Programmering og Problemløsning

14.1: Nedarvning

Resumé

- Med overloading kan vi genbruge navne til små variationer i inputparametre
- Association: “kender-til” - besked relation
- Aggregation: “har-en/flere” – udveksling af ejeskab
- Composition: “har-en/flere” – een ejer
- Overshadow: Navnesammenfald i nedarvning skygger i underklassen
- Abstrakte klasser og override: Abstrakte klasser kan kræve nedarvning og metodedefinitioner.

Interface (is-a)

Et fjernsyn og en bil har begge en knap. Man kan trykke på knappen, og resultatet er ganske forskelligt.

Interface (is-a)

Et fjernsyn og en bil har begge en knap. Man kan trykke på knappen, og resultatet er ganske forskelligt.

Listing 22.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```
1  type button =
2    abstract member press : unit -> string
3  type television () =
4    interface button with
5      member this.press () = "Changing channel"
6  type car () =
7    interface button with
8      member this.press () = "Activating wipers"
9  let pressIt (elm : #button) =
10   elm.press()
11
12 let t = television()
13 let c = car()
14 printfn "%s" (pressIt t)
15 printfn "%s" (pressIt c)
```

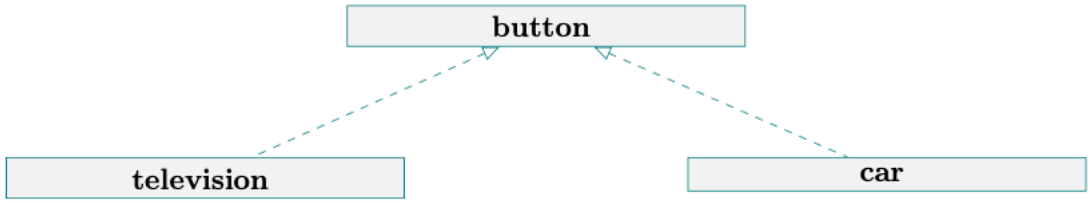
Interface (is-a)

Et fjernsyn og en bil har begge en knap. Man kan trykke på knappen, og resultatet er ganske forskelligt.

Listing 22.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```
type button =
  abstract member press : unit -> string
type television () =
  interface button with
    member this.press () = "Changing channel"
type car () =
  interface button with
    member this.press () = "Activating wipers"
let pressIt (elm : #button) =
  elm.press()

let t = television()
let c = car()
printfn "%s" (pressIt t)
printfn "%s" (pressIt c)
```



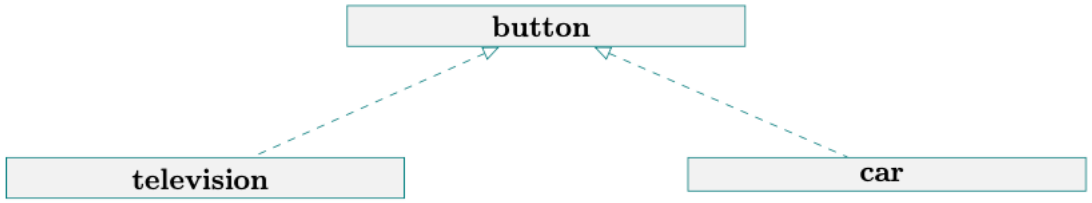
Interface (is-a)

Et fjernsyn og en bil har begge en knap. Man kan trykke på knappen, og resultatet er ganske forskelligt.

Listing 22.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```
type button =
  abstract member press : unit -> string
type television () =
  interface button with
    member this.press () = "Changing channel"
type car () =
  interface button with
    member this.press () = "Activating wipers"
let pressIt (elm : #button) =
  elm.press()

