Programmering og Problemløsning

13 December 2016

Christina Lioma

c.lioma@di.ku.dk

Announcement

No PoP classes (neither lectures, nor labs)
on January 2nd 2017

Today's lecture

- Class relationships (inheritance)
  - Scope
  - Accessibility
  - Constructors

Next(), Next(max), Next(min, max)

- *Next(): returns non-negative integer*
- *Next(max): returns non-negative integer up to but* ***excluding max***
- *Next(min, max): returns non-negative integer from* ***min (inclusive)*** *up to and* ***excluding max***

*https://docs.microsoft.com/en-us/dotnet/core/api/system.random#System_Random_Next_System_Int32_*

Next(), Next(max), Next(min, max)

- *Next(): returns ~~non-negative~~ integer*
- *Next(max): returns ~~non-negative~~ integer up to but **excluding max***
- *Next(min, max): returns ~~non-negative~~ integer from **min (inclusive)** up to and **excluding max***

*https://docs.microsoft.com/en-us/dotnet/core/api/system.random#System_Random_Next_System_Int32_*

Next(), Next(max), Next(min, max)

- *Next(): returns integer*
- *Next(max): returns integer up to but **excluding max***
- *Next(min, max): returns integer from **min (inclusive)** up to and **excluding max***
  - *1st value must be smaller or equal to 2nd value*

*https://docs.microsoft.com/en-us/dotnet/core/api/system.random#System_Random_Next_System_Int32_*

```
type Laser(power, accuracy) = class
    Power = … remaining battery power
    Accuracy = … in finding target
    Shoot() = … power decreases
    Scan() =  … power decreases but accuracy increases
end


type SpeedLaser(power, accuracy) = class
    Power = … remaining battery power
    Accuracy = … in finding target
    Shoot() = … power decreases
    Scan() =  … power decreases but accuracy increases
    SpeedShoot() = … shoots at tiny intervals
end
```

```
type Laser(power, accuracy) = class
    Power = … remaining battery power
    Accuracy = … in finding target
    Shoot() = … power decreases
    Scan() =  … power decreases but accuracy increases
end
```

identical

```
type SpeedLaser(power, accuracy) = class
    Power = … remaining battery power
    Accuracy = … in finding target
    Shoot() = … power decreases
    Scan() =  … power decreases but accuracy increases
    SpeedShoot() = … shoots at tiny intervals
end
```

```
type Laser(power, accuracy) = class
    Power = …
    Accuracy = …
    Shoot() = …
    Scan() =  …
end
```

Base class

```
type SpeedLaser (power, accuracy) = class
    inherit Laser(power, accuracy)
    SpeedShoot() = …
end
```

Derived class from the base

Inheritance

Inheritance

BaseClass (a.k.a. *Parent* or *Super* class)

- attributes
- methods

DerivedClass (a.k.a. *Child* or *Sub* class)
inherits **all** attributes & methods from Base

- newAttributes
- newMethods
- can add new attributes & methods in Derived, but
- Base cannot access them

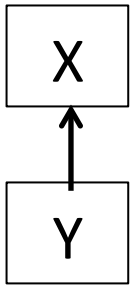
```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power

type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = Laser(90.0, 90.0)
let laser2 = SpeedLaser(100.0, 100.0)
laser1.Shoot()
laser2.Shoot()
```

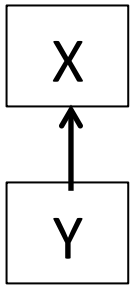
Power left: 89.000000
Power left: 99.000000

