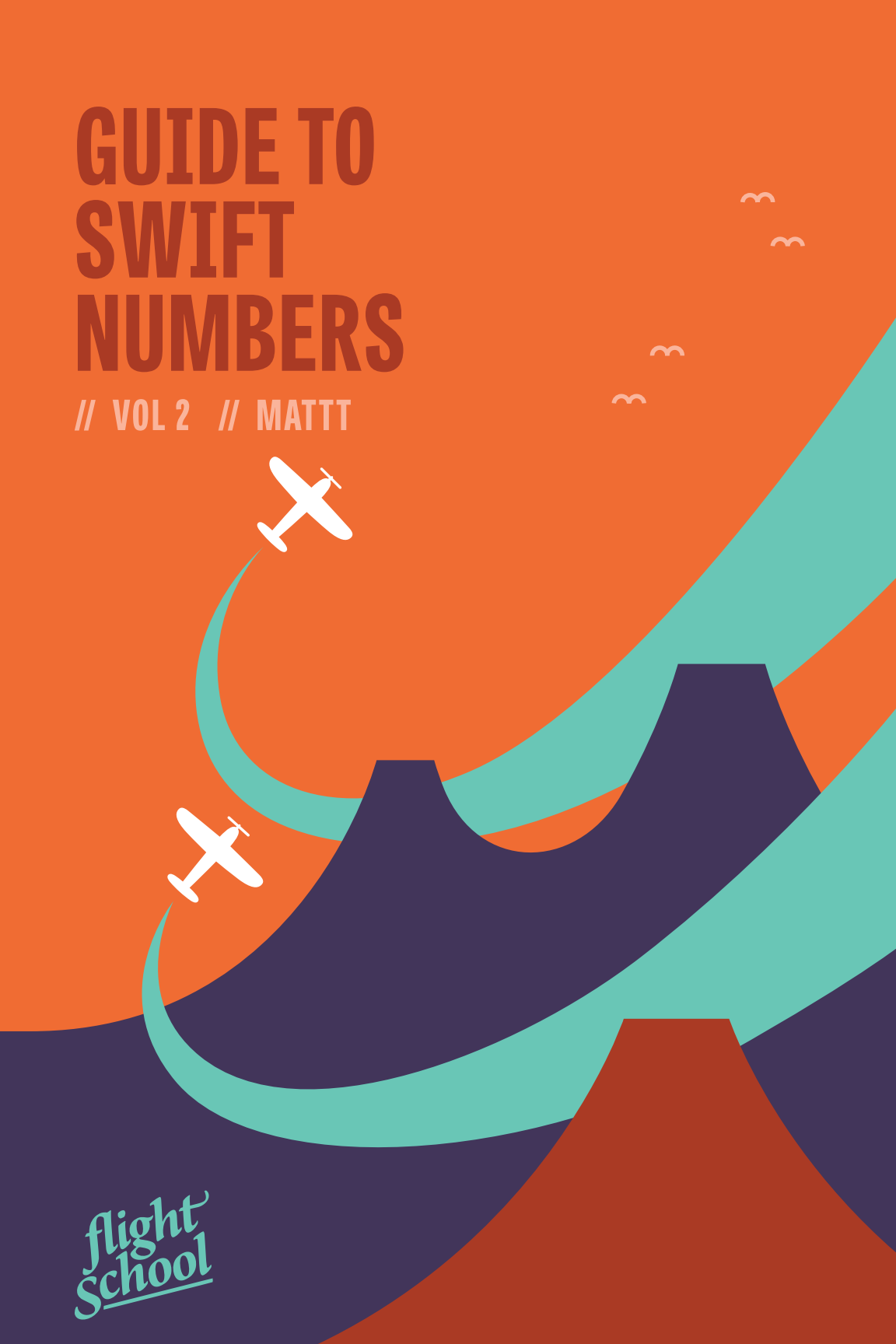
GUIDE TO SWIFT NUMBERS

// VOL 2 // MATTT



flight School

Flight School
An imprint of Read Evaluate Press, LLC
Portland, Oregon
https://readeval.press

Publisher	Mattt Zmuda
Illustrator	Lauren Mendez
Cover Designer	Mette Hornung Rankin
Editor	Morgan Tarling

ISBN

Printed in IBM Plex, designed by Mike Abbink at IBM in collaboration with Bold Monday.
Additional glyphs provided by Google Noto Sans.
Twemoji graphics made by Twitter and other contributors.
Cover type set in Tablet Gothic, designed by José Scaglione & Veronika Burian.

Contents

Introduction to Numbers 1
Number Formatting 36
Currencies and Monetary Amounts 61
Units and Measurements 82
Using Playgrounds as an Interactive Calculator 105


Flight School
Swift
UInt64
Unsigned Integers
UInt16
UInt32

Chapter 1:

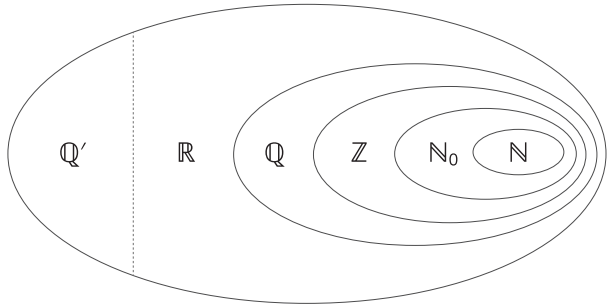
Introduction to Numbers

Numbers are mathematical objects that you can use to count and measure things.

When you use numbers for counting, like 1, 2, or 3, you're using what are called *natural numbers* ($\mathbb{N}$). If you include 0, we call them *whole numbers* ($\mathbb{N}_0$). And if you include negative numbers as well, you get the set of *integers* ($\mathbb{Z}$).

When you use numbers for measuring, they're often expressed as a ratio of integers, such as ½, ⅗, or ⅝. These are called *rational numbers* ($\mathbb{Q}$). Rationals include integers (and therefore natural numbers) because they can be expressed as a ratio to 1. Numbers that can't be expressed by a ratio of integers, like $\sqrt{2}$, φ, or π, are *irrational* ($\mathbb{Q}'$). We call the union of rational and irrational numbers *real numbers* ($\mathbb{R}$).

The relationship between these sets of numbers can be represented by a Euler diagram:



Numbers and Numerals

Numbers exist independently of their representation.

Perhaps the simplest and indeed one of the oldest ways to represent numbers is to use a *unary system*, like tallying. | is one, || is two, ||| is three.[1]

Unary notation has its drawbacks. For one, it takes a lot of space to represent large numbers. And humans are only able to perceive between 5 and 9 objects at once without subdividing.[2]

So instead of a (literally) 1-to-1 correspondence with quantity, you can instead assign numerical values to symbols, or *numerals*.

You're likely most familiar with writing numbers according to a *positional numeral system* — particularly using the Hindu–Arabic numeral system, in which numbers are represented by a sequence of ten distinct numerals, or *digits*.[3]

In Latin script, the glyphs from zero to nine are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In Arabic script, they're ٠, ١, ٢, ٣, ٤, ٥, ٦, ٧, ٨, and ٩.

A number consisting of a single digit expresses that digit's value. For example, 6 represents the number six.

To express larger numbers, you can arrange multiple digits right to left in ascending orders of magnitude. For example, the number 412 has 2 in the ones place, 1 in the tens place, and 4 in the hundreds place; four hundreds and one ten and two ones, or four hundred and twelve.

Number Bases

Another way of saying "ones place", "tens place", and "hundreds place" is to say the "10^0 place", "10^1 place", and "10^2 place". Number

---

1. You can see the influence of tallying in the first three characters of Brahmi numerals (𑁒 𑁓 𑁔) and Simplified Chinese characters (一 二 三).
2. That is, if you look at twelve apples, you'll perceive them as two groups of six apples or four groups of three apples. George A. Miller writes about this in his 1956 publication, *"The Magical Number Seven, Plus or Minus Two"*.
3. From the Latin *digiti*, meaning fingers, because most humans have ten fingers.

each position, starting with the rightmost at zero, and have the digit at that position represent its number value multiplied by ten to the power of that position.

Decimal

Digit	4	1	2
Place	10^2	10^1	10^0
Value	400	10	2

$$412 = 4 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 = 400 + 10 + 2$$

A *decimal* number system is base-10 because each place represents a different exponent of ten.

But you can use numbers other than 10 as a base.

A *binary* number is base-2 because each place represents a different exponent of two. As such, each digit goes up to, but not including, 2 (so, 0 or 1).

For example, the decimal number 9 is written in binary as 1001:

$$\begin{aligned} 1001_2 &= (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 2^3 + 0 + 0 + 2^0 \\ &= 8 + 0 + 0 + 1 \\ &= 9 \end{aligned}$$

Decimal (base 10)	Binary (base 2)
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Note: Understanding binary digits allows you to do more with your *own* digits (your fingers, that is). The conventional way of counting fingers is unary; most people can only count up to 10 on two hands. If you instead use your fingers as binary digits, you can extend your range significantly — up to 31 on five fingers and 1023 on ten!

Binary Numbers and Digital Systems

What makes binary numbers so useful is that they conveniently map onto a variety of physical quantities — whether it's the high or low voltage of a CPU transistor or the charge in a DRAM capacitor or pulses of light flowing through a fiber optic cable or the magnetic polarity of a hard drive region.

This is what it means when you hear something like "computers are made up of 1's and 0's".

Those 1's and 0's are known as *bits*.

A single bit can represent 2 states: `1` or `0`.

A pair of bits can represent 4 states: `11`, `10`, `01`, or `00`.

Three bits can represent 8 states, four bits can represent 16 states, and each additional bit doubles the number of possible states.