let t = television()
let c = car()
printfn "%s" (pressIt t)
printfn "%s" (pressIt c)
```



Fordele: Angiver egenskaber, semantisk graf
Bagdele: Risiko for megen up- og downcasting

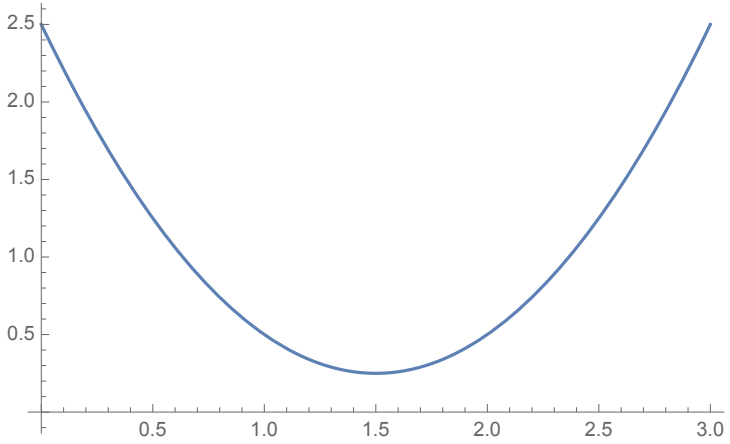
Operatorer: Komplekse tal

For reelle tal: $ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

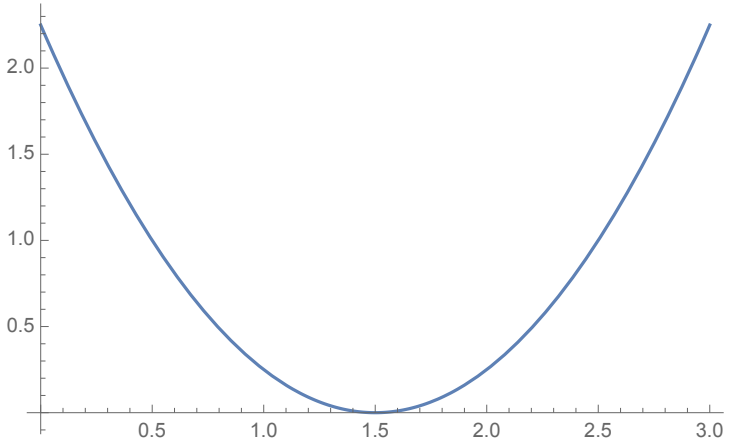
Operatorer: Komplekse tal

For reelle tal: $ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

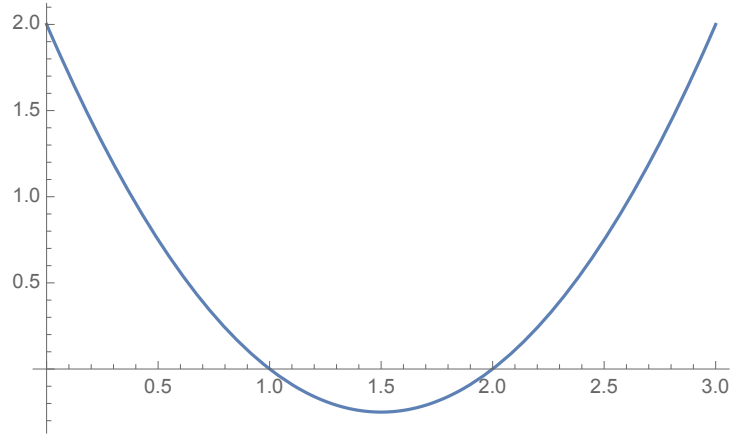
$b^2 - 4ac < 0$ 0 løsninger



