CHAPTER 2
Fundamentals

In Chapter 1, you wrote your first F# program. I broke it down to give you a feel for what you were doing, but much of the code is still a mystery. In this chapter, I provide the necessary foundation for you to understand that code fully, but more importantly, I present several more examples that you can use to grasp the basics of F# before you move on to the more complex features.

The first section of this chapter covers primitive types, like `int` and `string`, which are the building blocks for all F# programs. I then cover functions so you can manipulate data.

The fourth section details foundational types such as `list`, `option`, and `unit`. Mastering these types enables you to expand into the object-oriented and functional styles of F# code covered in later chapters.

By the end of this chapter, you will be able to write simple F# programs for processing data. In future chapters, you learn how to add power and expressiveness to your code, but for now, let's master the basics.

Primitive Types

A *type* is a concept or abstraction, and is primarily about enforcing safety. Types represent a proof of sorts if a conversion will work. Some types are straightforward (representing an integer), whereas others are far more abstract (like a function). F# is statically typed, meaning that type checking is done at compile time. For example, if a function accepts an integer as a parameter, you will get a compiler error if you try to pass in a string.

Like C# and VB.NET, F# supports the full cast and crew of *primitive .NET types*, which are standard across most programming languages. They are built into the F# language and separate from *user-defined types* that you define yourself.

To create a value, simply use a *let binding* via the `let` keyword. For example, the following code defines a new value `x` in an FSI session. You can do much more with `let` bindings, but we'll save that for Chapter 3:

```
> let x = 1;;

val x : int = 1
```

Numeric Primitives

Numeric primitives come in two varieties: integers and floating-point numbers. Integer types vary by size, so that some types take up less memory and can represent a smaller range of numbers. Integers can also be signed or unsigned based on whether or not they can represent negative values. Floating-point types vary in size too; in exchange for taking up more memory, they provide more precision for the values they hold.

To define new numeric values, use a `let` binding followed by an integer or floating-point literal with an optional suffix. The suffix determines the type of integer or floating-point number (a full list of available primitive numeric types and their suffixes is provided in Table 2-1):

```
> let answerToEverything = 42UL;;

val answerToEverything : uint64 = 42UL

> let pi = 3.1415926M;;

val pi : decimal = 3.1415926M

> let avogadro = 6.022e23;;

val avogadro : float = 6.022e23
```

Table 2-1. Numerical primitives in F#

Type	Suffix	.NET Type	Range
`byte`	`uy`	`System.Byte`	0 to 255
`sbyte`	`y`	`System.SByte`	−128 to 127
`int16`	`s`	`System.Int16`	−32,768 to 32,767
`uint16`	`us`	`System.UInt16`	0 to 65,535
`int, int32`		`System.Int32`	-2^{31} to $2^{31}-1$
`uint32`	`u`	`System.UInt32`	0 to $2^{32}-1$
`int64`	`L`	`System.Int64`	-2^{63} to $2^{63}-1$
`uint64`	`UL`	`System.UInt64`	0 to $2^{64}-1$
`float`		`System.Double`	A double-precision floating point based on the IEEE 64 standard. Represents values with approximately 15 significant digits.

Type	Suffix	.NET Type	Range
`float32`	`f`	`System.Single`	A single-precision floating point based on the IEEE 32 standard. Represents values with approximately 7 significant digits.
`decimal`	`M`	`System.Decimal`	A fixed-precision floating-point type with precisely 28 digits of precision.

F# will also allow you to specify values in hexadecimal (base 16), octal (base 8), or binary (base 2) using a prefix `0x`, `0o`, or `0b`:

```
> let hex = 0xFCAF;;

val hex : int = 64687

> let oct = 0o7771L;;

val oct : int64 = 4089L

> let bin = 0b00101010y;;

val bin : sbyte = 42y

> (hex, oct, bin);;
val it : int * int64 * sbyte = (64687, 4089L, 42)
```

If you are familiar with the IEEE 32 and IEEE 64 standards, you can also specify floating-point numbers using hex, octal, or binary. F# will convert the binary value to the floating-point number it represents. When using a different base to represent floating-point numbers, use the `LF` suffix for `float` types and `lf` for `float32` types:

```
> 0x401E000000000000LF;;
val it : float = 7.5

> 0x00000000lf;;
val it : float32 = 0.0f
```

Arithmetic

You can use standard arithmetic operators on numeric primitives. Table 2-2 lists all supported operators. Like most programming languages, integer division rounds down, discarding the remainder.

