Programmering og Problemløsning (PoP)

Klasser og Objekter

Ken Friis Larsen
kflarsen@diku.dk

Datalogisk Institut, Københavns Universitet (DIKU)

December, 2021

Dagens program

- *Object-Oriented Programming* endnu et programmeringsparadigme. Hvorfor?
- Klasser og objekter. Hvordan

Del I

Hvorfor Object-Oriented Programming (OOP)

Hvorfor?

- Vi skal arbejde fyld på en pizza, dvs en liste af ingredienser.
- Fx, vi vil gerne have en pizza med tomat, 75.5g ost og 6 skiver pepperoni

```
> let pizza = [ true; 75.5; 6 ];;
```

Hvorfor?

- Vi skal arbejde fyld på en pizza, dvs en liste af ingredienser.
- Fx, vi vil gerne have en pizza med tomat, 75.5g ost og 6 skiver pepperoni

```
> let pizza = [ true; 75.5; 6 ];;

  let pizza = [ true; 75.5; 6 ];;
  --------------------^^^^


error FS0001:
  All elements of a list must be of the same type as the first
  element, which here is 'bool'. This element has type 'float'.
```

Hvorfor?

- Vi skal arbejde fyld på en pizza, dvs en liste af ingredienser.
- Fx, vi vil gerne have en pizza med tomat, 75.5g ost og 6 skiver pepperoni

```
> let pizza = [ true; 75.5; 6 ];;

  let pizza = [ true; 75.5; 6 ];;
  --------------------^^^^


error FS0001:
  All elements of a list must be of the same type as the first
  element, which here is 'bool'. This element has type 'float'.
```

- Løsning: Vi skal have defineret en type til at holde styr på hvad slags ingredienser vi har.

Hvorfor?

```
type Topping =
    | Tomato
    | Cheese of float
    | Pepperoni of int

let isVegetarian topping =
    match topping with
        | Pepperoni _ -> false
        | _ -> true

let vegetarian toppings = List.forall isVegetarian toppings

let addExtra topping =
    match topping with
        | Tomato -> Tomato
        | Cheese g -> g + 20.0 |> Cheese
        | Pepperoni p -> p + 2 |> Pepperoni

let extraAll toppings = List.map addExtra toppings
```

Hvorfor?

- Alt virker nu:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6]
```

Hvorfor?

- Alt virker nu:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6]

> vegetarian pizza;;
val it : bool = false
```

Hvorfor?

- Alt virker nu:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6]

> vegetarian pizza;;
val it : bool = false

> extraAll pizza;;
val it : Topping list = [Tomato; Cheese 95.5; Pepperoni 8]
```

Hvorfor?

- Alt virker nu:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6]

> vegetarian pizza;;
val it : bool = false

> extraAll pizza;;
val it : Topping list = [Tomato; Cheese 95.5; Pepperoni 8]
```

- Faktisk, så vil skal der kunne komme skinke på pizzaer

Hvorfor?

```
type Topping =
    | Tomato
    | Cheese of float
    | Pepperoni of int
    | Ham of int                        // <-- Added
let isVegetarian topping =
    match topping with
        | Pepperoni _ -> false
        | Ham _ -> false                // <-- Added
        | _ -> true
let vegetarian toppings = List.forall isVegetarian toppings
let addExtra topping =
    match topping with
        | Tomato -> Tomato
        | Cheese g -> g + 20.0 |> Cheese
        | Pepperoni p -> p + 2 |> Pepperoni
        | Ham p       -> p + 1 |> Ham   // <-- Added
let extraAll toppings = List.map addExtra toppings
```

Hvorfor?

- Alt virker stadigvæk:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6; Ham 3];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6; Ham 3]

> vegetarian pizza;;
val it : bool = false

> extraAll pizza;;
val it : Topping list = [Tomato; Cheese 95.5; Pepperoni 8; Ham 4]
```

Hvorfor?

- Alt virker stadigvæk:

```
> let pizza = [Tomato; Cheese 75.5; Pepperoni 6; Ham 3];;
val pizza : Topping list = [Tomato; Cheese 75.5; Pepperoni 6; Ham 3]

> vegetarian pizza;;
val it : bool = false

> extraAll pizza;;
val it : Topping list = [Tomato; Cheese 95.5; Pepperoni 8; Ham 4]
```

- Faktisk, så skal der også kunne komme ananas og champion på pizza

Hvorfor?