A *byte* is a group of 8 bits. A single byte can represent as many as 2^8, or 256 possible values. Bytes are the fundamental unit for computers (bits are too small on their own to be particularly useful).

Tip: There's no fundamental reason why a byte is 8 bits; it's just what computers eventually standardized on. Historically, a byte was simply the number of bits needed to represent a character (hence the `char` type in C), and there are examples of computers that used 5, 6, and as many as 48 in the past.

Even though we often express bits and bytes using binary notation, it's important to remember that they don't have any inherent meaning. The same combination of 32 bits can be interpreted to be an integer or a floating-point number or a string, or any number of other things. It all depends on how a program chooses to understand the data.

Swift Built-In Number Types

The Swift standard library provides standard types for signed integers, unsigned integers, and floating-point numbers.

Signed Integers

Signed integers are used to represent positive and negative integers.

`Int8`, `Int16`, `Int32`, and `Int64` are signed integer types sized 8, 16, 32, and 64 bits, respectively.

`Int` is a platform-dependent signed integer type. On 64-bit platforms like macOS and iOS, `Int` is the same size as `Int64`. On 32-bit platforms, `Int` is the same size as `Int32`. In other words, `Int` takes on the native size of the platform.

You use `Int` to store integers unless your particular use case calls for an explicitly-sized type.

Unsigned Integers

Unsigned integers are used to represent positive integers. Unlike signed integers, unsigned integers can't represent negative numbers.

`UInt8`, `UInt16`, `UInt32`, and `UInt64` are unsigned integer types sized 8, 16, 32, and 64 bits, respectively.

`UInt` is a platform-dependent unsigned integer type. On 64-bit platforms like macOS and iOS, `UInt` is the same size as `UInt64`. On 32-bit platforms, `UInt` is the same size as `UInt32`.

You primarily use unsigned integer types to represent bit patterns and for interoperability with lower-level APIs. In most other circumstances you need to store an integer value, `Int` is preferred — even if the numbers to be stored are nonnegative.

Floating-Points

Floating-point numbers are used to represent *real numbers*, or numbers with a fractional component.

`Float` is a single-precision floating-point number type.
`Double` is a double-precision floating-point number type.
`Float80` is an extended double-precision floating-point number type.[4]

`Float` is typealiased to `Float32` and `Double` is typealiased to `Float64`.

You use `Double` to store floating-point numbers unless another type would be more appropriate for your use case.

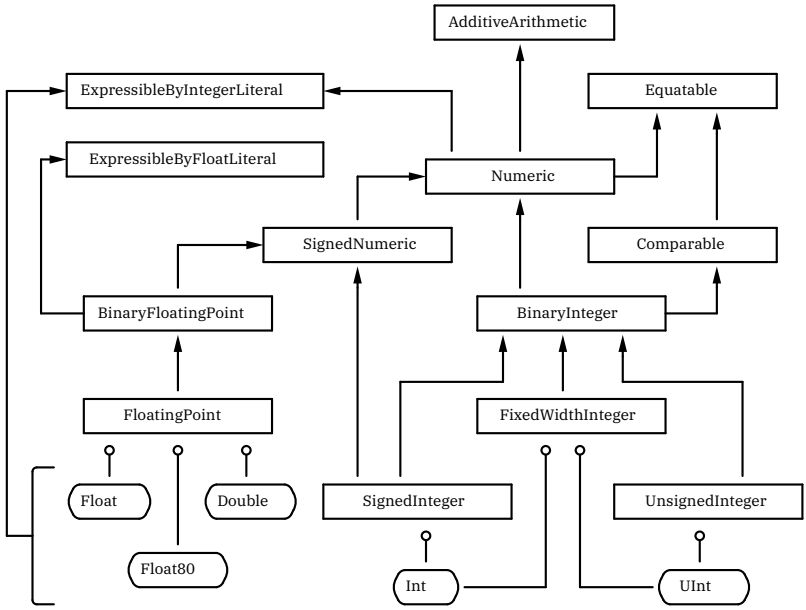
---

4. The terms *single*, *double*, and *extended* refer to the range and precision of numbers that those types can represent. They come from the original IEEE 754 specification for floating-point numbers, which is discussed later on.

Swift Number Protocols

As of Swift 4, the number system has been overhauled to adopt a protocol-oriented design.[5] Not only does that simplify the way you work with integer and floating-point numbers, but it allows anyone to define their own numeric types that look and feel like they're built-in.

Here's a diagram of how the built-in types and number protocols relate to one another:



Organizing types and responsibilities in this way provides more flexibility when extending functionality to existing types, and allows developers to more easily implement their own number types (as we'll discuss in later chapters).

---

5. For more information, check out the relevant Swift Evolution feature proposals:
   SE-0104: Protocol-oriented integers,
   SE-0113: Add integral rounding functions to FloatingPoint
   SE-0067: Enhanced Floating Point Protocols

We've already introduced the concrete types along the bottom. So let's get an understanding of how everything fits together by working our way up the type hierarchy.

...but before we do, let's talk about literals:

ExpressibleByIntegerLiteral and ExpressibleByFloatLiteral

A *literal* is a representation of a value in code, such as a number, string, array, or dictionary.

```
72              // Integer literal
1.618           // Floating-point literal
"hello"         // String literal
[2, 3, 5]       // Array literal
["b": "bravo"]  // Dictionary literal
```

Integer literals are inferred to be `Int` values and floating-point literals are inferred to be `Double` values unless an explicit type is specified.

```
let i = 10          // i is an Int
let u: UInt16 = 10  // u is a UInt16

let d = 1.23        // d is a Double
let f: Float = 1.23 // f is a Float
```

Any type can opt-in to be initializable from literal values by conforming to the corresponding protocol. For integer literals, it's the `ExpressibleByIntegerLiteral` protocol, and for floating-point literals, it's the `ExpressibleByFloatLiteral`.

As expected, each of the built-in integer types conforms to `ExpressibleByIntegerLiteral`. `Float`, `Double`, and `Float80` do too, in addition to conforming to `ExpressibleByFloatLiteral` (but you'll probably want to tack on a `.0` to any integer literals regardless, if only out of courtesy).

```
let f: Float = 2 // valid, but weird
```

Integer Literal Bases

Integer literals have a neat trick: in addition to their standard decimal form, they can be expressed in binary (base 2), octal (base 8), and hexadecimal (base 16).

- Binary integers begin with `0b`
- Octal integers begin with `0o`
- Decimal integers are unprefixed
- Hexadecimal integers begin with `0x`

```
0b11100001  // Binary
    0o341  // Octal
      225  // Decimal
     0xe1  // Hexadecimal
```

Binary

Digit	1	1	1	0	0	0	0	1
Place	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	128	64	32	0	0	0	0	1

Octal

Digit	3	4	1
Place	8^2	8^1	8^0
Value	192	32	1

Decimal

Digit	2	2	5
Place	10^2	10^1	10^0
Value	200	20	5

Hexadecimal

Digit	e	1
Place	16^1	16^0
Value	224	1

Binary literals are most useful for expressing *bit patterns*, which describe the layout of a sequence of 1's and 0's more clearly than

an equivalent decimal representation. For long bit patterns, use underscores (_) to group bits on byte boundaries (groups of 8), when applicable.

```
let bitPattern: UInt16 = 0b00110011_10101010
let bytes: [UInt8] = [0x44, 0x5A]
```

Hexadecimal literals are most often found in pairs, used to represent byte values. The hexadecimal digits for ten through fifteen can be interchangeably represented in lowercase (a-f) or uppercase (A-F) letters.

Tip: An old pastime of programmers is to spell out words using hexadecimal, like 0xBEEF, 0xEFFACE, and 0xDEC1DE. Tragically, the color represented by the RGB hex code #C0FFEE is a minty green color, rather than an evocatively satisfying mocha brown.

Octal literals are rarely used. The only time most programmers encounter them is when specifying permissions to the `chmod` system call.

Note: Another notable use for octal numbers is aeronautical. An air traffic controller may request that a pilot set their transponder to a particular code, consisting of four octal digits (for example, *"Cessna N12345, Squawk 4567"*). When the aircraft transponder receives an interrogation from the Secondary Surveillance Radar (SSR), it replies with that code and the pressure altitude. This information is used by ATC in conjunction with range and bearing information from their Primary Surveillance Radar (PSR) to distinguish between different aircraft in the vicinity.

FixedWidthInteger

Types that conform to the `FixedWidthInteger` protocol are integers that are stored in a constant number of bits. You can usually tell the number by how the type is named, but when in doubt, you can always call the required `bitWidth` type property.

```
UInt8.bitWidth // 8 (duh)
Int.bitWidth // 64 (oh, you fancy huh?)
```

Accessing the Minimum and Maximum Representable Values

Fixed-width integer types can represent at most 2^n distinct values, where *n* is equal to the number of bits. Typically, these values are contiguous — that is, without any gaps.

You can use the `min` and `max` type properties on fixed-width integer types to access the minimum and maximum representable values.

As discussed previously, both the `UInt8` and `Int8` types have a width of 8 bits and can, therefore, represent 2^8, or 256 distinct values. `UInt8` uses these values to represent the range 0 to 255, whereas `Int8` represents numbers between -128 to 127.

```
UInt8.min // 0
UInt.max // 255

Int8.min // -128
Int8.max // 127
```

Handling Overflows

What happens if you try to represent a number that exceeds the minimum or maximum value of a fixed-width integer type? You get an *overflow*.

Overflow can occur as the result of an arithmetic operation like addition or multiplication, or when initializing from a literal or different number type. For example, the maximum value of an `Int8` type is 127. If you try to add 1 to this number, overflow

occurs, because `Int8` can't represent the number 128. The same applies going the other direction: attempting to subtract 1 from the minimum value of an `Int8` type, -128, results in overflow, as well.

```
var i: Int8 = 127
i + 1 // error
```

By default, Swift checks for overflows when performing arithmetic operations and bit shifts, and crashes if overflow occurs.

