20 Classes and objects

Object-oriented programming is a programming paradigm that focusses on *objects* such as a person, place, thing, event, and concept relevant for the problem.

· object-oriented programming
· objects

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: Objects may contain data and code, which may be either *public* or *private*. An object's *members* are the object's public values and functions. Public values are called *properties* or *attributes*, and public functions are called *methods*, and these can be accessed using the "." notation similarly to modules and namespaces. Private values are called *fields* and private functions are just called *functions*, and these can only be used by code inside the object. The type definition of an object is called a *class*, while values of the class are called *objects*. When objects are created, a special function called the *constructor* is executed. Creating objects is also often referred to as *instantiating* objects.

· public
· private
· members
· properties
· attributes
· methods
· fields
· functions
· class
· objects
· constructor
· instantiation

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 Chapter 22.

· models
· object-oriented analysis
· object-oriented design

20.1 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 to distinguish it from a regular type definition.

· object
· class

Listing 20.1 A class definition.

```
type <classIdent> ({<arg>}) [as <selfIdent>]
  [class]
  [inherit <baseClassIdent>({<arg>})]
  {[let <binding>] | [do <statement>]}
  {(member | abstract member | default | override) <memberDef>}
  [end]
```

The <classIdent> is the name of the class, <arg> are its optional arguments, <selfIdent> is an optional *self identifier*, <baseClassIdent> is the name of another class that this class optionally builds upon using the inherit keyword (see Section 21.1), the optional let-bindings and do-statements define *fields* and *functions*, and <memberDef> are public members, i.e., *properties* and *methods*. Members may be regular members using the member keyword or abstract members using either the abstract

· self identifier
· fields
· functions
· properties
· methods

`member`, `default`, or `override` keywords (see Section 21.2). The *primary constructor* is everything until the first member. Mutably recursive class definition can be defined using the `and` keyword, e.g., `type aClass () = ... and bClass () = ...`. Do statements must use the `do`-keyword.

· primary constructor

An example of a simple program defining a class and creating objects of their type is given in Listing 20.2.

Listing 20.2 class.fsx:
A class defintion and an object of this class.

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

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

In the example, the class `aClass` is defined in the header in line 1, and it includes one integer argument, `anArgument`. Classes can also be defined without arguments, but the parentheses cannot be omitted. Together with the header, line 2–4 is the primary constructor. In the member section line 5–7 is the value `value : int` and function `scale : int -> int` defined using the name `this` as a *self identifier*. If not declared using the `as` keyword in the header, then the self identifier can be any valid identifier. In line 9 and 10 two objects `a` and `b` of type `aClass` are created, which implies that memory is reserved on *The Heap* (see Section 6.8) and the constructor is run for each of them. Thus, for `a`, `this.value` refers to the memory set aside for `a`, and for `b`, `this.value` refers to the memory set aside for `b`. In line 11 are shown examples of their use. Notice, that members are accessed outside the object using the "`.`" notation in the same manner as an application program would access elements of a module. In many languages, objects are instantiated using the *`new`* keyword, but in F# this is optional. I.e., `let a = aClass (2)` is identical to `let a = new aClass (2)`.

· self identifier
· The Heap
· `new`

Class types allow for defining code, which is executed when values of its type are created, i.e., when objects are instantiated. This initialization code is called the *constructor*, and in contrast to many other languages, the constructor is always stated as an integral part of the class header in F# as described above. It is called the *primary constructor*, its arguments are specified in the header, and the primary constructor's body is the `let` and `do`-bindings following the header. The values and variables in the constructor are called *fields*, while functions are just called *functions*. Note that 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 = ...`.

· constructor
· primary constructor
· fields
· functions
· `as`

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* or *attributes*. Note that the concept of attributes as member values is different from the concept of functions and `let`-binding attributes, which is specified with the `[<>]` notation. For this reason, this author prefers the name member property in F#. Member functions are called *methods*.

· `member`
· interface
· properties
· attributes
· methods

Note that members are immutable. In the example in Listing 20.2, line 6 and 7 defines a property and a method. Properties and methods belong to objects, and the implication is the example `value` and `scale` 'resides' on or 'belongs' to each object. The body of a member has access to arguments, the primary constructor's bindings, and to all class' members, regardless of the member's lexicographical order.

In the class definition in Listing 20.2 we bind the primary constructor's arguments to the property `this.value`. The prefix `this.` is a *self identifier* used in the definition of the class such that, e.g., `this.value` is the name of the `objectValue` value for the particular object being constructed. 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, however, **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`.**

· self identifier

Advice

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 20.3.

Listing 20.3 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)
```

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 20.3, 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 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. Said 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

20.2 Accessors

Methods are most often used as an interface between the fields of an object and the application program. Consider the example in Listing 20.4.

Listing 20.4 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
```

In the example, the 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 the 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 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 20.5. · accessors

Listing 20.5 classGetSet.fsx:
Members can act as variables with the built-in 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 property were a mutable value. If `set` is omitted, then the property 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 an *`Item`* property with extended `get` and `set` makes objects act as indexed variables as demonstrated in Listing 20.6. · `Item`

Listing 20.6 classGetSetIndexed.fsx:
Properties can act as index variables with the built-in 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 dimensional indexed properties are defined by adding more indexing arguments to the definition of `get` and `set` such as demonstrated in Listing 20.7.

Listing 20.7 classGetSetHigherIndexed.fsx:
Properties can act as index variables with the built-in getter and setter functions.

```
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
```

20.3 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 also Section 6.8. Consider the example in Listing 20.8. · The Heap

Listing 20.8 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.** For this reason, it is often seen that classes implement a copy function, returning a new object with copied values, e.g., Listing 20.9. Advice

Listing 20.9 classCopy.fsx:
A copy method is often needed. Compare with Listing 20.8.

```
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
```

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*.

· copy constructor

20.4 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 an example, consider a class whose objects each should 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. Better is to use a static field and delegate the administration of ids completely to the class' and object's constructors as demonstrated in Listing 20.10.

· `static`

Listing 20.10 classStatic.fsx:
Static fields and members are identical to all objects of the type.

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

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

Notice in the example line 2, 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, then 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.

20.5 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 20.11.

Listing 20.11 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
```

For mutually recursive classes, the keyword *and* must be used as shown in Listing 20.12.

· and

Listing 20.12 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
```

Here `anInt` and `aFloat` hold an integer and a floating point value respectively, and they both implement an addition of `anInt` and `aFloat` that returns and `aFloat`. Thus, they are mutually dependent and must be defined in the same `type` definition using `and`.

20.6 Function and operator overloading

It is often convenient to define different methods with the same name, but whose functionality depends on the number and type of arguments given. This is called *overloading* and F# supports method overloading. An example is shown in Listing 20.13.

· overloading

Listing 20.13 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
```

In the example we define an object, which can produce greetings strings on 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 20.12 the notation for addition is less than elegant. For such situations, F# supports *operator overloading*. All usual operators may be overloaded (see Section 6.3), 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 20.14.

· operator overloading

Listing 20.14 classOverloadOperator.fsx:
Operators can be overloaded using.

```
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
```

Thus, writing `v + w` is equivalent to writing `anInt.(+) (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 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, thus demonstrating the difference between unary and binary minus operators, where the binary minus would have been defined as `static member (-) (v : anInt, w : aFloat) = anInt ((float v.value) - w.value)`.

In Listing 20.14, 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 reducing the chance of introducing new mistakes by attempting to correct old. Secondly, if we later decide 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 `(+) (v, w)` outside a class will influence integer, real, string, etc. Thus, **operator overloading should only be done inside class definitions.** Advice

20.7 Additional constructors

Like methods, constructors can also be overloaded using the *new* keyword. E.g., the example in Listing 20.13 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 20.15.

· new

Listing 20.15 classExtraConstructor.fsx:
Extra constructors can be added using new.

```
type classExtraConstructor (name : string, greetings : string) =
  static let defaultGreetings = "Hello"
 // Additional constructor 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#"
```

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 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 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 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 20.16.

Advice

Listing 20.16 classDoThen.fsx:
The optional do- and then-keywords allows 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
```

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.

20.8 Interfacing with `printf` family

In previous examples, we accessed the property in order to print the content of the objects. Luckily, a more elegant solution is available. Objects can be printed directly, but the result is most often not very useful as can be seen in Listing 20.17.

