1 Classes and Objects

Object-oriented programming is a programming paradigm that focuses on objects such as a persons, places, things, events, and concepts relevant for the problem.

Object-oriented programming has a rich language for describing objects and their relations, which can seem overwhelming at first, and they will be explained in detail in this and following chapters. Here is a brief overview: The main programming structures are called a *classes* and *objects*. It is useful to think of classes as user defined types and objects as values of such types. However, there is more to classes and objects than types and values. Objects may contain both data and code, and it is sometimes useful to draw the corresponding class definition as shown in Figure 1.1. In this illustration, objects of type `aClass` will each contain an int and a pair of a

aClass
// The object's values (properties) aValue : int anotherValue : float*bool
// The object's functions (methods) aMethod: () -> int anotherMethod: float -> float

Figure 1.1: A class is sometimes drawn as a figure.

float and a boolean, and each object has two functions associated with them. The values stored in each object may differ, but the types are fixed by the class definition. It is common to call an object's values *properties* and an object's functions *methods*. In short, properties and methods are collectively called *members*. When an object is created, memory is set aside on *The Heap* to each object's property. Creating an object is commonly called *instantiation*. The members serve as the interface to each object, and each instantiated object will have the same type of members as all objects of that class, but their content may differ.

Object-oriented programming is an extension of data types, in the sense that objects contain 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* to be discussed in **??**.

1.1 Constructors and Members

A class is defined using the `type` keyword. Note that there are *always* parentheses after the class name to distinguish it from a regular type definition. The basic syntax for a class definition is as follows:

Listing 1.1: Syntax for simple class definitions.

```
type <classIdent> ({<arg>}) [as <selfIdent>]
  {let <binding> | do <statement>}
  {member <memberDef>}
```

The first line is the header of the class, where the `<classIdent>` is the name of the class, `<arg>` are its optional arguments, and `<selfIdent>` is an optional *self identifier*. The body of a class consists of the constructor and the member section. The header and the constructor section is often collectively called the *constructor*, and the body of the constructor consist of optional `let`-bindings and `do`-statements. Note that the `do`-statements in a class definition *must* use the `do`-keyword. The member section consisting of all the optional member definitions, where each definition use the `member`-keyword.

The header and constructor section is commonly called the *constructor*, and the constructor is executed at instantiation. In contrast to many other languages, the constructor is always stated as the initial code of a class definition. The values and variables in the constructor are called *fields*, while functions are just called *functions*.

Members are declared using the *`member`*-keyword, which defines 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. Member values are called *properties*, and member functions are called *methods*. Note that members are immutable. The body of a member has access to the arguments, the constructor's bindings, and to all class members, regardless of the member's lexicographical order. In contrast, 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 = ...`.

Consider the example in Figure 1.2. Here we have defined a class `car`, instantiated three objects, and bound them to the names `sportsCar`, `stationWagon`, and `miniBus`.

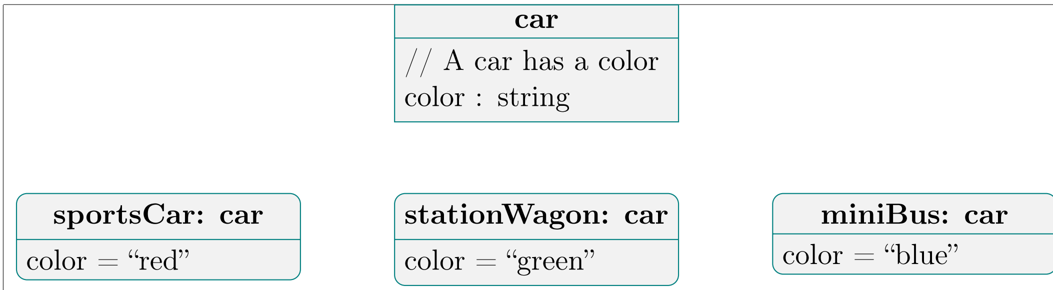

Figure 1.2: A class `car` is instantiated trice and bound to the names `sportsCar`, `stationWagon`, and `miniBus`, and each object's properties are set to different values.

Each object has been given different values for the `color` property. In F# this could look like the code in Listing 1.2. In the example, the class `car` is defined in lines 1–3.

