

These provide a kind of interface to the data type

(They’re really just functions that take an argument of that type)

An Example

Extending a record data type with members, turning the records into objects:

```
type vector2D =
  { x : float; y : float }
  member v.Length = sqrt (v.x**2.0 + v.y**2.0)
  member v.Scale(k) = { x = k*v.x; y = k*v.y }
  member v.X_shift(x_new) = { v with x = x + x_new }
  static member Zero = { x = 0.0; y = 0.0 }
  static member X_vector(x_in) = { x = x_in; y = 0.0 }
```

(All examples in this presentation are adapted from Syme et. al. *Expert F# 2.0*) `v` is like “`this`”, or “`self`” in other languages. In F# you can choose whatever identifier you want

Members without arguments are *properties*, members with arguments are *methods*

Members can be *static*, meaning they operate on types rather than values. We’ll see an example soon

The above is an immutable object type: no record fields can be updated

This is the object type resulting from the declaration:

```
type vector2D =
  { x : float; y : float }
  member Length : float
  member Scale : k:float -> vector2D
  member X_shift : x_new:float -> vector2D
  static member Zero : vector2D
  static member X_vector : x_in:float -> vector2D
```

The member types become part of the object type

Creating an object and using its methods and properties:

```
let vec = { x = 3.0; y = 4.0 } // create an object "vec"
vec.Length                     // length of vec = 5.0
let vec2 = vec.Scale(2.0)      // create a new object "vec2"
vec2.Length                    // length of vec = 10.0
vec.Length                     // length of vec is still 5.0
vector2D.Zero                  // a new object with x, y = 0.0
```

Note the immutability: `vec.Scale(2.0)` creates a new object, the old one is not affected

Static members are applied to the type, not to values of that type

Members vs. Functions

Consider this alternative, F# with ordinary functions:

```
type vector2D =
  { x : float; y : float }
let Length v       = sqrt (v.x**2.0 + v.y**2.0)
let Scale v k      = { x = k*v.x; y = k*v.y }
let Zero           = { x = 0.0; y = 0.0 }
let X_vector x_in = { x = x_in; y = 0.0 }
```

Perfectly possible, does the same thing. But we lose the bundling of members and record type into an object type

A Variation

Any F# type can be enriched with members into an object type:

```
type Tree<'a> = Leaf of 'a | Branch of Tree<'a> * Tree<'a>
  member t.Fringe =
    match t with
    | Leaf x -> [x]
    | Branch (t1,t2) -> t1.Fringe @ t2.Fringe
```

It doesn’t have to be a record type

Members can be recursive

Constructed Classes

Goes beyond the simple object types where ordinary F# types are extended with members

Adds a possibility to define entities local to objects

These entities can be precomputed

An Example of a Constructed Class

`vector2D` using a constructed class:

```
type vector2D(x : float; y : float) =
  let len = sqrt (x**2.0 + y**2.0)
  member v.Length = len
  member v.Scale(k) = vector2D(k*x, k*y)
  member v.X_shift(x_new) = vector2D(x = x + x_new, y = y)
  static member Zero = vector2D(x = 0.0, y = 0.0)
  static member X_vector(x_in) = vector2D(x = x_in, y = 0.0)
```

`vector2D` is a *constructor* (in the OO sense): a function that creates a new object

```
let v = vector2D(3.0, 4.0)
```

`len` will be computed then `vector2D` creates the new object

Arguments to members can be given both by *position*, or by *name*

```
let v = vector2D(3.0, 4.0)         \\ by position
let v = vector2D(x = 3.0, y = 4.0) \\ by name
let v = vector2D(y = 4.0, x = 3.0) \\ by name, order does not matter
```

The resulting type:

```
type vector2D =
  new : x:float * y:float -> Vector2D
  member Length : float
  member Scale k:float -> Vector2D
  member X_shift x_new:float -> Vector2D
  static member Zero : vector2D
  static member X_vector : x_in:float -> Vector2D
```

Note “`new`”, tells the type of the `vector2D` constructor

Named and Optional Arguments

With named arguments, it is convenient to make arguments optional and have a default value for them

Named arguments can be used with all method calls

Optional arguments are preceded by “`?`”

