Programmering og Problemløsning

3 December 2019

Christina Lioma

c.lioma@di.ku.dk

3/9	Scratch, imperativ programmering, problemløsning
5-12/9	LaTeX og indlejrede strukturer og syntaksfejl, rapportskrivning, problemløsning, kom i gang med F#
17-19/9	Værdier, funktioner, variable og procedurer, program flow, kommentarer
24-26/9	Namespaces and Modules, afprøvning af programmer
1-31/10	Lister, rekursion, typer og mønstergenkendelse, sumtyper og træer
19-28/11	Højereordens funktioner, fejl og untagelser, input/output, internet
3-12/12	**Klasser og objekter**, nedarvning, objektorienteret design
17/12-16/1	UML diagrammer, abstrakte klasser og interfaces, programeksempler, WinForms

Today's lecture

- Object-Oriented Programming (OOP) paradigm
- What is an object
- What is a class
- What is an object instance
  - How to create an object instance
  - How to use an object instance

Object-Oriented Programming

Paradigm: model or set of examples for doing something

Object-Oriented Programming

Paradigm: model or set of examples for doing something

Programming paradigm: how you organise your programs

Object-Oriented Programming

Paradigm: model or set of examples for doing something

Programming paradigm: how you organise your programs

1. Functional
2. Imperative
3. Object-Oriented

Object-Oriented Programming

Paradigm: model or set of examples for doing something

Programming paradigm: how you organise your programs

1. Functional: evaluating functions
2. Imperative: executing statements
3. Object-Oriented

Object-Oriented Programming

Paradigm: model or set of examples for doing something

Programming paradigm: how you organise your programs

1. Functional: evaluating functions
2. Imperative: executing statements
3. Object-Oriented: objects

What is an object

“An object is an abstract data type”

What is an object

"An object is an abstract data type"

An object is a *thing*

- e.g. person, car, country, notion of gravity, music concert…

What is an object

"An object is an abstract data type"

An object is a *thing*

e.g. person, car, country, notion of gravity, music concert…

Attributes

Behaviour

What is an object

"An object is an abstract data type"

An object is a *thing*

e.g. person, car, country, notion of gravity, music concert…

Attributes: name, legs, mouth, brain…

Behaviour: walks, talks, thinks…

What is an object

"An object is an abstract data type"

An object is a *thing*

- e.g. person, car, country, notion of gravity, music concert…
- Attributes: name, legs, mouth, brain…
- Behaviour: walks, talks, thinks…

Programming object

- Attributes: data
- Methods: functions that operate on that data (and possibly other data too)

What is an object

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

What is an object

Encapsulation

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

What is an object

Encapsulation

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

Abstract data type

What is an object

Encapsulation

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

Abstract data type

Built-in data types: integer, float, string…

What is an object

Encapsulation

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

Abstract data types: we invent them

Built-in data types: integer, float, string…

What is an object

Encapsulation

Attributes (data)
Methods (functions)

} glued together into one unit, called *object*

Data Abstraction

Abstract data types: we invent them

(Built-in data types: integer, float, string…)

Program for bank account transactions

Bank account as an object

Program for bank account transactions

Bank account as an object:

Attributes:

- Account number
- Name of account holder
- Amount of money in the account

Methods:

- Take money out
- Put money in

Account object:

number, holder, amount, withdraw, deposit

Account object:

number, holder, amount, withdraw, deposit

All accounts have the above

Account object:

number, holder, amount, withdraw, deposit

All accounts have the above

All accounts can be described by a common template

Account object:

number, holder, amount, withdraw, deposit

All accounts have the above

All accounts can be described by a common template

Class: a template for a collection of objects with the same characteristics

Define a class in F# (implicit)

```
type Account(number : int, holder : string) = class
    let mutable amount = 0

    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount

    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
end
```

Define a class in F# (implicit)

```
type Account(number : int, holder : string) = class
```

- Class declaration

Define a class in F# (implicit)

type Account(number : int, holder : string) = **class**

- Class declaration
- Class constructor (or primary constructor)

Define a class in F# (implicit)

type Account(number : int, holder : string) = **class**

- Class declaration
- Class constructor (or primary constructor)
- Initialises number & holder
  - Can be accessed anywhere inside the class

Define a class in F# (implicit)

type Account(number : int, holder : string) = **class**

- Class declaration
- Class constructor (or primary constructor)
- Initialises number & holder
  - Can be accessed anywhere inside the class
- Class constructor is embedded into class declaration

