Chapter 1

Derived Classes

Abstract Sometimes, classes can be grouped into a hierarchy such as foxes and fish both being animals. In this chapter, you will learn how object-oriented programming uses inheritance to benefit from such a hierarchy. For example, super-classes such as animals may contain elements, which are shared by the sub-classes foxes and fish, e.g., they both have a weight, and the sub-classes need only implement the differences, e.g., foxes live on land and fish in water. In this chapter you will learn how to

- construct a class hierarchy
- utilize the common super-class to have simple polymorphy, e.g., make lists that contains both foxes and fish
- create abstract classes similar to interface files, for a user (programmer) to later produce an implementation
- make and use class-interfaces to make your objects useful in generic functions such as sorting

1.1 Inheritance

Sometimes it is useful to derive new classes from old ones 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, and 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 1.1 and is called an *is-a relation*. Is-a relations can be implemented using

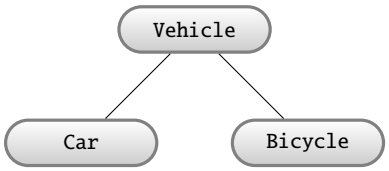

Fig. 1.1 Both a car and a bicycle is a (type of) vehicle.

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 possibly add new members. Another advantage is that objects from derived classes can be made to look like as if they were objects of the base class while still containing all their data. Such mascarading is useful when, for example, listing cars and bicycles in the same list.

In F#, inheritance is indicated using the `inherit` keyword in the class definition. An extensions of the syntax in Listing 1.1 is:

Listing 1.1: A class definition with inheritance.

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

New syntactical elements are: the `inherit` keyword, which indicates that this is a derived class and where `<baseClassIdent>` is the name of the base class. Further, members may be regular members using the `member` keyword as discussed in the previous chapter, and members can also be other types, as indicated by the keywords: `abstract member`, `default`, and `override`.

An example of defining base and derived classes for vehicles is shown In Listing 1.2. In the example, a simple base class `vehicle` is defined to include `wheels` as its single member. The derived classes inherit all the members of the base class, but do not have access to any non-members of the base constructor. I.e., `car` and `bicycle` automatically have the `wheels` property. Both derived classes additional members `maxPassengers` and `mustUseHelmet`, respectively.

Listing 1.2 vehicle.fsx:
New classes can be derived from old ones.

```
/// All vehicles have wheels
type vehicle (nWheels : int ) =
  member this.wheels = nWheels

/// A car is a vehicle with 4 wheels
type car (nPassengers : int) =
  inherit vehicle (4)
  member this.maxPassengers = nPassengers

/// A bicycle is a vehicle with 2 wheels
type bicycle () =
  inherit vehicle (2)
  member this.mustUseHelmet = true

let aVehicle = vehicle (1)
let aCar = car (4)
let aBike = bicycle ()
printfn "aVehicle has %d wheel(s)" aVehicle.wheels
printfn "aCar has %d wheel(s) with room for %d passenger(s)"
   aCar.wheels aCar.maxPassengers
printfn "aBike has %d wheel(s). Is helmet required? %b"
   aBike.wheels aBike.mustUseHelmet
```

```
$ dotnet fsi vehicle.fsx
aVehicle has 1 wheel(s)
aCar has 4 wheel(s) with room for 4 passenger(s)
aBike has 2 wheel(s). Is helmet required? true
```

Derived classes can replace base class members by defining new members *overshadow* the base members. The base members are still available through the `base`-keyword. Consider the example in the Listing 1.3. In this case, we have defined three greetings: `greeting`, `hello`, and `howdy`. The two later inherit `member this.str = "hi"` from `greeting`, but since they both also define a member property `str`, these overshadow the one from `greeting`. In `hello` and `howdy` the base value of `str` is available as `base.str`.