```
type Topping =
    | Tomato
    | Cheese of float
    | Pepperoni of int
    | Ham of int
    | Pineapple of int array
    | Mushrooms of int array

let addExtra topping =
    match topping with
        | Tomato -> Tomato
        | Cheese g -> g + 20.0 |> Cheese
        | Pepperoni p -> p + 2  |> Pepperoni
        | Ham p       -> p + 1  |> Ham
        | Pineapple ps -> Array.map (fun x -> x+1) ps |> Pineapple
        | Mushrooms ms -> ( Array.iteri (fun i x -> ms.[i] <- x+1) ms
                          ; topping )
```

Små Ændringer Har Stor Konsekvenser

```
let arr1 = [| 1; 2; 3|]
let pizza1 = [ Tomato; Pineapple arr1; Mushrooms arr1 ]

let arr2 = [| 1; 2; 3|]
let pizza2 = [ Tomato; Mushrooms arr2; Pineapple arr2 ] ;;
```

Små Ændringer Har Stor Konsekvenser

```
let arr1 = [| 1; 2; 3|]
let pizza1 = [ Tomato; Pineapple arr1; Mushrooms arr1 ]

let arr2 = [| 1; 2; 3|]
let pizza2 = [ Tomato; Mushrooms arr2; Pineapple arr2 ] ;;

> extraAll pizza1;;
val it : Topping list = [Tomato; Pineapple [|2; 3; 4|]; Mushrooms [|2; 3; 4|]]

> extraAll pizza2;;
val it : Topping list = [Tomato; Mushrooms [|2; 3; 4|]; Pineapple [|3; 4; 5|]]
```

Del II

Klasser og Objekter

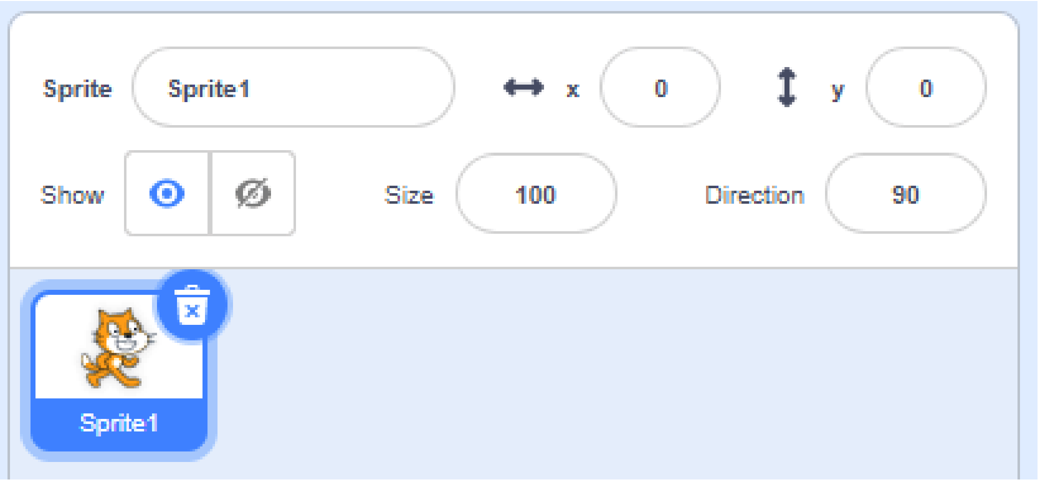
What Er Et Objekt *(Object)*

- Et objekt er en *indkapsling* af data, ved at hæfte data sammen med de funktioner der arbejder på dem:
  - *Properties*, data (a.k.a *attributes*, *fields*)
  - *Methods*, funktioner
- Et objekt er en *værdi* som har en *type*, vi skaber typisk objekter ud fra en *klasse* som erklærer en ny type.
- Ofte bruges objekter og klasser til at opnå *data abstraktion*.

I Har Allerede Arbejdet Med Objekter

I Har Allerede Arbejdet Med Objekter

I Har Allerede Arbejdet Med Objekter



Sprite: Sprite1

x: 0

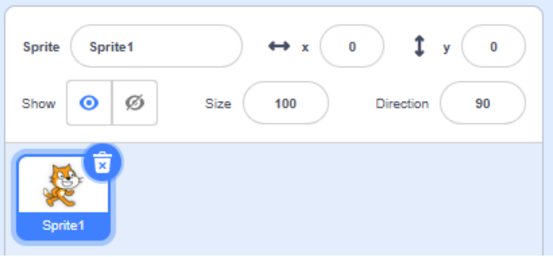
y: 0

Show: ◉ ⌀

Size: 100

Direction: 90

Sprite Properties

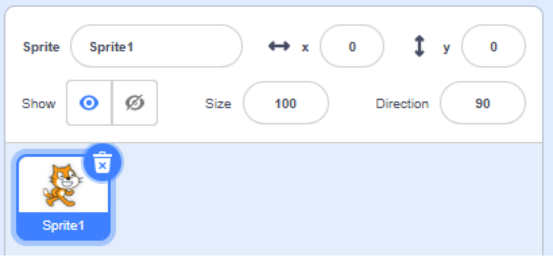


Sprite Sprite1 ↔ x 0 ↕ y 0

Show Size 100 Direction 90

```
type Sprite() =
  member this.X = 0
  member this.Y = 0
  member this.Direction = 90
  member this.Size = 100
  member this.Show = true
  member this.Image = "Sprite1"
```

Sprite Properties



Sprite: Sprite1

x: 0

y: 0

Show

Size: 100

Direction: 90