Listing 1.2 car.fsx:
Defining a class car, and making three instances of it. See also Figure 1.2.

```
1 type car (aColor : string) =
2   // Member section
3   member this.color = aColor
4 
5 let sportsCar = car ("red")
6 let stationWagon = car ("green")
7 let miniBus = car ("blue")
8 printfn "%s %s %s" sportsCar.color stationWagon.color
    miniBus.color
```

```
1 $ fsharpc --nologo car.fsx && mono car.exe
2 red green blue
```

Its header includes one string argument, `aColor`. The body of the constructor is empty, and the member section consists of lines 2–3. The class defines one property `color : string`. Note that when referring to a member inside an object, then we must use a *self identifier*. Here we use `this` as the self identifier, and as the example shows, we need not declare it in the class' header. A self identifier refers to the memory set aside to the particular instance of an object. It is common among other programming languages to use `this` as self identifier. 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. Nevertheless, **consistency in the name used as self-**

identifier is strongly encouraged, preferably using a name that reflects the nature of the reference, such as `this` or `me`. The objects are instantiated in lines 5–7, and the value of their properties are accessed in line 8. In many languages, objects are instantiated using the *new* keyword, but in F# this is optional. I.e., `let sportsCar = car ("red")` is identical to `let sportsCar = new car ("red")`. Note that both the self identifier and member access uses the "`.`" notation.

A more advanced implementation of a `car` class might include notions of a fuel gauge, fuel economy, and the ability to update the fuel gauge as the car is driven. An example of an implementation of this is given In Listing 1.3. Here in line 1, the constructor has

Listing 1.3 class.fsx:
Extending Listing 1.2 with fields and methods.

```
1  type car (econ : float, fuel : float) =
2    // Constructor body section
3    let mutable fuelLeft = fuel // liters in the tank
4    do printfn "Created a car (%.1f, %.1f)" econ fuel
5    // Member section
6    member this.fuel = fuelLeft
7    member this.drive distance =
8      fuelLeft <- fuelLeft - econ * distance / 100.0
9
10 let sport = car (8.0, 60.0)
11 let economy = car (5.0, 45.0)
12 sport.drive 100.0
13 economy.drive 100.0
14 printfn "Fuel left after 100km driving:"
15 printfn " sport: %.1f" sport.fuel
16 printfn " economy: %.1f" economy.fuel
```

```
$ fsharpc --nologo class.fsx && mono class.exe
Created a car (8.0, 60.0)
Created a car (5.0, 45.0)
Fuel left after 100km driving:
 sport: 52.0
 economy: 40.0
```

2 arguments: the fuel economy parameter and the initial amount of fuel in the tank. Thus, we are able to create 2 different cars with different fuel economy, as shown in lines 10–11. The amount of fuel left en each car object is stored in the mutable field `fuelLeft`. This is an example of a state of an object: It can be accessed outside the object by the `fuel` property, and it can be updated by the `drive` method.

Field names and functions defined in the constructor do not use the self identifier and cannot be accessed outside and object using the "." notation. However, they are available in both the constructor and the member section following the regular scope rules. Fields are a common way to hide implementation details, and they are *private* to the object or class in contrast to members that are *public*.

1.2 Accessors

Methods are most often used as an interface between the fields of an object and the application program. Consider the example in Listing 1.4. In the example, the

Listing 1.4 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
```

data contained in objects of type `aClass` is stored in the mutable field `v`. Since only members can be accessed from an application, it is not possible to retrieve or change the data of these object of class `aClass` directly. We could have programmed `v` as a member instead, i.e., `member this.v = 1`, however, often we are in a situation, where there is a range of possible choices of data representation, details of which we do wish to share with an application program. E.g., implementation details of arrays are not important for our ability to use them in applications. What matters is that the members that work on the array elements are well defined and efficient. Thus, the example demonstrates how we can build two simple methods `setValue` and `getValue` to set and get the data stored `v`. By making a distinction between the internal representation and how members give access to the data, we retain the possibility to change the internal representation without having to reprogram all the application programs. Analogously, we can change the engine in a car from one type to another without having to change the car's interaction with the driver and the road: steering wheel, pedals, wheels etc.

Such functions are called *accessors*. Internal states with setters and getters are a typical construction, since they allow 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 1.5. The expression defining `get: () -> 'a` and

Listing 1.5 classGetSet.fsx:
Members can act as variables with the built-in get and set 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
```

`set: 'a -> ()`, where `'a` is any type, can be any usual expression. The application calls the `get` and `set` as if the property were a mutable value. If `set` is omitted, then the property acts 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 an *`Item`* property with extended `get` and `set` makes objects act as indexed variables, as demonstrated in Listing 1.6. Higher dimensional indexed properties are defined by adding more indexing arguments to the definition of `get` and `set`, such as demonstrated in Listing 1.7.