$b^2 - 4ac = 0$ 1 løsning



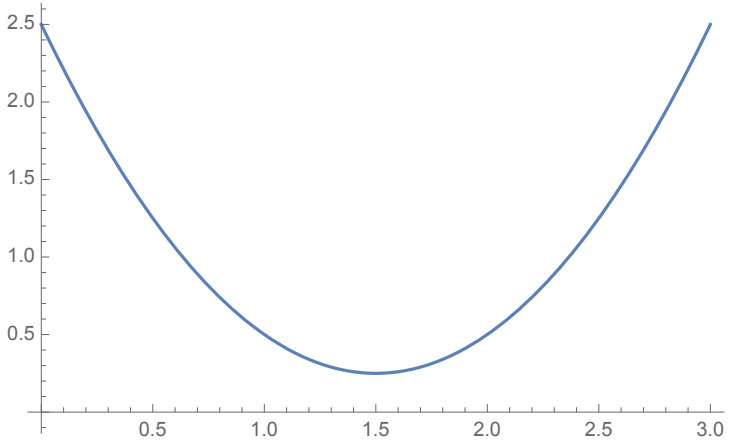
$b^2 - 4ac > 0$ 2 løsninger



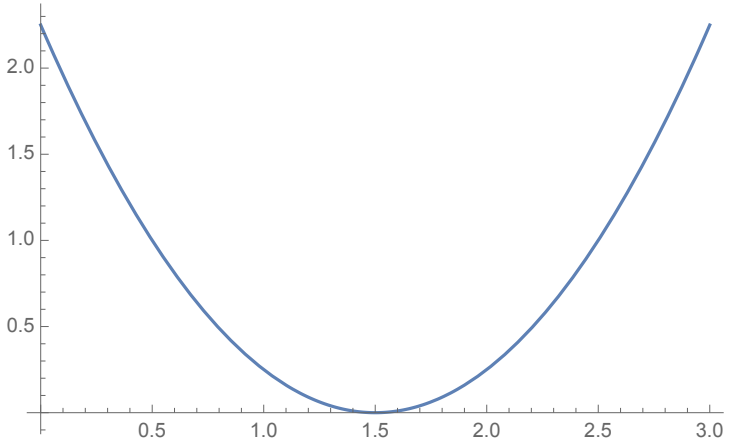
Operatorer: Komplekse tal

For reelle tal: $ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

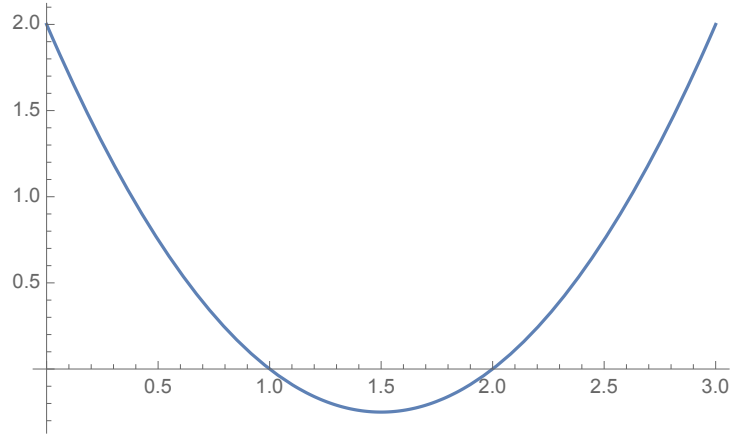
$b^2 - 4ac < 0$ 0 løsninger



$b^2 - 4ac = 0$ 1 løsning



$b^2 - 4ac > 0$ 2 løsninger



Imaginært tal: $x = a + ib$

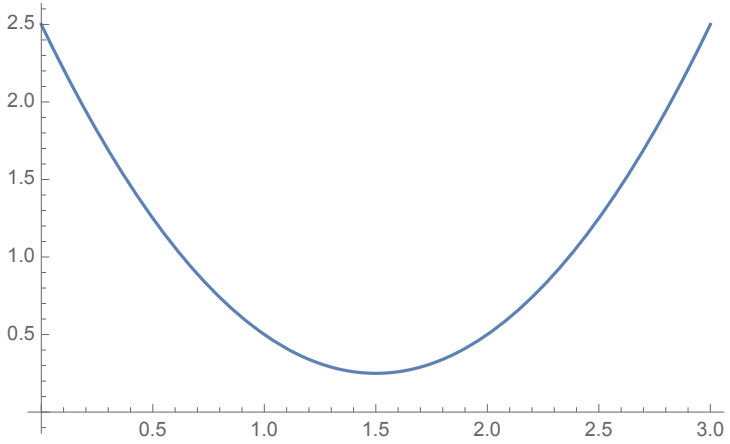
$$i = \sqrt{-1}$$

Altid 2 løsninger!

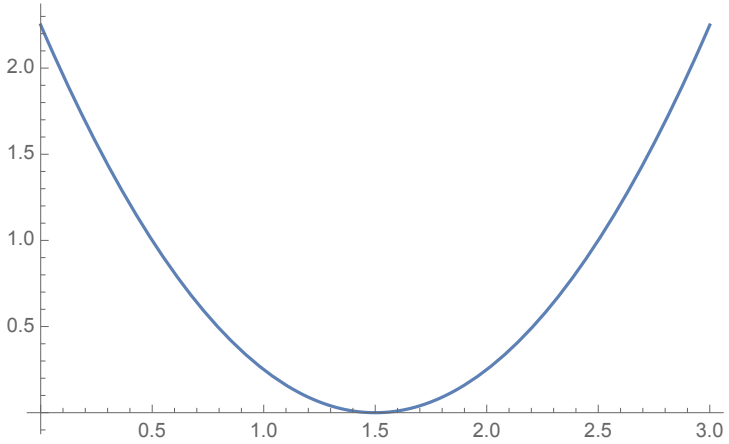
Operatorer: Komplekse tal

