header
primary constr
body
add. constr.
members

Chapter 19

Object-oriented programming

also known as
or attributes

Object-oriented programming is a programming paradigm that focusses on *objects* such as a person, place, thing, event, and concept relevant for the problem. Objects may contain data and code, which in the object-oriented paradigm are called *properties* and *methods*. Object-oriented programming is an extension of data types, in the sense that objects contains both data and functions in a similar manner as a module, but object-oriented programming emphasizes the semantic unity of the data and functions. Thus, objects are often *models* of real world entities, and object-oriented programming leads to a particular style of programming analysis and design called *object-oriented analysis and design*.

· Object-oriented programming
· objects
· properties
· methods
· models
· object-oriented analysis
· object-oriented design

19.1 Constructors and members

An object is a variable of a class type. A class is defined using the "type" keyword, and there are *always* parentheses after the class name.

Listing 19.1, class.fsx:
A class defintion and an object of this class.

```
1  type aClass(anArgument : int) =
2    // Constructor section of aClass
3    do printfn "A class has been created"
4    let objectValue = anArgument
5    // Member section of aClass
6    member this.value = anArgument
7    member this.scale (factor : int) = factor * objectValue
8
9  let a = aClass(2)
10 printfn "%d %d" a.value (a.scale(3))
```

```
1  $ fsharpc --nologo class.fsx && mono class.exe
2  A class has been created
3  2 6
```

In the example, the class `aClass` is defined in the header in line 1, and it includes one integer argument, `anArugment`. Classes can also be defined without arguments, but the parentheses cannot be omitted. The body of the class line 2–7 is indicated by the indentation. The arguments are immutable. The body consists of two parts: The *constructor section* in line 2–4 and *member section* in line 5–7. In

· constructor section
· member section

but the constructor is implied in line 1.
primary

relate to encapsulation.

sometimes called

line 9 is an object a of `aClass` type created, which implies that memory is reserved and initialisation code is run, and in line 10 is the object used.

or main

In object-oriented lingo, the initialization code is called the *constructor*, and in contrast to many other languages, the constructor is implicitly stated. It is called the *primary constructor*, the arguments given in the header are the primary constructor's arguments, and ~~the primary constructor's body are the~~ "`let`" and "`do`" statements following the header.[1] Members are not available in the constructor unless the self-identifier has been declared in the header using the keyword *"as"*, e.g., `type classMutable(name : string) as this = ....` → elaborate here. Why do this? Example (concrete).

· constructor
· primary constructor
· "as"

→ In this example the constructor's body . . .

not here if at all.

Class~~es~~ may have members declared using the keyword *"member"*, which must come after the constructor, and which define~~s~~ values~~,~~ and functions that are accessible from outside the class using the "."-notation. In this manner, the members define the *interface* between the internal bindings in the constructor and an application program. In object-oriented lingo the value and variable members are called *properties* and functions are called *methods*. Members are values, i.e., immutable. In the example, line 6 and 7 define~~s~~ a property and a method.

terminology

is this introduced?

· "member"
· interface
· properties
· methods

At the begin. of the chapter, you introduced properties & methods. Now you suddenly talk of members & functions. Then you resolve the confusion by mapping properties to members and methods to functions. But the whole confusion should be removed.

In the class definition in Listing 19.1 we bind the primary constructor's arguments to the member values `this.value`. The prefix `this.` is a *self-identifier* used in the definition of the class such that `this.value` is the name of the `objectValue` value for the particular object being constructed. E.g., when a is created in line 9, `this.value` refers to `a.value`. As a quirk, F# is very flexible regarding what name can be used for the self-identifier, and the member section could as well have been `self.value`, `__.value`, or anything else, and it need not be the same in every member definition. In Listing 19.1 we also declare a member function, `this.scale : int -> int`. ★

· self-identifier

not anything!

vague

unwrap this: functions that can be defined to be accessible outside the class or only inside the class (refer to chapt. x).

??

The body of member has access to arguments, the primary constructor's bindings, and to all class members, regardless of the member's lexicographical order.

add frame

Member values and member functions belong~~s~~ to objects, and the implication is the example `value` and `scale` '~~resides' on~~ or 'belongs' to each object.

→ I find this very confusing. I will discuss it with you.

