Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2018-11-29 11:22:06+01:00

Contents

19 | Working With Files

An important part of programming is handling data. A typical source of data is hard-coded bindings and expressions from libraries or the program itself, and the result is often shown on a screen as text output on the console. This is a good starting point when learning to program, and one which we have relied heavily upon in this book until now. However, many programs require more: We often need to ask a user to input data via, e.g., typing text on a keyboard, clicking with a mouse, or striking a pose in front of a camera. We also often need to load and save data to files, retrieve and deposit information from the internet, and visualize data graphically, as sounds, or by controlling electrical appliances. Graphical user interfaces will be discussed in Chapter 23, and here we will concentrate on working with the console, files, and the general concept of streams.

File and stream input and output are supported via built-in namespaces and classes. For example, the `printf` family of functions discussed in Section 6.5 is defined in the `Printf` module of the `Fsharp.Core` namespace, and it is used to put characters on the `stdout` stream, i.e., to print on the screen. Likewise, `ReadLine` discussed in Section 6.6 is defined in the `System.Console` class, and it fetches characters from the `stdin` stream, that is, reads the characters the user types on the keyboard until newline is pressed.

A *file* on a computer is a resource used to store data in and retrieve data from. Files are often associated with a physical device, such as a hard disk, but can also be a virtual representation in memory. Files are durable, such that other programs can access them independently, given certain rules for access. A file has a name, a size, and a type, where the type is related to the basic unit of storage such as characters, bytes, and words, (`char`, `byte`, and `int32`). Often data requires a conversion between the internal format to and from the format stored in the file. E.g., floating point numbers are sometimes converted to a UTF8 string using `fprintf` in order to store them in a file in a human-readable form, and interpreted from UTF8 when retrieving them at a later point from the file. Files have a low-level structure, which varies from device to device, and the low-level details are less relevant for the use of the file and most often hidden for the user. Basic operations on files are *creation*, *opening*, *reading from*, *writing to*, *closing*, and *deleting*.

· file

· create file
· open file
· read file
· write file
· close file
· delete file
· stream

A *stream* is similar to files in that they are used to store data in and retrieve data from, but streams only allow for handling of data one element at a time, like the readout of a thermometer: we can make temperature readings as often as we like, making notes and thus saving a history of temperatures, but we cannot access the future. Hence, streams are in principle without an end, and thus have infinite size, and data from streams are programmed locally by considering the present and previous elements. In contrast, files are finite in size and allow for global operations on all the file's data. Files may be considered a stream, but the opposite is not true.[1]

[1] Jon: **Maybe add a figure illustrating the difference between files and streams.**

19.1 Command Line Arguments

Compiled programs may be started from the console with one or more arguments. E.g., if we have made a program called `prog`, then arguments may be passed as `mono prog arg1 arg2 ...`. To read the arguments in the program, we must define a function with the ***EntryPoint*** attribute, and this function must be of type `string array -> int`. · `EntryPoint`

Listing 19.1: Defining an entry point function with arguments from the console.

```
[<EntryPoint>]
let <funcIdent> <arg> =
  <bodyExpr>
```

`<funcIdent>` is the function's name, `<arg>` is the name of an array of strings, and `<bodyExpr>` is the function body. Return value 0 implies a successful execution of the program, while a non-zero value means failure. The entry point function can only be in the rightmost file in the list of files given to `fsharpc`. An example is given in Listing 19.2.

**Listing 19.2 commandLineArgs.fsx:
Interacting with a user with `ReadLine` and `WriteLine`.**

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    0  // Signals that program terminated successfully
```

An example execution with arguments is shown in Listing 19.3.

Listing 19.3: An example dialogue of running Listing 19.2.

```
$ fsharpc --nologo commandLineArgs.fsx
$ mono commandLineArgs.exe Hello World
Arguments passed to function : [|"Hello"; "World"|]
```

In Bash, the return value is called the *exit status* and can be tested using Bash's `if` statements, as demonstrated in Listing 19.4. · exit status

Listing 19.4: Testing return values in Bash when running Listing 19.2.