Swift provides arithmetic operator variants prefixed with an ampersand (&) that ignore overflow and instead allow values to *wrap-around*. Consider the maximum `UInt8` value, `0b11111111` (255): adding one results in the number `0b100000000` (256). However, because `UInt8` only stores 8 bits, it discards the most significant bit (2^8), resulting in the number `0b00000000` (0).

```
let u = UInt8.max // 255
u &+ 1 // 0
```

There are also methods like `addingReportingOverflow(_:)`, which report whether overflow occurs by returning the result along with a Boolean in the tuple return value.

```
let u = UInt8.max // 255
u.addingReportingOverflow(1) // (0, true)
```

For operations that you know cannot overflow, you can opt out of overflow checks by calling the `unsafeAdding(_:)`, `unsafeSubtracting(_:)`, `unsafeMultiplied(by:)` or `unsafeDivided(by:)` method. Or you can cast your fate to the wind and opt out of overflow checking entirely, by compiling with the `-Ounchecked` optimization mode.

Bitwise Operations

`FixedWidthInteger` types can perform *bitwise operations*, which operate on the level of individual bits. These include left and right bit shifts (<<, >>), AND (&), OR (|), XOR (^), and NOT (~).

Bitwise operations are among the fastest and most efficient calculations that can be done on a CPU and are especially useful in low-level programming capacities. Most Swift applications fall a bit higher up the food chain, so you may not ever use them directly.[6]

Byte Order

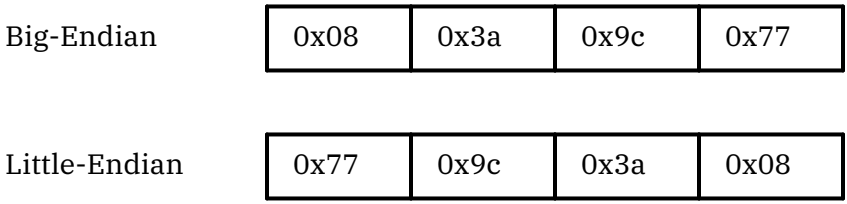
Just like how some writing systems are left-to-right, and others are right-to-left, computers can differ in how they send and store data for multi-byte integers.

> **Tip:** Bytes are often represented using hexadecimal integer literals because a byte conveniently fits into two base-16 digits.

For example, consider the number 0x083a9c77 (138,058,871 in decimal), which can be represented by a 4-byte unsigned integer (`UInt32`). The leftmost byte, 0x08 is known as the *most significant byte*, because it contributes the greatest magnitude of value. In contrast, the rightmost byte is the *least significant byte*, because it has the lowest magnitude of value.

When a number starts with its most significant byte, it's called *big-endian* format. Conversely, when a number starts with its least significant byte, it's called *little-endian* format.

Going back to our example, here's how the 4 bytes of the 32 bit unsigned integer would be stored in each format:



6. If you're interested in learning more, *The Swift Programming Language's* chapter on Advanced Operators is an excellent treatment of the subject.

In Swift, you can call the `bigEndian` and `littleEndian` properties on fixed-width integer types to get the representation in your desired byte order.

You don't typically have to worry about this unless you're building something that directly interfaces with a system that uses a different byte order. If you're using a high-level framework like Foundation, details like byte order should all be taken care of for you.

SignedInteger and UnsignedInteger

A *signed* integer type can represent both positive and negative numbers, whereas an *unsigned* integer type can only represent positive numbers.

These `SignedInteger` and `UnsignedInteger` protocols don't have many requirements and are provided instead as a way to distinguished signedness in generic constraints or extensions.[7]

Signed Integer Notations

It's clear how unsigned integer values correspond to numbers, but how do signed integers work? Rather than jumping right to the answer,[8] we can learn a lot by trying to find a solution for ourselves.

> **Tip:** For simplicity, we'll use 3-bit integers in the following examples.

One approach would be to reserve the most significant bit to represent the sign (0 for positive, 1 for negative) with the remaining bits representing the magnitude. For example, `0b001` is 1 and `0b101` is -1. This is known as *sign magnitude notation*.

---

7. The Swift standard library doesn't have an `Integer` type, but if it did, it'd be defined as the composition of these two protocols (`typealias Integer = SignedInteger & UnsignedInteger`).

8. tl;dr: Swift signed integers use two's complement notation.

Bit Pattern	Unsigned Value	Sign Magnitude Notation Value
0b000	0	0
0b001	1	1
0b010	2	2
0b011	3	3
0b100	4	-0
0b101	5	-1
0b110	6	-2
0b111	7	-3

There are a few problems with sign magnitude notation. First is that there are two values for 0: `0b000` and `0b100`. This redundancy is confusing and reduces the effective range of the type by 1. Second, arithmetic operations are complicated. Consider the sum of 1 and -2; adding the bits directly produces an incorrect result:

Decimal

Digit	4	1	2
Place	10^2	10^1	10^0
Value	400	10	2

-3? Yikes.

Alright, let's try a different idea: what if, instead of a sign bit, we start with an offset, or *bias*, and subtract that amount from the unsigned value? For a 3-bit number, set the offset to $-2^{(3-1)}$, or -4 (half the total range we want to represent).

This approach is called *excess notation*[9] :

9. Written as "excess(N)" where N is the number "in excess" subtracted from the unsigned value.

Bit Pattern	Unsigned Value	Excess(8) Notation Value
0b000	0	-4
0b001	1	-3
0b010	2	-2
0b011	3	-1
0b100	4	0
0b101	5	1
0b110	6	2
0b111	7	3

Excess notation solves the problem of double zeros, but arithmetic is still counter-intuitive. Again, consider the bitwise addition of 1 and -2:

$$\begin{array}{r} 101 \\ +010 \\ \hline 111 \end{array}$$

3? 🤢

Is there maybe a way to combine aspects of both solutions? What if we took the sign bit from sign magnitude notation and combined it with the bias in excess notation? Well, you end up with something called *two's complement*, which is, incidentally, how Swift and most other languages implement signed integers.

In two's complement notation, the most significant bit is the sign bit. When it's 0, the number is positive. When it's 1, the number is negative, and a bias is subtracted.

The easiest way to get a handle on it is to see everything in a correspondence table.

Bit Pattern	Unsigned Value	Two's Complement Notation Value
0b000	0	0
0b001	1	1
0b010	2	2
0b011	3	3
0b100	4	-4
0b101	5	-3
0b110	6	-2
0b111	7	-1

So far, so good. Let's see how arithmetic works in practice, once again by summing 1 and -2 in bitwise fashion:

$$\begin{array}{r} 001 \\ +110 \\ \hline 111 \end{array}$$

-1. *Yowza yowza bo-bowza!* Arithmetic works the same for signed two's complement as for unsigned integers, and the leading sign bit makes it easy to tell whether a number is positive or negative. The unsigned 0 corresponds to signed zero, and we avoid any positive/negative zero confusion. It's no wonder why two's complement is the most common way to implement signed integers in software.

BinaryInteger

Let's say you have a number stored in a `UInt64` variable that you want to pass to a method that takes a `UInt32` parameter. Or maybe you have a signed integer and want an unsigned integer (or vice-versa). Whenever you convert a number to a different type, there's a possibility that the number cannot be represented by that new type.

Binary integer values can be converted and cast to any types conforming to the `BinaryInteger` and `BinaryFloatingPoint` protocols according to the following strategies:

Range-Checked Conversion

If you're certain that an integer value can be represented by the new type, use the default `init(_:)` initializer to perform *range-checked conversion*. Passing a value that falls outside the representable range of the initializing type, results in an error.

```
let b = 747
let int16 = Int16(b) // success

let int8 = Int8(b) // error
```

Tip: You can also use the `numericCast(_:)` method to perform range-checked conversion if the destination type can be inferred.

Exact Conversion

If crashing is too dramatic a response to a failed conversion for your tastes, you can instead try *exact conversion* using the failable initializer `init?(exactly:)`. If the passed value cannot be represented, the initializer returns `nil`.

```
let p = 57
UInt(exactly: p) // UInt with value 57

let n = -33
UInt(exactly: n) // nil
```

You can also pass a floating-point number to this initializer to get an integer value when it is both representable and has no fractional component.[10]

```
Int8(exactly: 123.0) // 100
Int8(exactly: 1234.0) // nil (out-of-range for Int8)
Int8(exactly: 123.123) // nil (has fractional component)
```

10. Technically, the `init?(exactly:)` initializer required by `BinaryInteger` is for floating-point numbers only. The overloads that take integer types are convenience initializers required by the `SignedInteger` and `UnsignedInteger` protocols.

Clamping Conversion

Another option for handling out-of-range values is to take the closest representable value — that is, `max` for values larger than the upper bounds and `min` for negative values that exceed the lower bounds. This is called *clamping conversion* and is provided by the `init(clamping:)` initializer.

```
Int8(clamping: 0xA320) // 127
UInt8(clamping: -747) // 0
```

Bit Pattern Conversion

The final strategy is *bit pattern conversion*, which uses the `init(truncatingIfNeeded:)` initializer to truncate or expand the binary representation of the passed value to the size of the destination type.

```
let u: UInt8 = 0b11110000 // 240
Int8(truncatingIfNeeded: u) // -16 (0b11110000)
UInt16(truncatingIfNeeded: u) // 240 (0b00000000_11110000)
```

That covers all the protocols specific to integer types. Next up, we have floating-point numbers.

Unfortunately, there's no getting around it: floating-point math is *hard*.