As an aside, if we wanted to use a tuple argument for the class, then this must be explicitly annotated, since the call to the constructor looks identical. This is demonstrated in Listing 19.2.

Listing 19.2 classTuple.fsx:
Beware: Creating objects from classes with several arguments and tuples have the same syntax.

```
type vectorWTupleArgs(x : float * float) =
  member this.cartesian = x
type vectorWTwoArgs(x : float, y : float) =
  member this.cartesian = (x,y)
let v = vectorWTupleArgs(1.0, 2.0)
let w = vectorWTwoArgs(1.0, 2.0)
```

If I did not know, and read this for the first time, I would not get this.

is this introduced?

Whether the full list of arguments should be transported from the caller to the object as a tuple or not is a matter of taste that mainly influences the header of the class. The same cannot be said about how the elements of the vector are stored inside the object and made accessible outside the object. In Listing 19.2, the difference between storing the vector's elements in individual members `member this.x = x` and `member this.y = y` or as a tuple `member this.cartesian = (x, y)`, is that in order to access the first element in a vector `v` an application program would in the first case must write `v.x`, while in the second case the application program must first retrieve the tuple and then extract the first element, e.g., as `fst v.cartesian`. Which is to be preferred depends very much on the application: Is it the individual elements or the complete tuple of elements that is to have focus, when using the objects.

how does it influence it? by showing? ← too vague

→ expand

[1] Jon: define "`do`" statement somewhere used in loops!

★ Advice about consistent self-identifiers?

Class is like a library?
caller ⇒ application

Said ~~in a~~ differently, which choice will make the easiest to read application program with the lowest risk of programming errors. Hence, when designing classes, **consider carefully how application programs will use the class, and aim for simplicity and versatility while minimizing the risk of error in the application program.** Advice

19.2 Accessors

Methods are most often used as an interface between the local bindings of an object and the application program. Consider the example in Listing 19.3.

Listing 19.3, classAccessor.fsx:
Accessor methods interface with internal bindings.

```
type aClass () =
  let mutable v = 1
  member this.setValue (newValue : int) : unit =
    v <- newValue
  member this.getValue () : int = v
let a = aClass()
printfn "%d" (a.getValue ())
a.setValue(2)
printfn "%d" (a.getValue ())
```

```
$ fsharpc --nologo classAccessor.fsx && mono classAccessor.exe
1
2
```

In the above example, the methods `setValue` and `getValue` set~~s~~ and get~~s~~ the state of the objects. Such functions are called *accessors*. Internal states with setters and getters are a typical construction, since it allows for complicated computations, when states are read to and written from, and gives the designer of the class the freedom to change the internal representation while keeping the interface the same. Accessors are so common that F# includes a special syntax for them: Classes can be made to act like variables using "`member`"..."`with`"..."and" keywords and the special function bindings "`get()`" and "`set()`" as demonstrated in Listing 19.4. · accessors

So far, I would not know what set and get actually do.
I would be confused about:
- what type of complicated computations can be done (when I do not even understand the simple computations that they are meant to do)
- what 'states' exactly means
- what exactly is 'change internal representation'

All this should come <u>after</u> an explanation of what set & get do, and why, and they should be expanded with examples.

Listing 19.4, classGetSet.fsx:
Members can act as variables with the builtin getter and setter functions.

```
type aClass () =
  let mutable v = 0
  member this.value
    with get() = v
    and set(a) = v <- a

let a = aClass()
printfn "%d" a.value
a.value<-2
printfn "%d" a.value
```

```
$ fsharpc --nologo classGetSet.fsx && mono classGetSet.exe
0
2
```

The expression defining `get: () -> 'a` and `set: 'a -> ()`, where `'a` is any type, can be any usual expression. The application calls the `get` and `set` as if the member was a mutable value. If `set` is omitted, then the member act as a value rather than a variable, and values cannot be assigned to it in the application program.

Setters and getters are so common that F# has a short-hand for this using "`member val value = 0 with get, set`", which creates the internal mutable value `value`, but this is discouraged in this text.