Inheritance types

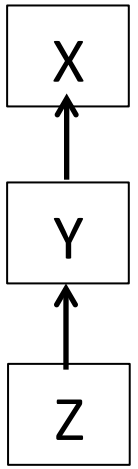


Single

Inheritance types

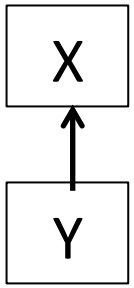


Single



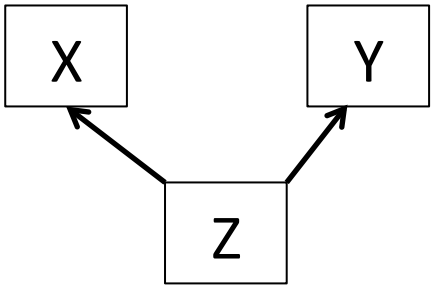
Multi-level

Inheritance types



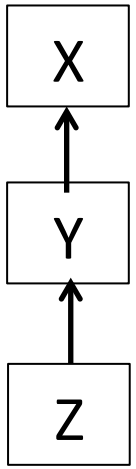
Single

F# allows



Multiple

F# does not allow



Multi-level

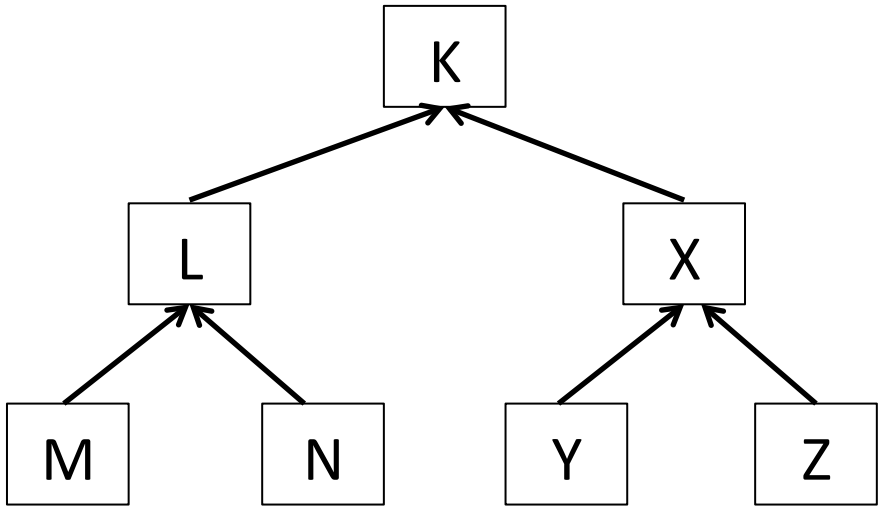
F# allows

Inheritance

- Code reusability (inherited class members)
- Code extensibility (new Derived class members extend Base)

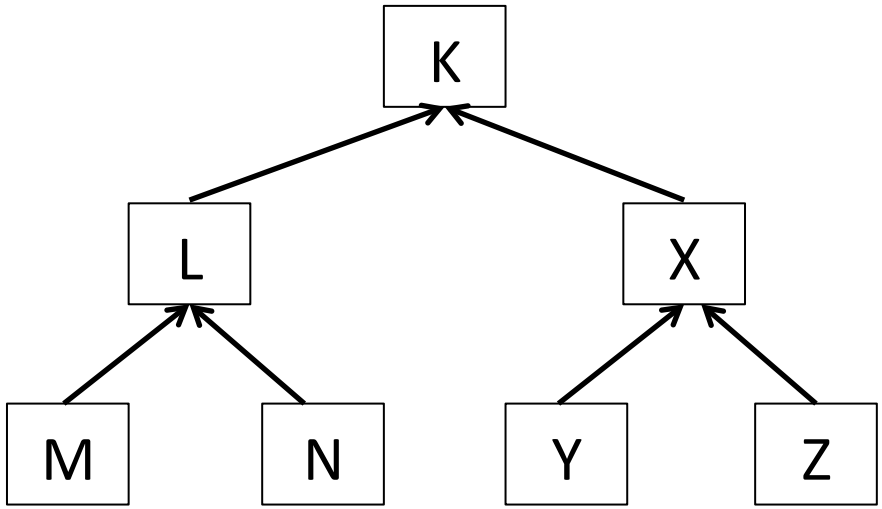
Inheritance

- Code reusability (inherited class members)
- Code extensibility (new Derived class members extend Base)
- If Base changes, all its Derived classes are affected