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 1.4. 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*

Listing 1.3 memberOvershadowingVar.fsx:
Inherited members can be overshadowed, but we can still access the base member. Compare with Listing 1.7.

```
/// hi is a greeting
type greeting () =
  member this.str = "hi"
/// hello is a greeting
type hello () =
  inherit greeting ()
  member this.str = "hello"
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  member this.str = base.str + " there"

let a = greeting ()
let b = hello ()
let c = howdy ()
printfn "%s, %s, %s" a.str b.str c.str
```

```
$ dotnet fsi memberOvershadowingVar.fsx
hi, hello, hi there
```

★ operator *:?>*, but the validity of the operation is checked at runtime. Thus, **avoid downcasting when possible.**

Listing 1.4 upCasting.fsx:
Objects can be upcasted resulting in an object to appear to be of the base type. Implementations from the derived class are ignored.

```
/// 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
```

```
$ dotnet fsi upCasting.fsx
hello howdy hi hello
```

1.2 Interfacing with the `printf` Family

In previous examples, we accessed the property in order to print the contents of 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 1.5. All classes

Listing 1.5 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 low-level
   information
```

```
$ dotnet fsi classPrintf.fsx
FSI_0001+vectorDefaultToString
```

implicitely inherit from a class with the peculiar name, *`System.Object`*, and as a consequence, all classes have a number of already defined members. 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 with the `%A` or `%O` placeholders in the formatting string, `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 1.6. Note, despite that `ToString()` returns a string, the `%s` placerholder only accepts values of the basic `string` type. We see

Listing 1.6 classToString.fsx:
Overriding `ToString()` function for better interaction with members of the `printf` family of procedures. Compare with Listing 1.5.

```
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
```

```
$ dotnet fsi classToString.fsx
(1.0, 2.0)
```

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.**

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)`", and "`[%A, %A]`". Considering each carefully, it seems that arguments can be made against all them. A common choice is to let the formatting be controlled by static members that can be changed by the application program through accessors.

1.3 Abstract Classes

In the previous sections, we have discussed inheritance as a method to modify and extend any class. I.e., the definition of the base classes were independent of 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 of derived classes and which impose design constraints of derived classes. Two such dependencies in F# are abstract classes and interfaces.

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. 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 1.7. 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

Listing 1.7 abstractClass.fsx:
In contrast to regular objects, upcasted derived objects use the derived implementation of abstract methods. Compare with Listing 1.3.

```
/// 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
```

```
$ dotnet fsi abstractClass.fsx
hello
howdy
```

of implementations being available in the abstract class, the abstract class still cannot be used to instantiate objects. The example in Listing 1.8 shows an extension of Listing 1.7 with a default implementation. In the example, the program in Listing 1.7 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 to provide an override member.

Note that even if all abstract members in an abstract class have 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 is an abstract class defining among other members, the `ToString` method with default implementation.

Listing 1.8 abstractDefaultClass.fsx:
Default implementations in abstract classes make implementations in derived classes optional. Compare with Listing 1.7.

```
/// 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
```

```
$ dotnet fsi abstractDefaultClass.fsx
hello
howdy
```

1.4 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's name. Consider the example in Listing 1.9. 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 1.9, the `printfn` function only needs to know the member's type, not its meaning. As a consequence,

Listing 1.9 classInterface.fsx:
Interfaces specify 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
```

```
$ dotnet fsi classInterface.fsx
value = 2, fct(80.0) = 11200
value = 6, fct(80.0) = 24.6914
```

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 implement 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.

1.5 Programming Intermezzo: Chess

To demonstrate the use of hierarchies, consider the following problem.

Problem 1.1

he 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 1.2.

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.

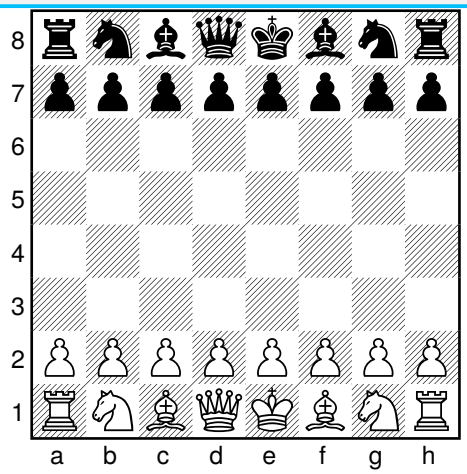

Fig. 1.2 Starting position for the game of chess.

