Programmering og Problemløsning

7 December 2018

Jon Sporring og Christina Lioma

```
type Robot(name : string) =
    member this.Name = name
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot("Bob")
bob.SayHello()
```

```
Hi, I'm Bob
```

```
type Robot(name : string) =
    member this.Name = name
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot("Bob")
bob.SayHello()
```

The class is called **Robot**

The object instance is called **bob**

Object instance bob has an property Name whose value is **Bob**

```
type Robot(name : string) =
    member this.Name = name
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot("Bob")
bob.SayHello()
```

- Class definition
- Class declaration & class primary constructor
- Instantiate new object
- Use instantiated object

```
type Robot(name : string) =
    member this.Name = name
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot("Bob")
bob.SayHello()
```

What if we wish to change the value of bob’s name?

```
let bob = Robot("Robert")
```

New object, old not changed

```
bob.Name <- "Robert"
```

“Property 'Name’ cannot be set”.

Inside the class: all fields and members are accessible
Outside the class: <u>only</u> members are accessible

```
type Robot(name : string) =
    let mutable theName = name

    member this.Name = theName
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot("Bob")
bob.SayHello()
bob.Name <- "Robert"
bob.SayHello()
```

“Property 'Name' cannot be set”

Inside the class: all fields and members are accessible

Outside the class: <u>only</u> members are accessible

```
type Robot(name : string) =
   let mutable theName = name

   member this.Name = theName
   member this.setName (aName: string) = theName <- aName
   member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot ("Bob")
bob.SayHello()
bob.setName("Robert")
bob.SayHello()
```

Hi, I’m Bob
Hi, I’m Robert

get() and set() syntax

<u>Without get() and set()</u>

```
let mutable internalName = initialValue
…
member this.propertyName = internalName
member this.setPropertyName (aValue) =
  internalName <- aValue
```

<u>With get() and set()</u>

```
let mutable internalName = initialValue
…
member alias.propertyName
    with get() = internalName
    and set(new-value) = internalName <- aValue
```

```
type Robot(name : string) =
    let mutable theName = name

    member this.Name = theName
    member this.setName (aName: string) = theName <- aName
    member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = Robot ("Bob")
bob.SayHello()
bob.setName("Robert")
bob.SayHello()
```

```
type Robot(name : string) =
   let mutable theName = name

   member this.Name
          with get() = theName
          and set(aName) = theName <- aName
   member this.SayHello() = printfn "Hi, I'm %s" this.Name

let bob = new Robot("Bob")
bob.SayHello()
bob.Name <- "Robert"
bob.SayHello()
```

Recab

```
type Class()
    property
    method()
```

Class declaration & class constructor

Class members

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

Make object instance

Use object instance

Recab

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

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

Static = same data for all objects, as if it were a module

”static” keyword makes fields and members identical for all objects

```
type SomeClass(property : int) = class
    static mutable i = 0
    member this.Property = property
    static member StaticProperty = “This is a static property”
    …
end
```

Laser factory

```
type Laser(name) =
    member this.Name = name
    member this.Fire() = printfn "%s is firing" this.Name

let laser1 = Laser("Super Laser")
let laser2 = Laser("Giga Laser")
let laser3 = Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

```
Super Laser is firing
Giga Laser is firing
Turbo Laser is firing
```

Laser factory: Unique id

```
type Laser(name) =
    static let mutable count = 0
    static do printfn "Laser class created"
    do count <- count + 1
    do printfn "Lasers created: %d" count
    member this.Name = name
    static member LaserCount = count
    member this.Fire() = printfn "%s is firing" this.Name

let laser1 = Laser("Super Laser")
let laser2 = Laser("Giga Laser")
let laser3 = Laser("Turbo Laser")
laser1.Fire()
laser2.Fire()
laser3.Fire()
```

What output does this give?

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

Calling static members without objects

```
type Laser(name) =
    static let mutable count = 0
    do count <- count + 1
    member this.Name = name
    static member LaserCount
        with get() = count
    member this.Fire() = printfn "%s is firing" this.Name

printfn "Laser count: %d" Laser.LaserCount
```

Will this run?

```
Laser count: 0
```

Recab Static class members:

- have the **same value for all their instances**
- can be accessed:
  - **before** any object is instantiated
  - **without** any object being instantiated
  - without reference to any instance (but with **direct reference to the class**)

Mutual dependent classes

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

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

- Although the main reason for creating classes is to encapsulate fields and functions, 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)

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

type Drone() = class end

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

Recap today's lecture

- Data hiding
- Access modifiers
- Instance and Static members