Listing 20.17 classPrintf.fsx:
Printing classes yields low-level information about the class.

```
type vectorDefaultToString (x : float, y : float) =
  member this.x = (x,y)

let v = vectorDefaultToString (1.0, 2.0)
printfn "%A" v // Printing objects gives lowlevel information
```

```
$ fsharpc --nologo classPrintf.fsx && mono classPrintf.exe
ClassPrintf+vectorDefaultToString
```

All classes are given default members through a process called *inheritance*, to be discussed below in Section 21.1. One example is the `ToString() : () -> string` function, which is useful in conjunction with, e.g., `printf`. When an object is given as argument to a `printf` function, then `printf` calls the object's `ToString()` function. The default implementation returns low-level information about the object, as can be seen above, but we may *override* this member using the *`override`*-keyword as demonstrated in Listing 20.18.[1]

· inheritance
· override
· `override`

Listing 20.18 classToString.fsx:
Overriding `ToString()` function for better interaction with members of the `printf` family of procedures. Compare with Listing 20.17.

```
type vectorWToString (x : float, y : float) =
  member this.x = (x,y)
  // Custom printing of objects by overriding this.ToString()
  override this.ToString() =
    sprintf "(%A, %A)" (fst this.x) (snd this.x)

let v = vectorWToString(1.0, 2.0)
printfn "%A" v // No change in application but result is better
```

```
$ fsharpc --nologo classToString.fsx && mono classToString.exe
(1.0, 2.0)
```

We see that as a consequence, the `printf` statement is much simpler. However beware, an application program may require other formatting choices than selected at the time of designing the class, e.g., in our example the application program may prefer square brackets as delimiters for vector tuples. So in general **when designing an override to `ToString()`, choose simple, generic formatting for the widest possible use.**

Advice

The most generic formatting is not always obvious, and in the vector case some candidates for the formatting string of `ToString()` are "`%A %A`", "`%A, %A`", "`(%A, %A)`", or "`[%A, %A]`". Considering each carefully it seems that arguments can be made against all. A common choice is to let the formatting be controlled by static members that can be changed by the application program by accessors.

[1]Jon: **something about ToString not working with 's' format string in printf.**

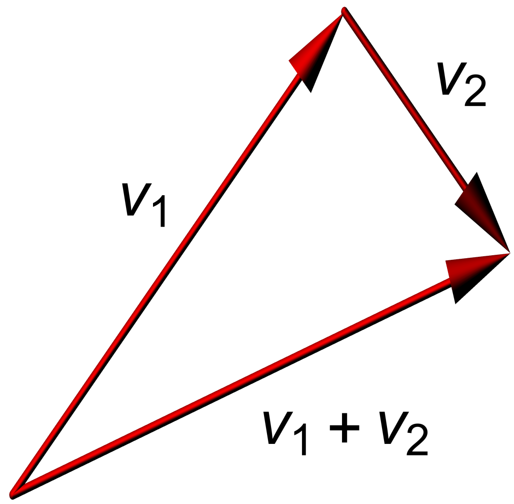

Figure 20.1: Illustration of vector addition in two dimensions.

20.9 Programming intermezzo

Consider the following problem.

Problem 20.1

A Euclidean vector is a geometric object that has a direction and a length and two operations: vector addition and scalar multiplication, see Figure 20.1. Define a class for a vector 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,

$$\vec{v} = (x, y), \tag{20.1}$$

the basic operations are defined as

$$\vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2), \tag{20.2}$$
$$a\vec{v} = (ax, ax), \tag{20.3}$$
$$\mathrm{dir}(\vec{v}) = \tan\frac{y}{x},\ x \neq 0, \tag{20.4}$$
$$\mathrm{len}(\vec{v}) = \sqrt{x^2 + y^2}, \tag{20.5}$$

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. The polar representation of vectors is also a tuple of real values (θ, l), where θ and l are the vector's direction and length. This representation is closely tied to the definition of a vector, and with 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,

$$\vec{v} = (\theta, l), \tag{20.6}$$

their basic operations are defined as

$$x(\theta, l) = l\cos(\theta), \tag{20.7}$$
$$y(\theta, l) = l\sin(\theta), \tag{20.8}$$
$$\vec{v}_1 + \vec{v}_2 = (x(\theta_1, l_1) + x(\theta_2, l_2), y(\theta_1, l_1) + y(\theta_2, l_2)) \tag{20.9}$$
$$a\vec{v} = (\theta, al), \tag{20.10}$$

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 representation uses 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`. Further, 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 implied longer programs. Thus, we will also provide an additional constructor on implicit Cartesian form, since this is the most common representation.

· discriminated unions

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 20.19.

Listing 20.19 vectorApp.fsx:
An application using the library in Listing 20.20.

```
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
```

The application of the vector class seems natural, makes use of the optional discriminated unions, and uses the infix operators "+" and "*" in a manner close to standard arithmetic, and interacts smoothly with the `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 20.20.

Listing 20.20 vector.fs:
A library serving the application in Listing 20.21.

```
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
```

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 respect dynamic scope.

The output of our combined library and application is shown in Listing 20.21.

Listing 20.21: Compiling and running the code from Listing 20.20 and 20.19.

```
$ 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
```

The output is as expected and for the vector class, our solution seems to be a good compromise between versatility and syntactical bloating.

21 | Derived classes

21.1 Inheritance

Sometimes it is useful to derive new classes from old in order to reuse code or to emphasize a program structure. For example, consider the concepts of a *car* and *bicycle*. They are both *vehicles* that can move forward and turn, but a car can move in reverse, has 4 wheels uses gasoline or electricity, while a bicycle has 2 wheels and needs to be pedaled. Structurally we can say that "a car is a vehicle" and "a bicycle is a vehicle". Such a relation is sometimes drawn as a tree as shown in Figure 21.1 and is called an *is-a relation*. Is-a relations can be implemented using class *inheritance*, where vehicle is called the *base class* and car and bicycle are each a *derived class*. The advantage is that a derived class can inherent the members of the base class, *override* and add possibly new members. Inheritance is indicated using the `inherit` keyword. Listing 20.1 shows the syntax for class definitions using inheritance, and an example of defining base and derived classes for vehicles is shown In Listing 21.1.

· is-a relation
· inheritance
· base class
· derived class
· override

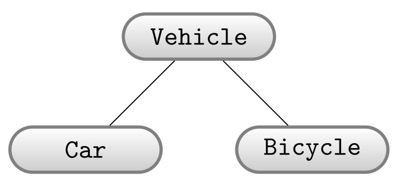

Figure 21.1: Both a car and a bicycle is a (type of) vehicle.

Listing 21.1 vehicle.fsx:
New classes can be derived from old.

```
/// A general vehicle, which moves on a plane and has a heading
type vehicle () =
  let mutable p = (0.0, 0.0) // coordinate on a plane
  let mutable d = 0.0 // heading direction in radians
  member this.dir with get() = d
  member this.pos with get() = p and set aPos = p <- aPos
  member this.turn angle = // turn heading
    d <- d + angle
  member this.forward step = // move forward (abs step)
    let s = abs step
    let vec = (s * (cos d), s * (sin d))
    p <- (fst p + fst vec, snd p + snd vec)
/// A car is a vehicle, has wheels and can move in reverse
type car (name) =
  inherit vehicle () // inherit dir, pos, turn, and forward
  member this.wheels = 4 // A car has 4 wheels
  member this.revese step = // move backwards (abs step)
    let s = - abs step
    let vec = (s * (cos this.dir), s * (sin this.dir))
    this.pos <- (fst this.pos + fst vec, snd this.pos + snd vec)
/// A bicycle is a vehicle and has wheels
type bicycle () =
  inherit vehicle () // inherit dir, pos, turn, and forward
  member this.wheels = 2 // A bike has 4 wheels

let aVehicle = vehicle () // has dir, pos, turn, forward
let aCar = car () // has dir, pos, turn, forward, wheels, reverse
let aBike = bicycle () // has dir, pos, turn, forward, wheels
printfn "The car aCar has %d wheels" aCar.wheels
printfn "The bicycle aBike has %d wheels" aBike.wheels
```

```
$ fsharpc --nologo vehicle.fsx && mono vehicle.exe
The car aCar has 4 wheels
The bicycle aBike has 2 wheels
```

In the example, a base class `vehicle` is defined with members `dir`, `pos`, `turn`, and `forward`. The derived classes inherit all the members of the base class, but do not have access to any non-members of the base's constructor. I.e., `car` and `bicycle` automatically have methods `turn` and `forward`, and properties `dir` and `pos` with their accessors, but they do not have direct access to the fields `p` and `d`. Both derived classes additionally define a property `wheels` and `car` also define a method `reverse`. Note that inheritance is one-way, and in spite that both derived classes define a member `wheels`, the base class does not have a `wheels` member.

