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 candiateRelativeMoves?

Nej! Abstrakte metoder beholder deres implementation ved upcasting

Spørgsmål til koden?

Design en objektorienteret løsning til følgende:

Use-case: Online auktion

En online auktion er et offentligt salg af f.eks. ting eller tjenesteydelser til højstbydende på nettet. Auktionen forestås auktionarius, som er en betroet mellemmand eller system. Auktionarius modtager varen til salg fra sælger, udstiller varen, modtager bud fra køber, og afgør vinderen.

En typisk auktion vil bestå af en udstillingsperiode for et antal vare, og dernæst et købsvindue, hvor købere kan byde på en bestemt vare. Køberne kan se hinandens bud, og den, som indenfor købsvinduet har budt højest, har købt produktet. Køber betaler det budte beløb til sælger, og et auktionssalær til auktionarius samt moms til staten.