For reelle tal: $ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

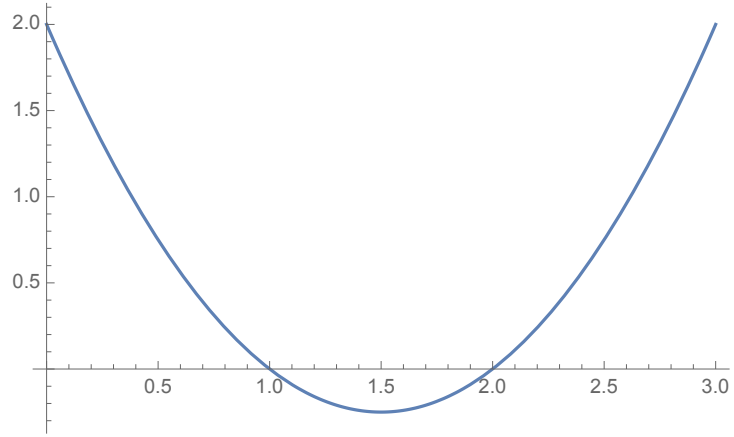
$b^2 - 4ac < 0$ 0 løsninger



$b^2 - 4ac = 0$ 1 løsning



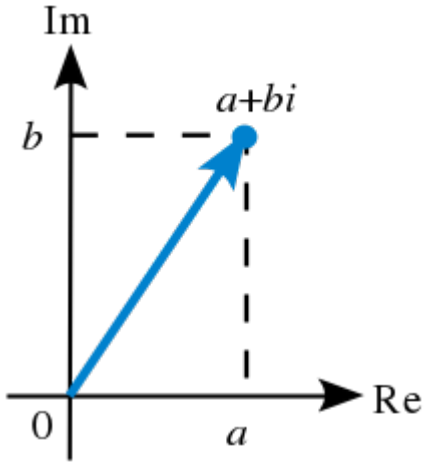
$b^2 - 4ac > 0$ 2 løsninger



Imaginært tal: $x = a + ib$

$i = \sqrt{-1}$

Altid 2 løsninger!



Komplekse tal

https://en.wikipedia.org/wiki/Complex_number

Kompleks konstant: $i^2 = -1$

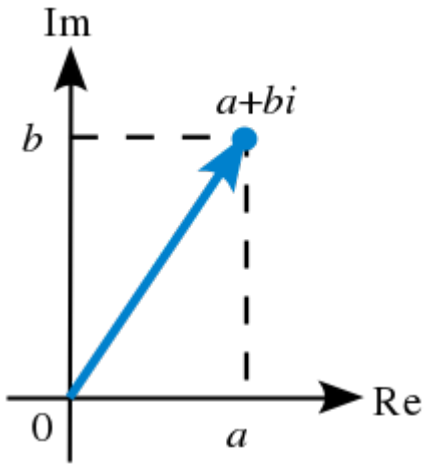
Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...



Komplekse tal: Mutable