Derived classes can replace base class members by defining new members *overshadow* the base' member. The base' members are still available using the *`base`*-keyword. Consider the example in the Listing 21.2.

· overshadow
· `base`

Listing 21.2 memberOvershadowing.fsx:
Inherited members can be overshadowed, but we can still access the base' member.

```
/// A counter has an internal state initialized at instantiation and
/// is incremented in steps of 1
type counter (init : int) =
  let mutable i = init
  member this.value with get () = i and set (v) = i <- v
  member this.inc () = i <- i + 1
/// A counter2 is a counter which increments in steps of 2.
type counter2 (init : int) =
  inherit counter (init)
  member this.inc () = this.value <- this.value + 2
  member this.incByOne () = base.inc () // inc by 1 implemented in base

let c1 = counter (0) // A counter by 1 starting with 0
printf "c1: %d" c1.value
c1.inc() // inc by 1
printfn " %d" c1.value
let c2 = counter2 (1) // A counter by 2 starting with 1
printf "c2: %d" c2.value
c2.inc() // inc by 2
printf " %d" c2.value
c2.incByOne() // inc by 1
printfn " %d" c2.value
```

```
$ fsharpc --nologo memberOvershadowing.fsx
$ mono memberOvershadowing.exe
c1: 0 1
c2: 1 3 4
```

In this case, we have defined two counters, each with an internal field `i` and with members `value` and `inc`. The `inc` method in `counter` increments `i` with 1, and in `counter2` the field `i` is incremented with 2. Note how `counter2` inherits both members `value` and `inc`, but overshadows `inc` by defining its own. Note also how `counter2` defines another method `incByOne` by accessing the inherited `inc` method using the `base` keyword.

Even though derived classes are different from their base, the derived class includes the base class, which can be recalled using *upcasting* by the upcast operator "*:>*". At compile-time this operator removes the additions and overshadowing of the derived class, as illustrated in Listing 21.3.

· upcast
· :>

Listing 21.3 upCasting.fsx:
Objects can be upcasted resulting in an object as if it were its base. Implementations from the derived class are ignored.

```fsharp
/// hello holds property str
type hello () =
  member this.str = "hello"
/// howdy is a hello class and has property altStr
type howdy () =
  inherit hello ()
  member this.str = "howdy"
  member this.altStr = "hi"

let a = hello ()
let b = howdy ()
let c = b :> hello // a howdy object as if it were a hello object
printfn "%s %s %s %s" a.str b.str b.altStr c.str
```

```
$ fsharpc --nologo upCasting.fsx && mono upCasting.exe
hello howdy hi hello
```

Here `howdy` is derived from `hello`, overshadows `str`, and adds property `altStr`. By upcasting object `b`, we create object `c` as a copy of `b` with all its fields, functions, and members as if it had been of type `hello`. I.e., `c` contains the base class version of `str` and does not have property `altStr`. Objects `a` and `c` are now of same type and can be put into, e.g., an array as `let arr = [|a, c|]`. Previously upcasted objects can also be downcasted again using the *downcast* operator `:?>`, but the validity of the operation is checked at runtime. Thus, **avoid downcasting when possible.**

· downcast
· `:?>`
Advice

In the above, inheritance is used to modify and extend any class. I.e., the definition of the base classes were independent on the definitions of inherited classes. In that sense, the base classes were oblivious to any future derivation of them. Sometimes it is useful to define base classes, which are not independent on derived classes, and which impose design constraints on derived classes. Two such dependencies in F# are abstract classes and interfaces to be described in the following sections.

21.2 Abstract class

An *abstract class* contains members defined using the *`abstract member`* and optionally the *`default`* keywords. An `abstract member` in the base class is a type definition, and derived classes must provide an implementation using the *`override`* keyword. Optionally, the base class may provide a default implementation using the `default` keyword, in which case overriding is not required in derived classes. Objects of classes containing abstract members without default implementations cannot be instantiated, but derived classes that provide the missing implementations can be. Note that abstract classes must be given the *[<AbstractClass>]* attribute. Note also that in contrast to overshadowing, upcasting keeps the implementations of the derived classes. Examples of this are shown in Listing 21.4.

· abstract class
· `abstract member`
· `default`
· `override`
· [<AbstractClass>]

Listing 21.4 abstractClass.fsx:
In contrast to regular objects, upcasted derived object use the derived implementation of abstract methods.

```
/// An abstract class for general greeting classes with property str
[<AbstractClass>]
type greeting () =
  abstract member str : string
/// hello is a greeting
type hello () =
  inherit greeting ()
  override this.str = "hello"
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  override this.str = "howdy"

let a = hello ()
let b = howdy ()
let c = [| a :> greeting; b :> greeting |] // arrays of greetings
Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
$ fsharpc --nologo abstractClass.fsx && mono abstractClass.exe
hello
howdy
```

In the example, we define a base class and two derived classes. Note how the abstract member is defined in the base class using the "`:`"-operator as a type declaration rather than a name binding. Note also that since the base class does not provide a default implementation, the derived classes supply an implementation using the `override`-keyword. In the example, objects of `baseClass` cannot be created, since such objects would have no implementation for `this.hello`. Finally, the two different derived and upcasted objects can be put in the same array, and when calling their implementation of `this.hello` we still get the derived implementations, which is in contrast to overshadowing.

Abstract classes may also specify a default implementation, such that derived classes have the option of implementing an overriding member, but are not forced to. In spite that implementations are available in the abstract class, the abstract class still cannot be used to instantiate objects. Such a variant is shown in Listing 21.5.

Listing 21.5 abstractDefaultClass.fsx:
Default implementations in abstract classes makes implementations in derived classes optional. Compare with Listing 21.4.

```
/// An abstract class for general greeting classes with property str
[<AbstractClass>]
type greeting () =
  abstract member str : string
  default this.str = "hello" // Provide default implementation
/// hello is a greeting
type hello () =
  inherit greeting ()
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  override this.str = "howdy"

let a = hello ()
let b = howdy ()
let c = [| a :> greeting; b :> greeting |] // arrays of greetings
Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
$ fsharpc --nologo abstractDefaultClass.fsx
$ mono abstractDefaultClass.exe
hello
howdy
```

In the example, the program in Listing 21.4 has been modified such that `greeting` is given a default implementation for `str`, in which case, `hello` does not need to supply one. However, in order for `howdy` to provide a different greeting, it still needs provide an override member.

Note that even if all abstract members in an abstract class has defaults, objects of its type can still not be created, but must be derived as, e.g., shown with `hello` above.

As a side note, every class implicitly derives from a base class *`System.Object`* which, which is an abstract class defining among other members the `ToString` method with default implementation. · `System.Object`

21.3 Interfaces

Inheritance of an abstract base class allows an application to rely on the definition of the base regardless of any future derived classes. This gives great flexibility, but at times even less knowledge is needed about objects in order to write useful applications. This is what *interfaces* offer. An interface specifies which members must exist but nothing more. Interfaces are defined as an abstract class *without arguments* and *only with abstract members*. Classes implementing interfaces must specify implementations for the abstract members using the *`interface with`* keywords. Objects of classes implementing interfaces can be upcasted as if they had an abstract base class of the interface' name. Consider the example in Listing 21.6.

· interface

· `interface with`

Listing 21.6 classInterface.fsx:
Interfaces specifies which members classes contain, and with upcasting gives more flexibility than abstract classes.

```
/// An interface for classes that have method fct and member value
type IValue =
  abstract member fct : float -> float
  abstract member value : int
/// A house implements the IValue interface
type house (floors: int, baseArea: float) =
  interface IValue with
    // calculate total price based on per area average
    member this.fct (pricePerArea : float) =
      pricePerArea * (float floors) * baseArea
    // return number of floors
    member this.value = floors
/// A person implements the IValue interface
type person(name : string, height: float, age : int) =
  interface IValue with
    // calculate body mass index (kg/(m*m)) using hypothetic mass
    member this.fct (mass : float) = mass / (height * height)
    // return the length of name
    member this.value = name.Length
  member this.data = (name, height, age)

let a = house(2, 70.0) // a two storage house with 70 m*m base area
let b = person("Donald", 1.8, 50) // a 50 year old person 1.8 m high
let lst = [a :> IValue; b :> IValue]
let printInterfacePart (o : IValue) =
  printfn "value = %d, fct(80.0) = %g" o.value (o.fct 80.0)
List.iter printInterfacePart lst
```

```
$ fsharpc --nologo classInterface.fsx && mono classInterface.exe
value = 2, fct(80.0) = 11200
value = 6, fct(80.0) = 24.6914
```

Here, two distinctly different classes are defined: `house` and `person`. These are not related by inheritance since no sensible common structure seems available. However, they share structures in the sense that they both have an integer property and a `float -> float` method. For each of the derived classes, these members have different meanings. Still, some treatment of these members by an application will only rely on their type and not their meaning. E.g., in Listing 21.6 the `printfn` function only needs to know the member's type not their meaning. As a consequence, the application can upcast them both to the implicit abstract base class `IValue`, put them in an array, and apply a function using the member definition of `IValue` with the higher-order `List.iter` function. Another example could be a higher order function calculating average values: For average values of the number of floors and average value of the length of people's names, the higher order function would only need to know that both of these classes implements the `IValue` interfaces in order to calculate the average of list of either objects types.

