

Programmering og Problemløsning

16.1: Objektorienteret design og programmering

Mere om dobbelt List.map

Alle kombinationer af bogstaver og tal:

```
> List.map (fun d -> "a"+d) ["1"; "2"; "3"];;  
val it : string list = ["a1"; "a2"; "a3"]
```

```
> List.map (fun l -> List.map (fun d -> l+d) ["1";"2";"3"]) ["a"; "b"];;  
val it : string list list = [["a1"; "a2"; "a3"]; ["b1"; "b2"; "b3"]]
```

Alle kombinationer af funktioner og tal:

```
> let indToRel = [  
    fun elm -> (elm,0); // South by elm  
    fun elm -> (-elm,0); // North by elm  
    fun elm -> (0,elm); // West by elm  
    fun elm -> (0,-elm) // East by elm  
]  
- List.map (fun e -> List.map e [1..7]) indToRel;;  
val it : (int * int) list list =  
  [[(1, 0); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0); (7, 0)];  
   [(-1, 0); (-2, 0); (-3, 0); (-4, 0); (-5, 0); (-6, 0); (-7, 0)];  
   [(0, 1); (0, 2); (0, 3); (0, 4); (0, 5); (0, 6); (0, 7)];  
   [(0, -1); (0, -2); (0, -3); (0, -4); (0, -5); (0, -6); (0, -7)]]
```

Mere om dobbelt List.map

Alle kombinationer af funktioner og tal:

```
> let indToRel = [  
  fun elm -> (elm,0); // South by elm  
  fun elm -> (-elm,0); // North by elm  
  fun elm -> (0,elm); // West by elm  
  fun elm -> (0,-elm) // East by elm  
]  
- List.map (fun e -> List.map e [1..7]) indToRel;;
```

Kombinationer med alternative List.map function og currying:

```
> let altMap lst e = List.map e lst  
- List.map (altMap [1..7]) indToRel;;  
val it : (int * int) list list =  
  [[(1, 0); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0); (7, 0)];  
   [(-1, 0); (-2, 0); (-3, 0); (-4, 0); (-5, 0); (-6, 0); (-7, 0)];  
   [(0, 1); (0, 2); (0, 3); (0, 4); (0, 5); (0, 6); (0, 7)];  
   [(0, -1); (0, -2); (0, -3); (0, -4); (0, -5); (0, -6); (0, -7)]]
```

Mere om dobbelt List.map

Kombinationer med alternative List.map function og currying:

```
> let altMap lst e = List.map e lst
- List.map (altMap [1..7]) indToRel;;
val it : (int * int) list list =
  [[(1, 0); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0); (7, 0)];
   [(-1, 0); (-2, 0); (-3, 0); (-4, 0); (-5, 0); (-6, 0); (-7, 0)];
   [(0, 1); (0, 2); (0, 3); (0, 4); (0, 5); (0, 6); (0, 7)];
   [(0, -1); (0, -2); (0, -3); (0, -4); (0, -5); (0, -6); (0, -7)]]
```

Kombinationer med swap og currying:

```
> let swap f a b = f b a
- List.map (swap List.map [1..7]) indToRel;;
val it : (int * int) list list =
  [[(1, 0); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0); (7, 0)];
   [(-1, 0); (-2, 0); (-3, 0); (-4, 0); (-5, 0); (-6, 0); (-7, 0)];
   [(0, 1); (0, 2); (0, 3); (0, 4); (0, 5); (0, 6); (0, 7)];
   [(0, -1); (0, -2); (0, -3); (0, -4); (0, -5); (0, -6); (0, -7)]]
```

Upcasting og downcasting med [<AbstractClass>]

Upcasting i pieces.fs

```
let pieces = [|  
  king (White) :> chessPiece;  
  rook (White) :> chessPiece;  
  king (Black) :> chessPiece |]
```

Upcasting:

- + vi kan maskere forskellige brikker som same type i listen
- Når vi skal bruge brikkernes specielle træk skal vi downcaste, men til hvilken type?

Skal brættet bruge downcasting for at få adgang til candidateRelativeMoves?

Nej! Abstrakte metoder beholder deres implementation ved upcasting

Hvad gør?

```
member this.availableMoves (board : Board) : (Position list * chessPiece list) =  
    board.getVacantNNeighbours this (*//$\label{chessPieceEnd}$*)
```

Hvad gør?

```
member this.getVacantNNeighbours (piece : chessPiece) : (Position list *
chessPiece list) =
  match piece.position with
  | None ->
    ([], [])
  | Some p ->
    let convertNWrap =
      (relativeToAbsolute p) >> this.getVacantNOccupied
    let vacantPieceLists = List.map convertNWrap piece.candidateRelativeMoves
    // Extract and merge lists of vacant squares
    let vacant = List.collect fst vacantPieceLists
    // Extract and merge lists of first obstruction pieces and filter out own
pieces
    let opponent =
      vacantPieceLists
      |> List.choose snd
    (vacant, opponent) (*// $\\label{chessBoardEnd}$*)
```

Spørgsmål til koden?