This topic outlines the built-in features that support object-oriented programming in F#.

Recommended reading: Expert F# Chapter 6.
“Rolling our own objects” is sufficient for many purposes in a functional language with mutable reference cells and records.

However, in order to interoperate well with other languages, a standard object system needs to be adopted.

F# includes a form of objects based closely on .NET.

These objects are also very similar to Java, with support for:
- Classes.
- Constructors.
- Fields/Instance variables
- Methods
- Static fields and methods
- Subtyping
- Access to “this”
- Properties (not in Java)
- Overloading
- (Single) Inheritance.
- Interfaces and abstract classes.

The concepts involved should be familiar, so we will mostly just see examples of the syntax, and a few features that differ from Java.
An example of a class definition

- The following F# code defines a class for called Vector2D

```fsharp
type Vector2D(dx: float, dy: float) =
    let len = sqrt(dx * dx + dy * dy)
    member this.DX = dx
    member this.DY = dy
    member this.Length = len
    member this.Scale(k) = Vector2D(k*dx, k*dy)
    member this.ShiftXY(x,y) = Vector2D(dx=dx+x, dy=dy+y)
    static member Zero = Vector2D(dx=0.0, dy=0.0)
```

- The (standard) constructor takes the two arguments dx and dy and returns an object with type Vector2D.
- The arguments are automatically stored along with the calculated length, just like the when creating a function closure.
- Other (instance) variables can be defined using let (like len above).
- Members (or methods) are essentially F# functions that also give a name for the object the method is called on – this in this case.
- A member with no arguments (like DX, DY, Length) is a get method for a property. (See the end of this topic for set methods.)
- Static members belong to the class, rather than a particular instance.
Creating and using an instance

- With this class defined, we can create an instance via the constructor, and use the members.

  ```
  let v = Vector2D(3.0, 4.0);
  val v : Vector2D
  > v.Length;;
  val it : float = 5.0
  > v.Scale(2.0).Length;;
  val it : float = 10.0
  ```

- The type signature for Vector2D the right may be useful (the interactive top-level prints this)

- Note: `new` is generally optional.

  ```
  type Vector2D =
  class
      new : dx:float * dy:float -> Vector2D
      member Scale : k:float -> Vector2D
      member ShiftX : x:float -> Vector2D
      member ShiftXY : x:float * y:float -> Vector2D
      member ShiftY : y:float -> Vector2D
      member DX : float
      member DY : float
      member Length : float
  static member OneX : Vector2D
  static member OneY : Vector2D
  static member Zero : Vector2D
  end
  ```
Actions and multiple constructors

- Another example that shows two more features:
  - actions performed by the standard constructor (via `do`)
  - a second constructor (via `new`) which must call the primary one

```haskell
type UnitVector2D(dx,dy) =
    let tolerance = 0.000001
    let length = sqrt(dx * dx + dy * dy)
    do if abs(length - 1.0) >= tolerance then
        failwith "not a unit vector";
    member v.DX = dx
    member v.DY = dy
new() = UnitVector2D (1.0,0.0)
```
**Interfaces**

- An interface is any class type with only abstract members.
- `abstract` specifies members that can have many different implementations.
- `new` directly creates an object implementing an interface (like records).
- Interfaces can also be inherited (like “implements” in Java, next 2 slides).

```csharp
open System.Drawing

type IShape =
    abstract Contains : Point -> bool
    abstract BoundingBox : Rectangle

let circle(center:Point,radius:int) =
{ new IShape with
    member x.Contains(p:Point) =
        let dx = float32 (p.X - center.X)
        let dy = float32 (p.Y - center.Y)
        sqrt(dx*dx+dy*dy) <= float32 radius
    member x.BoundingBox =
        Rectangle(center.X-radius, center.Y-radius, 2*radius+1, 2*radius+1 )
}
```
Subtyping and sequences

