

Programmering og Problemløsning (PoP)

Nedarvning

Ken Friis Larsen
kflarsen@diku.dk

Datalogisk Institut, Københavns Universitet (DIKU)

December, 2021

Nogle slides kraftigt inspireret af slides lavet af Jon Sparring

Dagens program

- ▶ Statiske (*static*) klasse elementer
- ▶ Nedarvning
- ▶ Objekt-orienteret analyse og design

Del I

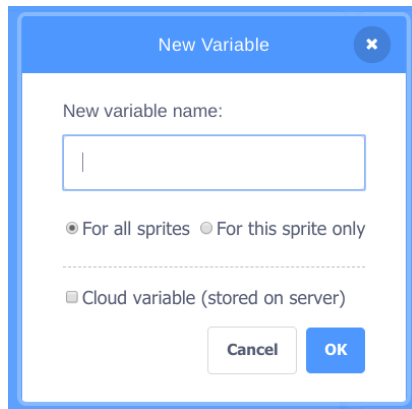
Statische (*static*) Klasse Elementer

Variabler

- ▶ Scratch har imperative variabler **variable**

Variabler

- ▶ Scratch har imperative variabler **variable**
- ▶ Når vi opretter en ny variable skal vi tage stilling til et underligt spørgsmål



The image shows the 'New Variable' dialog box in Scratch. It has a blue header with the title 'New Variable' and a close button (X). Below the header, there is a text input field labeled 'New variable name:'. Underneath the input field, there are two radio buttons: 'For all sprites' (which is selected) and 'For this sprite only'. Below these, there is a dashed line and a checkbox labeled 'Cloud variable (stored on server)'. At the bottom right, there are two buttons: 'Cancel' and 'OK'.

- ▶ Hvad er forskellen på *For all sprites* versus *For this sprite only*?

Static

- ▶ Et *static* medlem af en klasse deles mellem alle instanser af en klasse
- ▶ Simplificeret udgave af Sprite med **static** property

```
type SpriteS () =  
  static let mutable total = 0           // only once  
  let myID = ( total <- total + 1  
              ; total )  
  member obj.ID = myID  
  static member Total = total
```

```
> let (cat, dog, monkey) = (SpriteS(), SpriteS(), SpriteS());;  
val monkey : SpriteS  
val dog : SpriteS  
val cat : SpriteS
```

```
> let (cat, dog, monkey) = (SpriteS(), SpriteS(), SpriteS());;  
val monkey : SpriteS  
val dog : SpriteS  
val cat : SpriteS  
  
> (monkey.ID, dog.ID);;  
val it : int * int = (3, 2)
```



```
> let (cat, dog, monkey) = (SpriteS(), SpriteS(), SpriteS());;
val monkey : SpriteS
val dog : SpriteS
val cat : SpriteS

> (monkey.ID, dog.ID);;
val it : int * int = (3, 2)

> dog.Total;;
```

```
> let (cat, dog, monkey) = (SpriteS(), SpriteS(), SpriteS());;
val monkey : SpriteS
val dog : SpriteS
val cat : SpriteS
```

```
> (monkey.ID, dog.ID);;
val it : int * int = (3, 2)
```

```
> dog.Total;;
```

```
dog.Total;;
^^^^^^^^^^
```

error FS0809: Property 'Total' is **static**

```
> let (cat, dog, monkey) = (SpriteS(), SpriteS(), SpriteS());;
val monkey : SpriteS
val dog : SpriteS
val cat : SpriteS
```

```
> (monkey.ID, dog.ID);;
val it : int * int = (3, 2)
```

```
> dog.Total;;
```

```
dog.Total;;
^^^^^^^^^^
```

error FS0809: Property 'Total' is **static**

```
> SpriteS.Total;;
val it : int = 3
```