```
$ fsharpc --nologo commandLineArgs.fsx
$ if mono commandLineArgs.exe Hello World; then echo
   "success"; else echo "failure"; fi
Arguments passed to function : [|"Hello"; "World"|]
success
```

Also in Bash, the exit status of the last executed program can be accessed using the `$?` built-in environment variable. In Windows, this same variable is called `%errorlevel%`.

Stream	Description
`stdout`	Standard output stream used to display regular output. It typically streams data to the console.
`stderr`	Standard error stream used to display warnings and errors, typically streams to the same place as `stdout`.
`stdin`	Standard input stream used to read input, typically from the keyboard input.

Table 19.1: Three built-in streams in `System.Console`.

Function	Description
`Write: string -> unit`	Write to the console. E.g., `System.Console.Write "Hello world"`. Similar to `printf`.
`WriteLine: string -> unit`	As `Write`, but followed by a newline character, e.g., `WriteLine "Hello world"`. Similar to `printfn`.
`Read: unit -> int`	Wait until the next key is pressed, and read its value. The key pressed is echoed to the screen.
`ReadKey: bool -> System.ConsoleKeyInfo`	As `Read`, but returns more information about the key pressed. When given the value `true` as argument, then the key pressed is not echoed to the screen. E.g., `ReadKey true`.
`ReadLine unit -> string`	Read the next sequence of characters until newline from the keyboard, e.g., `ReadLine ()`.

Table 19.2: Some functions for interacting with the user through the console in the `System.Console` class. Prefix "`System.Console.`" is omitted for brevity.

19.2 Interacting With the Console

From a programming perspective, the console is a stream: A program may send new data to the console, but cannot return to previously sent data and make changes. Likewise, the program may retrieve input from the user, but cannot go back and ask the user to have input something else, nor can we peek into the future and retrieve what the user will input in the future. The console uses three built-in streams in `System.Console`, listed in Table 19.1. On the console, the standard output and error streams are displayed as text, and it is typically not possible to see a distinction between them. However, command-line interpreters such as Bash can, and it is possible from the command-line to filter output from programs according to these streams. However, a further discussion on this is outside the scope of this text. In `System.Console` there are many functions supporting interaction with the console, and the most important ones are shown in Table 19.2. Note that you must supply the empty argument "`()`" to the `Read` functions in order to run most of the functions instead of referring to them as values. A demonstration of the use of `Write`, `WriteLine`, and `ReadLine` is given in Listing 19.5.

Listing 19.5 userDialogue.fsx:
Interacting with a user with `ReadLine` and `WriteLine`. The user typed "3.5" and "7.4".

```
System.Console.WriteLine "To perform the multiplication of a
  and b"
System.Console.Write "Enter a: "
let a = float (System.Console.ReadLine ())
System.Console.Write "Enter b: "
let b = float (System.Console.ReadLine ())
System.Console.WriteLine ("a * b = " + string (a * b))
```

```
$ fsharpc --nologo userDialogue.fsx && mono userDialogue.exe
To perform the multiplication of a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.9
```

The functions `Write` and `WriteLine` act as `printfn` without a formatting string. These functions have many overloaded definitions, the description of which is outside the scope of this book. **For writing to the console, `printf` is to be preferred.** Advice

Often `ReadKey` is preferred over `Read`, since the former returns a value of type `System.ConsoleKeyInfo` which is a structure with three properties:

`Key`: A `System.ConsoleKey` enumeration of the key pressed. E.g., the character 'a' is `ConsoleKey.A`.

`KeyChar`: A unicode representation of the key.

`Modifiers`: A `System.ConsoleModifiers` enumeration of modifier keys shift, crtl, and alt.

An example of a dialogue is shown in Listing 19.6.

Listing 19.6 readKey.fsx:
Reading keys and modifiers. The user pressed 'a', 'shift-a', and 'crtl-a', and the program was terminated by pressing 'crtl-c'. The 'alt-a' combination does not work on MacOS.

```
open System

printfn "Start typing"
while true do
  let key = Console.ReadKey true
  let shift =
    if key.Modifiers = ConsoleModifiers.Shift then "SHIFT+"
   else ""
  let alt =
    if key.Modifiers = ConsoleModifiers.Alt then "ALT+" else ""
  let ctrl =
    if key.Modifiers = ConsoleModifiers.Control then "CTRL+"
   else ""
  printfn "You pressed: %s%s%s%s" shift alt ctrl
   (key.Key.ToString ())
```