Floating-point math is hard for profound reasons. Because what it attempts to model are real numbers, which are *uncountably infinite*. That is, given any two real numbers, there are an infinite number of real numbers that fall between them. Infinity as a concept is mind-blowing enough as it is. But then you learn that, in fact, there's more than one infinity, and the infinity you're most familiar with ($|\mathbb{N}| = \aleph_0$) is actually pretty minuscule compared to other infinities. It's all quite overwhelming.

But floating-point math is hard for stupid reasons, too. For example, simple numbers like 0.1 can't be represented exactly in binary, which leads to unexpected results, like:

```
0.1 + 0.2 == 0.3 // false (!)
```

More on that later.

Anyway, just because something is hard doesn't mean that we can't begin to understand it. After reading through the rest of the chapter, you should come away with a working understanding of floating-point numbers, and know how to avoid the most common issues a programmer faces when using them.

FloatingPoint

Types conforming to the `FloatingPoint` protocol are implemented according to the IEEE 754 specification, which defines the requirements for working with floating-point numbers in hardware and software.

All floating-point numbers can be described in terms of four components:

- A *sign*, positive or negative
- A *radix*, or base for the representation of floating-point numbers
- A *biased exponent*, such that the sum of the exponent and a constant (bias) produces a nonnegative range
- A *significand*,[11] containing the significant digits

The general form for a floating-point value can be written as:

$$(-1)^{\text{sign}} \times \text{significand} \times \text{radix}^{\text{exponent}}$$

11. Sometimes referred to as a *mantissa*, though this term is discouraged.

This is quite similar to scientific notation in that it allows numbers small and large to be represented to a great degree of precision.

A floating-point number is *normalized* if the significand is between 1 and the radix.[12] For example, the number 12.34×10^4 isn't normalized because the significand, 12, is greater than its radix, 10; the normalized form of this number is 1.234×10^5.

Normalized numbers are faster for computers to operate on, and maximizes the number of significant digits represented.

Special Floating-Point Values

The IEEE 754 specification defines several special values, including positive and negative zero, positive and negative infinity, and NaN ("not a number") values.

Positive and Negative Zero

As we learned by looking at sign magnitude notation for signed integers, the use of a sign bit causes positive and negative zero values to exist. This is the case for IEEE 754 floating-point numbers.

Swift built-in floating-point types implement equality in such a way that doesn't distinguish between the two varieties of zero. This equality check is equivalent to calling the `isZero` property.

```
-0.0 == 0.0 // true
0.0.isZero // true
-0.0.isZero // true
```

Positive and Negative Infinity

If a floating-point number has a magnitude that would exceed the representable range, it's considered to be infinity.

---

12. If the significand is outside this range, you can arrive at normal form by successively multiplying or dividing by the radix (thereby changing the exponent) until the number is within this range.

You can determine whether a number is finite or infinite using the corresponding `isFinite` and `isInfinite` properties.

```
(0.0).isFinite // true
Double.infinity.isFinite // false
```

If you need to distinguish between positive and negative infinity, you can do so by explicitly checking the `floatingPointClass` property.

```
(-Double.infinity).floatingPointClass == .negativeInfinity
```

The rules for operating with infinity can be complex, but, in general, remain true to real number arithmetic as much as possible. For example, adding or subtracting finite values to infinite values results in infinity, as does multiplying or dividing by a (nonzero) finite value.

NaN (“not a number”)

NaN values occur when you perform an invalid operation on a floating-point number, such as dividing by zero or taking the square root of a negative number.

The result of most arithmetic operations that involve an NaN is NaN. NaN isn’t equal to any floating-point value, including itself. To check whether a result is “not a number”, use the `isNaN` property.

```
Double.nan == Double.nan // false
Double.nan.isNaN // true
```

IEEE 754 distinguishes between “signaling” and “quiet” NaN values, the former raising an error when involved in an arithmetic operation. Swift doesn’t expose these errors in its standard library, but you can test floating-point environment state using the Darwin APIs `feclearexcept(_:)` and `fetestexcept(_:)`.

```
import Darwin

feclearexcept(FE_INVALID)
Double.signalingNaN + 1
fetestexcept(FE_INVALID) == 0 // false, therefore Invalid
```

Tip: This approach can also be used to detect other floating-point errors, including overflow (`FE_OVERFLOW`), underflow (`FE_UNDERFLOW`), intermediate rounding (`FE_INEXACT`), and division by zero (`FE_DIVBYZERO`).

BinaryFloatingPoint

`BinaryFloatingPoint` values represent floating-point numbers whose radix, or base, is 2.

$$(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$$

Accessing Sign, Exponent, and Significand

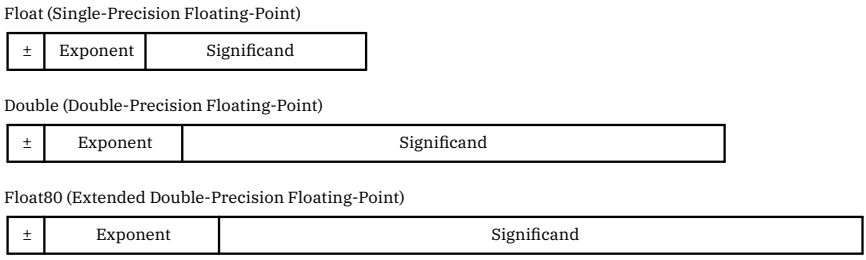
Binary floating-point numbers have a *range*, corresponding to the least and greatest representable values, as well as a *precision*, the number of significant digits that can be represented in a format (or, the number of digits to which a result is rounded). The range is determined by its exponent and its precision by its significand.

You can get the number of bits in the exponent and significand for a `BinaryFloatingPoint` type by accessing its `exponentBitCount` and `significandBitCount` properties.

```
Float.exponentBitCount // 8
Float.significandBitCount // 23

Double.exponentBitCount // 11
Double.significandBitCount // 52

Float80.exponentBitCount // 15
Float80.significandBitCount // 63
```



For most applications, floating-point numbers are distinguished by their precision rather than their magnitude. Here are some general guidelines for how the significand sizes of the built-in `BinaryFloatingPoint` types correspond to decimal precision:

Type	Significand Bits	Decimal Digits
`Float`	23	~7
`Double`	52	~16
`Float80`	63	~19

Comparing Floating-Point Numbers

Let's revisit that claim from before: why doesn't `0.1 + 0.2 == 0.3`? Because that's not what's actually being added.[13]

```
let lhs = 0.1 // lhs: Double = 0.10000000000000001
let rhs = 0.2 // rhs: Double = 0.20000000000000001
let sum = lhs + rhs // sum: Double = 0.30000000000000004
```

A fraction may be represented by a single digit in one base, but have nonterminating, repeating digits in another base.[14] For example, the quantity ⅓ is 0.333... in decimal, but in *duodecimal* (base-12), it's 0.4.

Binary numbers have it pretty rough in this department because fractions terminate only if their denominator has 2 as its sole prime factor. As a result, a lot of numbers end up being approximations when represented in binary.

That's why it's crucial to express floating-point numbers in normal form: you want to squeeze as many digits into the significand in order to minimize error later on. Except at the edge of the representable range, any leading zeroes are wasted bits that could instead be represented by the exponent, for which a set number of bits are already allocated.

So how do you compare two floating-point numbers for approximate equality? You *could* compare the difference between them to a fixed amount, or *epsilon*. Unfortunately, this approach — quite literally — doesn't scale. A difference of, say, 0.00001 is small relative to the number 1, but enormous for the number 1E-100.

---

13. To be clear, floating-point literals in code are infinitely precise. Any loss in precision happens when the `ExpressibleByFloatLiteral` initializer, `init(floatLiteral:)`, is called and the value is created.

14. Never mind the irrational numbers, which have nonterminating, nonrepeating digits in *every* natural base.

Instead of getting caught up in the uncountably infinite nature of real numbers, let's remind ourselves that `Float` and `Double` values are finite, consisting of small, discrete steps.

The distance between representable numbers is called an *ULP*, or unit of least precision, and this distance grows as numbers get larger and shrinks as numbers get smaller. For everyday `Float` values that hover around 1.0, representable numbers are spaced out by one ten-millionth.[15] At the extreme ends, ULPs can be as small as 10^{-45} and as large as 10^{31}.

```
Float.ulpOfOne                          // 0.00000011920929
Float.greatestFiniteMagnitude.ulp       // 2.02824096E+31
Float.leastNormalMagnitude.ulp          // 1.40129846E-45
```

We might say that two floating-point numbers are *approximately equal* if they have equal or adjacent representations — that is, within one ULP of each other.

Doing this calculation in such a way that handles all the special floating-point values correctly can be quite difficult. But lucky for us, the Swift standard library already provides this functionality by way of the `nextUp` and `nextDown` properties in `FloatingPoint` types. Here's a simple implementation that defines a custom `==~` operator:

```
infix operator ==~ : ComparisonPrecedence
func ==~<T> (lhs: T, rhs: T) -> Bool where T: FloatingPoint  {
    return lhs == rhs || lhs.nextDown == rhs || lhs.nextUp ==
rhs
}
```

15. C defines a constant called `FLT_EPSILON`, which is the ULP for numbers between 1.0 and 2.0. Programmers often use this for floating-point comparison by mistake. The Swift standard library uses a more descriptive name, `ulpOfOne`, as a way to prevent API misuse.

Let's put this to the test:

```
0.1 + 0.2 == 0.3 // false
0.1 + 0.2 ==~ 0.3 // true
```

As a general solution, this works pretty well — it even handles special values like infinity and NaN without so much as a hiccup.