An optional argument with type `'a` will have type `'a option` within the object type declaration

An argument given with value `v` will have the value `Some v` inside

If the argument is not given, it will have the value `None`

It is the responsibility of the programmer to write code that uses this distinction to provide a default value

Optional Arguments, Example

We turn `x` and `y` into optional arguments with default `0.0`:

```
type vector2D(?x : float; ?y : float) =
  let x = match x with
          | None   -> 0.0
          | Some v -> v
  let y = match y with
          | None   -> 0.0
          | Some v -> v
     .....
```

Note the new local definitions of `x` and `y` – not the same as the arguments `x` and `y`!

Optional Arguments, Continued

A builtin function to use with optional arguments:

```
defaultArg : ’a option -> ’a -> ’a
```

Its definition;

```
DefaultArg arg default =
  match arg with
  | None   -> default
  | Some a -> a
```

An example of its use:

```
let x = defaultArg x 0.0
let y = defaultArg y 0.0
...
```

Mutable Object Types

One idea with object-orientation is to encapsulate side-effects into objects

This reduces the risks with the side-effects

Side-effects means we should have mutable data inside objects

F# supports this

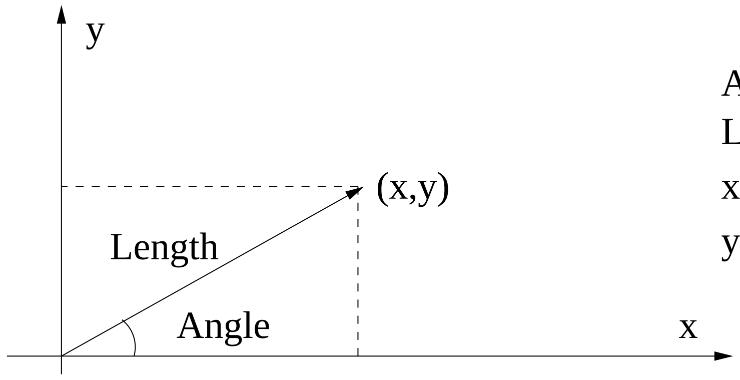
Object-type internal variables can be declared `mutable`

Members are defined with `get` and `set` methods:

- The `get` method returns the current value for the member
- The `set` method sets a new value for the member by setting new values for the object-internal mutable variables

An Example

The 2D-vector again, but now with two different views:



Angle = atan(y/x)
Length = sqrt(x**2 + y**2)
x = Length*cos(Angle)
y = Length*sin(Angle)

An object representing a 2D-vector will have x and y as mutable state

However, we will also provide methods for $Length$ and $Angle$

Getting $Length$ and $Angle$ will compute them from x and y

Setting $Length$ or $Angle$ will set x and y

Object Type Declaration

```
type mutVector2D(x : float; y : float) =
  let mutable current_x = x
  let mutable current_y = y
  member v.x with get () = current_x and set x = current_x <- x
  member v.y with get () = current_y and set y = current_y <- y
  member v.Length
    with get () = sqrt(current_x**2.0 + current_y**2.0)
    and  set len = let theta = v.Angle
                   current_x <- len*cos theta
                   current_y <- len*sin theta
  member v.Angle
    with get () = atan2 current_y current_x
    and  set theta = let len = v.Length
                     current_x <- len*cos theta
                     current_y <- len*sin theta
```

Resulting type:

```
type mutVector2D =
  new : float * float -> mutVector2D
  member x : float with get,set
  member y : float with get,set
  member Length : float with get,set
  member Angle: float with get,set
```

How to Use

```
> let v = mutVector2D(3.0, 4.0);;
val v : mutVector2D
> (v.x, v.y);;
val it : float * float = (3.0,4.0)
> (v.Length, v.Angle);;
val it : float * float = (5.0,0.927295218)
> v.Length <- 10.0;;
val it : unit = ()
> (v.x, v.y);;
val it : float * float = (6.0,8.0)
> (v.Length, v.Angle);;
val it : float * float = (10.0,0.927295218)
> v.x <- 1.0 ; v.y <- 1.0;;
val it : unit = ()
> (v.Length, v.Angle);;
val it : float * float = (1.414213562,0.7853981634)
```