```
1type complex (a : float, b : float) =
2  let mutable u = (a, b)
3  member this.re = fst u
4  member this.im = snd u
5  member this.add (v : complex) =
6    u <- (this.re + v.re, this.im + v.im)
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

```
14
15let x = complex(1.0, 2.0)
16let y = complex(2.5,-1.2)
```

Komplekse tal: Mutable

```
1type complex (a : float, b : float) =
2  let mutable u = (a, b)
3  member this.re = fst u
4  member this.im = snd u
5  member this.add (v : complex) =
6    u <- (this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12 member this.copy () =
13   complex(this.re, this.im)
14
15let x = complex(1.0, 2.0)
16let y = complex(2.5,-1.2)
17let z = x.copy()
18z.add(y)
19printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

Komplekse tal: Mutable

```
1type complex (a : float, b : float) =
2  let mutable u = (a, b)
3  member this.re = fst u
4  member this.im = snd u
5  member this.add (v : complex) =
6    u <- (this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12 member this.copy () =
13   complex(this.re, this.im)
14
15let x = complex(1.0, 2.0)
16let y = complex(2.5,-1.2)
17let z = x.copy()
18z.add(y)
19printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

Komplekse tal: Mutable

```
1type complex (a : float, b : float) =
2  let mutable u = (a, b)
3  member this.re = fst u
4  member this.im = snd u
5  member this.add (v : complex) =
6    u <- (this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12 member this.copy () =
13   complex(this.re, this.im)
14
15let x = complex(1.0, 2.0)
16let y = complex(2.5,-1.2)
17let z = x.copy()
18z.add(y)
19printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexSideEffect.fsx

Komplekse tal: Mutable

```
1type complex (a : float, b : float) =
2  let mutable u = (a, b)
3  member this.re = fst u
4  member this.im = snd u
5  member this.add (v : complex) =
6    u <- (this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12 member this.copy () =
13   complex(this.re, this.im)
14
15let x = complex(1.0, 2.0)
16let y = complex(2.5,-1.2)
17let z = x.copy()
18z.add(y)
19printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state

complexSideEffect.fsx

Komplekse tal: This

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member this.re = fst _u
4  member this.im = snd _u
5  member this.add (v : complex) : complex =
6    complex(this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x.add(y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexThis.fsx

Komplekse tal: This

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member this.re = fst _u
4  member this.im = snd _u
5  member this.add (v : complex) : complex =
6    complex(this.re + v.re, this.im + v.im)
7  override this.ToString () =
8    if this.im >= 0.0 then
9      sprintf "(%g + i %g)" this.re this.im
10   else
11     sprintf "(%g - i %g)" this.re (- this.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x.add(y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult

complexThis.fsx

Komplekse tal: U

```
type complex (a : float, b : float) =
  let _u = (a, b)
  member u.re = fst _u
  member u.im = snd _u
  member u.add (v : complex) : complex =
    complex(u.re + v.re, u.im + v.im)
  override u.ToString () =
    if u.im >= 0.0 then
      sprintf "(%g + i %g)" u.re u.im
    else
      sprintf "(%g - i %g)" u.re (- u.im)

let x = complex(1.0, 2.0)
let y = complex(2.5,-1.2)
let z = x.add(y)
printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexU.fsx

Komplekse tal: U

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  member u.add (v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x.add(y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexU.fsx

Komplekse tal: U

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  member u.add (v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x.add(y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult
- U: programming notation easier

complexU.fsx

Komplekse tal: Static

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member add (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = complex.add(x, y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexStatic.fsx

Komplekse tal: Static

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member add (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = complex.add(x, y)
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult
- U: programming notation easier
- Static: Like a module, usage slightly more natural

complexStatic.fsx

Komplekse tal: Operator

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x + y
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexOperator.fsx

Komplekse tal: Operator

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12
13let x = complex(1.0, 2.0)
14let y = complex(2.5,-1.2)
15let z = x + y
16printfn "%A + %A = %A" x y z
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult
- U: programming notation easier
- Static: Like a module, usage slightly more natural
- Operator: ‘normal’ usage

complexOperator.fsx

Komplekse tal: Constructors

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  new (a : float) = complex(a, 0.0)
4  member u.re = fst _u
5  member u.im = snd _u
6  static member (+) (u : complex, v : complex) : complex =
7    complex(u.re + v.re, u.im + v.im)
8  static member (+) (u : complex, v : float) : complex =
9    complex(u.re + v, u.im)
10 static member (+) (u : float, v : complex) : complex =
11   v + u
12 override u.ToString () =
13   if u.im >= 0.0 then
14     sprintf "(%g + i %g)" u.re u.im
15   else
16     sprintf "(%g - i %g)" u.re (- u.im)
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5)
20printfn "%A + %A = %A" x y (x+y)
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexNew.fsx

Komplekse tal: Constructors

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  new (a : float) = complex(a, 0.0)
4  member u.re = fst _u
5  member u.im = snd _u
6  static member (+) (u : complex, v : complex) : complex =
7    complex(u.re + v.re, u.im + v.im)
8  static member (+) (u : complex, v : float) : complex =
9    complex(u.re + v, u.im)
10 static member (+) (u : float, v : complex) : complex =
11   v + u
12 override u.ToString () =
13   if u.im >= 0.0 then
14     sprintf "(%g + i %g)" u.re u.im
15   else
16     sprintf "(%g - i %g)" u.re (- u.im)
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5)
20printfn "%A + %A = %A" x y (x+y)
```