Depending on your use case, you may additionally define an absolute margin to compensate for large relative errors at extreme values, and/or specify an acceptable difference in ULPs:

```
extension FloatingPoint {
    func isApproximatelyEqual(to other: Self,
                              within margin: Self?,
                              maximumULPs ulps: Int = 1) ->
Bool {
        precondition(margin?.sign != .minus && ulps > 0)

        guard self != other else {
            return true
        }

        let difference = abs(self - other)
        if let margin = margin, difference > margin {
            return false
        } else {
            return difference <= (self.ulp * Self(ulps))
        }
    }
}
```

Ultimately, it's up to you to evaluate what comparisons are appropriate for your application.

Converting from Decimal to Binary Floating-Point

The best way to demystify how floating-point numbers work is to go through the exercise of manually converting from a decimal number to a `Float` in IEEE 754 normal form.

First, pick a number. Let's use 0.0765, the imperial unit constant for air density used in aerodynamic calculations.[16]

Step 1: Check the Sign

If the number is positive, the sign bit is 0. If it's negative, the sign bit is 1.

0.0765 is positive, so the sign bit is 0.

Step 2: Factor the Significand

A binary floating-point number is in normal form if its significand is in the range 1 to 2. We can determine this by dividing by increasing powers of two until we get a result in that range:

$$0.0765 / 2^{-1} = 0.153$$
$$0.0765 / 2^{-2} = 0.306$$
$$0.0765 / 2^{-3} = 0.612$$
$$0.0765 / 2^{-4} = \mathbf{1.224}$$

The bias for a single-precision floating-point number is 127, or half of the unsigned value representable by the 8 exponent bits. Adding the power of -4 determined in the previous step, we get an exponent of 123 (`0b01111011`).

Step 3: Express the Significand Fraction in Binary

The significand determined in Step 2 is 1.224. The unit value 1 is implied, so our task is to express the significand fraction (0.224) in binary. To do this, we successively multiply the fractional digits

16. $\rho = 1.225$ kg/m³ (0.0765 lb/ft³) at sea level and 15°C.
Flight School usually is all about SI units; we're using feet and pounds here because 0.0765 makes for a more interesting conversion than 1.225.

by 2 and take the unit digit (1 or 0) until the sequence terminates, repeats, or we run out of digits (for `Float`, we get up to 23 binary significand digits):

$$
\begin{aligned}
0.224 \times 2 &= \mathbf{0}.448 \\
0.448 \times 2 &= \mathbf{0}.896 \\
0.896 \times 2 &= \mathbf{1}.792 \\
0.792 \times 2 &= \mathbf{1}.584 \\
0.584 \times 2 &= \mathbf{1}.168 \\
0.168 \times 2 &= \mathbf{0}.336 \\
0.336 \times 2 &= \mathbf{0}.672 \\
0.672 \times 2 &= \mathbf{1}.344 \\
0.344 \times 2 &= \mathbf{0}.688 \\
0.688 \times 2 &= \mathbf{1}.376 \\
0.376 \times 2 &= \mathbf{0}.752 \\
0.752 \times 2 &= \mathbf{1}.504 \\
0.504 \times 2 &= \mathbf{1}.008 \\
0.008 \times 2 &= \mathbf{0}.016 \\
0.016 \times 2 &= \mathbf{0}.032 \\
0.032 \times 2 &= \mathbf{0}.064 \\
0.064 \times 2 &= \mathbf{0}.128 \\
0.128 \times 2 &= \mathbf{0}.256 \\
0.256 \times 2 &= \mathbf{0}.512 \\
0.512 \times 2 &= \mathbf{1}.024 \\
0.024 \times 2 &= \mathbf{0}.048 \\
0.048 \times 2 &= \mathbf{0}.096 \\
0.096 \times 2 &= \mathbf{0}.192 \\
&\dots
\end{aligned}
$$

Step 4: Put Everything Together

Combining the sign bit (`0b0`), the exponent (`0b01111011`), and the significand (`0b00111001010110000001000`), we arrive at the final `Float` bit pattern:

±	Exponent	Significand
0	01111011	00111001010110000001000

We can verify this programmatically:

```
let bitPattern: UInt32 = 0b00111101100111001010110000001000
let string = String(bitPattern, radix: 2, uppercase: false)
let paddedString = String(repeating: "0", count: 32 -
string.count) + string
paddedString == "00111101100111001010110000001000" // true
Float(bitPattern: bitPattern) == Float(0.0765) // true
```

Converting from Binary Floating-Point to Decimal

Going the other way is also an important exercise. Let's complete the loop by deciphering a binary floating-point number, say `0b11000001000111001110100000001010`.

Step 0: Check for Special Cases

Before we do anything else, we should check to make sure our bit pattern corresponds to a number, not a special value like positive infinity or NaN.

Glossing over the finer details of how to do that for expediency, let's assume that what we have is an honest-to-goodness number. (*Scout's honor!*)

Step 1: Separate the Bits Into Components

First, we divide the bit pattern into three groups:

±	Exponent	Significand
1	10000010	00111001110100000001010

The first bit shows us the sign of the number, the next 8 bits give us the exponent, and the last 23 bits give us the fraction.

Step 2: Read the Sign Bit

The number is positive if the sign bit is 0 and negative if the sign bit is 1.

In this case, the sign bit is 1, so the number is negative.

Step 3: Determine the Exponent

`0b10000010` is equal to 130. If we subtract the `Float` exponent bias of 127, we get 3.

Step 4: Convert the Significand Bits into a Fraction

The significand is the fractional component of the floating-point number. It's what comes after the radix point[17] when written in binary. Reading left to right, each binary digit corresponds to decreasing, negative powers of two. To get its value, we multiply each binary digit by the corresponding power of 2 and sum the results:

$$= 00111001110100000001010_2$$

$$= (0 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + \ldots$$

$$= 0 + 0 + \frac{1}{8} + \frac{1}{16} + \ldots$$

$$= 0.22583127$$

Adding one to the fraction, we get a significand value of 1.22583127.

17. A *radix point* is the general term for the divider between integral and fractional number components. A decimal point is an example of a radix point.

Step 5: Put Everything Together

$$(-1)^{\text{sign}} \times \text{significand} \times \text{radix}^{\text{exponent}}$$

$$(-1)^{1} \times 1.22583127 \times 2^{3}$$

$$-1.22583127 \times 8$$

$$-9.80665016$$

Our mystery IEEE 754 number is — in actuality — the decimal floating-point value -9.80665016.

We can check out work in code:

```
let bitPattern: UInt32 = 0b11000001000111001110100000001010
Float(bitPattern: bitPattern) // -9.80665016
```

The original value used to produce this was `-9.80665`, the standard gravity acceleration (in m/s^2), so this result isn't too far off.

AdditiveArithmetic, Numeric, and SignedNumeric

Types conforming to the `Numeric` protocol support the elementary arithmetic operations (`+`, `-`, `*`) as well as their mutating counterparts (`+=`, `-=`, `*=`). In Swift 5, the addition and subtraction operators have been factored out of the `Numeric` protocol to support generic algorithms on both scalars and vectors.[18]

If a type conforms to `SignedNumeric`, it can represent either a positive or negative quantity. Both `Numeric` and `SignedNumeric` inherit the `ExpressibleByIntegerLiteral` and `Equatable` protocols.

`AdditiveArithmetic`, `Numeric` and `SignedNumeric` serve as the primary extension point for custom number types that wrap a built-in number type.

---

18. For more information, see SE-0233: "Make Numeric Refine a new AdditiveArithmetic Protocol"

With a solid foundation of how numbers work on computers in general and in Swift specifically, we can take a look at different ways that numbers are used.

In Chapter 2, we'll look at how to format numbers in a locale-appropriate way for a global audience using `NumberFormatter`.

Chapter 3 discusses the correct way to work with monetary amounts and implements a custom `Money` type that takes advantage of several number protocols discussed in this chapter.

Chapter 4 and Chapter 5 demonstrate the power of Foundation's units and measurements APIs by showing how they can be used for dimensional analysis and engineering calculations.

Recap

- Numbers exist independently of their representation.
- A bit is a binary (base 2) number, and a byte is a collection of 8 bits.
- Unless you have a reason to do otherwise, use `Int` to represent whole numbers and `Double` to represent fractions.
- Swift uses protocols to represent numbers coherently in the type system.
- All number types can be expressed by integer literals; floating-point types can also be expressed by floating-point literals.
- Fixed-width integers store values in a set number of bits, and exceeding these bounds results in overflow. You can manipulate these bits directly using bitwise operations. Multi-byte values may exist in big- or little-endian format.
- Unsigned integers can represent only positive numbers. Signed integers can represent both positive and negative numbers using two's complement notation.
- Binary integer values can be converted and cast to different binary integer types using range-checked, exact, clamping, or bit pattern conversions.
- Floating-point numbers represent fractional numbers with a particular range and precision according to the IEEE 754 specification.
- Binary floating-point values approximate fractional numbers using binary components for sign, exponent, and significand.
- Numeric types support the elementary arithmetic operators `+`, `-`, and `*`.


N12,34

Chapter 2:
Number Formatting

Aircraft operating in the United States are required to display their registration N-Number. The guidelines for how these identification marks should appear are spelled out in Title 14, Part 45 of the Code of Federal Regulations (CFR), under "Identification and Registration Marking".[1]

Markings must meet the following criteria:

- ☑ **Height**: at least 12 inches
- ☑ **Width**: ⅔ of character height
- ☑ **Thickness**: solid, ⅙ thickness of character height
- ☑ **Spacing**: ≥ ¼ of the character width
- ☑ **Uniformity**: equal sizing on both sides of the aircraft

Of course, aircraft registration N-numbers are numbers in name only (they're really just alphanumeric strings). But their lesson about the importance of meeting expectations certainly applies to numbers, too.