Static Methods

- Det er lidt grimt med imperativ kode som udtryk

```
type SpriteS () =  
    static let mutable total = 0  
    let myID = ( total <- total + 1 // me no like  
                ; total )  
    member obj.ID = myID  
    static member Total = total
```

Static Methods

- ▶ Det er lidt grimt med imperativ kode som udtryk

```
type SpriteS () =  
    static let mutable total = 0  
    let myID = ( total <- total + 1 // me no like  
                ; total )  
    member obj.ID = myID  
    static member Total = total
```

- ▶ Vi indfører en *statisk metode* NextID

```
type SpriteS () =  
    static let mutable total = 0  
    let myID = SpriteS.NextID()  
    member obj.ID = myID  
    static member Total = total  
    static member NextID () = // it's a very NICE  
        total <- total + 1  
        total
```

Static Opsummering

- ▶ Statiske klasse elementer (*members*) har *samme værdi for alle instanser af en klasse*
- ▶ Statiske klasse elementer *kan tilgås*:
 - ▶ **før** at nogen objekter er instantieret
 - ▶ **uden** at nogen objekter er instantieret
 - ▶ uden et objekt, i stedet bruge en *direkte reference til klassen*

Del II

Nedarvning

Fra Generelt Til Specialisering

- ▶ Vi vil ofte gerne *generalisere* over en type med nogle specifikke (eller *specialiserede*) elementer.
- ▶ For eksempel:

```
type price = int
```

```
type topping =  
  | Tomato of price  
  | Cheese of price * float  
  | Pepperoni of price * int
```

- ▶ Hvor topping er det generelle (abstrakte) begreb og Tomato, Cheese og Pepperoni er specifikke elementer

Funktioner På Toppings

- Vi kan definere funktioner der virker på toppings

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

```
let addExtra topping =  
  match topping with  
  | Tomato p -> Tomator <| p + 500  
  | Cheese (p, g) -> Cheese (p + 500, g + 20.0)  
  | Pepperoni(p, slices) -> Pepperoni(p+1000, slices+2)
```

Funktioner På Toppings

- ▶ Vi kan definere funktioner der virker på toppings

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

```
let addExtra topping =  
  match topping with  
    | Tomato p -> Tomator <| p + 500  
    | Cheese (p, g) -> Cheese (p + 500, g + 20.0)  
    | Pepperoni(p, slices) -> Pepperoni(p+1000, slices+2)
```

- ▶ Faktisk, så vil jeg jo gerne have flere toppings som fx ananas, skinke, og ...

Toppings Som Objekter

- ▶ Alle toppings har en pris

```
type Topping (price_init : price) =  
  let mutable price = price_init  
  member top.Price with get() = price  
                        and set p = price <- p
```

Toppings Som Objekter

- ▶ Alle toppings har en pris

```
type Topping (price_init : price) =  
  let mutable price = price_init  
  member top.Price with get() = price  
                        and set p = price <- p
```

- ▶ Automatic property

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set
```

Nedarvning

- ▶ Tomato, Cheese og Pepperoni *nedarver* fra Topping

```
type Tomato () =  
    inherit Topping(500)
```

```
type Cheese (amount : float) =  
    inherit Topping(700)  
    member val Amount = amount with get, set
```

```
type Pepperoni(slices : int) =  
    inherit Topping(700)  
    member val Slices = slices with get, set
```

- ▶ Objekter af typen Tomato *er* også en Topping
- ▶ Objekter af typen Cheese *er* også en Topping
- ▶ Objekter af typen Pepperoni *er* også en Topping

```
let tomato = Tomato();;  
> val tomato : Tomato  
let cheese = Cheese(75.5) ;;  
> val cheese : Cheese  
  
tomato.Price;;  
> val it : int = 500  
cheese.Amount <- 90.0;;  
val it : unit = ()
```