As a final note, inheritance ties classes together in a class hierarchy. Abstract members enforce inheritance and impose constraints on the derived classes. Like abstract classes, interfaces impose constraints on derived classes, but without requiring a hierarchical structure.

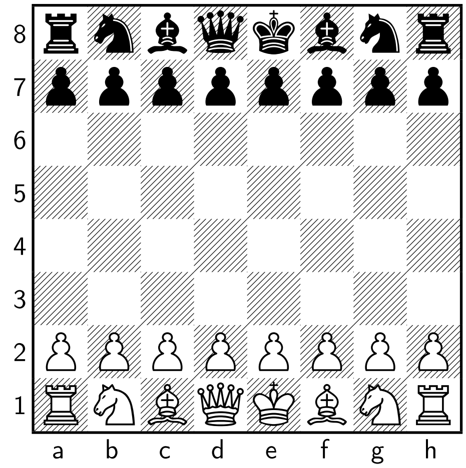

Figure 21.2: Starting position for the game of chess.

21.4 Programming intermezzo: Chess

To demonstrate the use of hierarchies, consider the following problem

Problem 21.1

The game of chess is a turn-based game for two, which consists of a board of 8×8 squares and a set of 16 black and 16 white pieces. A piece can be either a king, queen, rook, bishop, knight or pawn and each piece has a specific movement pattern on the board. Pieces are added to, moved on, and removed from the board during the game, and there can be at most one piece per square. A piece strikes another piece of opposing color by moving to its square and the piece of opposing color is removed from the game. The game starts with the configuration shown in Figure 21.2.

Make a program that allows two humans to play simple chess using only kings and rooks. The king must be able to move to all neighboring squares not occupied by a piece of the same color and cannot move onto a square, where it can be struck in the next turn. The rook must be able to move in horizontal and vertical lines until a piece of the same color or up to and including a piece of opposing color. Make a program that allows two humans to play simple chess.

Since we expect that the solution to the above problem is going to be a relatively long program, we have decided to split the code into a library and an application program. Before writing a library, it is often useful to start thinking about how the library should be used. Thus we start by sketching the application program, and in the process consider options for the main methods and properties to be used.

We also foresee future extensions to include more pieces, but also that these pieces will obey the same game mechanics that we design for the present problem. Thus, we will put the main part of the library in a file defining the module called `Chess` and the derived pieces in another file defining the module `Pieces`.

Every game needs a board, and we will define a class `Board`. A board is like an array, so it seems useful to be able to move pieces by index notation. Thus, the board must have a two-dimensional `Item` property. We also decide that each position will hold an option type such when a square is empty it holds `None` otherwise it holds piece `p` as `Some p`. Although chess notation would be neat, for ease of programming we will let index (0,0) correspond to position a1 in chess notation etc. The most common operation will probably be to move pieces around, so we will give the board a `move` method. We will most likely also like to print the board with pieces in their right locations. For simplicity we choose to override the `ToString` method in `Board`, and that this method also prints information about each individual piece such as where it is, where it can move to, and which pieces it can either protect or hit. The pieces that a piece can protect or hit we will call the piece' neighbor pieces.

A piece can be one of several types, so this gives a natural hierarchical structure, which is well suited for inheritance, Each piece must be given a color, which may conveniently be given as argument at instantiation. Thus, we have decided to make a base class called `chessPiece` with argument `Color`, and derived classes `king` and `rook`. The color may conveniently define as a discriminated union type of either `White` or `Black`. Each piece will also override the `ToString` method for ease of printing. The override will be used in conjunction with the board's override, so it should only give information about the piece' type and color. For compact printing, we will use a single letter for the type of piece, upper case if white, and lower case if black. We expect the pieces also to need to know something about the relation to board, so we will make a `position` property, which holds the coordinates of the piece, and we will make a `availableMoves` method that lists the possible moves, a piece can make. Thus, we produce the application in Listing 21.7, and an illustration of what the program should do is shown in Figure 21.3.

Listing 21.7 chessApp.fsx:
A chess application.

```
open Chess
open Pieces
/// Print various information about a piece
let printPiece (board : Board) (p : chessPiece) : unit =
  printfn "%A: %A %A" p p.position (p.availableMoves board)

// Create a game
let board = Chess.Board () // Create a board
// Pieces are kept in an array for easy testing
let pieces = [|
  king (White) :> chessPiece;
  rook (White) :> chessPiece;
  king (Black) :> chessPiece |]
// Place pieces on the board
board.[0,0] <- Some pieces.[0]
board.[1,1] <- Some pieces.[1]
board.[4,1] <- Some pieces.[2]
printfn "%A" board
Array.iter (printPiece board) pieces

// Make moves
board.move (1,1) (3,1) // Moves a piece from (1,1) to (3,1)
printfn "%A" board
Array.iter (printPiece board) pieces
```

At this point, we are fairly happy with the way the application is written. The double bookkeeping of pieces in an array and on the board seems a bit excessive, but for testing, it seems useful to be able to easily access all pieces both those in play and struck. Although the `position` property of a `chessPiece` could be replaced by a function searching for a specific piece on the board, we have a hunch that we will need to retrieve a piece' position often, and that this double will most likely save execution time later.

Continuing our outer to inner approach, as a second step, we consider the specific pieces: They will inherit a base piece and implement the details that are special for that piece. Each piece is signified by its color and its type, and each type has a specific motion pattern. Since we have already decided to use discriminated unions for the color, it seems natural to let the color be part of the constructor of

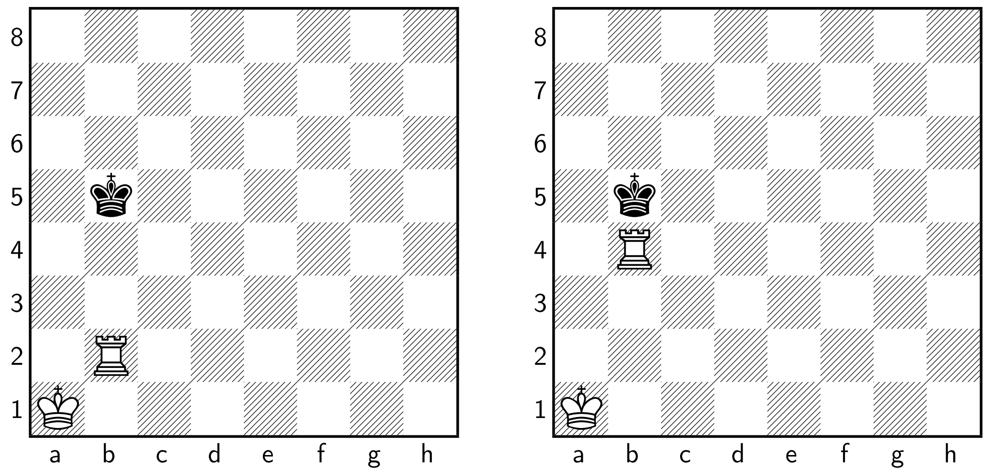

Figure 21.3: Starting at the left and moving white rook to b4.

the base class. As in the example application in Listing 21.7, pieces are upcasted to `chessPiece`, then the base class must know how to print the piece type. For this, we will define an abstract property, such that everything needed for overriding `ToString` is available to the base class, but also such that the name of the type of the piece is set in the derived class.

For a piece on the board, its available moves depend on its type and the other pieces. The application program will need to make a decision on whether to move the piece depending on which vacant squares, it can move to, and its relation to its neighbors, i.e., is the piece protecting one of its own color, or does it have the opportunity to hit an opponent. Thus given the board with all the pieces, it seems useful that `availableMoves` returns two lists: a list of vacant squares and a list of neighboring pieces of either color. Each piece has certain movement pattern, which we will specify regardless of the piece' position on the board and relation to other pieces. Thus, this will be an abstract member called `candiateRelativeMoves` implemented in the derived pieces. These candidate relative moves are then to be sifted for legal moves, and the process will be the same for all pieces, which thus can be implemented in the base class as the `availableMoves`.