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult
- U: programming notation easier
- Static: Like a module, usage slightly more natural
- Operator: ‘normal’ usage
- New: easier creation

complexNew.fsx

Komplekse tal: Overload

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  static member (+) (u : complex, v : float) : complex =
8    complex(u.re + v, u.im)
9  static member (+) (u : float, v : complex) : complex =
10   v + u
11 override u.ToString () =
12   if u.im >= 0.0 then
13     sprintf "(%g + i %g)" u.re u.im
14   else
15     sprintf "(%g - i %g)" u.re (- u.im)
16
17let x = complex(1.0, 2.0)
18let y = 2.5
19printfn "%A + %A = %A" x y (x+y)
20printfn "%A + %A = %A" y x (y+x)
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

complexOverload.fsx

Komplekse tal: Overload

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  static member (+) (u : complex, v : float) : complex =
8    complex(u.re + v, u.im)
9  static member (+) (u : float, v : complex) : complex =
10   v + u
11 override u.ToString () =
12   if u.im >= 0.0 then
13     sprintf "(%g + i %g)" u.re u.im
14   else
15     sprintf "(%g - i %g)" u.re (- u.im)
16
17let x = complex(1.0, 2.0)
18let y = 2.5
19printfn "%A + %A = %A" x y (x+y)
20printfn "%A + %A = %A" y x (y+x)
```

Kompleks konstant: $i^2 = -1$

Imaginære del: $\mathrm{Im}(a + ib) = b$

Reelle del: $\mathrm{Re}(a + ib) = a$

Lig med: $(a + ib) = (c + id) \Leftrightarrow a = c \text{ and } b = d$

Addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

...

- Mutable: copy needed, difficult to remember state
- This: No state to remember, operator difficult
- U: programming notation easier
- Static: Like a module, usage slightly more natural
- Operator: ‘normal’ usage
- New: easier creation
- Overload: type mixing in usage

complexOverload.fsx

Komplekse tal: equality

(=) operator eksisterer ikke!

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Komplekse tal: equality

(=) operator eksisterer ikke!

```
> let x = complex(1.0, 2.0);;
val x : complex = (1 + i 2)

> x.GetType();;
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Komplekse tal: equality

(=) operator eksisterer ikke!

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

```
> let x = complex(1.0, 2.0);;
val x : complex = (1 + i 2)

> x.GetType();;
val it : System.Type =
  FSI_0005+complex
    {Assembly = FSI-ASSEMBLY, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null;
     AssemblyQualifiedName = "FSI_0005+complex, FSI-ASSEMBLY, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null";
     Attributes = AutoLayout, AnsiClass, Class, NestedPublic, Serializable;
     BaseType = System.Object;
     ContainsGenericParameters = false;
     CustomAttributes = seq
                          [[Microsoft.FSharp.Core.CompilationMappingAttribute((Microsoft.FSharp.Core.SourceConstructFlags)3)];
                           [System.SerializableAttribute()]];
     DeclaredConstructors = [|Void .ctor(Double, Double)|];
     DeclaredEvents = [||];
     DeclaredFields = [|System.Tuple`2[System.Double,System.Double] _u|];
     DeclaredMembers = [|Double get_re(); Double get_im();
                         complex op_Addition(complex, complex);
                         System.String ToString(); Void .ctor(Double, Double);
                         Double re; Double im;
                         System.Tuple`2[System.Double,System.Double] _u|];
     DeclaredMethods = [|Double get_re(); Double get_im();
                         complex op_Addition(complex, complex);
                         System.String ToString()|];
     DeclaredNestedTypes = seq [];
     DeclaredProperties = [|Double re; Double im|];
```

Komplekse tal: equality

(=) operator eksisterer ikke!

