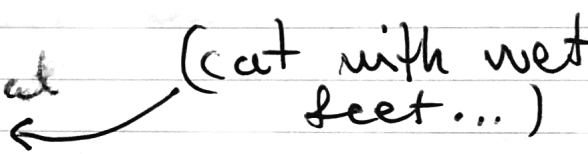


Byref / Inref / Outref

- Byref is a low level construct used for a) optimization b) communication with the outside world.
 - Byref's are managed (restrictive) pointers
 - inref is read-only
 - outref is write-only
 - byref is bidirectional  (cat with wet feet...)
- "Don't use more power than necessary"

```
let f (x : byref<string>) =  
  x ← "hello from f"
```

```
let mutable s = ""  
f &s
```

Exceptions

- Exceptions are necessary to deal with failed IO. Also a lot of .NET (C#) are tossing exceptions around like poisonous candy.
- Exceptions inherit from System.Exception (if you need to roll your own)
- we can raise exceptions using raise, failwith and invalidArg

let fails x = failwith "oh no!"

```
[ try  
  fails 1.0  
with  
  | ex →  
    printfn "%s" ex.message  
  0.0
```

```
[ try  
  fails 42.0  
finally  
  printfn "whatever."  
  1.0
```

Units of Measure

- Units of measure prevent us from comparing apples and oranges
- We can attach units of measure to numeric types
- Units can be multiplied, divided and powered ($*$, $/$ and $^$), and the compiler knows how to simplify them.

[<Measure>] type cm

[<measure>] type cl = cm^3

[<Measure>] type cm2 = cm^2

let a = 2.0 <cm>

let b = 1.0 <cl>

let (c: float <cm2>) = b / a

let d = a + b // error!

- One can also create generic units.

Loose ends and odd bits

- Module and type level access control:

Everything is public by default, but can be controlled using: public, private, internal

- Sometimes we write an implicitly generic function, which we use in different contexts, and we get a type error.

This happens because the function gets specialized too early!

```
let add x y = x + y
```

```
let f(x:int) = add x x
```

```
let g(x:float) = add x x
```

- This can be solved by inlining:

```
let inline add x y = x + y
```

- When creating IDisposable objects you must use the new keyword. Instead of let you can also use the use keyword.

- Loops: for i = 1 to 10 do ...

```
for i in enumerable do ...
```

```
while (true) do ...
```

Active patterns

- Active patterns allow us to program pattern matching!
- Can be a very powerful technique to create compact and readable code!

← banana chip

```
let (| Even | odd |) (x : int) =  
  if x % 2 = 0 then Even else Odd
```

```
let f x =  
  match x with  
  | Even → ...  
  | Odd → ...
```

```
let (| Pairup |) x = (x, x+1)
```

```
let f x =  
  match x with  
  | Pairup (a, b) → ...
```

- Incomplete patterns must return Option:

```
let (| Email | _ |) (s : string) =  
  if Regex.Match(s, ...) then Some s  
  else None
```