Defining a *Item* member with extended `get` and `set` makes objects act as indexed variables as demonstrated in Listing 19.5. · Item

Listing 19.5, classGetSetIndexed.fsx:
Members can act as index variables with the builtin getter and setter functions.

```
type aClass(size : int) =
  let arr = Array.create<int> size 0
  member this.Item
    with get(ind : int) = arr.[ind]
    and set(ind : int) (p : int) = arr.[ind] <- p

let a = aClass(3)
printfn "%A" a
printfn "%d %d %d" a.[0] a.[1] a.[2]
a.[1] <- 3
printfn "%d %d %d" a.[0] a.[1] a.[2]
```

```
$ fsharpc --nologo classGetSetIndexed.fsx && mono classGetSetIndexed.exe
ClassGetSetIndexed+aClass
0 0 0
0 3 0
```

Higher order indexed variables are defined by adding more indexing arguments to the definition of `get` and `set`.

Combinations of non-static member definitions is shown in Listing 19.6.

Listing 19.6 classMemberDefinition.fsx:
A large variation of class member definitions. This program intentionally does not compile, but demonstrates variation that will, and problematic lines are indicated by the in-code comments.

```
type Test() =
  let letV = 0
  [<DefaultValue>] val mutable valMutableV : int // Discouraged
  let mutable letMutableV = 0
  member val memberValV = 0 // Discouraged
  member this.memberThisV = 0
  member mutable this.memberMutableThisV = 0 // Error, mutable members are illegal
  member val memberValGetV = 0 with get
  member val memberValSetV = 0 with set // Error, must have get
  member val memberValGetSetV = 0 with get, set
  member this.memberThisGetV
    with get() = letMutableV // Read from object variable
  member this.memberThisSetV
    with set(value) = letMutableV <- value // Save to object variable
  member this.memberThisGetSetV
    with get() = letMutableV // Read from object variable
    and set(value) = letMutableV <- value // Save to object variable

let t = Test()
printfn "%A" t.letV // Error: internal
t.letV <- 1 // Error: internal
printfn "%A" t.valMutableV
t.valMutableV <- 1
printfn "%A" t.letMutableV // Error: internal
t.letMutableV <- 1 // Error: internal
printfn "%A" t.memberValV
t.memberValV <- 1 // Error: read-only
printfn "%A" t.memberThisV
t.memberThisV <- 1 // Error: read-only
printfn "%A" t.memberValGetV
t.memberValGetV <- 1 // Error: read-only
printfn "%A" t.memberValGetSetV
t.memberValGetSetV <- 1
printfn "%A" t.memberThisGetV
t.memberThisGetV <- 1 // Error: read-only
printfn "%A" t.memberThisSetV // Error: write-only
t.memberThisSetV <- 1
printfn "%A" t.memberThisGetSetV
t.memberThisGetSetV <- 1
```

confusing

The val-keyword in this context has not been discussed previously.[2] `staticMemberV` and `staticMemberValV` have the same interface. The `[<DefaultValue>] val mutable valMutableV : int` has not been discussed and is discouraged, but gives a mutable member value that is initialized to the type's default value, e.g., `Unchecked.defaultof<int>` in this case. *[<DefaultValue>]* is called an *attribute*, but will not be discussed further.[3] Defining mutable member variables is illegal, but allowed using explicit

· [<DefaultValue>]
· attribute

[2] Jon: maybe mention "Explicit Field", https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/members/explicit-fields-the-val-keyword.

[3] Jon: Should attributes be included https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/attributes?

get and set functions. The definitions for `valMutableV`, `memberValGetSetV` and `memberThisGetSetV` gives the same interface to a mutable variable, but with slight variation in how their initial value is set, and how get and set actions can be programmed. In general, **minimize the use of constructions using the "`val`"-keyword in class definitions for brevity.** Advice All the above holds for static definitions except `[<DefaultValue>] static val mutable staticValMutableV : int` is illegal.

↪ This last sentence is ungrammatical.

19.3 Objects are reference types

Objects are reference type values, implying that copying objects copies their references not their values. Consider the example in Listing 19.7.

Listing 19.7, classReference.fsx:
Objects are reference types means assignment is aliasing.

```
1 type aClass () =
2   let mutable v = 0
3   member this.value with get() = v and set(a) = v <- a
4
5 let a = aClass()
6 let b = a
7 a.value<-2
8 printfn "%d %d" a.value b.value
```

```
1 $ fsharpc --nologo classReference.fsx && mono classReference.exe
2 2 2
```

Thus, the binding to b in line 6 is an alias to a, not a copy, and changing object a also changes b! This is a common cause of error, and you should **think of objects as arrays.** Advice For this reason, it is often seen that classes implement a copy function, returning a new object with copied values, e.g., Listing 19.8.