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

```
> let x = complex(1.0, 2.0);;
val x : complex = (1 + i 2)

> x.GetType();;
val it : System.Type =
  FSI_0005+complex
    {Assembly = FSI-ASSEMBLY, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null;
     AssemblyQualifiedName = "FSI_0005+complex, FSI-ASSEMBLY, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null";
     Attributes = AutoLayout, AnsiClass, Class, NestedPublic, Serializable;
     BaseType = System.Object;
     ContainsGenericParameters = false;
     CustomAttributes = seq
                          [[Microsoft.FSharp.Core.CompilationMappingAttribute((Microsoft.FSharp.Core.SourceConstructFlags)3)];
                           [System.SerializableAttribute()]];
     DeclaredConstructors = [|Void .ctor(Double, Double)|];
     DeclaredEvents = [||];
     DeclaredFields = [|System.Tuple`2[System.Double,System.Double] _u|];
     DeclaredMembers = [|Double get_re(); Double get_im();
                         complex op_Addition(complex, complex);
                         System.String ToString(); Void .ctor(Double, Double);
                         Double re; Double im;
                         System.Tuple`2[System.Double,System.Double] _u|];
     DeclaredMethods = [|Double get_re(); Double get_im();
                         complex op_Addition(complex, complex);
                         System.String ToString()|];
     DeclaredNestedTypes = seq [];
     DeclaredProperties = [|Double re; Double im|];
```

Komplekse tal: equality

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12 override u.Equals obj =
13   match obj with
14     :? complex as v -> u.re = v.re && u.im = v.im
15     | _ -> false
16 override u.GetHashCode() = hash _u
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5,-1.2)
20printfn "%A = %A = %A" x y (x=y)
21printfn "%A = %A = %A" x x (x=x)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Komplekse tal: equality

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12 override u.Equals obj =
13   match obj with
14     :? complex as v -> u.re = v.re && u.im = v.im
15     | _ -> false
16 override u.GetHashCode() = hash _u
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5,-1.2)
20printfn "%A = %A = %A" x y (x=y)
21printfn "%A = %A = %A" x x (x=x)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Compare types operator: “:?”

Komplekse tal: equality

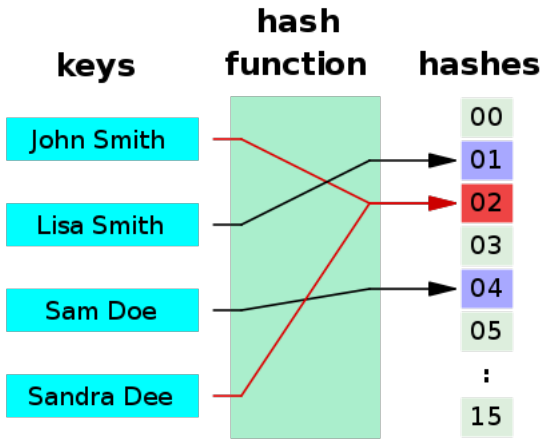
```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12 override u.Equals obj =
13   match obj with
14     :? complex as v -> u.re = v.re && u.im = v.im
15     | _ -> false
16 override u.GetHashCode() = hash _u
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5,-1.2)
20printfn "%A = %A = %A" x y (x=y)
21printfn "%A = %A = %A" x x (x=x)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Compare types operator: “:?”

Hash function: map from infinite domain to fixed sized domain. For computational and storage efficient data access



https://en.wikipedia.org/wiki/Hash_function

Komplekse tal: equality

```
1type complex (a : float, b : float) =
2  let _u = (a, b)
3  member u.re = fst _u
4  member u.im = snd _u
5  static member (+) (u : complex, v : complex) : complex =
6    complex(u.re + v.re, u.im + v.im)
7  override u.ToString () =
8    if u.im >= 0.0 then
9      sprintf "(%g + i %g)" u.re u.im
10   else
11     sprintf "(%g - i %g)" u.re (- u.im)
12 override u.Equals obj =
13   match obj with
14     :? complex as v -> u.re = v.re && u.im = v.im
15     | _ -> false
16 override u.GetHashCode() = hash _u
17
18let x = complex(1.0, 2.0)
19let y = complex(2.5,-1.2)
20printfn "%A = %A = %A" x y (x=y)
21printfn "%A = %A = %A" x x (x=x)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

Compare types operator: “:?”

```
sporring@Jons-mac 14.1-3 % fsharpi complexEqual.fsx
(1 + i 2) = (2.5 - i 1.2) = false
(1 + i 2) = (1 + i 2) = true
```