Table 2-2. Arithmetic operators

Operator	Description	Example	Result
`+`	Addition	`1 + 2`	`3`
`-`	Subtraction	`1 - 2`	`-1`
`*`	Multiplication	`2 * 3`	`6`
`/`	Division	`8L / 3L`	`2L`
`**`	Power[a]	`2.0 ** 8.0`	`256.0`

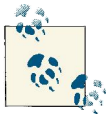
Operator	Description	Example	Result
%	Modulus	7 % 3	1

[a] Power, the ** operator, only works for float and float32 types. To raise the power of an integer value, you must either convert it to a floating-point number first or use the pown function.

By default, arithmetic operators do not check for overflow, so if you exceed the range allowed by an integer value by addition, it will overflow to be negative (similarly, subtraction will result in a positive number if the number is too small to be stored in the integer type):

```
> 32767s + 1s;;
val it : int16 = -32768s

> -32768s + -1s;;
val it : int16 = 32767s
```



If integer overflow is a cause for concern, you should consider using a larger type or using checked arithmetic, discussed in Chapter 7.

F# features all the standard mathematical functions you would expect, with a full listing in Table 2-3.

Table 2-3. Common mathematical functions

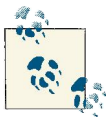
Routine	Description	Example	Result
abs	Absolute value of a number	abs -1.0	1.0
ceil	Round up to the nearest integer	ceil 9.1	10.0
exp	Raise a value to a power of e	exp 1.0	2.718
floor	Round down to the nearest integer	floor 9.9	9.0
sign	Sign of the value	sign -5	-1
log	Natural logarithm	log 2.71828	1.0
log10	Logarithm in base 10	log10 1000.0	3.0
sqrt	Square root	sqrt 4.0	2.0
cos	Cosine	cos 0.0	1.0
sin	Sine	sin 0.0	0.0
tan	Tangent	tan 1.0	1.557
pown	Compute the power of an integer	pown 2L 10	1024L

Conversion Routines

One of the tenets of the F# language is that there are no implicit conversions. This means that the compiler will not automatically convert primitive data types for you behind the scenes, such as converting an `int16` to an `int64`. This eliminates subtle bugs by removing surprise conversions. Instead, to convert primitive values, you must use an explicit conversion function listed in Table 2-4. All of the standard conversion functions accept all other primitive types—including strings and chars.

Table 2-4. Numeric primitive conversion routines

Routine	Description	Example	Result
`sbyte`	Converts data to an `sybte`	`sbyte -5`	`-5y`
`byte`	Converts data to a `byte`	`byte "42"`	`42uy`
`int16`	Converts data to an `int16`	`int16 'a'`	`97s`
`uint16`	Converts data to a `unit16`	`uint16 5`	`5us`
`int32, int`	Converts data to an `int`	`int 2.5`	`2`
`uint32`	Converts data to a `uint32`	`uint32 0xFF`	`255`
`int64`	Converts data to an `int64`	`int64 -8`	`-8L`
`uint64`	Converts data to a `unit64`	`uint64 "0xFF"`	`255UL`
`float`	Converts data to a `float`	`float 3.1415M`	`3.1415`
`float32`	Converts data to a `float32`	`float32 8y`	`8.0f`
`decimal`	Converts data to a `decimal`	`decimal 1.23`	`1.23M`



Although these conversion routines accept strings, they parse strings using the underlying `System.Convert` family of methods, meaning that they throw `System.FormatException` exceptions for invalid inputs.

BigInteger

If you are dealing with data larger than 2^{64}, F# has the `bigint` type for representing arbitrarily large integers. (`bigint` type is simply an alias for the `System.Numerics.BigInteger` type.)

`bigint` is integrated into the F# language, and uses the `I` suffix for literals. Example 2-1 defines data storage sizes as `bigints`.

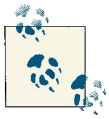
Example 2-1. The BigInt type for representing large integers

```
> open System.Numerics

// Data storage units
let megabyte  = 1024I    * 1024I
let gigabyte  = megabyte * 1024I
```

```
let terabyte  = gigabyte * 1024I
let petabyte  = terabyte * 1024I
let exabyte   = petabyte * 1024I
let zettabyte = exabyte  * 1024I;;

val megabyte : BigInteger = 1048576
val gigabyte : BigInteger = 1073741824
val terabyte : BigInteger = 1099511627776
val petabyte : BigInteger = 1125899906842624
val exabyte : BigInteger = 1152921504606846976
val zettabyte : BigInteger = 118059162071741130342 4
```



Although `bigint` is heavily optimized for performance, it is much slower than using the primitive integer data types.

Bitwise Operations

Primitive integer types support bitwise operators for manipulating values at a binary level. Bitwise operators are typically used when reading and writing binary data from files. See Table 2-5.