Listing 1.6 classGetSetIndexed.fsx:
Properties can act as indexed variables with the built-in get and set 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
```

Listing 1.7 classGetSetHigherIndexed.fsx:
Getters and setters for higher dimensional index variables.

```
type aClass (rows : int, cols : int) =
  let arr = Array2D.create<int> rows cols 0
  member this.Item
    with get (i : int, j : int) = arr.[i,j]
    and set (i : int, j : int) (p : int) = arr.[i,j] <- p

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

```
$ fsharpc --nologo classGetSetHigherIndexed.fsx
$ mono classGetSetHigherIndexed.exe
ClassGetSetHigherIndexed+aClass
0 0 0
0 3 0
```

1.3 Objects are Reference Types

Objects are reference type values, implying that copying objects copies their references, not their values, and their content is stored on *The Heap*, see **??**. Consider the example in Listing 1.8. 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.** For this reason, it is often seen that classes implement a copy function returning a new object with copied values, as shown in Listing 1.9. In the example, we see that since `b` now is a copy, we do not change it by changing `a`. This is called a *copy constructor*.

Listing 1.8 classReference.fsx:
Objects assignment can cause 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
```

1.4 Static Classes

Classes can act as modules and hold data which is identical for all objects of its type. These are defined using the *`static`*-keyword. And since they do not belong to a single object, but are shared between all objects, they are defined without the self-identifier and accessed using the class name, and they cannot refer to the arguments of the constructor. For example, consider a class whose objects each hold a unique identification number (id): When an object is instantiated, the object must be given the next available identification number. The next available id could be given as an argument to the constructor, however, this delegates the task of maintaining the uniqueness of ids to the application program. It is better to use a static field and delegate the administration of ids completely to the constructors, as demonstrated in

Listing 1.9 classCopy.fsx:
A copy method is often needed. Compare with Listing 1.8.

```
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
```

Listing 1.10. Notice in line 2 that a static field `nextAvailableID` is created for the value to be shared by all objects. The initialization of its value is only performed once, at the beginning of program execution. However, every time an object is instantiated, the value of `nextAvailableID` is copied to the object's field `studentID` in line 3, and `nextAvailableID` is updated. The static field can be accessed with a static accessor, as demonstrated in line 7. Notice how this definition does not include a self-identifier, and that the member is accessible from the application in line 11 using the class' name, in both cases since it is not a member of any particular object.

1.5 Recursive Members and Classes

The members of a class are inherently recursive: static and non-static methods may recurse using the self identifier and other members regardless of their lexicographical scope. This is demonstrated in Listing 1.11. For mutually recursive classes, the keyword *and* must be used, as shown in Listing 1.12. Here `anInt` and `aFloat` hold an integer and a floating point value respectively, and they both implement an addition of `anInt` an `aFloat` that returns and `aFloat`. Thus, they are mutually dependent and must be defined in the same `type` definition using `and`.

Listing 1.10 classStatic.fsx:
Static fields and members are identical to all objects of the type.

```
type student (name : string) =
  static let mutable nextAvailableID = 0 // A global id for all objects
  let studentID = nextAvailableID // A per object id
  do nextAvailableID <- nextAvailableID + 1
  member this.id with get () = studentID
  member this.name = name
  static member nextID = nextAvailableID // A global member
let a = student ("Jon") // Students will get unique ids, when instantiated
let b = student ("Hans")
printfn "%s: %d, %s: %d" a.name a.id b.name b.id
printfn "Next id: %d" student.nextID // Accessing the class's member
```