Komplekse tal: List.sort (comparable)

```
> List.sort;;
 val it : ('a list -> 'a list) when 'a : comparison
```

```
type complex (a : float, b : float) =
  let _u = (a, b)
  member u.re = fst _u
  member u.im = snd _u
  static member (+) (u : complex, v : complex) : complex =
    complex(u.re + v.re, u.im + v.im)
  override u.ToString () =
    if u.im >= 0.0 then
      sprintf "(%g + i %g)" u.re u.im
    else
      sprintf "(%g - i %g)" u.re (- u.im)
  override u.Equals obj =
    match obj with
      :? complex as v -> u.re = v.re && u.im = v.im
      | _ -> false
  override u.GetHashCode() = hash _u
  interface System.IComparable with
    member u.CompareTo obj =
      match obj with
        :? complex as v -> compare u.re v.re
        | _ -> invalidArg "obj" "cannot compare values of different types"

let x = complex(1.0, 2.0)
let y = complex(2.5,-1.2)
let lst = [y;x]
printfn "List.sort %A = %A" lst (List.sort lst)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

System.IComparable

- CompareTo

Komplekse tal: List.sort (comparable)

```
> List.sort;;
val it : ('a list -> 'a list) when 'a : comparison
```

```
type complex (a : float, b : float) =
  let _u = (a, b)
  member u.re = fst _u
  member u.im = snd _u
  static member (+) (u : complex, v : complex) : complex =
    complex(u.re + v.re, u.im + v.im)
  override u.ToString () =
    if u.im >= 0.0 then
      sprintf "(%g + i %g)" u.re u.im
    else
      sprintf "(%g - i %g)" u.re (- u.im)
  override u.Equals obj =
    match obj with
      :? complex as v -> u.re = v.re && u.im = v.im
      | _ -> false
  override u.GetHashCode() = hash _u
  interface System.IComparable with
    member u.CompareTo obj =
      match obj with
        :? complex as v -> compare u.re v.re
        | _ -> invalidArg "obj" "cannot compare values of different types"

let x = complex(1.0, 2.0)
let y = complex(2.5,-1.2)
let lst = [y;x]
printfn "List.sort %A = %A" lst (List.sort lst)
```

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

System.IComparable

- CompareTo

Komplekse tal: List.sort (comparable)

System.object klassen:

- Equals
- Finalize
- GetHashCode
- GetType
- ToString

System.IComparable

- CompareTo

```
> List.sort;;
 val it : ('a list -> 'a list) when 'a : comparison
```

```
type complex (a : float, b : float) =
  let _u = (a, b)
  member u.re = fst _u
  member u.im = snd _u
  static member (+) (u : complex, v : complex) : complex =
    complex(u.re + v.re, u.im + v.im)
  override u.ToString () =
    if u.im >= 0.0 then
      sprintf "(%g + i %g)" u.re u.im
    else
      sprintf "(%g - i %g)" u.re (- u.im)
  override u.Equals obj =
    match obj with
      :? complex as v -> u.re = v.re && u.im = v.im
      | _ -> false
  override u.GetHashCode() = hash _u
  interface System.IComparable with
    member u.CompareTo obj =
      match obj with
        :? complex as v -> compare u.re v.re
        | _ -> invalidArg "obj" "cannot compare values of different types"

let x = complex(1.0, 2.0)
let y = complex(2.5,-1.2)
let lst = [y;x]
printfn "List.sort %A = %A" lst (List.sort lst)
```

```
sporring@Jons-mac 14.1-3 % fsharpi complexCompareTo.fsx
List.sort [(2.5 - i 1.2); (1 + i 2)] = [(1 + i 2); (2.5 - i 1.2)]
```

Opsummering

- Interfaces: angiver egenskaber på tværs af klasser
- Komplekse tal:
  - Mutable,
  - immutable,
  - self-identifier,
  - static,
  - operator,
  - extra constructors,
  - operator overloading
- Copy constructor
- equality og comparable type begrænsninger

Opsummering

- Interfaces: angiver egenskaber på tværs af klasser
- Komplekse tal:
  - Mutable,
  - immutable,
  - self-identifier,
  - static,
  - operator,
  - extra constructors,
  - operator overloading
- Copy constructor
- equality og comparable type begrænsninger

God Jul