```
type Sprite() =
  member this.X = 0
  member this.Y = 0
  member this.Direction = 90
  member this.Size = 100
  member this.Show = true
  member this.Image = "Sprite1"
```

- Sprite er klasse (en ny type)
- X, Y, Direction, Size, Show, Image er properties

Konstruktør *(Constructor)*

- Ofte vil vi gerne give nogle parameter til *konstruktøren*:

```
type Sprite(x:int, y:int, dir:int) =
  let d = dir % 360
  member this.X = x
  member this.Y = y
  member this.Direction = d
```

- `Sprite` har følgende *klasse signatur*:

```
type Sprite =
  class
    new : x:int * y:int * dir:int -> Sprite
    member Direction : int
    member X : int
    member Y : int
  end
```

Objeker Fra En Klasse

- Vi kan lave en *instans* af en klasse (skabe et objekt):

```
> let cat = Sprite(23, 42, 450);;
val cat : Sprite
```

- Og tilgå properties via . (dot)

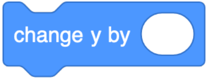
```
> cat.X;;
val it : int = 23

> cat.Direction;;
val it : int = 90
```

Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
```

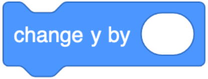
Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
```



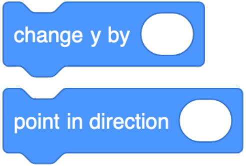
Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
```



Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
```



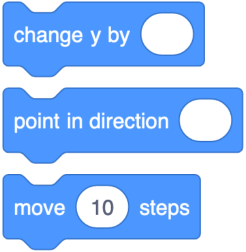
Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
  member this.PointInDir d = dir <- d % 360
```

change y by
point in direction

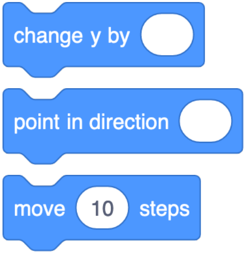
Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
  member this.PointInDir d = dir <- d % 360
```



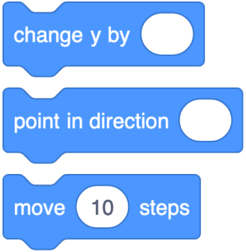
Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
  member this.PointInDir d = dir <- d % 360
  member this.MoveSteps num =
      let fnum = float num
      x <- x + (dir |> toRad |> sin |> ( * ) fnum |> int)
      y <- y + (dir |> toRad |> cos |> ( * ) fnum |> int)
```



Ændring Af Objekters Tilstand

```
type Sprite(x_init:int, y_init:int, d_init) =
  let mutable dir = d_init % 360
  let mutable x = x_init
  let mutable y = y_init
  member this.X = x
  member this.Y = y
  member this.Direction = dir
  member this.ChangeY by = y <- y + by
  member this.PointInDir d = dir <- d % 360
  member this.MoveSteps num =
      let fnum = float num
      x <- x + (dir |> toRad |> sin |> ( * ) fnum |> int)
      y <- y + (dir |> toRad |> cos |> ( * ) fnum |> int)
```



```
let toRad deg =
    deg |> float |> ( / ) 180.0 |> ( * ) System.Math.PI
```

Data Abstraktion

```
type Sprite =
  class
    new : x_init:int * y_init:int * d_init:int -> Sprite
    member X : int
    member Y : int
    member Direction : int
    member ChangeY : by:int -> unit
    member PointInDir : d:int -> unit
    member MoveSteps : num:int -> unit
  end
```

Data Abstraktion

```
> let cat = Sprite(23, 42, 450);;
val cat : Sprite

> (cat.X, cat.Y, cat.Direction);;
val it : int * int * int = (23, 42, 90)

> cat.MoveSteps 10;;
val it : unit = ()

> (cat.X, cat.Y, cat.Direction);;
val it : int * int * int = (23, 52, 90)
```

Data Abstraktion

```
> cat.y;;
```

Data Abstraktion

```
> cat.y;;

  cat.y;;
  ----^

error: The field, constructor or member 'y' is not defined.
```

Data Abstraktion

```
> cat.y;;

  cat.y;;
  ----^

error: The field, constructor or member 'y' is not defined.

> cat.X <- 420;;
```

Data Abstraktion

```
> cat.y;;

  cat.y;;
  ----^

error: The field, constructor or member 'y' is not defined.

> cat.X <- 420;;

  cat.X <- 420;;
  ^^^^^

error FS0810: Property 'X' cannot be set
```

Opsummering

- Bivirkninger på imperative data-strukturer kan hurtigt give anledning til subtile fejl, der kan være svære at finde.
- Et objekt er en *indkapsling* af data, ved at hæfte data sammen med de funktioner der arbejder på dem:
  - *Properties*, data (a.k.a *attributes*, *fields*)
  - *Methods*, funktioner
- Vi skaber objekter som instanser af *klasser*
- Ofte bruges objekter og klasser til at opnå *data abstraktion*. Det vil sige, vi kontrollerer hvilke funktioner der må ændre ved data.