Many pieces move in runs, e.g., the rook can move horizontally and vertically until there is another piece. Vacant squares behind the blocking piece are unavailable. For a rook, we thus must analyze four runs: northward, eastward, southward, and westward. For each run, we must consult the board to see, how many vacant fields there are in that direction, and which is the piece blocking if any. Thus, we decide that the board must have a function that can analyze a list of runs and that the result is concatenated into a single list of vacant squares and a single list of neighboring pieces if any. This function we call `getVacentNNeighbours`. And so we arrive at Listing 21.8.

Listing 21.8 pieces.fs:
An extension of chess base.

```
1 module Pieces
2 open Chess
3 /// A king is a chessPiece which moves 1 square in any direction
4 type king(col : Color) =
5   inherit chessPiece(col)
6   override this.nameOfType = "king"
7   // king has runs of 1 in 8 directions: (N, NE, E, SE, S, SW, W, NW)
8   override this.candiateRelativeMoves =
9       [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
10      [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
11 /// A rook is a chessPiece which moves horisontally and vertically
12 type rook(col : Color) =
13   inherit chessPiece(col)
14   // rook can move horisontally and vertically
15   // Make a list of relative coordinate lists. We consider the
16   // current position and try all combinations of relative moves
17   // (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...; (0,-7).
18   // Some will be out of board, but will be assumed removed as
19   // illegal moves.
20   // A list of functions for relative moves
21   let indToRel = [
22     fun elm -> (elm,0); // South by elm
23     fun elm -> (-elm,0); // North by elm
24     fun elm -> (0,elm); // West by elm
25     fun elm -> (0,-elm) // East by elm
26     ]
27   // For each function in indToRel, we calculate List.map f [1..7].
28   // swap converts (List.map fct indices) to (List.map indices fct).
29   let swap f a b = f b a
30   override this.candiateRelativeMoves =
31     List.map (swap List.map [1..7]) indToRel
32   override this.nameOfType = "rook"
```

The king has the simplest relative movement candidates being the hypothetical eight neighboring squares. For rooks, the relative movement candidates are somewhat more complicated. For rooks, we would like to use `List.map` to convert a list of single indices into double indices to calculate each run. And we have gathered all the elemental functions for this in `indToRel`. E.g., function at index 0, we may write `List.map indToRel.[0] indices`. However, we would also like to use `List.map` to perform this operation for all elemental functions in `indToRel`. Direct joining such two applications of `List.map` does not work, since `List.map` takes a function and a list as its arguments, and for the second application, these two arguments should switch order. I.e., the first time it is `indices` that takes the role of the list, while the second it is `indToRel` that takes the role of the list. A standard solution in functional programming is to use currying and the ***swap*** function as illustrated in line 31. · `swap`
The function is equivalent to the anonymous function `fun elm -> swap List.map indices elm`, and since `swap` swaps the arguments of a function, this reduces to `fun elm -> List.map elm indices`, which is exactly what is needed.

The final step will be to design the `Board` and `chessPiece` classes. The Chess module implements discriminated unions for color and an integer tuple for a position. These are shown in Listing 21.9.

Listing 21.9 chess.fs:
A chess base: Module header and discriminated union types.

```
module Chess
type Color = White | Black
type Position = int * int
```

The `chessPiece` will need to know what a board is, so we must define it as a mutually recursive class with `Board`. Further, since all pieces must supply an implementation of `availableMoves`, we set it to be abstract by the abstract class attribute and with an abstract member. The board will need to be able to ask for a string describing each piece and to keep the board on the screen we include an abbreviated description of the piece's properties color and piece type. The result is shown in Listing 21.10.

Listing 21.10 chess.fs:
A chess base. Abstract type chessPiece.

```
/// An abstract chess piece
[<AbstractClass>]
type chessPiece(color : Color) =
  let mutable _position : Position option = None
  abstract member nameOfType : string // "king", "rook", ...
  member this.color = color // White, Black
  member this.position // E.g., (0,0), (3,4), etc.
    with get() = _position
    and set(pos) = _position <- pos
  override this.ToString () = // E.g. "K" for white king
    match color with
      White -> (string this.nameOfType.[0]).ToUpper ()
      | Black -> (string this.nameOfType.[0]).ToLower ()
  /// A list of runs, which is a list of relative movements, e.g.,
  /// [[(1,0); (2,0);...]; [(-1,0); (-2,0)]...]. Runs must be
  /// ordered such that the first in a list is closest to the piece
  /// at hand.
  abstract member candiateRelativeMoves : Position list list
  /// Available moves and neighbours ([(1,0); (2,0);...], [p1; p2])
  member this.availableMoves (board : Board) : (Position list *
   chessPiece list) =
    board.getVacantNNeighbours this
```

Our `Board` class is by far the largest and will be discussed by Listing 21.11–21.13. The constructor is shown in Listing 21.11.

Listing 21.11 chess.fs:
A chess base: the constructor

```
/// A board
and Board () =
  let _array = Collections.Array2D.create<chessPiece option> 8 8 None
  /// Wrap a position as option type
  let validPositionWrap (pos : Position) : Position option =
    let (rank, file) = pos // square coordinate
    if rank < 0 || rank > 7 || file < 0 || file > 7
    then None
    else Some (rank, file)
  /// Convert relative coordinates to absolute and remove out
  /// of board coordinates.
  let relativeToAbsolute (pos : Position) (lst : Position list) :
   Position list =
    let addPair (a : int, b : int) (c : int, d : int) : Position =
      (a+c,b+d)
    // Add origin and delta positions
    List.map (addPair pos) lst
    // Choose absolute positions that are on the board
    |> List.choose validPositionWrap
```

For memory efficiency, the board has been implemented using a `Array2D`, since pieces will move around often. For later use in the members shown in Listing 21.13 we define tow functions that converts relative coordinates into absolute coordinates on the board, and removes those that fall outside the board. These are called `validPositionWrap` and `relativeToAbsolute`.

For ease of use in an application, `Board` implements `Item`, such that the board can be read and writing to using array notation. And `ToString` is overridden, such that an application may print the board anytime using a `printf` function. This is shown in Listing 21.12.

Listing 21.12 chess.fs:
A chess base: Board header, constructor, and non-static members.

```
43    /// Board is indexed using .[,] notation
44    member this.Item
45      with get(a : int, b : int) = _array.[a, b]
46      and set(a : int, b : int) (p : chessPiece option) =
47        if p.IsSome then p.Value.position <- Some (a,b)
48        _array.[a, b] <- p
49    /// Produce string of board for, e.g., the printfn function.
50    override this.ToString() =
51      let rec boardStr (i : int) (j : int) : string =
52        match (i,j) with
53          (8,0) -> ""
54          | _ ->
55            let stripOption (p : chessPiece option) : string =
56              match p with
57                None -> ""
58                | Some p -> p.ToString()
59            // print top to bottom row
60            let pieceStr = stripOption _array.[7-i,j]
61            //let pieceStr = sprintf "(%d, %d)" i j
62            let lineSep = " " + String.replicate (8*4-1) "-"
63            match (i,j) with
64            (0,0) ->
65              let str = sprintf "%s\n| %1s " lineSep pieceStr
66              str + boardStr 0 1
67            | (i,7) ->
68              let str = sprintf "| %1s |\n%s\n" pieceStr lineSep
69              str + boardStr (i+1) 0
70            | (i,j) ->
71              let str = sprintf "| %1s " pieceStr
72              str + boardStr i (j+1)
73      boardStr 0 0
```

Note that for efficiency, location is also stored in each piece, so `set` also needs to update the particular piece' position as done in line 47. Note also that the board is printed with the first coordinate of the board being rows and second columns and such that element (0,0) is at the bottom right complying with standard chess notation.

The main computations are done in the static methods of the board as shown in Listing 21.13.