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 that when a square is empty it holds `None`, and 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's 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 be 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's 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 their 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 1.10, and an illustration of what the program should do is shown in Figure 1.3. 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's position often, and that this double bookkeeping 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 the base class. As in the example application in Listing 1.10, pieces are upcasted to `chessPiece`, thus, the base class must know how to print the piece type. For this, we will define an

Listing 1.10 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
```

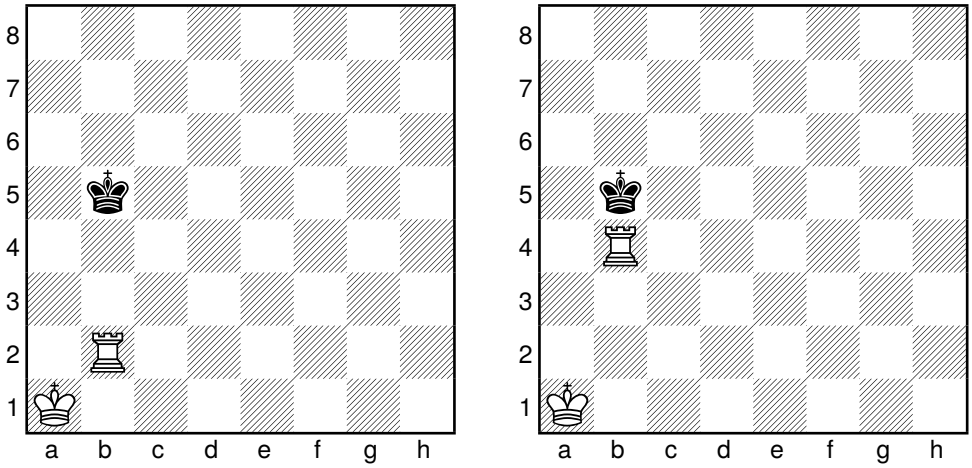

Fig. 1.3 Starting at the left and moving white rook to b4.

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's piece. 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 a 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 `candidateRelativeMoves` 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. Thus, sifting 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 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 1.11.

Listing 1.11 pieces.fs:
An extension of chess base.

```
module Pieces
open Chess
/// A king moves 1 square in any direction
type king(col : Color) =
  inherit chessPiece(col)
  // A king has runs of length 1 in 8 directions:
  //   (N, NE, E, SE, S, SW, W, NW)
  override this.candidateRelativeMoves =
      [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
      [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
  override this.nameOfType = "king"
/// A rook moves horizontally and vertically
type rook(col : Color) =
  inherit chessPiece(col)
  // A rook can move horizontally and vertically
  // Make a list of relative coordinate lists. We consider the
  // current position and try all combinations of relative
  // moves (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...;
   (0,-7).
  // Some will be out of board, but will be assumed removed as
  // illegal moves.
  // A list of functions for relative moves
  let indToRel = [
    fun elm -> (elm,0); // South by elm
    fun elm -> (-elm,0); // North by elm
    fun elm -> (0,elm); // West by elm
    fun elm -> (0,-elm) // East by elm
    ]
  // For each function f in indToRel, we calculate
  //   List.map f [1..7].
```

The king has the simplest relative movement candidates, being the hypothetical eight neighboring squares. Rooks have a considerably longer list of candidates of relative moves, since it potentially can move to all 7 squares northward, eastward, southward, and westward. This could be hardcoded as 4 potential runs, `[[(1,0); (2,0); ... (7,0)]; [(-1,0); (-2,0); ... (0,-7)]]`. Each run will be based on the list `[1..7]`, which gives us the idea to use `List.map` to convert a list of single indices `[1..7]` into lists of runs as required by `candidateRelativeMoves`. Each run may be generated from `[1..7]` as

```
South: List.map (fun elm -> ( elm, 0)) [1..7]
North: List.map (fun elm -> (-elm, 0)) [1..7]
West:  List.map (fun elm -> (0,  elm)) [1..7]
East:  List.map (fun elm -> (0, -elm)) [1..7]
```

and which can be combined as a list of 4 lists of runs. Further, since functions are values, we can combine the 4 different anonymous functions into a list of functions and use a for-loop to iterate over the list of functions. This is shown in Listing 1.12. However, this solution is imperative in nature and does not use the elegance of the

Listing 1.12 imperativeRuns.fsx:
Calculating the runs of a rook using imperative programming.