Listing 19.8, classCopy.fsx:
A copy method is often needed. Compare with Listing 19.7.

```
type aClass () =
  let mutable v = 0
  member this.value with get() = v and set(a) = v <- a
  member this.copy() =
    let o = aClass()
    o.value <- v
    o
let a = aClass()
let b = a.copy()
a.value<-2
printfn "%d %d" a.value b.value
```

```
$ fsharpc --nologo classCopy.fsx && mono classCopy.exe
2 0
```

In the example we see that since b now is a copy, we do not change it by changing a.

19.4 Static classes

Classes can act as modules and hold properties and methods that are identical for all objects of ~~its~~ their type. These are defined as *static members* using the *"static"*-keyword. They are accessed using the class name and not the object name~~s~~. An example is given in Listing 19.9.

ignore, my fault

· static members
· "static"

Listing 19.9, classStatic.fsx:
Static local variables and members are identical to all objects of the type.

```
1 type aClass(v : int) =
2   static let mutable c = 1
3   member this.value = c * v
4   static member factor with get() = c and set(a) = c <- a
5 let a = aClass(2)
6 let b = aClass(3)
7 printfn "%d %d" a.value b.value
8 aClass.factor <- 2 // Change value for all objects
9 printfn "%d %d" a.value b.value
```

```
$ fsharpc --nologo classStatic.fsx && mono classStatic.exe
2 3
4 6
```

In line 8 the static member function is access changing the local static mutable variable, and since it is static, it is the same variable for both object a and b. Hence next time a.value and b.value are accessed, they both have changed.

19.5 Mutual recursive classes

Classes are inherently recursive and the body of the class can refer to static members of itself. For mutually recursive classes, use the keyword *"and"* as shown in Listing 19.10.

· "and"

Listing 19.10, classAssymetry.fsx:
Mutually recursive classes are defined using the " " keyword.

```
type anInt(v : int) =
  member this.value = v
  member this.add (w : float) : aFloat = aFloat((float this.value) + w)
and aFloat(w : float) =
  member this.value = w
  member this.add (v : int) : aFloat = aFloat((float v) + this.value)
let a = anInt(2)
let b = aFloat(3.2)
let c = a.add(b.value)
let d = b.add(a.value)
printfn "%A %A %A %A" a.value b.value c.value d.value
```

```
$ fsharpc --nologo classAssymetry.fsx && mono classAssymetry.exe
2 3.2 5.2 5.2
```

Here anInt and aFloat are mutually dependent, and thus must be defined in the same "type" definition using "and".

19.6 Function and operator overloading

Member functions may be overloaded, as shown in Listing 19.11.

Listing 19.11, classOverload.fsx:
Overloading methods. Members set : int -> () and set : int * int -> () are permitted since they differ in argument number or type.

```
type Greetings () =
  let mutable greetings = "Hi"
  let mutable name = "Programmer"
  member this.str = greetings + " " + name
  member this.setName(newName : string) : unit =
    name <- newName
  member this.setName(newName : string, newGreetings : string) : unit =
    greetings <- newGreetings
    name <- newName
let a = Greetings()
printfn "%s" a.str
a.setName("F# programmer")
printfn "%s" a.str
a.setName("Expert", "Hello")
printfn "%s" a.str
```

```
$ fsharpc --nologo classOverload.fsx && mono classOverload.exe
Hi Programmer
Hi F# programmer
Hello Expert
```

In the example two methods are defined both named set, but with different number of arguments. This is called *overloading* and is allowed as long as the arguments differ in number or type.

· overloading

In Listing 19.10 the notation for addition is less than elegant. For such situations, F# supports operator overloading. To overload the "+" operator we overload its functional equivalence (+) as a static member as shown in Listing 19.12.