Listing 21.13 chess.fs:
A chess base: Board static members.

```
  /// Move piece by specifying source and target coordinates
  member this.move (source : Position) (target : Position) : unit =
    this.[fst target, snd target] <- this.[fst source, snd source]
    this.[fst source, snd source] <- None
  /// Find the tuple of empty squares and first neighbour if any.
  member this.getVacantNOccupied (run : Position list) : (Position list
   * (chessPiece option)) =
    try
      // Find index of first non-vacant square of a run
      let idx = List.findIndex (fun (i, j) -> this.[i,j].IsSome) run
      let (i,j) = run.[idx]
      let piece = this.[i, j] // The first non-vacant neighbour
      if idx = 0
      then ([], piece)
      else (run.[..(idx-1)], piece)
    with
      _ -> (run, None) // outside the board
  /// find the list of all empty squares and list of neighbours
  member this.getVacantNNeighbours (piece : chessPiece) : (Position list
   * chessPiece list)  =
    match piece.position with
      None ->
        ([],[])
      | Some p ->
        let convertNWrap =
          (relativeToAbsolute p) >> this.getVacantNOccupied
        let vacantPieceLists = List.map convertNWrap
   piece.candiateRelativeMoves
        // Extract and merge lists of vacant squares
        let vacant = List.collect fst vacantPieceLists
        // Extract and merge lists of first obstruction pieces and
   filter out own pieces
        let opponent =
          vacantPieceLists
          |> List.choose snd
        (vacant, opponent)
```

A chess piece must implement `candiateRelativeMoves`, and we decided in Listing 21.10 that moves should be specified relative to the piece' position. Since the piece does not know, which other pieces are on the board, it can only specify all potential positions. For convenience, we will allow pieces to also specify positions outside the board, such that, e.g., the rook can specify the 7 nearest neighboring squares up, down, left, and right regardless that some may be outside the board. Thus `getVacantNNeighbours` must first convert the relative positions to absolute and clip any outside the board. This is done by `relativeToAbsolute`. Then for each run, the first occupied square must be identified. Since `availableMoves` must return two lists, vacant squares, and immediate neighbors, this structure is imposed on the output of `convertNWrap` as well. This is computed in `getVacantNOccupied` by use of the built-in `List.findIndex` function. This function returns the index of the first element in a list for which the supplied function is true and otherwise throws an exception. Exceptions are always somewhat inelegant, but in this case, it is harmless, since the exception signifies a valid situation where no pieces exist on the run. After having analyzed all runs independently, then all the vacant lists are merged and all the neighboring pieces are merge and both are returned to the caller.

Compiling the library files with the application and executing gives the result shown in Listing 21.14.

Listing 21.14: Running the program. Compare with Figure 21.3.

```
$ fsharpc --nologo chess.fs pieces.fs chessApp.fsx && mono chessApp.exe
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   | k |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   | R |   |   |   |   |   |   |
 --------------------------------
| K |   |   |   |   |   |   |   |
 --------------------------------

K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3); (1, 4); (1, 5);
   (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0);
   (3, 0)], [])
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   | k |   |   |   |   |   |   |
 --------------------------------
|   | R |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
|   |   |   |   |   |   |   |   |
 --------------------------------
| K |   |   |   |   |   |   |   |
 --------------------------------

K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
R: Some (3, 1) ([(2, 1); (1, 1); (0, 1); (3, 2); (3, 3); (3, 4); (3, 5);
   (3, 6); (3, 7); (3, 0)],
 [k])
k: Some (4, 1) ([(3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4, 0); (3,
   0)], [R])
```

We see that the program has correctly determined that initially, the white king has the white rook as its neighbor and due to its location in the corner only has two free positions to move to. The white rook has many and the black king as its neighbor. The black king is free to move to all its eight neighboring fields. After moving the white rook to (3,1) or b4 in regular chess notation, then the white king has no neighbors, the white rook and the black king are now neighbors with an appropriate restriction on their respective vacant squares. These simple use-tests are in no way a thorough test of the quality of the code, but they give us a good indication that our library offers a tolerable interface for the application, and that at least major parts of the code function as expected. Thus, we conclude this intermezzo.

22 The object-oriented programming paradigm

Object-oriented programming is a paradigm for encapsulating data and methods into cohesive units. Key features of object-oriented programming are: · Object-oriented programming

Encapsulation
: Data and methods are collected into a cohesive unit, and an application program need only focus on how to use the object, not on its implementation details.

Inheritance
: Objects are organized in a hierarchy of gradually increased specialty. This promotes a design of code that is of general use, and code reuse by specializing the general to the specific.

Polymorphism
: By overriding methods from a base class, derived classes define new data types while their methods still produce results compatible with the base class definitions.

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technologic constraints, while the design should include technological constraints such as defined by the targeted language and hardware. · what · how

The primary steps for *object oriented analysis and design* are: · object oriented analysis and design

1. identify objects,
2. describe object behavior,
3. describe object interactions,
4. describe some details of the object's inner workings,
5. write a precise description for classes, properties and methods using, e.g., F#'s XML documentation standard,
6. write mockup code,
7. write unit-tests and test the basic framework using the mockup code,
8. replace the mockup with real code while testing to keep track of your progress. Extend the unit-test as needed,
9. evaluate code in relation to the desired goal

10. complete your documentation both in-code and outside.

Step 1–4 are the analysis phase and gradually stops in step 4, while the design phase gradually starts step 4 and gradually stops when actual code is written in step 7. Notice that the last steps are identical with imperative programming Chapter 12. Programming is never a linear experience, and you will often need to go back to previous steps to update or change decisions. You should not refrain from improving your program design and implementation, but you should always be mindful of the goal. Often the perfect solution is much less than needed to complete a task, often less will suffice.

An object-oriented analysis can be a daunting process. A good starting point is a *use case*, *problem statement*, or a *user story*, which in human language describes of a number of possibly hypothetical interactions between a user and a system performs in order to solve some task. Two useful methodologies for performing an object-oriented analysis is the method of nouns-and-verbs and the unified modeling language described in the following sections.

· use case
· problem statement
· user story

22.1 Identification of objects, behaviors, and interactions by nouns-and-verbs

A key point in object-oriented programming is that objects should to a large extent be independent and reusable. As an example, the type `int` models the concept of integer numbers. It can hold integer values from -2,147,483,648 to 2,147,483,647, and a number of standard operations and functions are defined for it. We may use integers in many different programs, and it is certain that the original designers did not foresee our use, but strived to make a general type applicable for many uses. Such a design is a useful goal, when designing objects, that is, our objects should model the general concepts and be applicable in future uses.

Analyzing a specific use-case, good candidates for objects are persons, places, things, events, concept etc., which are almost always characterized by being *nouns* in the text. Interactions between objects are actions that bind objects together, and actions are often associated with *verbs*. When choosing methods, it is important to maintain an object centered perspective, i.e., for a general-purpose object, we should limit the need for including information about other objects. E.g., a value of type `int` need not know anything about the program, in which it is being used.

· nouns
· verbs

Said briefly, the *nouns-and-verbs method* is:

· nouns-and-verbs method

> Nouns are object candidates, verbs are candidate methods, that describe interactions between objects.

22.2 Class diagrams in the Unified Modelling Language

Having found an initial list of candidate objects and interactions, it is often useful to make a drawing of these relations and with an increased focus on the object's inner workings. A *class diagram* is a schematic drawing of the program highlighting its object-oriented structure, and we will use the *Unified Modelling Language 2* (*UML*) [5] standard. The standard is very broad, and here we will discuss structure diagrams for use of describing objects.

· class diagram
· Unified Modelling Language 2
· UML

A class is drawn as a shown in Figure 22.1. In UML, classes are represented as boxes with their class name. Depending on the desired level of details zero or more properties and methods are described. These describe the basic interface to the class and objects of its type. Abstract members that require an implementation, are shown in cursive. Here we have used F# syntax, to conform with this book theme,

ClassName
value-identifier : type value-identifier : type = default value
function-identifier (arg : type) (arg : type) ... : type *function-identifier (arg : type) (arg : type) ... : type*

Figure 22.1: A UML diagram for a class, consists of it's name, zero or more attributes, and zero or more methods.

<<interface>> **InterfaceName**
value-identifier : type value-identifier : type = default value
function-identifier (arg : type) (arg : type) ... : type

Figure 22.2: An interface is a class that requires an implementation.

but typically C# syntax is used. Interfaces are a special type of class that require an implementation. To highlight this, UML uses the notation shown in Figure 22.2.[1]

Relations between classes and objects are indicated by lines and arrows. The most common ones are summarized in Figure 22.3. Their meaning will be described in detail in the following.

Classes may inherit other classes, where the parent is called the base class and the children its derived classes. Such a relation is often called an *is-a* relation, since the derived class *is a* kind of base class. An illustration of inheritance in UML is shown in Figure 22.4. Here two classes inherit the base class. The syntax is analogous for interfaces, except a stippled line is used to indicate that a derived class implements an interface, as shown in Figure 22.5.

· inheritance
· is-a
· interface

Other relations between classes are association, aggregation, and composition:

· association

Association
: In associated relations, one class knows about the other, e.g., uses it as arguments of a function or similar.