- Inheritance of implementations tends not to be used as often in F# - composition of functions is preferred.
- However, subtypes are common, particularly subtypes of .NET types
- Here are the definitions for the .NET enumerables and collections
  - A `seq` is defined as an abbreviation for `IEnumerable<'a>'

```fsharp
type ISequencer<'a> =
    abstract Current : 'a
    abstract MoveNext : unit -> bool

type ISequence<'a> =
    abstract GetEnumerator : unit -> ISequencer<'a>

type ICollection<'a> =
    inherit ISequence<'a>  // Inheritance implies subtyping
    abstract Count : int
    abstract IsReadOnly : bool
    abstract Add : 'a -> unit
    abstract Clear : unit -> unit
    abstract Contains : 'a -> bool
    abstract CopyTo : 'a[] * int -> unit
    abstract Remove : 'a -> unit
```
Implementation inheritance

- Here’s an example that includes implementation inheritance.
- The first class has abstract and implemented (default) members.
- The second inherits the first, replacing (via override) one inherited member and implementing 2 others, leaving one method abstract.

```fsharp

type TextOutputSink() =
    abstract WriteChar : char -> unit
    abstract WriteString : string -> unit
    default x.WriteString(s) = s |> String.iter x.WriteChar

open System.Text

type ByteOutputSink() =
    inherit TextOutputSink()
    abstract WriteByte : byte -> unit
    abstract WriteBytes : byte[] -> unit
    default sink.WriteChar(c) =
        sink.WriteByte(Encoding.UTF8.GetBytes([|c|]))
    override sink.WriteString(s) =
        sink.WriteByte(Encoding.UTF8.GetBytes(s))
    default sink.WriteBytes(b) =
        b |> Array.iter (fun c -> sink.WriteByte(c))
```

Other object-oriented features

- `box(x)` converts a type like `int` to a subtype of `System.Object`.
- `unbox<type>(x)` attempts to convert an object to `type`.
- `x :> type` casts to a supertype at compile time.
- `x :?> type` attempts to cast to a subtype at runtime.
- Members can also be defined for unions and records.

```csharp
type Tree<'a> =
    | Tree of 'a * Tree<'a> * Tree<'a>
    | Tip of 'a

member t.Size =
  match t with
  | Tree(_,l,r) -> 1 + l.Size + r.Size
  | Tip _ -> 1
```
Overloading

- Static methods for operators can be overloaded, but often the types of the arguments need to be included in the definition, particularly when many members have the same number of arguments.

- In the following types for vectors and points are defined, with subtraction on points returning a vector, and subtracting a vector from a point returning a vector.

```plaintext
type Vector =
{ DX:float; DY:float }

member v.Length = sqrt(v.DX*v.DX + v.DY*v.DY)

type Point =
{ X:float; Y:float }

static member (-) (p1:Point,p2:Point) =

static member (-) (p:Point,v:Vector) =
{ X=p.X-v.DX; Y=p.Y-v.DY }
```
Named and optional arguments

- Type members and constructors can have named and optional args.
- These are particularly useful for allowing APIs to be used easily.
- Within the body, optional arguments have type of the form 'a option.
- A useful library function is defaultArg : 'a option -> 'a -> 'a

```csharp
open System.Drawing

type LabelInfo(?
  text: string
, ?
  font: Font
) =
  let text = defaultArg text ""
  // if text is None use ""
  let font = match font with
    | None -> new Font(FontFamily.GenericSansSerif, 12.0f)
    | Some v -> v

  member x.Text = text
  member x.Font = font

LabelInfo ( text= "Hello World" );
val it : LabelInfo =
{ Font = [Font: Name=Microsoft Sans Serif, Size=12]:
  Text = "Hello World" 
}
```
Extra arguments and get/set for properties

- Unused named arguments are converted into calls to modify properties.
- This is useful to make imperative APIs appear more functional. Thus:

```csharp
open System.Windows.Forms
let form = new Form(Visible=true, TopMost=true, Text="Welcome")
```

- Actually means:

```csharp
let form =
    let tmp = new Form()
    tmp.Visible <- true
    tmp.TopMost <- true
    tmp.Text <- "Welcome"
    tmp
```

- Properties are actually just members that have a get and/or set member, as in the following example (with mutable variables via let and <-).

```csharp
type LabelInfoWithPropertySetting() =
    let mutable text = ""
    // the default
    let mutable font = new Font(FontFamily.GenericSansSerif, 12.0f)
    member x.Text with get() = text and set(v) = text <- v
    member x.Font with get() = font and set(v) = font <- v
```