```
$ fsharpc --nologo readKey.fsx && mono readKey.exe
Start typing
You pressed: A
You pressed: SHIFT+A
You pressed: CTRL+A
```

19.3 Storing and Retrieving Data From a File

A file stored on the filesystem has a name, and it must be opened before it can be accessed and closed when finished. Opening files informs the operating system that your program is now going to use the file. While a file is open, the operating system will protect it depending on how the file is opened. E.g., if you are going to write to the file, then this typically implies that no one else may write to the file at the same time, since simultaneous writing to a file may leave the resulting file in an uncertain state. Sometimes the operating system will realize that a file that was opened by a program is no longer being used, e.g., since the program is no longer running, but **it is good practice always to release reserved files, e.g., by closing them as soon as possible, such that other programs may have access to it.** On the other hand, it is typically safe for several programs to read the same file at the same time, but it is still important to close files after their use, such that the operating system can effectively manage the computer's resources. Reserved files are just one of the possible obstacles that you may meet when attempting to open a file. Other points of failure may be that the file does not exist, your program may not have sufficient rights for accessing it, or the device where the file is stored may have unreliable access. Thus, **never assume that accessing files always works, but program defensively, e.g., by checking the return status of the file accessing functions and by `try` constructions.**

Advice

Advice

Data in files may have been stored in various ways, e.g., it may contain UTF8 encoded characters or sequences of floating point numbers stored as raw bits in chunks of 64 bits, or it may be a sequence of bytes that are later going to be interpreted as an image in jpeg

System.IO.File	Description
`Open:` `(path : string) * (mode : FileMod` `-> FileStream`	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see Table 19.4. Other programs are not allowed to access the file before this program closes it.
`OpenRead: (path : string)` `-> FileStream`	Request the opening of a file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenText: (path : string)` `-> StreamReader`	Request the opening of an existing UTF8 file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenWrite: (path : string)` `-> FileStream`	Request the opening of a file on `path` for writing with `FileMode.OpenOrCreate`. Other programs may not access the file before this program closes it.
`Create: (path : string)` `-> FileStream`	Request the creation of a file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file before this program closes it.
`CreateText: (path : string)` `-> StreamWriter`	Request the creation of an UTF8 file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file before this program closes it.

Table 19.3: The family of `System.IO.File.Open` functions. See Table 19.4 for a description of `FileMode`, Tables 19.5 and 19.6 for a description of `FileStream`, Table 19.7 for a description of `StreamReader`, and Table 19.8 for a description of `StreamWriter`.

or tiff format. To aid in retrieving the data, F# has a family of open functions, all residing in the `System.IO.File` class. These are described in Table 19.3.

For the general `Open` function, you must also specify how the file is to be opened. This is done with a special set of values described in Table 19.4. An example of how a file is opened and later closed is shown in Listing 19.7.

FileMode	Description
`Append`	Open a file and seek to its end, if it exists, or create a new file. Can only be used together with FileAccess.Write. May throw `IOException` and `NotSupportedException` exceptions.
`Create`	Create a new file. If a file with the given filename exists, then that file is deleted. May throw the `UnauthorizedAccessException` exception.
`CreateNew`	Create a new file, but throw the `IOException` exception if the file already exists.
`Open`	Open an existing file. `System.IO.FileNotFoundException` exception is thrown if the file does not exist.
`OpenOrCreate`	Open a file, if it exists, or create a new file.
`Truncate`	Open an existing file and truncate its length to zero. Cannot be used together with `FileAccess.Read`.

Table 19.4: File mode values for the `System.IO.Open` function.

Listing 19.7 openFile.fsx:
Opening and closing a file, in this case, the source code of this same file.

```
let filename = "openFile.fsx"

let reader =
  try
    Some (System.IO.File.Open (filename,
   System.IO.FileMode.Open))
  with
    _ -> None

if reader.IsSome then
  printfn "The file %A was successfully opened." filename
  reader.Value.Close ()
```

```
$ fsharpc --nologo openFile.fsx && mono openFile.exe
The file "openFile.fsx" was successfully opened.
```

Notice how the example uses the defensive programming style, where the `try`-expression is used to return the optional datatype, and further processing is made dependent on the success of the opening operation.

In F#, the distinction between files and streams is not very clear. F# offers built-in support for accessing files as bytes through the `System.IO.FileStream` class, and for characters in a particular encoding through the `System.IO.TextReader` and `System.IO.TextWriter`.

A successfully opened `System.IO.FileStream` file, e.g., using `System.IO.File.OpenRead` from Table 19.3, will result in an `FileStream` object. From this object we can extract information about the file, such as the permitted operations and more listed in Table 19.5.
This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 19.6. E.g., we may `Seek` a particular position in the file, but only within the range of legal postions from 0 until the length of the file. Most operating systems do not necessarily write information to files immediately after one of the `Write` functions, but will often for optimization purposes collect information in a buffer that is to be written to a device in batches. However, sometimes is is useful to be

Property	Description
`CanRead`	Gets a value indicating whether the current stream supports reading. (Overrides Stream.CanRead.)
`CanSeek`	Gets a value indicating whether the current stream supports seeking. (Overrides Stream.CanSeek.)
`CanWrite`	Gets a value indicating whether the current stream supports writing. (Overrides Stream.CanWrite.)
`Length`	Gets the length of a stream in bytes. (Overrides Stream.Length.)
`Name`	Gets the name of the FileStream that was passed to the constructor.
`Position`	Gets or sets the current position of this stream. (Overrides Stream.Position.)

Table 19.5: Some properties of the `System.IO.FileStream` class.

`Method`	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Read byte[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadByte ()`	Read a byte from the file and advances the read position to the next byte.
`Seek int * SeekOrigin`	Sets the current position of this stream to the given value.
`Write byte[] * int * int`	Writes a block of bytes to the file stream.
`WriteByte byte`	Writes a byte to the current position in the file stream.

Table 19.6: Some methods of the `System.IO.FileStream` class.

able to force the operating system to empty its buffer to the device. This is called *flushing* and can be forced using the `Flush` function. · flushing

Text is typically streamed through the `StreamReader` and `StreamWriter`. These may be considered higher-order stream processing, since they include an added interpretation of the bits to strings. A `StreamReader` has methods similar to a `FileStream` object and a few new properties and methods, such as the `EndOfStream` property and `ReadToEnd` method, see Table 19.7. Likewise, a `StreamWriter` has an added method for automatically flushing after every writing operation. A simple example of opening a text-file and processing it is given in Listing 19.8.

Property/Method	Description
`EndOfStream`	Check whether the stream is at its end.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Peek ()`	Reads the next character, but does not advance the position.
`Read ()`	Reads the next character.
`Read char[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadLine ()`	Reads the next line of characters until a newline. Newline is discarded.
`ReadToEnd ()`	Reads the remaining characters until end-of-file.

Table 19.7: Some methods of the `System.IO.StreamReader` class.

Property/Method	Description
`AutoFlush : bool`	Gets or sets the auto-flush. If set, then every call to `Write` will flush the stream.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Write 'a`	Writes a basic type to the file.
`WriteLine string`	As `Write`, but followed by newline.

Table 19.8: Some methods of the `System.IO.StreamWriter` class.

Listing 19.8 readFile.fsx:
An example of opening a text file and using the `StreamReader` properties and methods.