```
$ fsharpc --nologo classStatic.fsx && mono classStatic.exe
Jon: 0, Hans: 1
Next id: 2
```

1.6 Function and Operator Overloading

It is often convenient to define different methods that have the same name, but with functionalities that depend on the number and type of arguments given. This is called *overloading*, and F# supports method overloading. An example is shown in Listing 1.13. In the example we define an object which can produce greetings strings of the form <greeting> <name>, using the `str` member. It has a default greeting "Hi" and name "Programmer", but the name can be changed by calling the `setName` accessor with one argument, and both greeting and name can be changed by calling the overloaded `setName` with two arguments. Overloading in class definition is allowed as long as the arguments differ in number or type.

In Listing 1.12, the notation for addition is less than elegant. For such situations, F# supports *operator overloading*. All usual operators may be overloaded (see **??**), and in contrast to regular operator overloading, the compiler uses type inference to decide which function is to be called. All operators have a functional equivalence, and to overload the binary "+" and unary "-" operators, we overload their functional equivalence `(+)` and `(~-)` as static members. This is demonstrated in Listing 1.14. Thus, writing `v + w` is equivalent to writing `anInt.(+) (v, w)`. Presently, the

Listing 1.11 classRecursion.fsx:
Members can recurse without the `rec` keyword and refer to other members regardless of their lexicographical scope.

```
type twice (v : int) =
  static member fac n = if n > 1 then n * (twice.fac (n-1)) else 1 // No rec
  member this.copy = this.twice // No lexicographical scope
  member this.twice = 2*v

let a = twice (2)
let b = twice.fac 3
printfn "%A %A %A" a.copy a.twice b
```

```
$ fsharpc --nologo classRecursion.fsx && mono classRecursion.exe
4 4 6
```

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 using the "~"-lexeme, as illustrated in the above example for unitary minus. With the unitary minus, we are able to subtract objects of `anInt` by first negating the right-hand operand and then adding the result to the left-hand operand. In contrast, the binary minus would have been defined as `static member (-) (v : anInt, w : aFloat) = anInt ((float v.value) - w.value)`.

In Listing 1.14, 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 reduces the chance of introducing new mistakes by attempting to correct old ones. Secondly, if we later decide to change the internal representation, then we only need to update one version of the multiplication function, hence we reduce programming time and risk of errors as well.

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

Listing 1.12 classAssymetry.fsx:
Mutually recursive classes are defined using the and keyword.

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

definitions.

1.7 Additional Constructors

Like methods, constructors can also be overloaded by using the *`new`* keyword. E.g., the example in Listing 1.13 may be modified, such that the name and possibly greeting is set at object instantiation rather than by using the accessor. This is illustrated in Listing 1.15. The top constructor that does not use the `new`-keyword is called the *primary constructor*. The body of the additional constructor must call the primary constructor, and the body cannot extend the primary constructor's fields and functions. It is useful to **think of the primary constructor as a superset of arguments and the additional ones as subsets or specializations.** 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 property `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 result in an

Listing 1.13 classOverload.fsx:
Overloading methods `set : int -> ()` and `set : int * int -> ()` is 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
```

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 1.16. The `do`-keyword is often understood to be implied by F#, e.g., in front of all `printf`-statements, but in the above examples they are required where used. This may change in future releases of F#. F# allows for many additional constructors, but they must be distinguishable by type.

Listing 1.14 classOverloadOperator.fsx:
Operators can be overloaded using their functional equivalents.

```
type anInt (v : int) =
  member this.value = v
  static member (+) (v : anInt, w : anInt) = anInt
  (v.value + w.value)
  static member (~-) (v : anInt) = anInt (-v.value)
and aFloat (w : float) =
  member this.value = w
  static member (+) (v : aFloat, w : aFloat) = aFloat
  (v.value + w.value)
  static member (+) (v : anInt, w : aFloat) =
    aFloat ((float v.value) + w.value)
  static member (+) (w : aFloat, v : anInt) = v + w //
  reuse def. above
  static member (~-) (v : aFloat) = aFloat (-v.value)

let a = anInt (2)
let b = anInt (3)
let c = aFloat (3.2)
let d = a + b // anInt + anInt
let e = c + a // aFloat + anInt
let f = a + c // anInt + aFloat
let g = -a // unitary minus anInt
let h = a + -b // anInt + unitary minus anInt
printf "a=%A, b=%A, c=%A, d=%A" a.value b.value c.value
  d.value
printf ", e=%A, f=%A, g=%A, h=%A" e.value f.value g.value
  h.value
```

```
$ fsharpc --nologo classOverloadOperator.fsx
$ mono classOverloadOperator.exe
a=2, b=3, c=3.2, d=5, e=5.2, f=5.2, g=-2, h=-1
```

Listing 1.15 classExtraConstructor.fsx:
Extra constructors can be added, using `new`.

```
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.str = greetings + " " + name

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

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

Listing 1.16 classDoThen.fsx:
The optional `do`- and `then`-keywords allow for computations before and after the primary constructor is called.

