Programmering og Problemløsning

9 December 2016

Christina Lioma
c.lioma@di.ku.dk

Today’s lecture

- Class member definition
- Flow of execution
- Scope

```
type Class()
    attribute
    method()

let myInstance = new Class()
myInstance.Method()
```

type Class()	Class declaration & class constructor
attribute method()	Class members
let myInstance = new Class()	Make object instance
myInstance.Method()	Use object instance

```
type Class()
    attribute
        with get()
                                          Make accessible
        and set(…)
    method()

let myInstance = new Class()
myInstance.Method()
myInstance.Attribute <- …                 Access directly
```

Class defines two major aspects of an instance:

- The attributes that are used in each instance
- The operations that are performed on each instance

Class defines two major aspects of an instance:

- The attributes that ~~are~~ **can be** used in each instance

- The operations that ~~are~~ **can be** performed on each instance

Class defines two major aspects of an instance:

- The attributes that ~~are~~ can be used in each instance **but not their values**
- The operations that ~~are~~ can be performed on each instance

Class defines two major aspects of an instance:

- The attributes that ~~are~~ can be used in each instance **but not their values**
- The operations that ~~are~~ can be performed on each instance

Only operation performed on the class: constructor

An instance is related to the class from which it was created ("instance-of" relationship)

```
type Robot(name) = class
   member x.Name = name
   member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Laser(name) = class
   member x.Name = name
   member x.Fire() = printfn “%s is firing" x.Name
end
let Bob = new Laser("Bob")
Bob.SayHello()
```

```
type Robot(name) = class
    member x.Name = name
    member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Laser(name) = class
    member x.Name = name
    member x.Fire() = printfn “%s is firing" x.Name
end
let Bob = new Laser("Bob")
Bob.SayHello()
```

Why is this not working?

```
type Robot(name) = class
   member x.Name = name
   member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Laser(name) = class
   member x.Name = name
   member x.Fire() = printfn “%s is firing" x.Name
end
let Bob = new Laser("Bob")
Bob.SayHello()
```

the laser cannot use the robot’s method

```
type Robot(name) = class
    member x.Name = name
    member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Laser(name) = class
    member x.Name = name
    member x.Fire() = printfn “%s is firing" x.Name
end
let Bob = new Laser("Bob")
Bob.Fire()
```

the laser can only fire

Two (or more) classes can contain members (attributes or methods) that have the **same** name and/or value and/or operation

Two (or more) classes can contain members (attributes or methods) that have the **same** name and/or value and/or operation

Even though these look the same, they are **different** because they belong to different classes (scope)

Two (or more) classes can contain members (attributes or methods) that have the **same** name and/or value and/or operation

Even though these look the same, they are **different** because they belong to different classes (scope)

Each instance can use any/all class members, but only from its own class

```
type Robot(name) = class
   member x.Name = name
   member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Laser(name) = class
   member x.Name = name
   member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let Bob = new Laser("Bob")
Bob.SayHello()
```

This is allowed (not recommended)

Alternative syntax to define several classes

```
type Robot(name) =
    member x.Name = …
    member x.SayHello() = …

and Laser(name) =
    member x.Name = …
    member x.Fire() = …
```

- Although the main reason for creating classes is to encapsulate data & methods, it is possible to have a class that has no data or methods (**empty class**)

- Although the main reason for creating classes is to encapsulate data & methods, it is possible to have a class that has no data or methods (**empty class**)

- Why? Early development – class not fully identified or implemented (stub)

- Although the main reason for creating classes is to encapsulate data & methods, it is possible to have a class that has no data or methods (**empty class**)

- Why? Early development – class not fully identified or implemented (stub)

- Looks empty, but memory space is allocated to it

```
type Robot(name) = class
    member x.Name = name
    member x.SayHello() = printfn "Hi, I'm %s" x.Name
end
let bob = new Robot("Bob")
bob.SayHello()

type Drone() = class end

type Laser(name) = class
    member x.Name = name
    member x.Fire() = printfn “%s is firing" x.Name
end
let Bob = new Laser("Bob")
Bob.Fire()
```

Instance vs static class members

- Instance attribute (*laser serial number*): can have a different value in each instance

- Static attribute (*number of lasers created*): always has the same value in all instances

Instance vs static class members

- Instance attribute (*laser serial number*): can have a different value in each instance

- Static attribute (*number of lasers created*): always has the same value in all instances

```
member x.InstanceAttribute = …
static member StaticAttribute = …
```

```
type Laser(name) = class
	member x.Name = name
	member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

Extend this program so that it prints out the total number of lasers created (work in groups – 5 minutes)

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

What output does this give?

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

```
Lasers created: 1
Lasers created: 2
Lasers created: 3
Super Laser is firing
Giga Laser is firing
Turbo Laser is firing
```

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
```

~~laser1.Fire()~~

~~laser2.Fire()~~

~~laser3.Fire()~~

```
Lasers created: 1
Lasers created: 2
Lasers created: 3
```

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
~~laser1.Fire()~~
~~laser2.Fire()~~
~~laser3.Fire()~~
```

```
Lasers created: 1
Lasers created: 2
Lasers created: 3
```

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
~~laser1.Fire()~~
~~laser2.Fire()~~
~~laser3.Fire()~~
```

*executes when instance is **built**, not used*

```
Lasers created: 1
Lasers created: 2
Lasers created: 3
```

let and *do* bindings in class definition

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions
- *let* before *do*. Why?