```
let printFile (reader : System.IO.StreamReader) =
  while not(reader.EndOfStream) do
    let line = reader.ReadLine ()
    printfn "%s" line

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

```
$ fsharpc --nologo readFile.fsx && mono readFile.exe
let printFile (reader : System.IO.StreamReader) =
  while not(reader.EndOfStream) do
    let line = reader.ReadLine ()
    printfn "%s" line

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

Here the program reads the source code of itself, and prints it to the console.

Function	Description
`Copy (src : string, dest : string)`	Copy a file from `src` to `dest`, possibly overwriting any existing file.
`Delete string`	Delete a file
`Exists string`	Checks whether the file exists
`Move (from : string, to : string)`	Move a file from `src` to `to`, possibly overwriting any existing file.

Table 19.9: Some methods of the `System.IO.File` class.

Function	Description
`CreateDirectory string`	Create the directory and all implied sub-directories.
`Delete string`	Delete a directory.
`Exists string`	Check whether the directory exists.
`GetCurrentDirectory ()`	Get working directory of the program.
`GetDirectories (path : string)`	Get directories in `path`.
`GetFiles (path : string)`	Get files in `path`.
`Move (from : string, to : string)`	Move a directory and its content from `src` to `to`.
`SetCurrentDirectory : (path : string) -> unit`	Set the current working directory of the program to `path`.

Table 19.10: Some methods of the `System.IO.Directory` class.

19.4 Working With Files and Directories.

F# has support for managing files, summarized in the `System.IO.File` class and summarized in Table 19.9.

In the `System.IO.Directory` class there are a number of other frequently used functions, summarized in Table 19.10.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 19.11.

19.5 Reading From the Internet

The internet is a global collection of computers that are connected in a network using the internet protocol suite TCP/IP. The internet is commonly used for transport of data such as emails and for offering services such as web pages on the World Wide Web. Web resources are identified by a *Uniform Resource Locator* (*URL*), popularly known as a web page, and an URL contains information about where and how data from the web page is to be obtained. E.g., `https://en.wikipedia.org/wiki/F_Sharp_(programming_language)` contains 3 pieces of information: `https` is the protocol to be used to interact with the resource, `en.wikipedia.org` is the host's name, and `wiki/F_Sharp_(programming_language)` is the

· Uniform Resource Locator
· URL

Function	Description
`Combine string * string`	Combine two paths into a new path.
`GetDirectoryName (path: string)`	Extract the directory name from `path`.
`GetExtension (path: string)`	Extract the extension from `path`.
`GetFileName (path: string)`	Extract the name and extension from `path`.
`GetFileNameWithoutExtension (path : string)`	Extract the name without the extension from `path`.
`GetFullPath (path : string)`	Extract the absolute path from `path`.
`GetTempFileName ()`	Create a uniquely named and empty file on disk and return its full path.

Table 19.11: Some methods of the `System.IO.Path` class.

filename.

F#'s `System` namespace contains functions for accessing web pages as stream, as illustrated in Listing 19.9.

Listing 19.9 webRequest.fsx:
Downloading a web page and printing the first few characters.

```
/// Set up a url as a stream
let url2Stream url =
    let uri = System.Uri url
    let request = System.Net.WebRequest.Create uri
    let response = request.GetResponse ()
    response.GetResponseStream ()

/// Read all contents of a web page as a string
let readUrl url =
    let stream = url2Stream url
    let reader = new System.IO.StreamReader(stream)
    reader.ReadToEnd ()

let url = "http://fsharp.org"
let a = 40

let html = readUrl url
printfn "Downloaded %A. First %d characters are: %A" url a
   html.[0..a]
```

