Chapter 1

Working With Files

Abstract Most programs do not live in isolation but interacts with the world around it. For example, many programs read and write from and to files or the internet. This is in general known as input and output. We have previously seen how to interact with the user by reading and writing to the terminal. In this chapter you will learn how to:

- Giving programs to ability to take arguments from the terminal,
- Read and write from to and from files,
- Handle errors by catching exceptions and if needed, throwing them yourself, and

Particularly with the topic of exceptions, we are leaving the domain of functional programming, and where errors before to a large extend were caught before a given program was run, we will now learn how to write programs, that can handle errors while they are running.

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. 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, `printf` is a family of functions 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. An example execution with arguments is shown in Listing 1.3.

**Listing 1.2 commandLineArgs/Program.fs:
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
```

In Bash, the return value is called the *exit status* and can be tested using Bash's `if`

Listing 1.3: An example dialogue of running Listing 1.2.

```
$ cd commandLineArgs; dotnet run Hello World
Arguments passed to function : [|"Hello"; "World"|]
```

statements, as demonstrated in Listing 1.4. Also in Bash, the exit status of the last

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

```
$ cd commandLineArgs; if dotnet run Hello World; then echo
   "success"; else echo "failure"; fi
Arguments passed to function : [|"Hello"; "World"|]
success
```

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

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.

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 1.5. The functions `Write` and `WriteLine` act as `printfn` without a formatting string. These functions have

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 multiply 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))
```

```
$ dotnet fsi userDialogue.fsx
To multiply a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.900000000000002
```

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.

1.3 Exceptions

Exceptions are runtime errors, such as division by zero. E.g., attempting integer division by zero halts execution and a long somewhat cryptic error message is written to screen, as illustrated in Listing 1.7. The error message contains much information. The first part, `System.DivideByZeroException: Attempted to divide by zero` is the error-name with a brief ellaboration. Then follows a list libraries that were involved when the error occurred, and finally F# states that it `Stopped due to error`. `System.DivideByZeroException` is a built-in exception type, and the built-in integer division operator chooses to raise the exception when the undefined

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 ())
```

```
$ dotnet fsi readKey.fsx
Start typing
You pressed: A
You pressed: SHIFT+A
You pressed: CTRL+A
```

Listing 1.7: Division by zero halts execution with an error message.

```
> 3 / 0;;
System.DivideByZeroException: Attempted to divide by zero.
   at <StartupCode$FSI_0002>.$FSI_0002.main@() in /Users/jrh630/repositories/fsharp-book/src/stdin:line 1
```

division by zero is attempted. Many times such errors can be avoided by clever program design. However, this is not always possible or desirable, which is why F# implements exception handling for graceful control.

Exceptions are a basic-type called `exn`, and F# has a number of built-in ones, a few of which are listed in Table 1.3.

Attribute	Description
`ArgumentException`	Arguments provided are invalid.
`DivideByZeroException`	Division by zero.
`NotFiniteNumberException`	floating point value is plus or minus infinity, or Not-a-Number (NaN).
`OverflowException`	Arithmetic or casting caused an overflow.
`IndexOutOfRangeException`	Attempting to access an element of an array using an index which is less than zero or equal or greater than the length of the array.

Table 1.3 Some built-in exceptions. The prefix `System.` has been omitted for brevity.

Exceptions are handled by the `try`–keyword expressions. We say that an expression may *raise* or *cast* an exception and that the `try`–expression may *catch* and *handle* the exception by another expression.

Exceptions like in Listing 1.7 may be handled by `try`–`with` expressions, as demonstrated in Listing 1.8. In the example, when the division operator raises the

Listing 1.8 exceptionDivByZero.fsx:
A division by zero is caught and a default value is returned.

```
let div enum denom =
  try
    enum / denom
  with
    | :? System.DivideByZeroException -> System.Int32.MaxValue
printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZero.fsx
3 / 1 = 3
3 / 0 = 2147483647
```

`System.DivideByZeroException` exception, then `try`–`with` catches it and returns the value `System.Int32.MaxValue`. Division by zero is still an undefined operation, but with the exception system, the program is able to receive a message about this undefined situation and choose an appropriate action.

The `try` expressions comes in two flavors: `try`–`with` and `try`–`finally` expressions.

The *`try`–`with`* expression has the following syntax,

Listing 1.9: Syntax for the `try`–`with` exception handling.