Listing 19.12, classOverloadOperator.fsx:
Operators can be overloaded using.

```
type anInt(v : int) =
  member this.value = v
  static member (+) (v : anInt, w : float) = (float v.value) + w
  static member (+) (w : float, v : anInt) = v + w
let a = anInt(2)
let b = a + 3.2
let c = 3.2 + a
printfn "%A %A %A" a.value b c
```

```
$ fsharpc --nologo classOverloadOperator.fsx && mono
   classOverloadOperator.exe
2 5.2 5.2
```

refer to earlier chapter where they have been defined

I don't get this

All usual operators may be overloaded and the compiler uses type inference to decide which function is to be called.[4] All operators have a functional equivalence, and writing `v + w` is equivalent to writing `vector.(+) (v, w)`. Presently the former is to be preferred, but at times, e.g., when using functions as arguments, it is useful to be able to refer to an operator by its function equivalent. Note that the functional equivalence of the multiplication operator `(*)` shares a prefix with the begin block comment lexeme "`(*`", which is why the multiplication function is written as `( * )`. Note also that unitary operators have a special notation, and in the above case unitary minus should be defined as `static member (~-) (v : anInt) = - v`. Can you give an example where this is useful?

refer to earlier chapter where this was introduced

In Listing 19.12, notice how the second (+) operator overloads the first by calling the first with the proper order of arguments. This is a general principle, **avoid duplication of code, reuse of existing code is almost always preferred.** Here it is to be preferred for two reasons. Firstly, if we discover a mistake in the multiplication code, then we need only correct it once, which implies that both multiplication methods are corrected once and reducing the chance of introducing new mistakes by attempting to correct~~ing~~ old. Secondly, if we later decide~~s~~ to change the internal representation of the vector, then we only need to update one version of the multiplication function, hence we reduce programming time and risk of errors as well. Advice

Beware that operator overloading outside class definitions overwrites *all* definitions of the operator. E.g., overloading `vector.(+) (v, w)` outside a class will influence integer, real, string, etc. Thus, **operator overloading should only be done inside class definitions.** [5] Advice

Overloading is a programming structure that does not work well in functional-first programming languages such as F#, since types are not easily inferred. Therefore, overloading has restricted usage. E.g., even in the above case, if the application program attempt~~s~~ed to define the following function `let f = anInt.(+)`, the functional equivalent of the "+" operator, then the compiler will not be able to identify, which of the two candiate functions ~~are~~ is to be bound to `f`, and will through throw an error. Although irrelevant until `f` is applied to arguments, an effective type inference system has yet to be implemented.

[4] Jon: **Something about which the syntax for new operators allowed.**

[5] Jon: **Something about which operators can be overloaded** https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/operator-overloading.

yes!
Also, why would we want to overload operators? Motivation

19.7 Additional constructors

→ expand, explain, motivate

The constructor itself can be overloaded, to allow for varying parameters at time of object creation. Additional constructors may be created using the *new*-keyword as illustrated in Listing 19.13.

· new

a.k.a. auxiliary or secondary

Listing 19.13, classExtraConstructor.fsx:
Extra constructors can be added using “ ”.

```
type classExtraConstructor(name : string, greetings : string) =
  static let defaultGreetings = "Hello"
  // Additional constructors are defined by new()
  new(name : string) =
    classExtraConstructor(name, defaultGreetings)
  member this.name = name
  member this.greeting = greetings + " " + name

let s = classExtraConstructor("F#") // Calling additional constructor
let t = classExtraConstructor("F#", "Hi") // Calling primary constructor
printfn "\"%A\", \"%A\"" s.greeting t.greeting
```

```
$ fsharpc --nologo classExtraConstructor.fsx && mono
  classExtraConstructor.exe
""Hello F#"", ""Hi F#""
```

The body of the additional constructor must call the primary constructor, and the body cannot extend the primary constructor's values and functions. It is useful to **think the primary constructor as a superset of arguments and the additional as subsets or specialisations.** As regular scope rules dictate, the additional constructor has access to the primary constructor's bindings. However, in order to access the object's members, the self-identifier has to be explicitly declared using the "as"-keyword in the header. E.g., writing `new(x : float, y : float) as alsoThis = ...`. However beware, even though the body of the additional constructor now may access the member value `alsoThis.x`, this value has first been created once the primary constructor has been called. E.g., calling the primary constructor in the additional constructor as `new(x : float, y : float) as alsoThis = classExtraConstructor(fst alsoThis.x, y, defaultSeparator)` will cause an exception at runtime. Code may be executed in additional constructors: Before the call to the primary constructor, "let" and "do" statements are allowed. If code is to be executed after the primary constructor has been called, then it must be preceded by the "then" keyword as shown in Listing 19.14.

Advice