```
$ fsharpc --nologo webRequest.fsx && mono webRequest.exe
Downloaded "http://fsharp.org". First 40 characters are:
   "<!DOCTYPE html>
<html lang="en">
<head>"
```

To connect to a URL as a stream, we first need first format the URL string as a *Uniform Resource Identifiers* (*URI*), which is a generalization of the URL concept, using the `System.Uri` function. Then we must initialize the request by the `System.Net.WebRequest` function, and the response from the host is obtained by the `GetResponse` method. Finally, we can access the response as a stream by the `GetResponseStream` method. In the end, we convert the stream to a `StreamReader`, such that we can use the methods from Table 19.7 to access the web page. [2]

· Uniform Resource Identifiers
· URI

19.6 Resource Management

Streams and files are examples of computer resources that may be shared by several applications. Most operating systems allow for several applications to be running in parallel, and to avoid unnecessarily blocking and hogging of resources, all responsible applications must release resources as soon as they are done using them. F# has language constructions for automatic releasing of resources: the `use` binding and the `using` function. These automatically dispose of resources when the resource's name binding falls out of scope. Technically, this is done by calling the ***`Dispose`*** method on objects that implement the ***`System.IDisposable`*** interface. See Section 21.4 for more on interfaces.

· `Dispose`
· `System.IDisposable`

The ***`use`*** keyword is similar to `let`:

· `use`

Listing 19.10: Use binding expression.

```
use <valueIdent> = <bodyExpr> [in <expr>]
```

A `use` binding provides a binding between the `<bodyExpr>` expression to the name `<valueIdent>` in the following expression(s), and adds a call to `Dispose()` on `<valueIdent>` if it implements `System.IDisposable`. See for example Listing 19.11.

Listing 19.11 useBinding.fsx:
Using `use` instead of `let` releases disposable resources at end of scope.

```
open System.IO

let writeToFile (filename : string) (str : string) : unit =
   use file = File.CreateText filename
   file.Write str
   // file.Dispose() is implicitly called here,
   // implying that the file is closed.

writeToFile "use.txt" "Using 'use' closes the file, when out
  of scope."
```

Here, `file` is an `System.IDisposable` object, and `file.Dispose()` is called automatically before `writeToFile` returns. This implies that the file is closed. Had we used `let` instead, then the file would first be closed when the program terminates.

The higher-order function ***`using`*** takes a disposable object and a function, executes the function on the disposable objects, and then calls `Dispose()` on the disposable object.

· `using`

[2]Jon: **This section could be extended...**

This is illustrated in Listing 19.12

Listing 19.12 using.fsx:
The using function executes a function on an object and releases its disposable resources. Compare with Listing 19.11.

```
open System.IO

let writeToFile (str : string) (file : StreamWriter) : unit =
    file.Write str

using (File.CreateText "use.txt") (writeToFile "Disposed after
   call.")
// Dispose() is implicitly called on the anonymous file
   handle, implying
// that the file is automatically closed.
```

The main difference between `use` and `using` is that resources allocated using `use` are disposed at the end of its scope, while `using` disposes the resources after the execution of the function in its argument. In spite of the added control of `using`, we **prefer `use` over `using` due to its simpler structure.** Advice

19.7 Programming intermezzo: Name of Existing File Dialogue

A typical problem when working with files is

Problem 19.1
Ask the user for the name of an existing file.

Such dialogues often require the program to aid the user, e.g., by telling the user which files are available, and by checking that the filename entered is an existing file. A solution could be,

Listing 19.13 filenamedialogue.fsx:
Ask the user to input a name of an existing file.

```
let getAFileName () =
  let mutable filename = Unchecked.defaultof<string>
  let mutable fileExists = false
  while not(fileExists) do
    System.Console.Write("Enter Filename: ")
    filename <- System.Console.ReadLine()
    fileExists <- System.IO.File.Exists filename
  filename