Inheritance

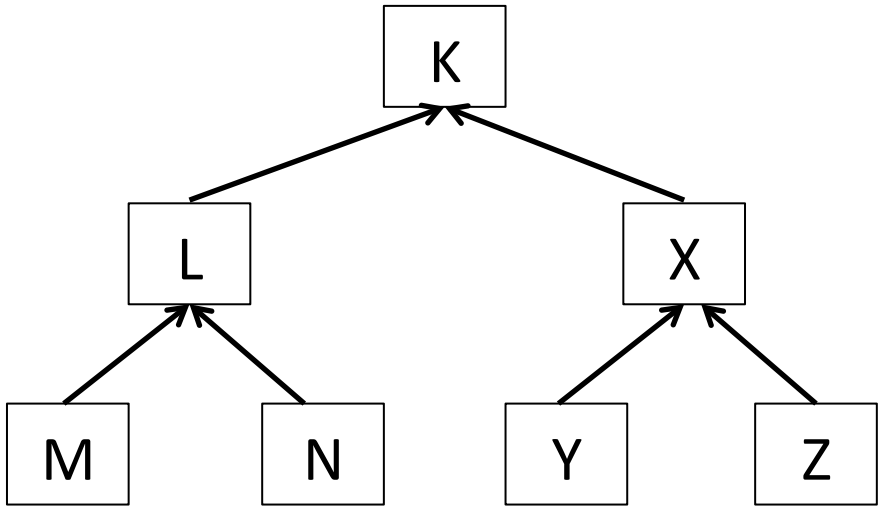
- Code reusability (inherited class members)
- Code extensibility (new Derived class members extend Base)
- If Base changes, all its Derived classes are affected



- In large class hierarchy, several class members remain unused even though memory is allocated to them

Inheritance

- Code reusability (inherited class members)
- Code extensibility (new Derived class members extend Base)
- If Base changes, all its Derived classes are affected



- In large class hierarchy, several class members remain unused even though memory is allocated to them
- If no base class is specified, we implicitly inherit from System.Object

What happens with inheritance?

- *Instance vs static members*
- *Accessibility with get() and set()*
- *Empty classes*
- *Additional constructors*

What happens with inheritance?

- *Instance vs static members: inherited without problems*
- *Accessibility with get() and set() : inherited without problems*
- *Empty classes: inherited without problems*
- *Additional constructors: special inheritance*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100, 100)
laser1.Shoot()
let laser2 = SpeedLaser(200, 200)
laser2.Shoot()
```

If BaseClass has additional constructors, are they inherited?

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100, 100)
laser1.Shoot()
let laser2 = SpeedLaser(200, 200)
laser2.Shoot()
```

If BaseClass has additional constructors, are they inherited?

*Line 12: “A unique overload for method 'Laser' could not be determined based on type information prior to this program point. A type annotation may be needed. Candidates: new : p:float * a:float -> Laser, new : p:int * a:int -> Laser”*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100, 100)
laser1.Shoot()
let laser2 = SpeedLaser(200, 200)
laser2.Shoot()
```

If BaseClass has additional constructors, are they inherited?

← ERROR OCCURS HERE

*Line 12: “A unique overload for method 'Laser' could not be determined based on type information prior to this program point. A type annotation may be needed. Candidates: new : p:float * a:float -> Laser, new : p:int * a:int -> Laser”*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100.0, 100.0)
laser1.Shoot()
let laser2 = SpeedLaser(200.0, 200.0)
laser2.Shoot()
```

OK, I will run both instances with floats (primary constructor)

← **CALLING PRIMARY CONSTRUCTOR**

← **CALLING PRIMARY CONSTRUCTOR**

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100.0, 100.0)
laser1.Shoot()
let laser2 = SpeedLaser(200.0, 200.0)
laser2.Shoot()
```

OK, I will run both instances with floats (primary constructor)

← CALLING PRIMARY CONSTRUCTOR

← CALLING PRIMARY CONSTRUCTOR

*Line 12: “A unique overload for method 'Laser' could not be determined based on type information prior to this program point. A type annotation may be needed. Candidates: new : p:float * a:float -> Laser, new : p:int * a:int -> Laser”*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100.0, 100.0)
laser1.Shoot()
let laser2 = SpeedLaser(200.0, 200.0)
laser2.Shoot()
```