```
type classDoThen (aValue : float) =
  // "do" is mandatory to execute code in the primary
   constructor
  do printfn "  Primary constructor called"
  // Some calculations
  do printfn "  Primary done" (* *)
  new () =
    // "do" is optional in additional constructors
    printfn "  Additional constructor called"
    classDoThen (0.0)
    // Use "then" to execute code after construction
    then
      printfn "  Additional done"
  member this.value = aValue

printfn "Calling additional constructor"
let v = classDoThen ()
printfn "Calling primary constructor"
let w = classDoThen (1.0)
```

```
$ fsharpc --nologo classDoThen.fsx && mono classDoThen.exe
Calling additional constructor
  Additional constructor called
  Primary constructor called
  Primary done
  Additional done
Calling primary constructor
  Primary constructor called
  Primary done
```

1.8 Programming Intermezzo: Two Dimensional Vectors

Consider the following problem.

Problem 1.1

A Euclidean vector is a geometric object that has a direction, a length, and two operations: vector addition and scalar multiplication, see Figure 1.3. Define a class for a vector in two dimensions.

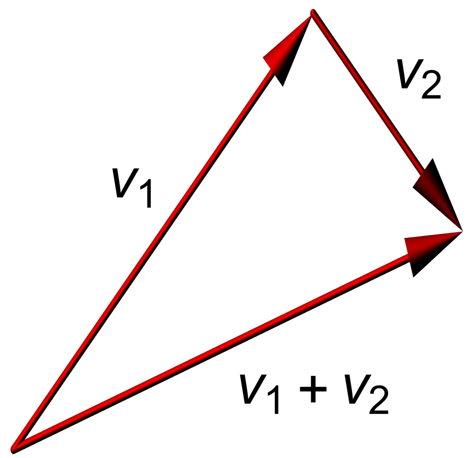

Figure 1.3: Illustration of vector addition in two dimensions.

An essential part in designing a solution for the above problem is to decide which representation to use internally for vectors. The Cartesian representation of a vector is as a tuple of real values (x, y), where x and y are real values, and where we can imagine that the tail of the vector is in the origin, and its tip is at the coordinate (x, y). For vectors on Cartesian form,

$$\vec{v} = (x, y), \tag{1.1}$$

the basic operations are defined as

$$\vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2), \tag{1.2}$$
$$a\vec{v} = (ax, ax), \tag{1.3}$$
$$\mathrm{dir}(\vec{v}) = \tan \frac{y}{x},\ x \neq 0, \tag{1.4}$$
$$\mathrm{len}(\vec{v}) = \sqrt{x^2 + y^2}, \tag{1.5}$$

where x_i and y_i are the elements of vector $\vec{v}_i$, a is a scalar, and dir and len are the direction and length functions, respectively. The polar representation of vectors is also a tuple of real values (θ, l), where θ is the vector's angle from the x-axis and l is the vector's length. This representation is closely tied to the definition of a vector, and has the constraint that $0 \leq \theta < 2\pi$ and $0 \leq l$. This representation reminds us that vectors do not have a position. For vectors on polar form,

$$\vec{v} = (\theta, l), \tag{1.6}$$

their basic operations are defined as

$$x(\theta, l) = l\cos(\theta), \tag{1.7}$$
$$y(\theta, l) = l\sin(\theta), \tag{1.8}$$
$$\vec{v}_1 + \vec{v}_2 = (x(\theta_1, l_1) + x(\theta_2, l_2), y(\theta_1, l_1) + y(\theta_2, l_2)) \tag{1.9}$$
$$a\vec{v} = (\theta, al), \tag{1.10}$$

where θ_i and l_i are the elements of vector $\vec{v}_i$, a is a scalar, and x and y are the Cartesian coordinate functions.

So far in our analysis, we have realized that:

- both the Cartesian and polar representations use a pair of reals to represent the vector,
- both require functions to calculate the elements of the other representation,
- the polar representation is invalid for negative lengths, and
- the addition operator under the polar representation is also more complicated and essentially requires access to the Cartesian representation.

The first step in shaping our solution is to decide on file structure: For conceptual separation, we choose to use a library and an application file. F# wants files to define namespaces or modules, so we choose the library to be a `Geometry` module, which implements the vector class to be called `vector`. Furthermore, when creating vector objects we would like to give the application program the ability to choose either Cartesian or polar form. This is can be done using *discriminated unions*. Discriminated unions allow us to tag values of possibly identical form, but they also lead to longer programs. Thus, we will also provide an additional constructor on implicit Cartesian form, since this is the most common representation of vectors.

A key point when defining libraries is to consider their interface with the application program. Hence, our second step is to write an application using the yet to be written library in order to get a feel for how such an interface could be. This is demonstrated in the application program Listing 1.17. The application of the vector class seems natural, makes use of the optional discriminated unions, uses the infix operators "`+`" and "`*`" in a manner close to standard arithmetic, and interacts smoothly with the

Listing 1.17 vectorApp.fsx:
An application using the library in Listing 1.18.

