1 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 **??**, 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 **??** 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 **??** 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*.

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 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`.

Listing 1.1: 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 1.2.

Listing 1.2 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 1.3.

Listing 1.3: An example dialogue of running Listing 1.2.

```
$ 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 1.4.

Listing 1.4: Testing return values in Bash when running Listing 1.2.

```
$ 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%`.

1.2 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 1.1. On the console, the standard output and

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 1.1: Three built-in streams in `System.Console`.

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 1.2. Note that you must supply the

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 1.2: Some functions for interacting with the user through the console in the `System.Console` class. Prefix “`System.Console.`” is omitted for brevity.

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 1.5.

Listing 1.5 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.** ⋆

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 1.6.

Listing 1.6 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
```

1.3 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.**

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 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 1.3.

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 1.4. An example of how a file is opened and later closed is shown in Listing 1.7.

Listing 1.7 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 a 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,

System.IO.File	Description
`Open: (path : string) * (mode : FileM -> FileStream`	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see Table 1.4. 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 1.3: The family of `System.IO.File.Open` functions. See Table 1.4 for a description of `FileMode`, Tables 1.5 and 1.6 for a description of `FileStream`, Table 1.7 for a description of `StreamReader`, and Table 1.8 for a description of `StreamWriter`.

and for characters in a particular encoding through the `System.IO.TextReader` and `System.IO.TextWriter`.

A successfully opened `System.IO.FileStream` file by, e.g., `System.IO.File.OpenRead` from Table 1.3, 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 1.5. This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 1.6. 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

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

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 1.5: Some properties of the `System.IO.FileStream` class.

in batches. However, sometimes is is useful to be 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.

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 1.6: Some methods of the `System.IO.FileStream` class.

Listing 1.8 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
```

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 1.7. Likewise, a `StreamWriter` has an added method

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 1.7: Some methods of the `System.IO.StreamReader` class.

for automatically flushing after every writing operation. A simple example of opening

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 1.8: Some methods of the `System.IO.StreamWriter` class.

a text-file and processing it is given in Listing 1.8. Here the program reads the source code of itself, and prints it to the console.

1.4 Working With Files and Directories.

F# has support for managing files, summarized in the `System.IO.File` class and summarized in Table 1.9.

In the `System.IO.Directory` class there are a number of other frequently used functions, summarized in Table 1.10.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 1.11.

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 1.9: Some methods of the `System.IO.File` class.

Function	Description
`CreateDirectory string`	Create the directory and all implied subdirectories.
`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 1.10: Some methods of the `System.IO.Directory` class.

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 1.11: Some methods of the `System.IO.Path` class.

1.5 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: The host uses the `https` protocol, `en.wikipedia.org` is its name, and `wiki/F_Sharp_(programming_language)` is the filename.

F#'s `System` namespace contains functions for accessing web pages as stream, as illustrated in Listing 1.9.

Listing 1.9 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 1.7 to access the web page.

1.6 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 ?? for more on interfaces.

The *use* keyword is similar to let:

Listing 1.10: 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 in contrast to let, it also adds a call to `Dispose()` on <valueIdent> if it implements `System.IDisposable`. See for example Listing 1.11.

Listing 1.11 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. This is illustrated in Listing 1.12

Listing 1.12 using.fsx:
The `using` function executes a function on an object and releases its disposable resources. Compare with Listing 1.11.

```
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.** ⋆

1.7 Programming intermezzo: Name of Existing File Dialogue

A typical problem when working with files is

Problem 1.1

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 1.13 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
```