OK, I will skip the inherited instance altogether

← **CALLING PRIMARY CONSTRUCTOR**

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)
let laser1 = Laser(100.0, 100.0)
laser1.Shoot()
let laser2 = SpeedLaser(200.0, 200.0)
laser2.Shoot()
```

OK, I will skip the inherited instance altogether
Now I cannot even run Base!

← **CALLING PRIMARY CONSTRUCTOR**

*Line 12: “A unique overload for method 'Laser' could not be determined based on type information prior to this program point. A type annotation may be needed. Candidates: new : p:float * a:float -> Laser, new : p:int * a:int -> Laser”*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p : float, a : float)
let laser1 = Laser(100.0, 100.0)
laser1.Shoot()
~~let laser2 = SpeedLaser(200.0, 200.0)~~
~~laser2.Shoot()~~
```

If Base has additional constructor(s), must specify which constructor is inherited

← **SPECIFY INHERITED CONSTRUCTOR**

Multiple constructor inheritance

Possible to inherit more than one constructor using F#'s explicit syntax:

https://msdn.microsoft.com/en-us/library/dd233225.aspx

Inheritance

BaseClass

- BaseAttributes
- BaseMethods

DerivedClass

- BaseAttributes
- BaseMethods
- NewAttributes
- NewMethods

Derived inherits **all** attributes & methods from *Base*

Inheritance

BaseClass

- BaseAttributes
- BaseMethods

DerivedClass

- BaseAttributes
- BaseMethods
- NewAttributes
- NewMethods

Derived inherits **all non-private** attributes & methods from *Base*

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

Power left: 79.000000

```
type Laser(p, a) =
        let mutable power = p
        let mutable accuracy = a
        member private x.Shoot() =
                power <- power – 1.0
                printfn “Power left: %f” power
type SpeedLaser(p, a) =
        inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

Make Shoot() private

Shoot is not accessible

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Accuracy = accuracy
    member private x.Power = power
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” x.Power
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

What if power is private?

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Accuracy = accuracy
    member private x.Power = power
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” x.Power
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

What if power is private?
It works

Power left: 79.000000

```
type Laser(p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Accuracy = accuracy
    member private x.Power =
        with get() = power
        and set(value) = power <- value
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” x.Power
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
printfn “Power left: %f” laser1.Power
```

What if power is private?
But this does not work

Power is not accessible

```
type Laser private (p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80, 90)
laser1.Shoot()
```

What if the Base primary constructor is private?

```
type Laser private (p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80, 90)
laser1.Shoot()
```

What if the Base primary constructor is private?

It works. Why?

Power left: 79.000000

```
type Laser private (p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80, 90)
laser1.Shoot()
```

Power left: 79.000000

What if the Base primary constructor is private?

It works. Why? Because it resolves the method overload

```
type Laser private (p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80, 90)
laser1.Shoot()
```

What if the Base primary constructor is private?

It works. Why? Because it resolves the method overload

Note: output is float!

Power left: 79.000000

```
type Laser private (p, a) =
    let mutable power = p
    let mutable accuracy = a
    member x.Shoot() =
        power <- power – 1.0
        printfn “Power left: %f” power
    new(p : int, a : int) =
        let floatP = float(p)
        let floatA= float(a)
        Laser(floatP, floatA)
type SpeedLaser(p, a) =
    inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

What if the Base primary constructor is private?

What about this?

```
type Laser private (p, a) =
        let mutable power = p
        let mutable accuracy = a
        member x.Shoot() =
                power <- power – 1.0
                printfn “Power left: %f” power
        new(p : int, a : int) =
                let floatP = float(p)
                let floatA= float(a)
                Laser(floatP, floatA)
type SpeedLaser(p, a) =
        inherit Laser(p, a)

let laser1 = SpeedLaser(80.0, 90.0)
laser1.Shoot()
```

What if the Base primary constructor is private?