It's not enough to toss a number into an interpolated string and call it a day. Although there's no threat of a federal agency shutting you down (maybe), your users probably won't be too impressed. This chapter will guide you through all of the unspoken rules and conventions for how to represent numbers to users around the world.

1. 14 CFR 45, courtesy of the US Government Publishing Office.

NumberFormatter

`NumberFormatter` (née `NSNumberFormatter`) is among the oldest APIs in Foundation, going back to the NeXT days when Mac OS X was but a twinkle in Scott Forstall's eye.

Like all `Formatter` subclasses, `NumberFormatter` is responsible for creating representations of objects — in this case, numbers — appropriate for the localization preferences of the user. In addition to any custom format you might specify, it has built-in styles for displaying the following:

- **Cardinals** (1 / one)
- **Ordinals** (1st)
- **Decimal** (1,234.56)
- **Scientific** (1.23456E3)
- **Percentages** (13%)
- **Currency** ($123.45 / USD1,234.57 / 1,234.57 US dollars)

Some of this functionality has been accumulated over the years, but `NumberFormatter` was pretty much always the one-stop shop we know it as today.

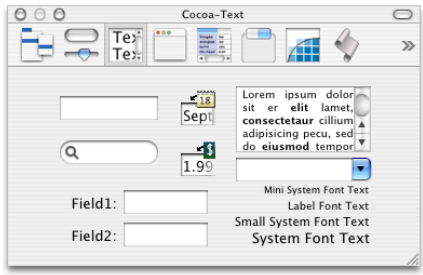
There are two reasons for this:

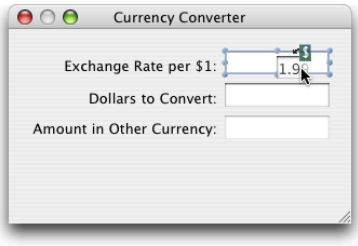
First, `NumberFormatter` is primarily a wrapper around ICU4C, the C interface to International Components for Unicode. If you look at the documentation for `UNumberFormat`, you'll notice a lot of shared terminology and functionality, in the same familiar "kitchen sink" form factor.

The second reason has to do with Interface Builder. Back when it was its own, separate app, developers would split their time evenly between IB the IDE.[2] In Interface Builder, you'd drag out UI elements, connect actions, outlets, and bindings to controllers, and

2. Apple's IDE was Project Builder until 2003 when it was replaced by Xcode. Xcode 4.0 unified Interface Builder and other tools into a single application.

then jump back to code for everything you couldn't do by mouse alone.[3] Whenever you had an `NSTextField` whose value was bound to an `NSNumber` value, you simply drag-and-drop'd an `NSNumberFormatter` on top of it and configured it as necessary in the inspector panel.





3. Or at least that was the idea. Next time you meet an old-school Cocoa developer, go ahead and ask them about those pre-iPhone days.

Formatting

To format a number, you create a `NumberFormatter` object, set its `numberStyle` to produce the desired output, make any other necessary configurations, and then pass the number to `string(for:)`.

```
let formatter = NumberFormatter()
formatter.numberStyle = .ordinal
formatter.string(for: 2) // 2nd
```

Alternatively, you can use the class method `localizedString(from :number:)`[4] to format a number using a particular style without creating a formatter. This approach can be convenient for one-off use cases, but doesn't allow for any customization.

for vs. from

`NumberFormatter` has two similarly-named methods that do pretty much the same thing:

```
func string(for obj: Any?) -> String?
func string(from number: NSNumber) -> String?
```

Why are there two, you ask?

In Objective-C, `NSFormatter` has the method `-stringForObject Value:`, which takes an object of type `id` (a pointer to any Objective-C object). The Swift version of this method is `string(for:)`.

In an effort to make the Objective-C API more explicitly-typed, `NSNumberFormatter` and other Foundation formatters also provide convenience methods like `-stringFromNumber:`, or `string(from:)` in Swift, that take an `NSNumber` instead of an `id`

4. The second parameter, `number:`, is conspicuously named, because the parameter takes a `Number Formatter.Style` value, rather than a number. Chock it up to an edge case from "The Great Renaming" of Swift 3.

argument . However, this extra type information in Objective-C is more of an impediment in Swift, as number types require an explicit downcast to `NSNumber`.

```
let number = 123.45 // Double
formatter.string(for: number)
formatter.string(from: number as NSNumber)
```

Which one do you use, then? Honestly, `string(for:)` works just fine. You'll probably tend towards that anyway, as a natural aversion to typing "`as NSNumber`".

Reusing Formatters

Creating and configuring formatters can be an expensive operation, so you want to avoid doing this within a tight loop. It's also a good idea to present information consistently in your app, which is easier to do when you aren't repeating formatter setup code across your codebase.

One strategy is to store a formatter instance as a property of a view controller. Whenever a number is formatted, such as when setting the text in a table view cell, you can use the shared formatter rather than creating a new one each time.

```
class ViewController: UIViewController {
    lazy var numberFormatter: NumberFormatter = {
        let numberFormatter = NumberFormatter()
        numberFormatter.numberStyle = .decimal
        return numberFormatter
    }()
}
```

Cardinal Numbers

Linguistically, *cardinal numbers* are used to denote quantity. A cardinal number can appear as words (one, two, three, four) or as numerals (1, 2, 3, 4).

By default, `NumberFormatter` represents numbers with cardinal numerals.[5] This corresponds to the none case in the `NumberFormatter.Style` enumeration. You can have a number formatter use cardinal words instead by setting its `numberStyle` property to `spellOut`.

Locale	Identifier	none	spellOut
U.S. English	en-US	4	four
Italian	it	4	quattro
Hindi	hi	४	चार
Simplified Chinese	zh-Hans	4	四

```
let formatter = NumberFormatter()
formatter.string(for: 4) // 4

formatter.numberStyle = .spellOut
formatter.string(for: 4) // four
```

Negative Numbers

Numbers less than zero are formatted differently to indicate that they're negative. At the time of writing, all of the available locales on Apple platforms denote a negative number using minus prefix.[6]

Locale	Identifier	none
U.S. English	en-US	-456
Swiss German	gsw	−456
Arabic	ar	٤٥٦-

5. As a result, `NumberFormatter` omits fractional components of numbers by default, which may be unexpected.

6. Even though Arabic text is read right-to-left, Arabic numerals are read the same as any other positional notations — with the caveat that the negative prefix appears before (i.e. to the right of) the numerals.

```
let formatter = NumberFormatter()
formatter.string(for: -456) // -456
```

The only real inconsistency among the available locales is whether they use a hyphen or a proper minus sign character.

- `U+002D - HYPHEN-MINUS`
- `U+2212 − MINUS SIGN`

If you're particularly bothered by this bit of typographic pedantry, you can coerce the world to do the right thing by explicitly setting a *proper* minus sign.[7]

```
let formatter = NumberFormatter()
formatter.negativePrefix = "\u{2212}" // U+2212 − MINUS SIGN
```

Non-Hindu-Arabic Numerals

People around the world primarily represent numbers using decimal positional notation with Hindu-Arabic numerals. However, some locales in the Middle-East, India and Southeast Asia use different numerals.

Script	Numerals
Hindu-Arabic	1234567890
Eastern Arabic	١٢٣٤٥٦٧٨٩٠
Western Arabic	۱۲۳۴۵۶۷۸۹۰
Burmese	၁၂၃၄၅၆၇၈၉၀
Bengali	১২৩৪৫৬৭৮৯০
Devanagari	१२३४५६७८९०
Dzongkhan	༡༢༣༤༥༦༧༨༩༠

7. Swift string literals support the full range of Unicode characters, but it's often helpful to use escaped Unicode scalars for code points whose representation could be confused with other characters.

```
let formatter = NumberFormatter()
formatter.locale = Locale(identifier: "ar-SA")
formatter.string(for: 1234567890) // ١٢٣٤٥٦٧٨٩٠
```

`NumberFormatter` automatically chooses the correct number system based on the user's localization preferences. The important takeaway here is that you shouldn't expect user input for numbers to always show up as ASCII digits (`0`-`9`).

Note: Why are they called Hindu-Arabic numerals when they're distinct from the Devanagari numerals used in Hindi as well as both Eastern and Western Arabic numerals? Persian and Arabic mathematicians in the 8th century C.E. called them "Hindu numerals" because it came from India. When they started to catch on in Europe in the 10th century C.E., mathematicians there called them "Arabic numerals" without being aware of their origin.

Chinese Numerals

Chinese and other languages in East Asia, have a traditional numbering system that's *multiplicative* rather than positional.[8] Place values like ten and one hundred are represented by distinct glyphs and preceded by a coefficient digit (1–9). For example, the number twelve thousand three hundred forty-five is represented as 一万二千三百四十五, or 一万 (1 × 10,000) + 二千 (2 × 1,000) + 三百 (3 × 100) + 四十 (4 × 10) + 五 (5).[9]

1	2	3	4	5	6	7	8	9	10	10^2	10^3	10^4	10^8
一	二	三	四	五	六	七	八	九	十	百	千	万	亿

8. The Unicode Common Locale Data Repository, or CLDR, distinguishes between default, native, and traditional numbering systems. A *default* system is the one used normally to represent numbers. A *native* system uses native digits, usually in the script of the locale's language, in decimal positional notation. A *traditional* system uses native digits and a different notation than decimal positional.