```
try
   <testExpr>
with
   [ | ] <pat1> -> <exprHndl1>
   | <pa2> -> <exprHndl2>
   | <pat3> -> <exprHndl3>
   ...
```

where `<testExpr>` is an expression which might raise an exception, `<patn>` is a pattern, and `<exprHndln>` is the corresponding exception handler. The value of the `try`–expression is either the value of `<testExpr>`, if it does not raise an exception, or the value of the exception handler `<exprHndln>` of the first matching pattern `<patn>`. The above is using lightweight syntax. Regular syntax omits newlines.

In Listing 1.8 *dynamic type matching* is used using the "`:?`" lexeme, i.e., the pattern matches exception with type `System.DivideByZeroException` at runtime. The exception value may contain furter information and can be accessed if named using the *as*–keyword, as demonstrated in Listing 1.10. Here the exception value is bound

Listing 1.10 exceptionDivByZeroNamed.fsx:
Exception value is bound to a name. Compare to Listing 1.8.

```
let div enum denom =
  try
    enum / denom
  with
    | :? System.DivideByZeroException as ex ->
      printfn "Error: %s" ex.Message
      System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZeroNamed.fsx
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

to the name `ex`.

All exceptions may be caught as the dynamic type `System.Exception`, and F# implements a short-hand for catching an exception and binding its value to a name as demonstrated in Listing 1.11 Finally, the short-hand may be guarded with a

Listing 1.11 exceptionDivByZeroShortHand.fsx:
An exception of type `System.Exception` is bound to a name. Compare to Listing 1.10.

```
let div enum denom =
  try
    enum / denom
  with
    | ex -> printfn "Error: %s" ex.Message; System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZeroShortHand.fsx
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

when–guard, as demonstrated in Listing 1.12. The first pattern only matches the `System.Exception` exception when `enum` is 0, in which case the exception handler returns 0.

Listing 1.12 exceptionDivByZeroGuard.fsx:
An exception of type `System.Exception` is bound to a name and guarded. Compare to Listing 1.11.

```
let div enum denom =
  try
    enum / denom
  with
    | ex when enum = 0 -> 0
    | ex -> System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
printfn "0 / 0 = %d" (div 0 0)
```

```
$ dotnet fsi exceptionDivByZeroGuard.fsx
3 / 1 = 3
3 / 0 = 2147483647
0 / 0 = 0
```

Thus, if you don't care about the type of exception, then you need only use the short-hand pattern matching and name binding demonstrated in Listing 1.11 and Listing 1.12, but if you would like to distinguish between types of exceptions, then you must use explicit type matching and possibly value binding demonstrated in Listing 1.8 and Listing 1.10

The *`try`–`finally`* expression has the following syntax,

Listing 1.13: Syntax for the `try`–`finally` exception handling.

```
try
   <testExpr>
finally
   <cleanupExpr>
```

The `try`–`finally` expression evaluates the `<cleanupExpr>` expression following evaluation of the `<testExpr>`, regardless of whether an exception is raised or not, as illustrated in Listing 1.14. Here, the `finally` branch is evaluated following the evaluation of the test expression regardless of whether the test expression raises an exception or not. However, if an exception is raised in a `try`–`finally` expression and there is no outer `try`–`with` expression, then execution stops without having evaluated the `finally` branch.

Exceptions can be raised using the *`raise`*-function,

Listing 1.15: Syntax for the `raise` function that raises exceptions.

```
raise (<expr>)
```

Listing 1.14 exceptionDivByZeroFinally.fsx:
The `finally` branch is executed regardless of an exception.

```
let div enum denom =
  printf "Doing division:"
  try
    printf " %d %d." enum denom
    enum / denom
  finally
    printfn " Division finished."

printfn "3 / 1 = %d" (try div 3 1 with ex -> 0)
printfn "3 / 0 = %d" (try div 3 0 with ex -> 0)
```

```
$ dotnet fsi exceptionDivByZeroFinally.fsx
Doing division: 3 1. Division finished.
3 / 1 = 3
Doing division: 3 0. Division finished.
3 / 0 = 0
```

An example of raising the `System.ArgumentException` is shown in Listing 1.16 In this example, division by zero is never attempted and instead an exception is raised

Listing 1.16 raiseArgumentException.fsx:
Raising the division by zero with customized message.