let listOfFiles = System.IO.Directory.GetFiles "."
printfn "Directory contains: %A" listOfFiles
let filename = getAFileName ()
printfn "You typed: %s" filename
```

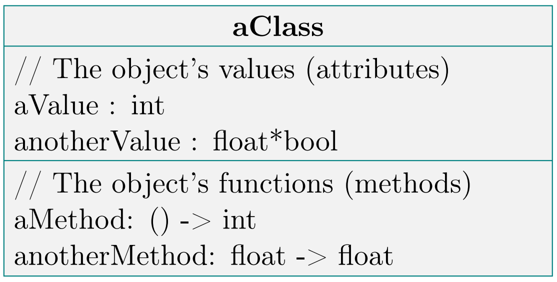
20 | Classes and Objects

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

· object-oriented programming

Object-oriented programming has a rich language for describing objects and their relations, which can seem overwhelming at first, and they will be explained in detail in this and following chapters. Here is a brief overview: The main programming structures are called a *classes* and *objects*. It is useful to think of classes as user defined types and objects as values of such types. However, there is more to classes and objects than types and values. Objects may contain both data and code, and it is sometimes useful to draw the corresponding class definition as shown in Figure 20.1. In this illustration, objects of type

· class
· object



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

Figure 20.1: A class is sometimes drawn as a figure.

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

· properties
· attributes
· methods
· members
· The Heap
· instantiate

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

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

Listing 20.1: Syntax for simple class definitions.

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

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

· self identifier
· constructor

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

· constructor
· fields
· functions

Members are declared using the *`member`*-keyword, which defines values and functions that are accessible from outside the class using the "`.`"-notation. In this manner, the members define the *interface* between the internal bindings in the constructor and an application program. Member values are called *properties* or *attributes*, and member functions are called *methods*. Note that members are immutable. The body of a member has access to the arguments, the constructor's bindings, and to all class members, regardless of the member's lexicographical order. In contrast, members are not available in the constructor unless the self identifier has been declared in the header using the keyword *`as`*, e.g., `type classMutable(name : string) as this = ....`

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

Consider the example in Figure 20.2. Here we have defined a class `car`, instantiated three

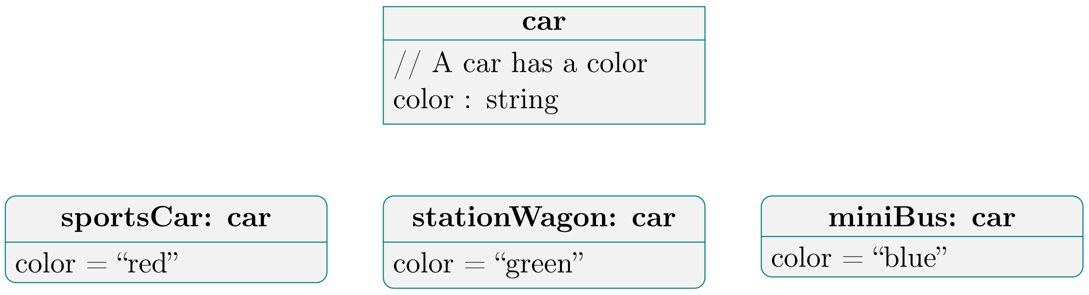


Figure 20.2: A class `car` is instantiated trice and bound to the names `sportsCar`, `stationWagon`, and `miniBus`, and each object's attributes are set to different values.

objects, and bound them to the names `sportsCar`, `stationWagon`, and `miniBus`. Each object has been given different values for the `color` attribute. In F# this could look like the code in Listing 20.2.

Listing 20.2 car.fsx:
Defining a class car, and making three instances of it. See also Figure 20.2.

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

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

In the example, the class `car` is defined in lines 1–3. Its header includes one string argument, `aColor`. The body of the constructor is empty, and the member section consists of lines 2–3. The class defines one attribute `color : string`. Note that when referring to a member inside an object, then we must use a *self identifier*. Here we use `this` as the self identifier, and as the example shows, we need not declare it in the class' header. A self identifier refers to the memory set aside to the particular instance of an object. It is common among other programming languages to use `this` as self identifier. F# is very flexible regarding what name can be used for the self-identifier, and the member section could as well have been `self.value`, `__.value`, or anything else, and it need not be the same in every member definition. Nevertheless, **consistency in the name used as self-identifier is strongly encouraged, preferably using a name that reflects the nature of the reference, such as `this` or `me`.** The objects are instantiated in lines 5–7, and the value of their attributes are accessed in line 8. In many languages, objects are instantiated using the *new* keyword, but in F# this is optional. I.e., `let sportsCar = car ("red")` is identical to `let sportsCar = new car ("red")`. Note that both the self identifier and member access uses the "`.`" notation.

· self identifier

Advice

· `new`

A more advanced implementation of a `car` class might include notions of a fuel gauge, fuel economy, and the ability to update the fuel gauge as the car is driven. An example of an implementation of this is given In Listing 20.3.

Listing 20.3 class.fsx:
Extending Listing 20.2 with fields and methods.

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

```
1 $ fsharpc --nologo class.fsx && mono class.exe
2 Created a car (8.0, 60.0)
3 Created a car (5.0, 45.0)
4 Fuel left after 100km driving:
5  sport: 52.0
6  economy: 40.0
```

Here in line 1, the constructor has 2 arguments: the fuel economy parameter and the initial amount of fuel in the tank. Thus, we are able to create 2 different cars with different fuel economy, as shown in lines 10–11. The amount of fuel left en each car object is stored in the mutable *field* `fuelLeft`. This is an example of a state of an object: It can be accessed outside the object by the `fuel` attribute, and it can be updated by the `drive` method.

· field

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

· private
· public

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.

```fsharp
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 a situation, where there is a range of possible choices of data representation, details of which we do wish to share with an application program. E.g., implementation details of arrays are not important for our ability to use them in applications. What matters is that the members that work on the array elements are well defined and efficient. Thus, the example demonstrates how we can build two simple methods `setValue` and `getValue` to set and get the data stored `v`. By making a distinction between the internal representation and how members give access to the data, we retain the possibility to change the internal representation without having to reprogram all the application programs. Analogously, we can change the engine in a car from one type to another without having to change the car's interaction with the driver and the road: steering wheel, pedals, wheels etc.

Such functions are called *accessors*. Internal states with setters and getters are a typical construction, since they allow for complicated computations when states are read to and written from, and gives the designer of the class the freedom to change the internal representation while keeping the interface the same. Accessors are so common that F# includes a special syntax for them: Classes can be made to act like variables using `member`...`with`...`and` keywords and the special function bindings `get()` and `set()`, as demonstrated in Listing 20.5.