9. Chinese numbers are grouped into *myriads* (10,000) rather than thousands (1,000), as in Western numbering systems.

`NumberFormatter` produces traditional Chinese numerals for supported locales when its `numberStyle` is set to `spellOut`.

```
let formatter = NumberFormatter()
formatter.numberStyle = .spellOut
formatter.locale = Locale(identifier: "zh-Hans")
formatter.string(for: 12345) // 一万二千三百四十五
```

Ordinal Numbers

Related to cardinals, *ordinal numbers* are used to denote rank or position in a sequence. An ordinal number can be written out ("first", "second", "third", "fourth") or abbreviated using numerals ("1st", "2nd", "3rd", "4th").

`NumberFormatter` produces numeral ordinals when its `numberStyle` is set to `ordinal`.[10]

Locale	Identifier	.ordinal
U.S. English	en-US	1st
German	de	1.
French	fr	1er
Italian	it	1º
Hindi	hi	१ला
Simplified Chinese	zh-Hans	第1

```
let formatter = NumberFormatter()
formatter.numberStyle = .ordinal
formatter.string(for: 1) // 1st
```

10. There's currently no ordinal counterpart to the cardinal `spellOut` style.

Indo-European Grammatical Gender

Several languages in the Indo-European language family inflect ordinals according to grammatical gender.

In French, the numeral abbreviation of the word “first” depends on the grammatical gender of the word it describes. For masculine nouns, the suffix is *-er* (“le 1er jour”), and for feminine nouns, the suffix is *-re* (“la 1re nuit”). For all other ordinal numbers, the suffix is *-e*. `NumberFormatter` only uses the masculine form, *-er*.

```
let formatter = NumberFormatter()
formatter.numberStyle = .ordinal
formatter.locale = Locale(identifier: "fr-FR")
formatter.string(for: 1) // 1er
```

Other Romance languages have designated ordinal indicators that are appended to numerals:

- `U+00AA ª FEMININE ORDINAL INDICATOR`
- `U+00BA º MASCULINE ORDINAL INDICATOR`

In both Italian and Spanish, only the masculine indicator is used.

```
let formatter = NumberFormatter()
formatter.numberStyle = .ordinal
formatter.locale = Locale(identifier: "it-IT")
formatter.string(for: 1) // 1º
```

Warning: `NumberFormatter` doesn’t provide a way to specify gender, and always uses the masculine form. To represent ordinal numbers correctly, use `NSLocalizedString` instead.

Sino-Tibetan / Altaic Classifiers

Some languages in the Sino-Tibetan and Altaic language families, including Chinese and Japanese, have *nominal classifiers*, which specify how a number quantifies something. For example, the

Chinese phrase “one ticket”, 「一张票」 (yī zhāng piào), uses the classifier for flat objects, 张, whereas the phrase “one pen”, 「一支笔」 (yī zhī bǐ), uses the classifier for long, round objects, 支.

In Japanese, rank or order may be indicated by one of several classifiers — most commonly 第 (だい) or 番 (ばん).

`NumberFormatter` doesn’t provide a way to specify order classifiers, and always uses 第.

```
let formatter = NumberFormatter()
formatter.numberStyle = .ordinal
formatter.locale = Locale(identifier: "ja-JP")
formatter.string(for: 8) // 第8
```

Decimal Numbers

The `decimal` style is used to represent real numbers. `NumberFormatter` uses the fractional and group separators appropriate for the selected locale and has configuration options for the number of digits to show and what rounding behavior to use, if applicable.

Locale	Identifier	`.decimal`
U.S. English	en-US	1,234,567.89
French	fr	1 234 567,89

```
let formatter = NumberFormatter()
formatter.numberStyle = .decimal
formatter.string(for: 1234567.89) // 1,234,567.89
```

Fractional and Group Separators

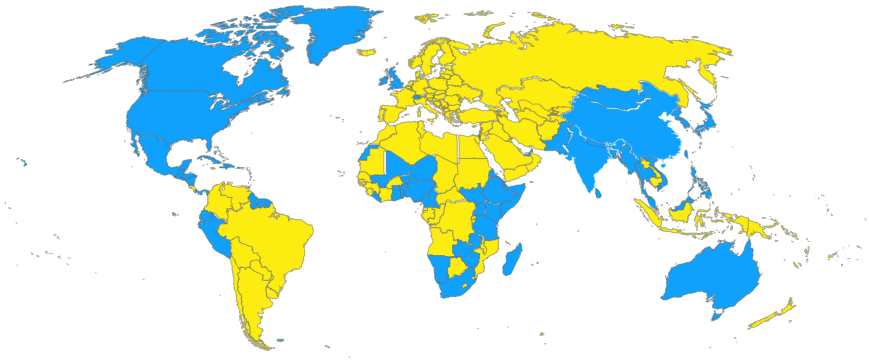
Historically English-speaking countries tend to use a period as a decimal point and a comma as a group separator for the thousands place. Most other countries instead use a comma as a decimal point and a space as a group separator instead.

```
let formatter = NumberFormatter()
formatter.numberStyle = .decimal

formatter.locale = Locale(identifier: "en-GB")
formatter.string(from: 1234567.89) // 1,234,567.89

formatter.locale = Locale(identifier: "fr-FR")
formatter.string(from: 1234567.89) // 1 234 567,89
```

There's a pretty even geographic split between the two, as you can see in the following map:



Countries that primarily use period as a decimal point are shaded blue. Countries that primarily use comma as a decimal point are shaded yellow. This data is based on the behavior of `NumberFormatter` at the time of writing.

Hindi locales are unique for grouping at the thousands place, but then every two digits afterward.

```
let formatter = NumberFormatter()
formatter.numberStyle = .decimal
formatter.locale = Locale(identifier: "hi-IN")
formatter.string(from: 1234567.89) // १२,३४,५६७.८९
```

Configuring Number Precision

There are three ways to specify how many digits are displayed in `NumberFormatter` representations of numbers:

- Enabling significant digits
- Setting maximum integer / fraction digits
- Using a custom format

A *significant digit* is any digit that indicates precision — this excludes any leading zeroes that indicate magnitude. Put another way: a digit significant if it appears in the significand of a number when it's expressed in scientific notation. For example, the digits 10203 are significant in the numbers 1,020,300 and 0.00010203.

You set `usesSignificantDigits` to `true` and specify maximum and minimum significant digits if you're representing numbers with precision. By default, `maximumSignificantDigits` is set to 6 and `minimumIntegerDigits` is set to 1.

```
let formatter = NumberFormatter()

formatter.usesSignificantDigits = true
formatter.maximumSignificantDigits = 2

formatter.string(from: 123) // 120
formatter.string(from: 123456) // 120000
formatter.string(from: 123.456) // 120
formatter.string(from: 1.230000) // 1.2
formatter.string(from: 0.00123) // 0.0012
```

Important: If the minimum number of digits is greater than the maximum number of digits for any of these properties `NumberFormatter` defers to the maximum value specified. Negative values are ignored.

When you want to enforce a fixed width for your representation, set `usesSignificantDigits` to `false` and specify maximum and minimum integer and/or fraction digits. By default, `minimumFractionDigits`, `maximumFractionDigits`, and `minimumIntegerDigits` are set to `0`; `maximumIntegerDigits` is cheekily set to 42.

```
let formatter = NumberFormatter()
formatter.usesSignificantDigits = false // default

formatter.minimumIntegerDigits = 4
formatter.minimumFractionDigits = 2

formatter.string(from: 123) // 0123.00
formatter.string(from: 123456) // 123456.00
formatter.string(from: 123.456) // 0123.46
formatter.string(from: 1.230000) // 0001.23
formatter.string(from: 0.00123) // 0000.00
```

Custom formats override the built-in number formatter styles, and we'll talk more about those later on.

Rounding Modes

When the number of digits exceeds what a `NumberFormatter` object is configured to show, it rounds the value according to the configuration specified in its `roundingMode` property.

The following rounding modes are available:

- `ceiling` and `floor` round toward positive and negative infinity, respectively
- `up` and `down` round away from and toward zero, respectively
- `halfUp` and `halfDown` round toward the closest value, rounding away from or toward zero, respectively, if equidistant
- `halfEven` round toward the closest value, rounding toward the nearest even number if equidistant

The following table provides a summary of how each of these rounding modes behaves:

Rounding Mode	1.2	1.22	1.25	1.27	-1.25
`ceiling`	1.2	1.3	1.3	1.3	-1.2
`floor`	1.2	1.2	1.2	1.2	-1.3
`up`	1.2	1.3	1.3	1.3	-1.3
`down`	1.2	1.2	1.2	1.2	-1.2
`halfUp`	1.2	1.2	1.3	1.3	-1.3
`halfDown`	1.2	1.2	1.2	1.3	-1.2
`halfEven`	1.2	1.2	1.2	1.3	-1.2

```
let formatter = NumberFormatter()
formatter.numberStyle = .decimal
formatter.maximumFractionDigits = 1

let numbers =
    [1.2, 1.22, 1.25, 1.27, -1.25]
let modes: [NumberFormatter.RoundingMode] =
    [.ceiling, .floor, .up, .down, .halfUp, .halfDown,
.halfEven]

for mode in modes {
    formatter.roundingMode = mode

    for number in numbers {
        formatter.string(for: number)
    }
}
```

Tip: You can specify different rounding behaviors by setting the `roundingBehavior` property to a `NSDecimalNumberHandler` value or using a custom number format.

Scientific Notation

Scientific notation.

Not a great choice for the general public, but engineering types go nuts for it.

Though, with so much emphasis placed on standardization within science and engineering, there's a surprising lack of consensus for how scientific notation should be represented globally.