```
let div enum denom =
  if denom = 0 then
    raise (System.ArgumentException "Error: \"division by
   0\"")
  else
    enum / denom

printfn "3 / 0 = %s" (try (div 3 0 |> string) with ex ->
   ex.Message)
```

```
$ dotnet fsi raiseArgumentException.fsx
3 / 0 = Error: "division by 0"
```

which must be handled by the caller. Note that the type of `div` is `int -> int -> int` because `denom` is compared with an integer in the conditional statement. This contradicts the typical requirements for `if` statements, where every branch has to return the same type. However, any code that explicitly raise exceptions are ignored, and the type is inferred by the remaining branches.

Programs may define new exceptions using the syntax,

Listing 1.17: Syntax for defining new exceptions.

```
exception <ident> of <typeId> {* <typeId>}
```

An example of defining a new exception and raising it is given in Listing 1.18. Here

Listing 1.18 exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

printfn "picky %A = %A" 3 (try picky 3 |> string with ex ->
   ex.Message)
printfn "picky %A = %A" 5 (try picky 5 |> string with ex ->
   ex.Message)
```

```
$ dotnet fsi exceptionDefinition.fsx
picky 3 = "3"
picky 5 = "Exception of type 'FSI_0001+DontLikeFive' was
   thrown."
```

an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. The example demonstrates that catching the exception as a `System.Exception` as in Listing 1.11, the `Message` property includes information about the exception name but not its argument. To retrieve the argument `"5 sucks"`, we must match the exception with the correct exception name, as demonstrated in Listing 1.19.

F# includes the *`failwith`* function to simplify the most common use of exceptions. It is defined as `failwith : string -> exn` and takes a string and raises the built-in `System.Exception` exception. An example of its use is shown in Listing 1.20. To catch the `failwith` exception, there are several choices. The exception casts a `System.Exception` exception, which may be caught using the `:?` pattern, as shown in Listing 1.21. However, this gives annoying warnings, since F# internally is built such that all exception match the type of `System.Exception`. Instead, it is better to either match using the wildcard pattern as in Listing 1.22, or use the built-in `Failure` pattern as in Listing 1.23. Notice how only the `Failure` pattern allows for the parsing of the message given to `failwith` as an argument.

Invalid arguments are such a common reason for failures, that a built-in function for handling them has been supplied in F#. The *`invalidArg`* takes 2 strings and raises the built-in `ArgumentException`, as shown in Listing 1.24. The `invalidArg` function raises an `System.ArgumentException`, as shown in Listing 1.25.

Listing 1.19 exceptionDefinitionNCatch.fsx:
Catching a user-defined exception.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

try
  printfn "picky %A = %A" 3 (picky 3)
  printfn "picky %A = %A" 5 (picky 5)
with
  | DontLikeFive msg -> printfn "Exception caught with
   message: %s" msg
```

```
$ dotnet fsi exceptionDefinitionNCatch.fsx
picky 3 = 3
Exception caught with message: 5 sucks
```

Listing 1.20 exceptionFailwith.fsx:
An exception raised by `failwith`.

```
if true then failwith "hej"
```

```
$ dotnet fsi exceptionFailwith.fsx
System.Exception: hej
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionFailwith.fsx:line 1
Stopped due to error
```

The `try` construction is typically used to gracefully handle exceptions, but there are times where you may want to pass on the bucket, so to speak, and re-raise the exception. This can be done with the *`reraise`*, as shown in Listing 1.26. The `reraise` function is only allowed to be the final call in the expression of a `with` rule.

1.4 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

Listing 1.21 exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    :? System.Exception -> printfn "So failed"
```

```
$ dotnet fsi exceptionSystemException.fsx


exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold



exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold

So failed
```

Listing 1.22 exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    _ -> printfn "So failed"
```

```
$ dotnet fsi exceptionMatchWildcard.fsx
So failed
```

will protect it depending on how the file is opened. E.g., if you are going to write to

Listing 1.23 exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
```

```
$ dotnet fsi exceptionFailure.fsx
The castle of "Arrrrg"
```

Listing 1.24 exceptionInvalidArg.fsx:
An exception raised by `invalidArg`. Compare with Listing 1.16.

```fsharp
if true then invalidArg "a" "is too much 'a'"
```

```
$ dotnet fsi exceptionInvalidArg.fsx
System.ArgumentException: is too much 'a' (Parameter 'a')
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionInvalidArg.fsx:line 1
Stopped due to error
```

Listing 1.25 exceptionInvalidArgNCatch.fsx:
Catching the exception raised by `invalidArg`.

```fsharp
let _ =
  try
    invalidArg "a" "is too much 'a'"
  with
    :? System.ArgumentException -> printfn "Argument is no
   good!"
```

```
$ dotnet fsi exceptionInvalidArgNCatch.fsx
Argument is no good!
```

Listing 1.26 exceptionReraise.fsx:
Reraising an exception.

```fsharp
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
      reraise()
```

```
$ dotnet fsi exceptionReraise.fsx
The castle of "Arrrrg"
System.Exception: Arrrrg
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionReraise.fsx:line 3
Stopped due to error
```

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 1.4.