```
let tomato = Tomato();;  
> val tomato : Tomato  
let cheese = Cheese(75.5) ;;  
> val cheese : Cheese  
  
tomato.Price;;  
> val it : int = 500  
cheese.Amount <- 90.0;;  
val it : unit = ()  
  
let pizza = [ tomato; cheese ];;
```

```

let tomato = Tomato();;
> val tomato : Tomato
let cheese = Cheese(75.5) ;;
> val cheese : Cheese

tomato.Price;;
> val it : int = 500
cheese.Amount <- 90.0;;
val it : unit = ()

let pizza = [ tomato; cheese ];;

let pizza = [ tomato; cheese ];;
-----^^^^^^

```

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


```
> let pizza : Topping list = [ tomato; cheese ];;

val pizza : Topping list =
    [FSI_0013.Pizza+Tomato; FSI_0013.Pizza+Cheese]

> let pizza2 = [ tomato :> Topping; cheese :> Topping ];;

val pizza2 : Topping list =
    [FSI_0013.Pizza+Tomato; FSI_0013.Pizza+Cheese]
```

Metoder På Toppings

- Lad os tilføje metoden AddVAT til Topping

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() =  
    topping.Price <- float topping.Price  
                      |> (*) 1.25 |> round |> int
```

Metoder På Toppings

- Lad os tilføje metoden AddVAT til Topping

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() =  
    topping.Price <- float topping.Price  
                      |> (*) 1.25 |> round |> int
```

- Vi kan bruge AddVAT på alle slags toppings

```
> List.map (fun (t : Topping) -> t.AddVAT(); t.Price)  
          [tomato; cheese];;
```

```
val it : price list = [625; 875]
```

Abstrakte Klasser

- ▶ Lad os tilføje `isVegetarian` og `addExtra` som metoder til `Topping`

Abstrakte Klasser

- Lad os tilføje isVegetarian og addExtra som metoder til Topping

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() = ...  
  member topping.IsVegetarian = ???  
  member topping.AddExtra () = ???
```

Abstrakte Klasser

- Lad os tilføje `isVegetarian` og `addExtra` som metoder til `Topping`

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() = ...  
  member topping.IsVegetarian = ???  
  member topping.AddExtra () = ???
```

- Vi må lave `Topping` til en *abstrakt klasse*

```
[<AbstractClass>]
```

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() = ...  
  abstract member IsVegetarian : bool  
  abstract member AddExtra : unit -> unit
```

Abstrakte Klasser

- Lad os tilføje `isVegetarian` og `addExtra` som metoder til `Topping`

```
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() = ...  
  member topping.IsVegetarian = ???  
  member topping.AddExtra () = ???
```

- Vi må lave `Topping` til en *abstrakt klasse*

```
[<AbstractClass>]  
type Topping (price_init : price) =  
  member val Price = price_init with get, set  
  member topping.AddVAT() = ...  
  abstract member IsVegetarian : bool  
  abstract member AddExtra : unit -> unit
```

- `AddExtra` og `IsVegetarian` *skal* implementeres af klasser der nedarver fra `Topping`

Override, Tomato

- Brug **override** for at implementere et abstrakt element

```
type Tomato () =  
  inherit Topping(500)  
  override __.IsVegetarian = true  
  override this.AddExtra () =  
    this.Price <- this.Price + 500
```


Override, Cheese og Pepperoni

```
type Cheese (amount : float) =  
  inherit Topping(700)  
  member val Amount = amount with get, set  
  override __.IsVegetarian = true  
  override this.AddExtra () =  
    this.Price <- this.Price + 500  
    this.Amount <- this.Amount + 20.0
```

```
type Pepperoni(slices : int) =  
  inherit Topping(700)  
  member val Slices = slices with get, set  
  override __.IsVegetarian = false  
  override this.AddExtra () =  
    this.Price <- this.Price + 1000  
    this.Slices <- this.Slices + 2
```