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
...
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions
- *let* before *do*. Why? *let* bindings initialise values, and *do* bindings operate on initialised values

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions
- *let* before *do*. Why? *let* bindings initialise values, and *do* bindings operate on initialised values
- *"do"* (in *do* binding): optional for modules but compulsory for classes

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions
- *let* before *do*. Why? *let* bindings initialise values, and *do* bindings operate on initialised values
- *"do"* (in *do* binding): optional for modules but compulsory for classes
- *let** & *do** (can have zero or more)

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

let and *do* bindings in class definition

- *let/do*: after class declaration but before member definitions
- *let* before *do*. Why? *let* bindings initialise values, and *do* bindings operate on initialised values
- *“do”* (in *do* binding): optional for modules but compulsory for classes
- *let** & *do** (can have zero or more)
- *let/do*: can be instance or static (instance by default)

```
type Laser(name) =
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
…
```

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        printfn "Lasers created: %i" count
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
```

```
type Laser(name) = class
      static let mutable count = 0
      do
            count <- count + 1
            ~~printfn "Lasers created: %i" count~~
      member x.Name = name
      static member LaserCount = count
      member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
```

How else can I display the number of lasers created (count)?

```
type Laser(name) = class
        static let mutable count = 0
        do
                count <- count + 1
                ~~printfn "Lasers created: %i" count~~
        member x.Name = name
        static member LaserCount = count
        member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
```

How else can I display the number of lasers created (count)?

1. *Define new method*
2. *Use get()*

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        ~~printfn "Lasers created: %i" count~~
    member x.Name = name
    static member LaserCount = count
    member x.Fire() = printfn "%s is firing" x.Name
    static member ShowCount() = printfn "Count is: %i" Laser.LaserCount
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
```

How else can I display the number of lasers created (count)?

1.Define new method

2. Use get()

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
    member x.Name = name
    static member LaserCount
        with get() = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
printfn “Laser count: %i” Laser.LaserCount
```

How else can I display the number of lasers created (count)?

1. *Define new method*
2. ***Use get()***

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
    member x.Name = name
    static member LaserCount
        with get() = count
    member x.Fire() = printfn "%s is firing" x.Name
end
let laser1 = new Laser("Super Laser”)
let laser2 = new Laser("Giga Laser”)
let laser3 = new Laser("Turbo Laser")
printfn “Laser count: %i” Laser.LaserCount
```

The scope of LaserCount is the whole class

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
    member x.Name = name
    static member LaserCount
        with get() = count
    member x.Fire() = printfn "%s is firing" x.Name
end
~~let laser1 = new Laser("Super Laser”)~~
~~let laser2 = new Laser("Giga Laser”)~~
~~let laser3 = new Laser("Turbo Laser")~~
printfn “Laser count: %i” Laser.LaserCount
```

Will this run?

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
    member x.Name = name
    static member LaserCount
        with get() = count
    member x.Fire() = printfn "%s is firing" x.Name
end
~~let laser1 = new Laser("Super Laser”)~~
~~let laser2 = new Laser("Giga Laser”)~~
~~let laser3 = new Laser("Turbo Laser")~~
printfn “Laser count: %i” Laser.LaserCount
```

Will this run? Yes! What’s its output?

When a class member is static

- it has the same value for all its object instances

When a class member is static

- it has the same value for all its object instances
- It can be accessed before any object is instantiated & without reference to any object instance

When a class member is static

- it has the same value for all its object instances
- It can be accessed before any object is instantiated & without reference to any object instance

```
type Laser(name) = class
    static let mutable count = 0
    do
        count <- count + 1
        member x.Name = name
    static member LaserCount
        with get() = count
    member x.Fire() = printfn "%s is firing" x.Name
end
printfn “Laser count: %i” Laser.LaserCount
```

Class defines two major aspects of an instance:

- The attributes that ~~are~~ can be used in each instance but not their values
- The operations that ~~are~~ can be performed on each instance

Only operations performed on the class: constructor, **selected do bindings, static methods**

An instance is related to the class from which it was created ("instance-of" relationship)

Recap today’s lecture

- Class member definition
- Flow of execution
- Scope