```
30 let mutable listOfRuns : ((int * int) list) list = []
31 for f in indToRel do
32   let run = List.map f [1..7]
33   listOfRuns <- run :: listOfRuns
```

functional programming paradigm. A direct translation into functional programming is given in Listing 1.13. The functional version is slightly longer, but avoids the

Listing 1.13 functionalRuns.fsx:
Calculating the runs of a rook using functional programming.

```
30 let rec makeRuns lst =
31   match lst with
32     | [] -> []
33     | f :: rest ->
34       let run = List.map f [1..7]
35       run :: makeRuns rest
36 makeRuns indToRel
```

mutable variable.

Generating lists of runs from the two lists `[1..7]` and `indToRel` can also be performed with two `List.maps`, as shown in Listing 1.14.

The anonumous function,

```
fun e -> List.map e [1..7],
```

Listing 1.14 ListMapRuns.fsx:
Calculating the runs of a rook using double `List.maps`.

```
30 List.map (fun e -> List.map e [1..7]) indToRel
```

is used to wrap the inner `List.map` functional. An alternative, sometimes seen is to use currying with argument swapping: Consider the function, `let altMap lst e = List.map e lst`, which reverses the arguments of `List.map`. With this, the anonymous function can be written as `fun e -> altMap [1..7] e` or simply replaced by currying as `altMap [1..7]`. Reversing orders of arguments like this in combination with currrying is what the *`swap`* function is for,

```
let swap f a b = f b a.
```

With swap we can write `let altMap = swap List.map`. Thus,

```
swap List.map [1..7]
```

is the same function as `fun e -> List.map e [1..7]`, and in which case we could rewrite the solution in Listing 1.14 as

```
List.map (swap List.map [1..7]) indToRel
```

if we wanted a very compact, but possible less readable solution.

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 1.15. The `chessPiece` will need to know what a board

Listing 1.15 chess.fs:
A chess base: Module header and discriminated union types.

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

is, so we must define it as a mutually recursive class with `Board`. Furthermore, 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 1.16.

Our `Board` class is by far the largest and will be discussed in Listing 1.17–1.19. The constructor is shown in Listing 1.17. For memory efficiency, the board has been

Listing 1.16 chess.fs:
A chess base. Abstract type chessPiece.

```
4  /// An abstract chess piece
5  [<AbstractClass>]
6  type chessPiece(color : Color) =
7    let mutable _position : Position option = None
8    abstract member nameOfType : string // "king", "rook", ...
9    member this.color = color // White, Black
10   member this.position // E.g., (0,0), (3,4), etc.
11     with get() = _position
12     and set(pos) = _position <- pos
13   override this.ToString () = // E.g. "K" for white king
14     match color with
15       White -> (string this.nameOfType.[0]).ToUpper ()
16       | Black -> (string this.nameOfType.[0]).ToLower ()
17   /// A list of runs, which is a list of relative movements,
      e.g.,
18   /// [[(1,0); (2,0);...]; [(-1,0); (-2,0)]...]. Runs must be
19   /// ordered such that the first in a list is closest to the
      piece
20   /// at hand.
21   abstract member candidateRelativeMoves : Position list list
22   /// Available moves and neighbours ([(1,0); (2,0);...],
      [p1; p2])
23   member this.availableMoves (board : Board) : (Position list
      * chessPiece list) =
24     board.getVacantNNeighbours this
```

implemented using a `Array2D`, since pieces will move around often. For later use, in the members shown in Listing 1.19 we define two functions that convert relative coordinates into absolute coordinates on the board, and remove 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 written 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 1.18. Note that for efficiency, location is also stored in each piece, so `set` also needs to update the particular piece's position, as done in line 48. 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.

Listing 1.17 chess.fs:
A chess base: the constructor