```
open Geometry
let v = vector(Cartesian (1.0,2.0))
let w = vector(Polar (3.2,1.8))
let p = vector()
let q = vector(1.2, -0.9)
let a = 1.5
printfn "%A * %A = %A" a v (a * v)
printfn "%A + %A = %A" v w (v + w)
printfn "vector() = %A" p
printfn "vector(1.2, -0.9) = %A" q
printfn "v.dir = %A" v.dir
printfn "v.len = %A" v.len
```

`printf` family. Thus, we have further sketched requirements to the library with the emphasis on application.

After a couple of trials, our library implementation has ended up as shown in Listing 1.18. Realizations achieved during writing this code are: Firstly, in order to implement a vector class using discriminated unions, we had to introduce a constructor with helper variables `_x`, `_y`, etc. The consequence is that the Cartesian and polar representation is evaluated once and only once every time an object is created. Unfortunately, discriminated unions do not implement guards on subsets, so we still have to cast an exception when the application attempts to create an object with a negative length. Secondly, for the `ToString` override we have implemented static members for typesetting vectors, since it seems more appropriate that all vectors should be typeset identically. Changing typesetting thus respects dynamic scope.

The output of our combined library and application is shown in Listing 1.19. The output is as expected, and for the vector class, our solution seems to be a good compromise between versatility and syntactical bloating.

Listing 1.18 vector.fs:
A library serving the application in Listing 1.19.

```
module Geometry
type Coordinate =
  Cartesian of float * float // (x, y)
  | Polar of float * float // (dir, len)
type vector(c : Coordinate) =
  let (_x, _y, _dir, _len) =
    match c with
      Cartesian (x, y) ->
        (x, y, atan2 y x, sqrt (x * x + y * y))
      | Polar (dir, len) when len >= 0.0 ->
        (len * cos dir, len * sin dir, dir, len)
      | Polar (dir, _) ->
        failwith "Negative length in polar representation."
  new(x : float, y : float) =
    vector(Cartesian (x, y))
  new() =
    vector(Cartesian (0.0, 0.0))
  member this.x = _x
  member this.y = _y
  member this.len = _len
  member this.dir = _dir
  static member val left = "(" with get, set
  static member val right = ")" with get, set
  static member val sep = ", " with get, set
  static member ( * ) (a : float, v : vector) : vector =
    vector(Polar (v.dir, a * v.len))
  static member ( * ) (v : vector, a : float) : vector =
    a * v
  static member (+) (v : vector, w : vector) : vector =
    vector(Cartesian (v.x + w.x, v.y + w.y))
  override this.ToString() =
    sprintf "%s%A%s%A%s" vector.left this.x vector.sep
  this.y vector.right
```

Listing 1.19: Compiling and running the code from Listing 1.18 and 1.17.

```
$ fsharpc --nologo vector.fs vectorApp.fsx && mono
    vectorApp.exe
1.5 * (1.0, 2.0) = (1.5, 3.0)
(1.0, 2.0) + (-1.796930596, -0.1050734582) =
    (-0.7969305964, 1.894926542)
vector() = (0.0, 0.0)
vector(1.2, -0.9) = (1.2, -0.9)
v.dir = 1.107148718
v.len = 2.236067977
```