Object Interface Types

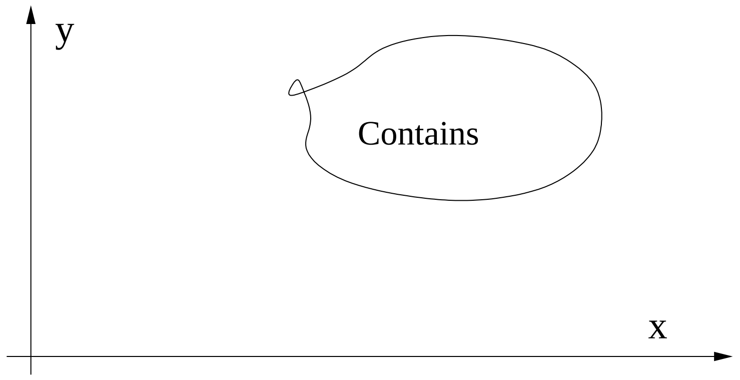
“Abstract” object type declarations, specify only members and their types, not their implementations

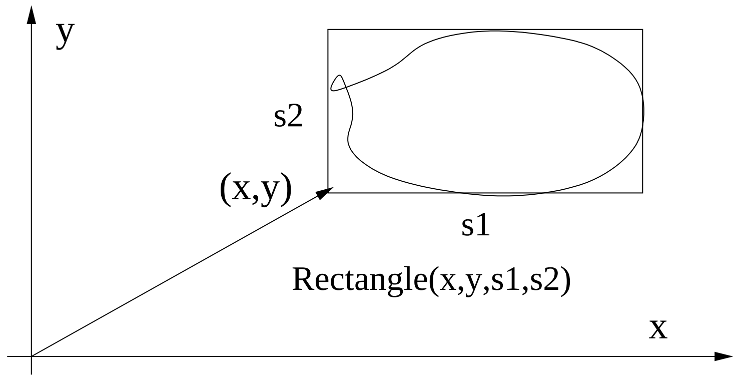
Concrete implementations are specified by separate declarations

By having different concrete implementations implement members differently, we achieve something similar to virtual methods

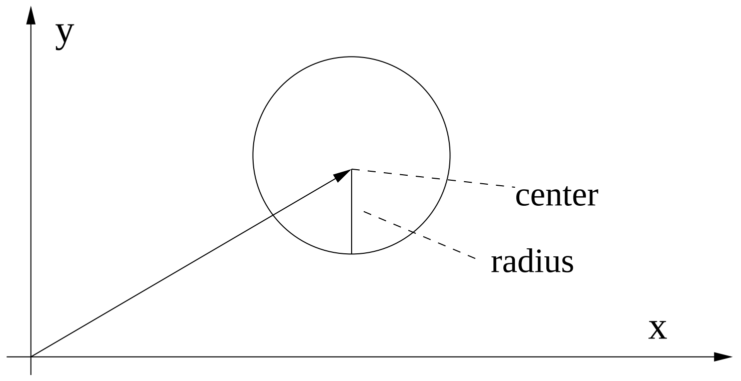
Example

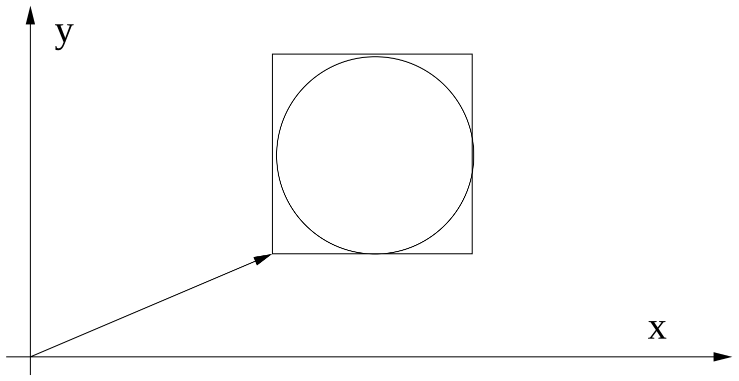
```
type Point = { X : float; Y : float }
type Rectangle = Rectangle of (float * float * float * float)

type IShape =
  abstract Contains : Point -> bool
  abstract Boundingbox : Rectangle
```