Override, Pineapple og Ham

- ▶ Vi kan tilføje nye slags toppings

```
type Pineapple() =  
  inherit Topping(700)  
  member val Pieces = 3 with get, set  
  override __.IsVegetarian = true  
  override this.AddExtra () =  
    this.Price <- this.Price + 1000  
    this.Pieces <- this.Pieces + 2
```

```
type Ham(slices : int) =  
  inherit Topping(500)  
  member val Slices = slices with get, set  
  override __.IsVegetarian = false  
  override this.AddExtra () =  
    this.Slices <- this.Slices + 2  
    this.Price <- this.Price + 500
```

Override, Pineapple og Ham

- ▶ Vi kan tilføje nye slags toppings

```
type Pineapple() =  
  inherit Topping(700)  
  member val Pieces = 3 with get, set  
  override __.IsVegetarian = true  
  override this.AddExtra () =  
    this.Price <- this.Price + 1000  
    this.Pieces <- this.Pieces + 2
```

```
type Ham(slices : int) =  
  inherit Topping(500)  
  member val Slices = slices with get, set  
  override __.IsVegetarian = false  
  override this.AddExtra () =  
    this.Slices <- this.Slices + 2  
    this.Price <- this.Price + 500
```

- ▶ *Uden at ændre ved den eksisterende kode*

Nedarvning Opsummering

- ▶ Vi kan bruge *nedarvning* til at specialisere en generel type (klasse)
- ▶ *Abstrakte metoder* skal implementeres når man nedarver
- ▶ Nedarvning giver os en *åben type* vi kan udvide uden at ændre ved eksisterende kode
(I modsætning til sum-typer som giver os en *lukket type*)

Del III

Objekt-Orienteret Design

Design Efter Navne- og Udsagnsord

- ▶ Navneord er kandidater til klasser, udsagnsord er kandidater til metoder, som får klasserne til at hænge sammen.
- ▶ Navneord (substantiv): Genstande, levende væsner, personer eller abstraktioner. F.eks.: (et) bord, (en) mus, Jon, (et) venskab.
- ▶ Udsagnsord (verbum): Handlinger, hændelser, tilstandsmåder. F.eks.: (at) løbe, (han) underviser, (han) er (lærer)

Case: Sænke Slagskibe

Dette er et spil for to personer, der kan spilles med papir og blyant. Der spilles på fire plader, to for hver spiller, og hver plade er inddelt i 10x10 felter. Hvert felt identificeres vha. dets række- og søjle-nummer.

Hver spiller får tildelt et antal skibe, som placeres på spillerens ene plade og markerer, hvor modstanderen har forsøgt at skyde. På den anden plade markerer spilleren tilsvarende, hvor de har forsøgt at ramme modstanderen.

Når skibene er placeret skiftes spillerne til at skyde på modstanderens felt, og modstanderen annoncerer ramt eller plask, alt efter om et skib blev ramt eller ej. Vinderen er den, der først får sænket alle modstanderes skibe.

Case: Sænke Slagskibe

Navneord

Dette er et **spil** for to **personer**, der kan spilles med **papir** og **blyant**. Der spilles på fire **plader**, to for hver **spiller**, og hver **plade** er inddelt i 10x10 **felter**. Hvert **felt** identificeres vha. dets **række-** og **søjle-nummer**.

Hver **spiller** får tildelt et antal **skibe**, som placeres på **spillerens** ene **plade** og markerer, hvor **modstanderen** har forsøgt at skyde. På den anden **plade** markerer **spilleren** tilsvarende, hvor de har forsøgt at ramme **modstanderen**.

Når **skibene** er placeret skiftes **spillerne** til at skyde på **modstanderens felt**, og **modstanderen** annoncerer ramt eller plask, alt efter om et **skib** blev ramt eller ej. **Vinderen** er den, der først får sænket alle **modstanderes skibe**.