`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 1.5. 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 1.4 The family of `System.IO.File.Open` functions. See Table 1.5 for a description of `FileMode`, Tables 1.6 and 1.7 for a description of `FileStream`, Table 1.8 for a description of `StreamReader`, and Table 1.9 for a description of `StreamWriter`.

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 1.5. An example of how a file is opened and later closed is shown in Listing 1.27. 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.

`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 1.5 File mode values for the `System.IO.Open` function.

Listing 1.27 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 ()
```

```
$ dotnet fsi openFile.fsx
```

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 by, e.g., `System.IO.File.OpenRead` from Table 1.4, 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 1.6. This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 1.7. 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 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.

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 1.6 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 1.7 Some methods of the `System.IO.FileStream` class.

Text is typically streamed through the `StreamReader` and `StreamWriter`. These may be considered higher-order stream processing, since they include an added

Listing 1.28 readFile.fsx:
An example of opening a text file and using the `StreamReader` properties and methods.

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

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

```
$ dotnet fsi readFile.fsx
let rec printFile (reader : System.IO.StreamReader) =
  if not(reader.EndOfStream) then
    let line = reader.ReadLine ()
    printfn "%s" line
    printFile reader

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

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 1.8. Likewise, a `StreamWriter` has an

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 1.8 Some methods of the `System.IO.StreamReader` class.

added method for automatically flushing after every writing operation. A simple

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 1.9 Some methods of the `System.IO.StreamWriter` class.

example of opening a text-file and processing it is given in Listing 1.28. Here the program reads the source code of itself, and prints it to the console.

1.5 Working With Files and Directories.

F# has support for managing files, summarized in the `System.IO.File` class and summarized in Table 1.10.

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 1.10 Some methods of the `System.IO.File` class.

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

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 1.11 Some methods of the `System.IO.Directory` class.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 1.12.

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 1.12 Some methods of the `System.IO.Path` class.

1.6 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.

We will limit our request to the present directory and use `System.Console.ReadLine` to get input from the user. Our strategy will be twofold. Firstly we will query the filesystem for the existing files using `System.IO.Directory.GetFiles`, and print these to the screen. Secondly, we will use `System.IO.File.Exists` to ensure that a file exists with the entered filename. We use the `Exists` function rather than examining the array obtained with `GetFiles`, since files may have been added or removed, since the `GetFiles` was called. A solution is shown in Listing 1.29. Note that it is programmed using a `while`-loop and with a flag `fileExists` used to exit the loop. The solution has a caveat: What should be done if the user decides not to enter a ⋆ filename at all. **Including a 'cancel'-option is a good style for any user interface, and should be offered when possible.** In a text-based dialogue, this would require us to use an input, which cannot be a filename, to ensure that all possible filenames and 'cancel'-option is available to the user. This problem has not been addressed in the code.

Listing 1.29 filenamedialogue.fsx:
Ask the user to input a name of an existing file.

```
let rec getAFileName () =
  System.Console.Write("Enter Filename: ")
  let filename = System.Console.ReadLine()
  if System.IO.File.Exists filename then filename
  else getAFileName ()

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

1.7 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 1.30: 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 1.31. Here, `file` is an `System.IDisposable` object, and

Listing 1.31 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" "'Use' cleans up, when out of scope."
```

`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 1.32

Listing 1.32 using.fsx:
The `using` function executes a function on an object and releases its disposable resources. Compare with Listing 1.31.

```
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.**

1.8 Key concepts and terms in this chapter

In this chapter, we have looked at how to interface programs with external data sources. Key concepts have been:

- **Arguments to programs** run in the terminal can be retrieved from within a program at running time
- A **file** is a general concept, for places to put data to or retrieve data from.
- Files are often a shared resource, and there may be other programs on a machine, which need access.
- To avoid chaos, the operating system **locks access to files** by other programs and therefore it is also important to **release the locks**, when your program is done.
- A **stream** is similar to a file for reading, however, they do not allow for random access.

- Errors may occur due to events outside the domain of control for a program. Such errors give rise to **exceptions**.
- Exceptions may be **caught** or **thrown** by our programs.