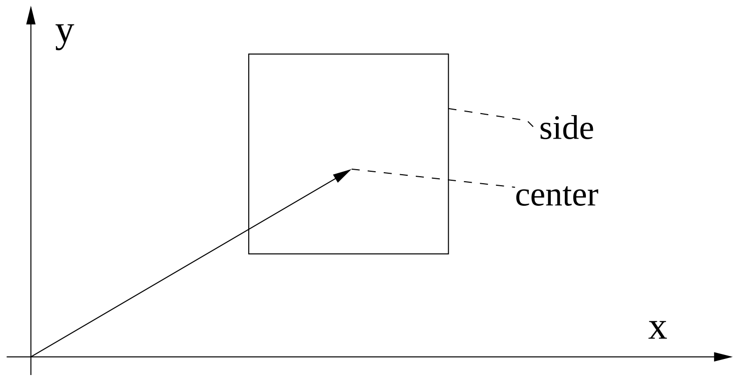


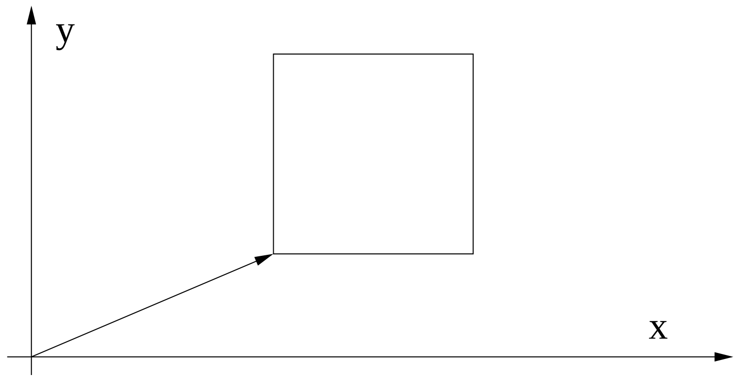
```
let circle(center:Point, radius:float) =
  { new IShape with
      member x.Contains(p:Point) =
        let dx = p.X - center.X
        let dy = p.Y - center.Y
        sqrt(dx**2.0 + dy**2.0) <= radius
      member x.Boundingbox =
        Rectangle (center.X - radius,center.Y - radius,
                   2.0*radius,2.0*radius) }
```





```
let square(center:Point, side:float) =
  { new IShape with
      member x.Contains(p:Point) =
        let dx = p.X - center.X
        let dy = p.Y - center.Y
        abs(dx) <= side/2.0 && abs(dy) <= side/2.0
      member x.Boundingbox =
        Rectangle (center.X - side/2.0,center.Y - side/2.0,
                   side,side) }
```





Object Expressions

`circle` and `square` are functions whose bodies are *object expressions* (the “`{ new IShape with ...}`”)

Object expressions are used to specify implementations for interface types

An object expression must give an implementation for each member of the interface type

In our example, the functions `circle` and `square` provide implementations of the `IShape` interface

Inheritance

Object interface types can inherit from each other

Thus, hierarchies of such types can be built

The keyword “`inherit`” specifies inheritance

```
type Blahonga =
  abstract xxx : ...

type FooBar =
  inherit Blahonga
  abstract yyy : ...
```

An implementation of `FooBar` must implement both `xxx` and `yyy`

Functional Programming Techniques and Object Expressions

Object expressions can take functions as arguments

In that way, object expressions can be abstracted further

An example:

- A simple interface `TextOutputSink` defining two methods: for writing a character, and for writing a string
- A function `SimpleOutputSink` returning an implementation

(See next page)

```
type TextOutputSink =
  abstract WriteChar : char -> unit
  abstract WriteString : string -> unit

let SimpleOutputSink(writechar) =
  { new TextOutputSink with
      member x.WriteChar(c) = writechar c
      member x.WriteString(s) =
        for c in s do writechar c }
```

`SimpleOutputSink` defines the simple pattern to write a string by writing it character by character