Case: Sænke Slagskibe

Navneord

Udsagnsord

Dette **er** et **spil** for to **personer**, der kan **spilles** med **papir** og **blyant**. Der **spilles** på fire **plader**, to for hver **spiller**, og hver **plade** **er inddelt** i 10x10 **felter**. Hvert **felt** **identificeres** vha. dets **række-** og **søjle-nummer**.

Hver **spiller** **får tildelt** et antal **skibe**, som **placeres** på **spillerens** ene **plade** og **markerer**, hvor **modstanderen** har **forsøgt** at **skyde**. På den anden **plade** **markerer** **spilleren** tilsvarende, hvor de har **forsøgt** at **ramme** **modstanderen**.

Når **skibene** er **placeret** **skiftes** **spillerne** til at **skyde** på **modstanderens** **felt**, og **modstanderen** **annoncerer** ramt eller plask, alt efter om et **skib** blev **ramt** eller ej. **Vinderen** **er** den, der først får **sænket** alle **modstanderes** **skibe**.

Case: Sænke slagskibe

- ▶ **Navneord**: Spil, person , papir , blyant , plade, felt, blyant , papir , rækkenummer, søjlenummer, spiller, skibe, modstander, (en) vinder.
- ▶ **Udsagnsord**: er, spille , inddele, identificere , tildele, placere, forsøge , skyde, markere, ramme, skifte , annoncere, sænke.

Case: Sænke slagskibe

- ▶ **Navneord**: Spil, person, papir, blyant, plade, felt, blyant, papir, rækkenummer, søjlenummer, spiller, skibe, modstander, (en) vinder.
- ▶ **Udsagnsord**: er, spille, inddele, identificere, tildele, placere, forsøge, skyde, markere, ramme, skifte, annoncere, sænke.

Relationer:

- ▶ Et **spil** **består** af 2 **spillere**.
- ▶ Hver **spiller** **har** 2 **plader**.
- ▶ Hver **plade** er **inddelt** i 10x10 **felter**.
- ▶ Hvert **felt** **har** et **række-** og et **søjle-nummer**.
- ▶ Hver **spiller** **tildeles** **skibe**.
- ▶ **Skibe** **placeres** på en **plade**.
- ▶ **Spiller** **skyder** på **modstanderen**.
- ▶ **Modstander** **annoncerer** ramt eller plask.
- ▶ **Spiller** **markerer** **skud**.
- ▶ **Skib** kan blive **ramt** og **sunket**.
- ▶ **Vinder** er den, som har **sunket** alle **modstanders skibe**.

Case: Sænke slagskibe

- ▶ **Navneord**: Spil, person, papir, blyant, plade, felt, blyant, papir, rækkenummer, søjlenummer, spiller, skibe, modstander, (en) vinder.
- ▶ **Udsagnsord**: er, spille, inddele, identificere, tildele, placere, forsøge, skyde, markere, ramme, skifte, annoncere, sænke.

Relationer:

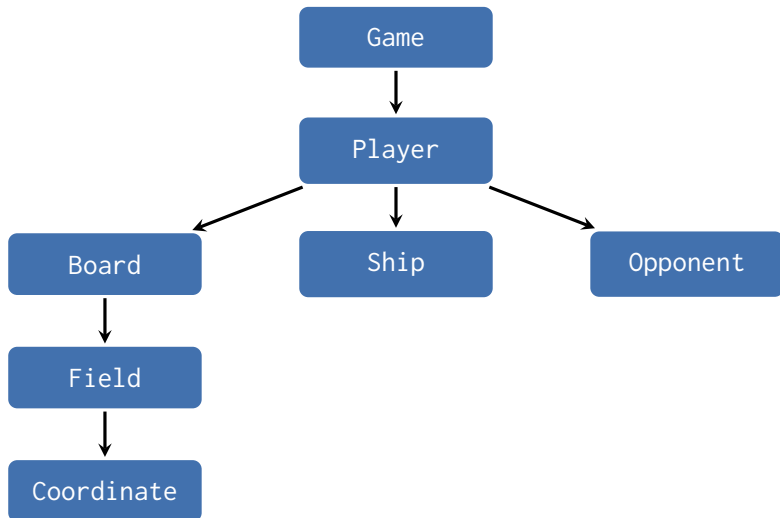
- ▶ Et **spil** **består** af 2 **spillere**.
- ▶ Hver **spiller** **har** 2 **plader**.
- ▶ Hver **plade** er **inddelt** i 10x10 **felter**.
- ▶ Hvert **felt** **har** et **række-** og et **søjle-nummer**.
- ▶ Hver **spiller** **tildeles** **skibe**.
- ▶ **Skibe** **placeres** på en **plade**.
- ▶ **Spiller** **skyder** på **modstanderen**.
- ▶ **Modstander** **annoncerer** ramt eller plask.
- ▶ **Spiller** **markerer** **skud**.
- ▶ **Skib** kan blive **ramt** og **sunket**.
- ▶ **Vinder** er den, som har **sunket** alle **modstanders skibe**.