What about this? Does not work because input arguments can only be integers (we have not inherited the method overload)

Inheritance and Method Overloading

Derived inherits **all non-private** attributes & methods from *Base*

If *Base* has additional constructors that overload its methods,
then the overload to be inherited must be specified

Inheritance and Method Overloading

Derived inherits **all non-private** attributes & methods from *Base*

If *Base* has additional constructors that overload its methods,
then the **non-private** overload to be inherited must be specified

Inheritance and Method Overloading

Derived inherits **all non-private** attributes & methods from *Base*

If *Base* has **non-private** constructors that overload its methods,
then the **non-private** overload to be inherited must be specified

Object-Oriented Programming Principles

- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Methods: functions designed to operate on object data (and maybe other data too)

Methods: functions designed to operate on object data (and maybe other data too)

Implement & expose object functionality

Methods: functions designed to operate on object data (and maybe other data too)

Implement & expose object functionality

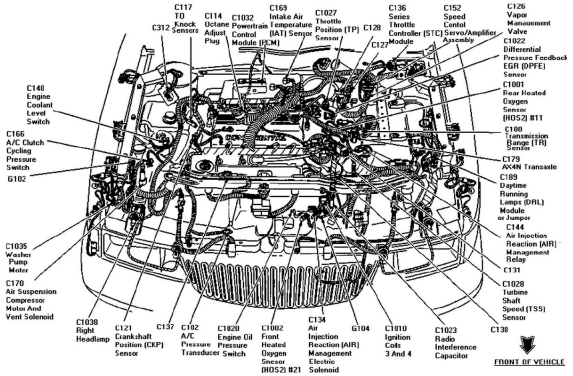
Our interaction with an object instance is defined by & limited to the methods offered by the class

Methods: functions designed to operate on object data (and maybe other data too)

Interface

Implement & expose object functionality

Our interaction with an object instance is defined by & limited to the methods offered by the class

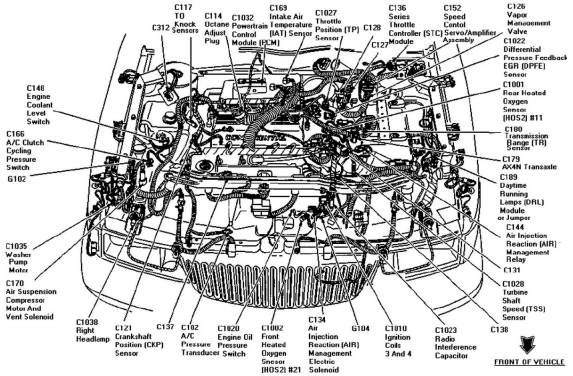


Methods: functions designed to operate on object data (**and maybe other data too**)

Interface

Implement & expose object functionality

Our interaction with an object instance is defined by & limited to the methods offered by the class



A method can operate on different types of data

A method can operate on different types of data

```
> 10+5;;
val it : int = 15
> "car"+"park";;
val it : string = "carpark”
```

A method can operate on different types of data

The operation changes according to the data type

```
> 10+5;;
val it : int = 15
> "car"+"park";;
val it : string = "carpark”
```

arithmetic sum

string concatenation

Polymorphism

A method can operate on different types of data

The operation changes according to the data type

The same method can take different *shapes*

```
> 10+5;;
val it : int = 15                    arithmetic sum
> "car"+"park";;
val it : string = "carpark”          string concatenation
```

Recap of today's lecture

- Class relationships (inheritance)
  - Scope
  - Accessibility
  - Constructors
- Polymorphism

Appendix:
Method overloading & Polymorphism

- **Overloading**: creating a method with the same name but a different amount of parameters or parameters of different data type

- **Polymorphism**: changing the functionality of a method across various types