· accessors

Listing 20.5 classGetSet.fsx:
Members can act as variables with the built-in get and set functions.

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

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

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

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 acts as a value rather than a variable, and values cannot be assigned to it in the application program.

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

Defining an *`Item`* property with extended `get` and `set` makes objects act as indexed variables, as demonstrated in Listing 20.6.

· `Item`

Listing 20.6 classGetSetIndexed.fsx:
Properties can act as indexed variables with the built-in get and set functions.

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

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

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

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:
Getters and setters for higher dimensional index variables.

```fsharp
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 Section 6.8. Consider the example in Listing 20.8.

· The Heap

Listing 20.8 classReference.fsx:
Objects assignment can cause aliasing.

```fsharp
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 example, consider a class whose objects each hold a unique identification number (id): When an object is instantiated, the object must be given the next available identification number. The next available id could be given as an argument to the constructor, however, this delegates the task of maintaining the uniqueness of ids to the application program. It is better to use a static field and delegate the administration of ids completely to the constructors, as demonstrated in Listing 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 line 2 that a static field `nextAvailableID` is created for the value to be shared by all objects. The initialization of its value is only performed once, at the beginning of program execution. However, every time an object is instantiated, the value of `nextAvailableID` is copied to the object's field `studentID` in line 3, and `nextAvailableID` is updated. The static field can be accessed with a static accessor, as demonstrated in line 7. Notice how this definition does not include a self-identifier, and that the member is accessible from the application in line 11 using the class' name, in both cases since it is not a member of any particular object.

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

In Listing 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 their functional equivalents.

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

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

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

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. In contrast, 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 reduces the chance of introducing new mistakes by attempting to correct old ones. Secondly, if we later decide to change the internal representation, then we only need to update one version of the multiplication function, hence we reduce programming time and risk of errors as well. 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 by 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 constructors are defined by new ()
  new (name : string) =
    classExtraConstructor (name, defaultGreetings)
  member this.name = name
  member this.str = greetings + " " + name

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

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

The top constructor that does not use the `new`-keyword is called the *primary constructor*. · primary constructor
The body of the additional constructor must call the primary constructor, and the body cannot extend the primary constructor's fields and functions. It is useful to **think of the primary constructor as a superset of arguments and the additional ones as subsets or specializations.** Advice 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.

Listing 20.16 classDoThen.fsx:
The optional do- and then-keywords allow for computations before and after the primary constructor is called.

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

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

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

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 Programming Intermezzo: Two Dimensional Vectors

Consider the following problem.

Problem 20.1

A Euclidean vector is a geometric object that has a direction, a length, and two operations: vector addition and scalar multiplication, see Figure 20.3. 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

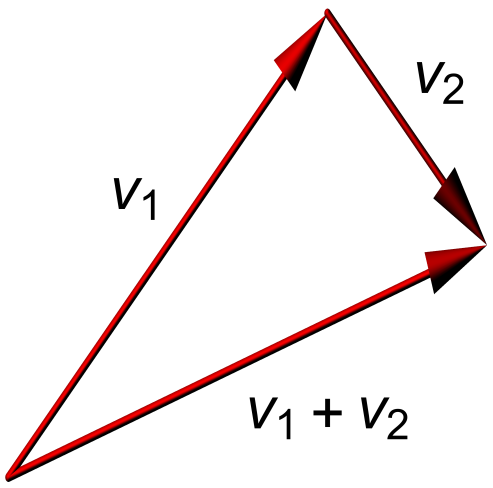


Figure 20.3: Illustration of vector addition in two dimensions.

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

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

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, respectively. The polar representation of vectors is also a tuple of real values (θ, l), where θ is the vector's angle from the x-axis and l is the vector's length. This representation is closely tied to the definition of a vector, and has the constraint that $0 \leq \theta < 2\pi$ and $0 \leq l$. This representation reminds us that vectors do not have a position. For vectors on polar form,

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

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 representations use a pair of reals to represent the vector,

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

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

· 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 20.17.

Listing 20.17 vectorApp.fsx:
An application using the library in Listing 20.18.

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

The application of the vector class seems natural, makes use of the optional discriminated unions, uses the infix operators "+" and "*" in a manner close to standard arithmetic, and interacts smoothly with the `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 20.18.