```
25 /// A board
26 and Board () =
27   let _board = Collections.Array2D.create<chessPiece option>
      8 8 None
28   /// Wrap a position as option type
29   let validPositionWrap (pos : Position) : Position option =
30     let (rank, file) = pos // square coordinate
31     if rank < 0 || rank > 7 || file < 0 || file > 7 then
32       None
33     else
34       Some (rank, file)
35   /// Convert relative coordinates to absolute and remove
36   /// out-of-board coordinates.
37   let relativeToAbsolute (pos : Position) (lst : Position
      list) : Position list =
38     let addPair (a : int, b : int) (c : int, d : int) :
      Position =
39       (a+c,b+d)
40     // Add origin and delta positions
41     List.map (addPair pos) lst
42     // Choose absolute positions that are on the board
43     |> List.choose validPositionWrap
```

Listing 1.18 chess.fs:
A chess base: Board header, constructor, and non-static members.

```
44  /// Board is indexed using .[,] notation
45  member this.Item
46    with get(a : int, b : int) = _board.[a, b]
47    and set(a : int, b : int) (p : chessPiece option) =
48      if p.IsSome then p.Value.position <- Some (a,b)
49      _board.[a, b] <- p
50  /// Produce string of board for, e.g., the printfn function.
51  override this.ToString() =
52    let mutable str = ""
53    for i = Array2D.length1 _board - 1 downto 0 do
54      str <- str + string i
55      for j = 0 to Array2D.length2 _board - 1 do
56        let p =  _board.[i,j]
57        let pieceStr =
58            match p with
59              None -> " ";
60              | Some p -> p.ToString()
61        str <- str + " " + pieceStr
62      str <- str + "\n"
63    str + "  0 1 2 3 4 5 6 7"
64
65  /// Move piece by specifying source and target coordinates
66  member this.move (source : Position) (target : Position) :
     unit =
67    this.[fst target, snd target] <- this.[fst source, snd
     source]
68    this.[fst source, snd source] <- None
69  /// Find the tuple of empty squares and first neighbour if
     any.
70  member this.getVacantNOccupied (run : Position list) :
     (Position list * (chessPiece option)) =
71    try
72      // Find index of first non-vacant square of a run
73      let idx = List.findIndex (fun (i, j) ->
    this.[i,j].IsSome) run
```

The main computations are done in the static methods of the board, as shown in Listing 1.19. A chess piece must implement `candidateRelativeMoves`, and we

Listing 1.19 chess.fs:
A chess base: Board static members.

```
74        let (i,j) = run.[idx]
75        let piece = this.[i, j] // The first non-vacant
     neighbour
76        if idx = 0 then
77          ([], piece)
78        else
79          (run.[..(idx-1)], piece)
80      with
81        _ -> (run, None) // outside the board
82    /// find the list of all empty squares and list of
     neighbours
83    member this.getVacantNNeighbours (piece : chessPiece) :
     (Position list * chessPiece list)  =
84      match piece.position with
85        None ->
86          ([],[])
87        | Some p ->
88          let convertNWrap =
89            (relativeToAbsolute p) >> this.getVacantNOccupied
90          let vacantPieceLists = List.map convertNWrap
     piece.candidateRelativeMoves
91          // Extract and merge lists of vacant squares
92          let vacant = List.collect fst vacantPieceLists
93          // Extract and merge lists of first obstruction pieces
94          let neighbours = List.choose snd vacantPieceLists
95          (vacant, neighbours)
```

decided in Listing 1.16 that moves should be specified relative to the piece's 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, all the neighboring pieces are merged and both are returned to the caller.

Compiling the library files with the application and executing gives the result shown in Listing 1.20. We see that the program has correctly determined that initially, the

Listing 1.20: Running the program. Compare with Figure 1.3.

```
$ dotnet fsi chess.fs pieces.fs chessApp.fsx
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
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)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
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])
```

white king has the white rook as its neighbors 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, and 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.

1.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at additional concepts from object-oriented programming focussing on class hierarchies. You have learned:

- how to create a new class by **inheriting** from another
- how to use **up-casting** and **down-casting** to mix objects of similar inheritance
- how to define abstract data types as **abstract classes**
- how to work with **class interfaces** to make objects compatible with generic functions