Relationer

- ▶ Et spil består af 2 spillere.
- ▶ Hver spiller har 2 plader.
- ▶ Hver plade er inddelt i 10x10 felter.
- ▶ Hvert felt har et række- og et søjle-nummer.
- ▶ Hver spiller tildeles skibe.
- ▶ Skibe placeres på en plade.
- ▶ Spiller skyder på modstanderen.
- ▶ Modstander annoncerer ramt eller plask.
- ▶ Spiller markerer skud.
- ▶ Skib kan blive ramt og sunket.
- ▶ Vinder er den, som har sunket alle modstanders skibe.

Relationer

- ▶ Et **spil** består af **spillere**.
- ▶ En **spiller** har en **modstander**.
- ▶ En **spiller** har **skibe**.
- ▶ En **spiller** har **plader**.
- ▶ En **plade** består af **felter**.
- ▶ Et **felt** har et **koordinat** (søjle- og række nummer).
- ▶ En **spiller** kan vinde.
- ▶ En **spiller** skyder på en **modstander**.
- ▶ En **spiller** markerer skud på en **plade**.
- ▶ Et **skib** kan blive ramt.



Kode (endelig)

```
/// A game is a battleship game
```

```
type Game() = class end
```

```
/// A player is a human player, that has 2 boards and
```

```
/// several ships
```

```
type Player() = class end
```

```
/// An opponent is another player
```

```
type Opponent() = class end
```

```
/// A ship can be damaged
```

```
type Ship() = class end
```

```
/// A board is a square set of fields with row-column
```

```
/// coordinates. Ships are placed on the board
```

```
type Board() = class end
```

```
/// A field is can be covered by a ship and can have been
```

```
/// shootend at
```

```
type Field() = class end
```

```
/// A coordinate is a location on a board
```

```
type Coordinate() = class end
```


How To Write A Battleship Game

How To Write A Battleship Game

How to draw an owl

1.



2.



1. Draw some circles

2. Draw the rest of the fucking owl

Opsummering

- ▶ *Design* efter navne- og udsagnsord
 - ▶ *Navneord* er kandidater til klasser,
 - ▶ *Udsagnsord* er kandidater til metoder, som klasserne kan bruge til at manipulere hinanden
- ▶ Vi starter meget stringent og systematisk
- ▶ Efterhånden som vi forstår problemet kan vi simplificere vores kode

Opsummering

- ▶ *Design* efter navne- og udsagnsord
 - ▶ *Navneord* er kandidater til klasser,
 - ▶ *Udsagnsord* er kandidater til metoder, som klasserne kan bruge til at manipulere hinanden
- ▶ Vi starter meget stringent og systematisk
- ▶ Efterhånden som vi forstår problemet kan vi simplificere vores kode
- ▶ Cliff-hanger: Jeg har været sparsom med den fulde sandhed.