Table 2-5. Bitwise operators

Operator	Description	Example	Result
`&&&`	And	`0b1111 &&& 0b0011`	`0b0011`
`|||`	Or	`0xFF00 ||| 0x00FF`	`0xFFFF`
`^^^`	Exclusive Or	`0b0011 ^^^ 0b0101`	`0b0110`
`<<<`	Left Shift	`0b0001 <<< 3`	`0b1000`
`>>>`	Right Shift	`0b1000 >>> 3`	`0b0001`

Characters

The .NET platform is based on Unicode, so most text is represented using 2-byte UTF-16 characters. To define a character value, you can put any Unicode character in single quotes. Characters can also be specified using a Unicode hexadecimal character code.

The following snippet defines a list of vowel characters and prints the result of defining a character using a hexadecimal value:

```
> let vowels = ['a'; 'e'; 'i'; 'o'; 'u'];;

val vowels : char list = ['a'; 'e'; 'i'; 'o'; 'u']

> printfn "Hex u0061 = '%c'" '\u0061';;
Hex u0061 = 'a'
val it : unit = ()
```

To represent special control characters, you need to use an escape sequence, listed in Table 2-6. An escape sequence is a backslash followed by a special character.

Table 2-6. Character escape sequences

Character	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab

If you want to get the numeric representation of a .NET character's Unicode value, you can pass it to any of the conversion routines listed in Table 2-4. Alternatively, you can get the byte representation of a character literal by adding a `B` suffix:

```
> // Convert value of 'C' to an integer
int 'C';;
val it : int = 67
> // Convert value of 'C' to a byte
'C'B;;
val it : byte = 67uy
```

Strings

String literals are defined by enclosing a series of characters, which can span multiple lines, in double quotes. To access a character from within a string, use the indexer syntax, `.[ ]`, and pass in a zero-based character index:

```
> let password = "abracadabra";;

val password : string = "abracadabra"

> let multiline = "This string
takes up
multiple lines";;

val multiline : string = "This string
takes up
multiple lines"

> multiline.[0];;
val it : char = 'T'
> multiline.[1];;
val it : char = 'h'
```

```
> multiline.[2];;
val it : char = 'i'
> multiline.[3];;
val it : char = 's'
```

Alternatively, F# supports triple quotes, which allow you to put quotation mark characters in strings without the need for escaping them individually. For example:

```
let xmlFragment = """<Ship Name="Prometheus"></foo>"""
```

If you want to specify a long string, you can break it up across multiple lines using a single backslash \. If the last character on a line in a string literal is a backslash, the string will continue on the next line after removing all leading whitespace characters:

```
> let longString = "abc-\
                       def-\
              ghi";;

val longString : string = "abc-def-ghi"
```

You can use the escape sequence characters (such as `\t` or `\\`) within a string if you want, but this makes defining file paths and registry keys problematic. You can define a verbatim string using the `@` symbol, which takes the verbatim text between the quotation marks and does not encode any escape sequence characters:

```
> let normalString = "Normal.\n.\n.\t.\t.String";;

val normalString : string = "Normal.
.
.       .       .String

> let verbatimString = @"Verbatim.\n.\n.\t.\t.String";;

val verbatimString : string = "Verbatim.\n.\n.\t.\t.String"
```

Similar to adding the `B` suffix to a character to return its byte representation, adding `B` to the end of a string will return the string's characters in the form of a byte array (arrays are covered in Chapter 4):

```
> let hello = "Hello"B;;

val hello : byte [] = [|72uy; 101uy; 108uy; 108uy; 111uy|]
```

Boolean Values

For dealing with values that can only be `true` or `false`, F# has the `bool` type (`System.Boolean`) as well as standard Boolean operators listed in Table 2-7.

Table 2-7. Boolean operators

Operator	Description	Example	Result
`&&`	And	`true && false`	`false`
`||`	Or	`true || false`	`true`
`not`	Not	`not false`	`true`

Example 2-2 builds truth tables for Boolean functions and prints them. It defines a function called `printTruthTable` that takes a function named `f` as a parameter. That function is called for each cell in the truth table and its result is printed. Later, the operators `&&` and `||` are passed to the `printTruthTable` function.

Example 2-2. Printing truth tables

```
> // Print the truth table for the given function
let printTruthTable f =
    printfn "       |true   | false |"
    printfn "       +-------+-------+"
    printfn " true  | %5b | %5b |" (f true  true) (f true  false)
    printfn " false | %5b | %5b |" (f false true) (f false false)
    printfn "       +-------+-------+"
    printfn ""
    ();;

val printTruthTable : (bool -> bool -> bool) -> unit

> printTruthTable (&&);;
       |true   | false |
       +-------+-------+
 true  |  true | false |
 false | false | false |
       +-------+-------+

val it : unit = ()
> printTruthTable (||);;
       |true   | false |
       +-------+-------+
 true  |  true |  true |
 false |  true | false |
       +-------+-------+

val it : unit = ()
```