Listing 20.18 vector.fs:
A library serving the application in Listing 20.19.

```fsharp
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 respects dynamic scope.

The output of our combined library and application is shown in Listing 20.19.

Listing 20.19: Compiling and running the code from Listing 20.18 and 20.17.

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

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. 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.

· is-a relation

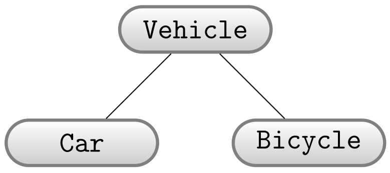


Figure 21.1: Both a car and a bicycle is a (type of) vehicle.

· inheritance
· base class
· derived class
· override

In F#, inheritance is indicated using the `inherit` keyword in the class definition. An extensions of the syntax in Listing 21.1 is:

Listing 21.1: A class definition with inheritance.

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

Extensions compared to Listing 21.1 are: `<baseClassIdent>` is the name of another class that this class optionally builds upon using the `inherit` keyword. Members may be regular members using the `member` keyword or other types of members indicated by the alternative keywords: `abstract member`, `default`, and `override`.

An example of defining base and derived classes for vehicles is shown In Listing 21.2.

Listing 21.2 vehicle.fsx:
New classes can be derived from old.

```fsharp
/// 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
```

```
$ fsharpc --nologo vehicle.fsx && mono vehicle.exe
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
```

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's constructor. I.e., `car` and `bicycle` automatically have the `wheels` attribute. Both derived classes additional members `maxPassengers` and `mustUseHelmet`, respectively.

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 21.3.

· overshadow
· `base`

Listing 21.3 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 21.4.

· upcast
· :>

Listing 21.4 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 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 21.5.

Listing 21.5 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
```

```
$ 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 21.6.[1]

· inheritance
· override
· `override`

Listing 21.6 classToString.fsx:
Overriding `ToString()` function for better interaction with members of the `printf` family of procedures. Compare with Listing 21.5.

```
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
```

```
$ 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)`", 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]Jon: **something about ToString not working with 's' format string in printf.**

21.3 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 21.7.

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

Listing 21.7 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 21.8.

Listing 21.8 abstractDefaultClass.fsx:
Default implementations in abstract classes makes implementations in derived classes optional. Compare with Listing 21.7.

```
/// 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
```

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

In the example, the program in Listing 21.7 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 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`

21.4 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 · interface · `interface with` can be upcasted as if they had an abstract base class of the interface' name. Consider the example in Listing 21.9.

Listing 21.9 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 21.9 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

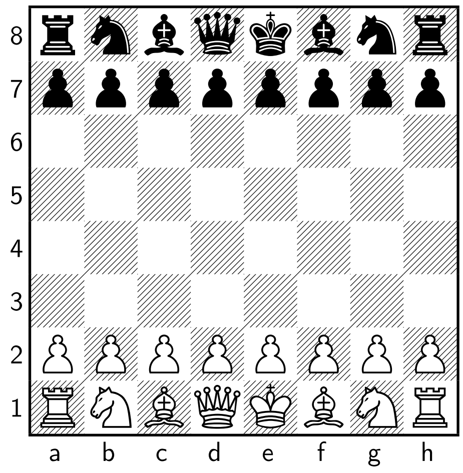


Figure 21.2: Starting position for the game of chess.

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.

21.5 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 21.10, and an illustration of what the program should do is shown in Figure 21.3.

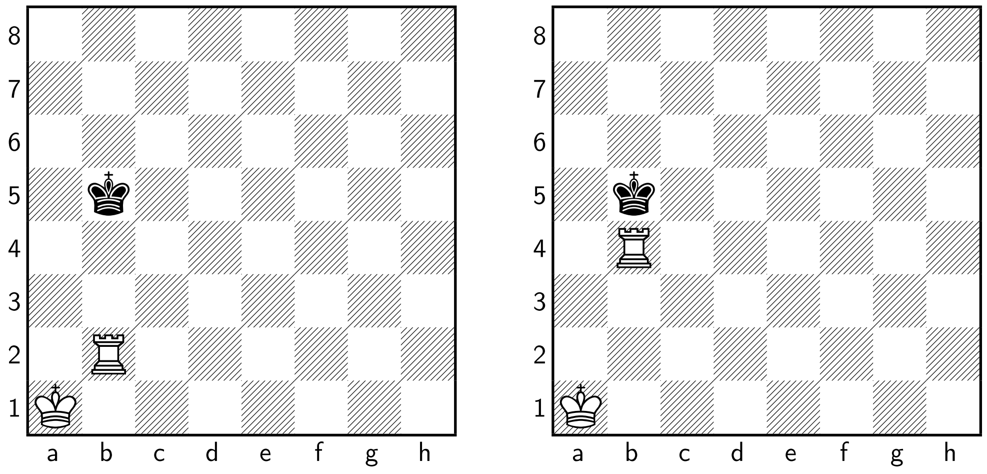


Figure 21.3: Starting at the left and moving white rook to b4.

Listing 21.10 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
```

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 the base class. As in the example application in Listing 21.10, 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 21.11.

Listing 21.11 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: 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.

· `swap`

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 21.12.

Listing 21.12 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 21.13.

Listing 21.13 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 21.14–21.16. The constructor is shown in Listing 21.14.

Listing 21.14 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 21.16 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 21.15.

Listing 21.15 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 21.16.

Listing 21.16 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 21.13 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 21.17.

Listing 21.17: 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.

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

[6] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[8] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[9] George Pólya. *How to solve it.* Princeton University Press, 1945.

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index