· aggregation

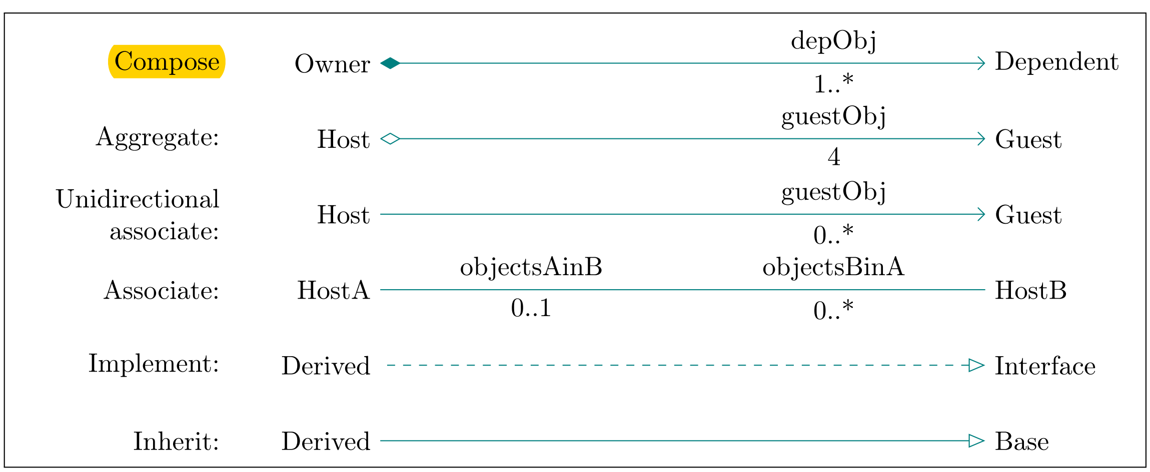

Figure 22.3: Arrows used in class diagrams to show relations between objects.

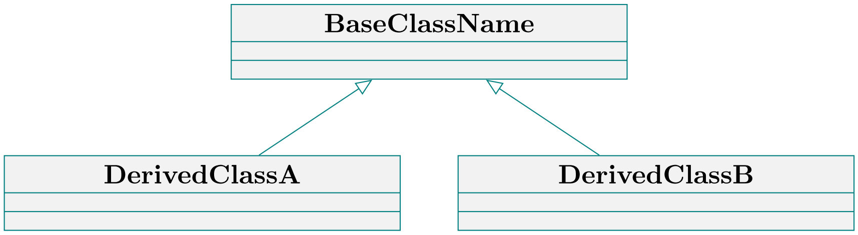

Figure 22.4: Inheritance is shown by a closed arrow head pointing to the base.

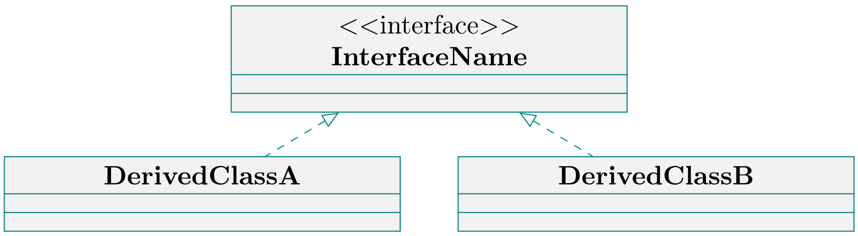

Figure 22.5: Implementations of interfaces is shown with stippled line and closed arrow head pointing to the base.

Aggregation

Aggregated relationships is a specialization of associations. In aggregated relations, the host object has a local copy of a guest object, but the host did not create the guest. E.g., the guest object is given as an argument to a function of the host, and the host makes a local alias for later use. When the host is deleted, then the guest is not.

· composition

· has-a

· package

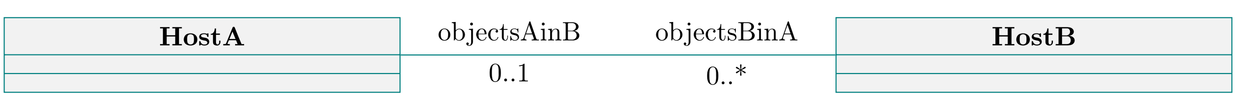

Figure 22.6: Bidirectional association is shown as a line with optional annotation.

n	exactly n instances
*	zero or more instances
n..m	n to m instances
n..*	from n to infinite instances

Table 22.1: Notation for association multiplicities is similar to F#'s slicing notation.

Figure 22.7: Unidirectional association shows a one-side *has-a* relations.

Composition

A composed relationship is a specialization of aggregations. In composed relations, the host creates the guest, and when the host is deleted so is the guest.

Aggregational and compositional relations are often called *has-a* relations since host objects have one or more guests either as aliases or as owner.

Bidirectional association means that classes know about each other. The UML notation is shown in Figure 22.6. Association may be annotated by an identifier and a multiplicity. In the figure, HostA has 0 or more variables of type HostB named objectsBinA, while HostB has 0 or 1 variables of HostA named objectsAinB. The multiplicity notation is very similar to F#'s slicing notation. Typical values are shown in Table 22.1. If the association is unidirectional, then an arrow is added for emphasis as shown in Figure 22.7. In this example, Host knows about Guest and has one instance of it, and Guest is oblivious about Host.

Aggregation is illustrated using a diamond tail and an open arrow as shown in Figure 22.8. Here the Host class has stored aliases to 4 different Guest objects. A stronger relation is composition. This is shown like aggregation but with a filled diamond as illustrated in Figure 22.9. In this example, Owner has created 1 or more objects of type Dependent, and when Owner is deleted so are these objects.

Finally, for visual flair, modules and namespaces are often visualized as a *package* as shown in Figure 22.10. A package is like a module in F#.

22.3 Programming intermezzo: designing a racing game

An example is the following *problem statement*: · problem statement

[1] Jon: **Add programming examples for each of these UML structures**

Figure 22.8: Aggregation relations are a subset of associations, where local aliases are stored for later use.

Figure 22.9: Composition relations are a subset of aggregation, where the host controls the lifetime of the guest objects.

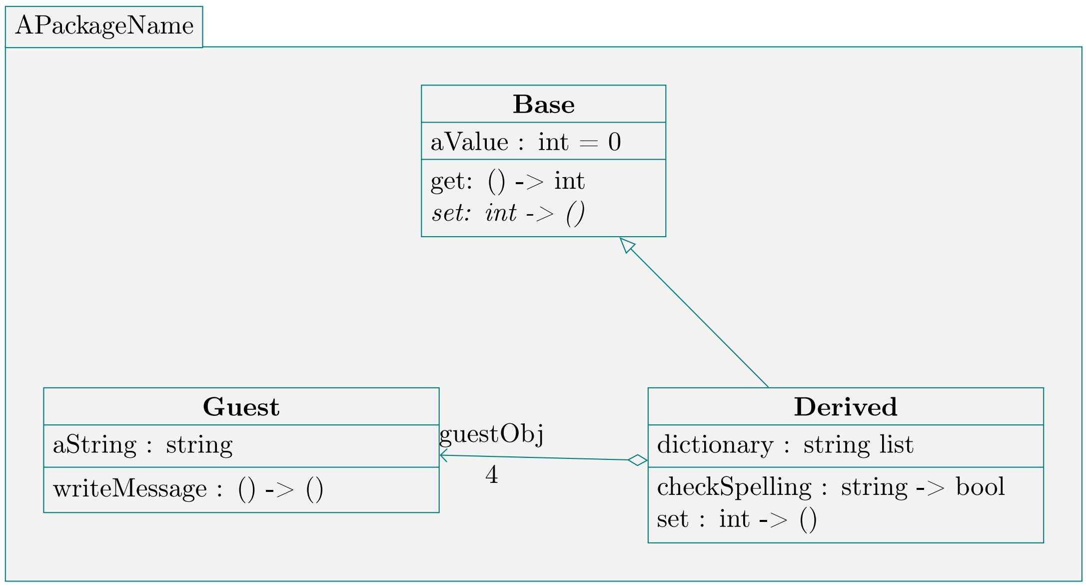

Figure 22.10: Packages are a visualizations of modules and namespaces.

Problem 22.1

Write a racing game, where each player controls his or her vehicle on a track. Each vehicle must have individual features such as top acceleration, speed, and handling. The player must be able to turn the vehicle left and right, and to accelerate up and down. At the beginning of the game, each vehicle is placed behind the starting line. Once the start signal is given, then the players may start to operate their vehicles. The player who first completes 3 rounds wins.

To seek a solution, we will use the *nouns-and-verbs method*. Below the problem statement is repeated with nouns and verbs highlighted.

