Active patterns

- Special functions, also called **pattern recognizers**, and are defined inside banana clips `(|...|)`.
- Active patterns (AP) are used in pattern matching, but they can also be called as ordinary functions.
- Single-case APs - used to **convert**/**decompose** values
    - `fsharp let (|RGB|) (col: System.Drawing.Color) = (col.R, col.G, col.B)`
- Multi-case APs - **partitioning** the input value space into multiple closed groups
    - `fsharp let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd`
- Partial APs - open ended partitioning where we only are interested in some of the possible input values. These APs must return an `option`.
    - `let (|INT|_|) (str: string) = let mutable intvalue = 0 if System.Int32.TryParse(str, &intvalue) then Some(intvalue) else None`
    - Need a "catch everything else" case in pattern matching (_)
        - `match str with | INT i -> printfn "Integer found: %d" i| ...| _ -> printfn "Input string did not match: %s." str`
- Parameterized APs - APs can receive arguments.
    - `fsharp let (|REGEX|_|) regex str = let m = Regex(regex).Match(str) if m.Success then Some (List.tail [ for x in m.Groups -> x.Value ]) else None match str with | REGEX "(\d{1,2})-(\d{1,2})-(\d{1,4})$" [INT m; INT d; INT y] -> new System.DateTime(y, m, d) | ... -> ...`

Units of measure

- Common units defined in `Microsoft.FSharp.Data.UnitSystems.SI.UnitNames`
- Use `[<Measure>]` on a type
- Use units on numeric types - `12<meter>`, `1.25<second>`, etc.
- Arithmetic involving values with units is checked, and values receive the computed units automatically.
    - `let velocity (dist: float<meter>) (time: float<second>) = dist/time`

Object orientation - Part I

- Classes
    - Defining classes
        - `type Pair(x: int, y: int) = let registerSecret(x, y, extract) = () let secret = x+y do registerSecret(x, y, secret) static member ZeroPair() = (0, 0) member self.CheckSecret(key) = secret=key`
    - Default constructor - `Pair(x: int, y: int)`
    - Local state - `secret`

    - Object initialization - `do ....`
    - Instance and static members
- Interfaces
    - Declaration
        - `type Interface1 = abstract member Method1 : int -> int`
    - Implementation
        - `type MyType(x: int) = ... interface IMyInterface with member this.Method() = ...`
        - Can expose interface members as regular members as well
        - `let createMyObject(x: int) = { new IMyInterface with member this.Method() = ... }`
    - Calling interface methods
        - Upcast to the interface type first
            - `(myObject :> IMyInterface).Method(...)`
- Inheritance
    - Declaration
        - `type MyDerived(...) = inherit MyBase(...)`
        - Only one direct base class is allowed => **no multiple inheritance**
        - Use `base` to refer to the base class instance in your derived instance methods.
        - The local state of the base class (`let` bindings, object initialization, etc.) is not available/accessible in a derived class.

What's next

- Object orientation - Part II: default/virtual/abstract methods, overrides, etc.
- Computation expressions
    - Asynchronous programming
    - Queries
- Quotations
- Scenarios
    - Relational data access - different ways to talk to a DB
    - Single Page Applications (SPAs) - write your client-side code in F#
    - Microservices, working with JSON
    - Structuring, organizing your code
        - Interfaces vs inheritance
    - Interoperating with C#
    - `<put your scenario here>`