Define a class in F# (implicit)

type Account(number : int, holder : string) = **class**

- Class declaration
- Class constructor (or primary constructor)
- Initialises number & holder
  - Can be accessed anywhere inside the class
- Class constructor is embedded into class declaration
- Class declaration & class constructor have the same parameters
  - These parameters automatically become immutable

Define a class in F# (implicit)

type Account(number : int, holder : string) = **class**

- Class declaration
- Class constructor (or primary constructor)
- Initialises number & holder
  - Can be accessed anywhere inside the class
- Class constructor is embedded into class declaration
- Class declaration & class constructor have the same parameters
  - These parameters automatically become immutable

int & string not necessary. *Type inference from usage*

Define a class in F# (implicit)

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
```

- Use *let-binding* to define *mutable* attribute

Define a class in F# (implicit)

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
```

- Use *let-binding* to define *mutable* attribute
- When the class is compiled, amount will be compiled as a class attribute
  - Can be accessed anywhere inside the class

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
```

- Class should have *both* attributes & methods as members
- Each member should be defined

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
```

- Class should have *both* attributes & methods as members
- Each member should be defined: self-identifier & .notation

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
```

- Class should have *both* attributes & methods as members
- Each member should be defined: self-identifier & .notation

Self-identifiers: x, me, self, this …

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
```

How to read this:

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
```

How to read this:

- We are defining a member of this class

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
```

How to read this:

- We are defining a member of this class
- This member is called Number

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
```

How to read this:

- We are defining a member of this class
- This member is called Number
- Number belongs to the object Account that is currently in scope and we refer to this by x

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
```

How to read this:

- We are defining a member of this class
- This member is called Number
- Number belongs to the object Account that is currently in scope and we refer to this by x
- The value of Number is given by number

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
end
```

- Methods take input inside brackets

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
end
```

- Methods take input inside brackets
- Methods use their input to operate on the only mutable attribute in this class, *amount*

OO program

Build the class that describes our objects (what we did now)

OO program

1. Build the class that describes our objects (what we did now)
2. Create instances of our objects by calling the class
3. Use the instances of our objects in the program

Create instance of Account

```
type Account(number : int, holder : string) = class
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
end
let max = new Account(123456, “Max Wilson”)
```

- new: creates an instance of class Account
- We pass parameters to the class constructor inside brackets

Class inference

```
type Person(name : string) = class
    member x.Name = name
    member x.SayHello() = printfn “Hi, I’m %s” x.Name
end
```

Class inference: omit *class* and *end*

```
type Person(name : string) = class
    member x.Name = name
    member x.SayHello() = printfn “Hi, I’m %s” x.Name
end
```

OR

```
type Person(name : string) =
    member x.Name = name
    member x.SayHello() = printfn “Hi, I’m %s” x.Name
```

Class & type inference: omit *class*, *end*, *string*

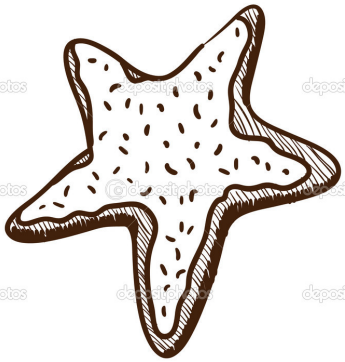
```
type Person(name : string) = class
    member x.Name = name
    member x.SayHello() = printfn “Hi, I’m %s” x.Name
end
```

OR

```
type Person(name) =
    member x.Name = name
    member x.SayHello() = printfn “Hi, I’m %s” x.Name
```

object class instance(s)

object



abstract idea of cookie

class

cookie cutter (template)

instance(s)

the actual cookie(s) we produce using the cutter

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
```

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

Holder: Max Wilson, Amount: 0

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

Build class

Instantiate object

Use object

Holder: Max Wilson, Amount: 0

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
max.Deposit(100)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
max.Deposit(100)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

```
Holder: Max Wilson, Amount: 0
Holder: Max Wilson, Amount: 100
```

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
max.Deposit(100)
max.Deposit(max.Number)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

?

```
type Account(number : int, holder : string) =
    let mutable amount = 0
    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount
    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount – value
let max = new Account(123456, “Max Wilson”)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
max.Deposit(100)
max.Deposit(max.Number)
printfn “Holder: %s, Amount: %i ” max.Holder max.Amount
```

```
Holder: Max Wilson, Amount: 0
Holder: Max Wilson, Amount: 123556
```

Recap today's lecture

- Object-Oriented Programming paradigm
- Object
- Class
- Object instance
- Build class, Create instance, Use instance

Next time: data hiding, access modifiers, instance and static members