> Write a racing game, where each player controls his or her vehicle on a track. Each vehicle must have individual features such as top acceleration, speed, and handling. The player must be able to turn the vehicle left and right, and to accelerate up and down. At the beginning of the game, each vehicle is placed behind the starting line. Once the start signal is given, then the players may start to operate their vehicles. The player who first completes 3 rounds wins.

The above nouns and verbs are candidates for objects, their behaviour and interaction. A deeper analysis is:

Identification of objects by nouns (Step 1):

Identified unique nouns are: racing game (game), player, vehicle, track, feature, top acceleration, speed, handling, beginning, starting line, start signal, rounds. From this list we seek cohesive units that are independent and reusable. The nouns

game, player, vehicle, and track

seems to fulfill these requirements, while all the rests seems to be features of the former and thus not independent concepts. E.g., top acceleration is a feature of a vehicle, and starting line is a feature of a track.

Object behavior and interactions by verbs (Step 2 and 3):
To continue our object oriented analysis, we will consider the object candidate identified above, and verbalize how they would act as models of general concepts useful in our game.

player The player is associated with the following verbs:

- A player controls/operates a vehicle.
- A player turns and accelerates a vehicle.
- A player completes a rounds.
- A player wins.

Verbalizing a player, we say that a player in general must be able to control the vehicle. In order to do this, the player must receive information about the track and all vehicles or at least some information about the nearby vehicles and track. And the player must receive information about the state of the game, i.e., when does the race start and stop.

vehicle A vehicle is controlled by a player and further associated with the following verbs:

- A vehicle has features top acceleration, speed, and handling.
- A vehicle is placed and on the track.

To further describe a vehicle we say that a vehicle is a model of a physical object, which moves around on the track under the influence of a player. A vehicle must have a number of attributes such as top acceleration, speed, and handling, and must be able to receive information about when to turn and accelerate. A vehicle must be able to determine its location in particular if it is on or off track and, and it must be able to determine if it has crashed into an obstacle such as another vehicle.

track A track is the place where vehicles operate and are further associated with the following verbs:

- A track has a starting line.
- A track has rounds.

Thus, a track is a fixed entity on which the vehicles race. It has a size and a shape, a starting and a finishing line, which may be the same, and vehicles may be placed on the track and can move on and possibly off the track.

game Finally, a game is associated with the following verbis:

- A game has a beginning and a start signal.
- A game can be won.

A game is the total sum of all the players, the vehicles, the tracks, and their interactions. A game controls the flow of a particular game including inviting players to race, sending the start signal, and monitoring when a game is finished and who won.

From the above we see that the object candidates feature seems to be a natural part of the description of the vehicle's attributes, and similarly, starting line may be an intricate part of a track. Also, many of the *verbs* used in the problem statement and in our extended verbalization of the general concepts indicate methods that are used to interact with the object. The object centered perspective tells us that for a general-purpose vehicle object, we need not include information about the player, analogous to a value of type `int` need not know anything the program, in which it is being used. In contrast, the candidate game is not as easily dismissed and could be used as a class which contains all the above, i.e., · verbs

With this description, we see that 'start signal' can be included as a natural part of the game object. Being confident that a good working hypothesis of the essential objects for the solution, we continue our investigating into further details about the objects and their interactions.

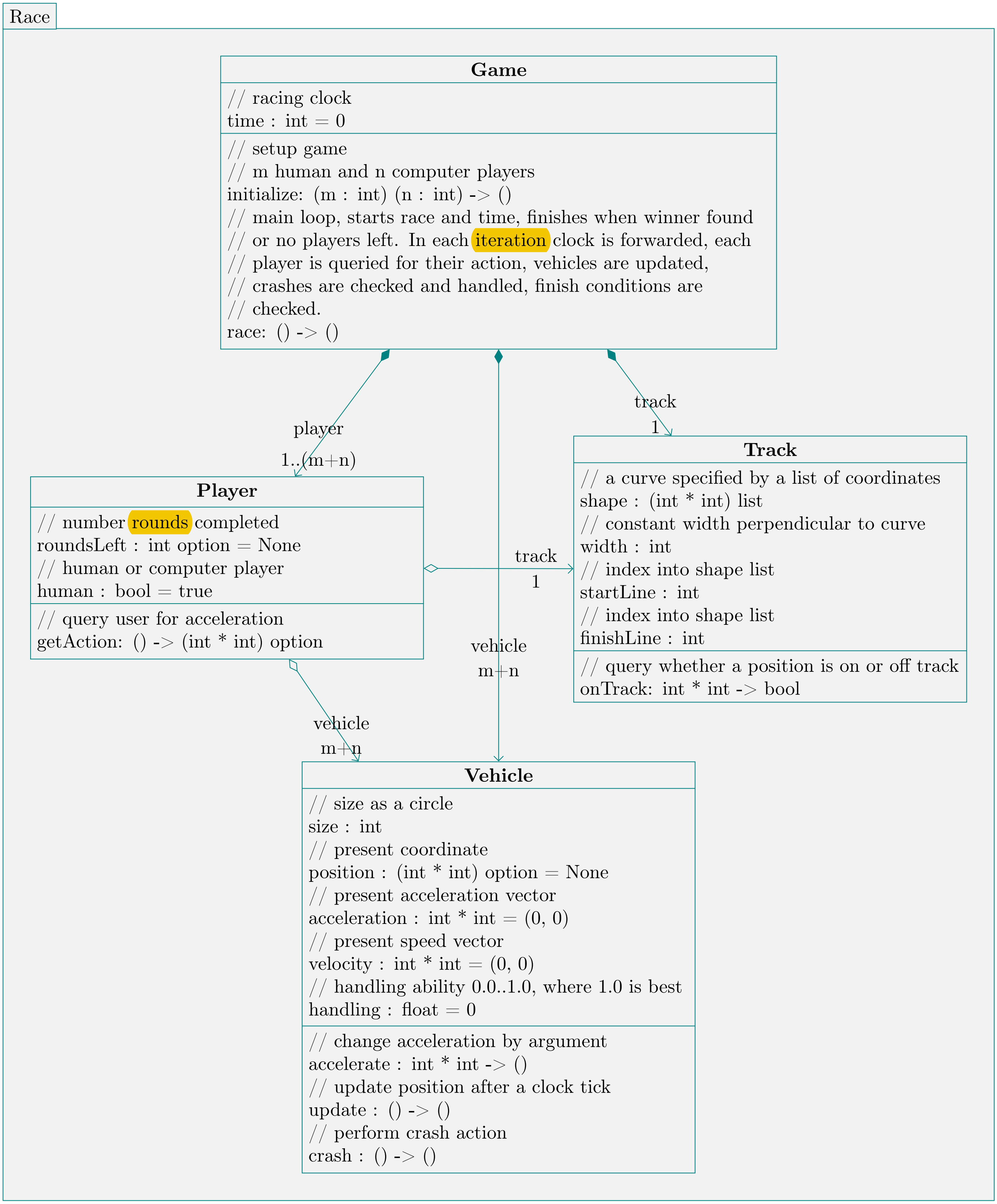

Figure 22.11: A class diagram for a racing game.

Analysis details (Step 4):

A class diagram of our design for the proposed classes and their relations is shown in Figure 22.11.

In the present description, there will be a single Game object, that initializes the other objects, and execute a loop updating the clock, query the players for actions, and informs the vehicles that they should move and under what circumstances. The track has been chosen to be dumb and does not participate much in the action. Player's method getAction will be an input from a user by keyboard, joystick or similar, but the complexity of the code for a computer player will be large since it needs to take a sensible decision based on the track and the location of the other vehicles. What at present is less clear, is whether it is the responsibility of Game or Vehicle to detect an off track or a crash event. If a vehicle is to do this, then each vehicle must have aggregated association to all other vehicles and obstacles. So, on the one hand, it would seem an elegant delegation of responsibilities that a vehicle knows, whether it has crashed into an obstacle or not, but on the other hand, it seems wasteful of memory resources to have duplicated references of all obstacles in every vehicle. The final choice is thus one of elegance versus resource management, and in the above, we have favored resource management. Thus, the main loop in Game must check all vehicles for a crash event, after the vehicle's positions have been updated, and in case inform the relevant vehicles.

Having created a design for a racing game, we are now ready to write start coding (Step 6–). It is not uncommon, that transforming our design into code will reveal new structures and problems, that possibly require our design to be updated. Nevertheless, a good design phase is almost always a sure course to avoid many problems once coding, since the design phase allows the programmer to think about the problem from a helicopter perspective before tackling details of specific sub-problems.