Locale	Identifier	`scientific`
U.S. English	en-US	1.23456789E4
POSIX	en-US_POSIX	1.234568E+004
French	fr	1,23456789E4
Greek	el	1,23456789e4
Swedish	sv	1,23456789×10^4
Arabic	ar	١٫٢٣٤٥٦٧٨٩اس٤

```
let formatter = NumberFormatter()
formatter.numberStyle = .scientific
formatter.string(for: 12345.6789) // 1.23456789E4
```

- The IEEE POSIX locale pads its exponent with leading 0s to 3 places (a fixed width is easier for a computer to parse).
- Locales that represent the decimal separator with a comma for decimal representations generally do the same for scientific notation.
- Some locales in Nordic-Baltic countries, including Estonia, Lithuania, and Sweden use the multiplication sign instead of the letter E.
- Some locales in Central and Southeastern European countries, including Greece, Slovakia, and Slovenia use a lowercase letter e.
- Locales that use Non-Hindu-Arabic numerals in other number styles use those same numerals for scientific notation as well.

But really, who cares, right? Most of the nerds who look at scientific notation all day probably wouldn't even notice if we swapped in Swedish numbers while they weren't looking. I mean, they can barely see anyway, what with their thick, dumb-looking glasses.

Other than that, there's not much else to say — oh, except that you should probably set `usesSignificantDigits` to `true` when `numberStyle` is `scientific`. Sigfigs are like catnip for the lab coat and pocket protector set.[11]

Percentages

Percentages express a ratio as a fraction of 100. `NumberFormatter` does this conversion for you automatically. For example, the number 0.12 is 12 percent, whereas the number 12 is 1200 percent.

11. This section was contributed by Flight School's resident bully, Chad.

Locale	Identifier	`percent`
U.S. English	en-US	12%
French	fr	12 %
Turkish	tr	%12
Basque	eu	% 12
Arabic	ar	٪ ١٢

```
let formatter = NumberFormatter()
formatter.numberStyle = .percent
formatter.string(for: 0.12) // 12%
```

There's worldwide variation in terms of whether the percent sign goes before or after the number, and whether there's a space in between.

Arabic even has its own percent sign, which uses the zero numeral from its numeral system.

- `U+066A ٪ ARABIC PERCENT SIGN`

Fortunately, `NumberFormatter` keeps track of these conventions so you don't have to.

Per-Mille

Along with the `percentSymbol` property that allows you to override the default locale symbol, `NumberFormatter` has a mysterious `perMillSymbol` property. The *per-mille* symbol (‰) expresses a ratio as a fraction of 1,000. It's used so infrequently that there's no consensus as whether it's spelled "per mil", "per mill", "permil", "permill", or "permille".

You can customize this property all you want, but there's not a single locale on Apple platforms that uses it with `percent` style. If you want it to make an appearance in your app, you'll need to specify a custom format.

Currency and Monetary Amounts

`NSNumberFormatter` has four *currency* styles that are used to represent monetary amounts using both the format and currency of the user's current locale.

The `currency` style uses the currency symbol:

Locale	Identifier	`currency`
U.S. English	en-US	$1,234.57
British English	en-GB	£1,234.57
German (Germany)	de-DE	1.234,57 €
Japan (Japanese)	ja-JP	¥1,235

The `currencyAccounting` style uses the currency symbol and denotes negative values with parentheses:

Locale	Identifier	`currencyAccounting`
U.S. English	en-US	($1,234.57)
British English	en-GB	(£1,234.57)
German (Germany)	de-DE	-1.234,57 €
Japan (Japanese)	ja-JP	(¥1,235)

Note: Developers often use `currencyAccounting` style and set `textAttributesForNegativeValues` to have `red` for `foregroundColor` when calling `attributedString(for:withDefaultAttributes:)`. According to some accountants, this practice may not be a good idea, as it draws an unnecessary amount of attention to something that's commonplace in any bookkeeping capacity. As always, be sure to consult with stakeholders, domain experts, and customers when designing your user interface.

The `currencyISOCode` style uses three-letter ISO 4217 currency codes:

Locale	Identifier	`currencyISOCode`
U.S. English	en-US	USD1,234.57
British English	en-GB	GBP1,234.57
German (Germany)	de-DE	1.234,57 EUR
Japan (Japanese)	ja-JP	JPY1,235

The `currencyPlural` style uses a localized, plural natural language representation of the currency:

Locale	Identifier	`currencyPlural`
U.S. English	en-US	1,234.57 US dollars
British English	en-GB	1,234.57 British pounds
German (Germany)	de-DE	1.234,57 Euro
Japan (Japanese)	ja-JP	1,235円

```
let formatter = NumberFormatter()

let identifiers =
    ["en-US", "en-GB", "de-DE", "ja-JP"]
let styles: [NumberFormatter.Style] =
    [.currency, .currencyAccounting, .currencyISOCode,
.currencyPlural]

for style in styles {
    formatter.numberStyle = style
    for identifier in identifiers {
        formatter.locale = Locale(identifier: identifier)
        formatter.string(for: 1234.567)
    }
}
```

`NumberFormatter` helpfully formats prices into the format that the user is most comfortable with. And normally, this is where we'd compare and contrast the different styles and how they're presented in various locales…

But we have something more important to discuss, and that's: Treating currency as a localization preference fundamentally changes the meaning of the number represented.

In all of the examples before — from cardinals and ordinals to decimals and percentages — all of the different localized representations are equivalent. You may not read them correctly (or be able to read them at all), but the meaning is exactly the same for each.

Currencies are different. They're units, not a formatting quirk. By treating $ and ¥ as interchangeable, `NumberFormatter` varies its representation by roughly two orders of magnitude. $300 is much more than ¥300.[12]

```
let price: Decimal = 300.00 // in USD

let formatter = NumberFormatter()
formatter.numberStyle = .currency

formatter.locale = Locale(identifier: "en-US")
formatter.string(for: price) // $300.00

formatter.locale = Locale(identifier: "ja-JP")
formatter.string(for: price) // ¥300 (!)
```

Warning: Setting `numberStyle` to any of the currency styles without explicitly setting the `currencySymbol` and/or `currencyCode` properties produces incorrect results.

We'll outline a comprehensive solution for working with and representing money in Chapter 3.

Custom Formats

If none of the built-in `NumberFormatter` styles offer exactly what you're looking for, you can set a custom number format.

12. At the time of writing, the USD/JPY ≅ 100.

The `format` property takes a valid Number Format Pattern as described in Unicode Technical Standard #35 Part 3:

Symbol	Meaning
`0`	Digit
`1-9`	Rounding increment
`@`	Significant digit
`#`	Digit, zero shows as absent
`.`	Decimal separator
`-`	Minus sign
`,`	Grouping separator

```
let formatter = NumberFormatter()
formatter.numberStyle = .decimal

// Format with thousands and decimal separator
// that rounds to the nearest five tenths
formatter.format = "#,##0.5"

formatter.locale = Locale(identifier: "en-US")
formatter.string(for: 1234.567) // 1,234.5

formatter.locale = Locale(identifier: "fr-FR")
formatter.string(for: 1234.567) // 1 234,5
```

Tip: You're more likely to use custom `format` strings for parsing numbers, rather than representing them. Given the various concerns of going in the opposite direction, parsing is an entire chapter unto itself and is left for a future Flight School guide.

We'll expand on our concerns about currency formatting in the next chapter.

`NumberFormatter` is an essential piece of any app developer's toolkit. However, as we saw, using it correctly requires careful consideration — more than a lot of folks realize. Some features work exactly as advertised. Others require additional configuration to ensure correct behavior. And a couple parts are incorrect to the point of being considered harmful.

Keep these in mind the next time you reach for `NumberFormatter`.

Recap

- Use the none and `spellOut` styles when presenting numbers as quantities.
- Use the `ordinal` style when presenting numbers as rank or order in a sequence.
- Use the `decimal` style when presenting numbers as measurement.
- Use the `scientific` style when presenting numbers as precise measurement in a science or engineering context.
- Use the `percent` style when presenting number as a ratio.
- Use one of the `currency` styles when presenting number as a monetary amount, but always set an explicit `currencySymbol` and/or `currencyCode`.


$
¥
€ $ ¥
JPY
USD

Chapter 3:

Currencies and Monetary Amounts

So there you are, wandering through international arrivals. You're exhausted, despite any sleep you did manage in the air.

What time is it? Who knows. You're acting on instinct at this point. And all that matters to you now is getting to your hotel and going to bed. Or maybe taking a shower.

Anyway, you don't have the energy to navigate the social exchange with the taxi driver to see if they take credit cards, so you decide to exchange cash-on-hand for the local currency as backup — if not for the cab fare, then definitely for some snacks.

After a polite but slightly awkward interaction with the clerk at the currency exchange kiosk, you get back a pile of what you're assured is, in fact, legal tender. Looking at the sea of unfamiliar faces adorning the brightly colored bills, you have your doubts.

Between the transaction fees and sleep deprivation, you immediately lose track how much that was supposed to be. You scrutinize the bills, attempting to discern their intrinsic worth. ("100... Is that a lot?"). Dead presidents can't help you now.

Sound a bit like how your app implements money? You wouldn't be alone if it did. Keep reading and you'll learn the right way to work with monetary amounts.

Representing Money

As we discussed in Chapter 1, binary floating-point numbers can't accurately represent most factors of 10. Nonintuitive results like 0.1 + 0.2 != 0.3 may be inconvenient in some situations, but when money's at stake, these kinds of errors can be catastrophic.

Warning: Don't represent monetary amounts with `Float` or `Double`.

However, there's more to it than that. An amount of money by itself is meaningless. Until it has an associated currency, it's just another number.

You could infer a currency for all amounts within a context, but this design limits your ability to do business with the rest of the world. And when the time does come for you to expand to new markets, you'll wish you'd done things right from the start.

Here are some qualities that you should look for in a `Money` type:

- **Precision**: You can express quantities exactly
- **Type-Safety**: You can't combine or compare monetary amounts with different currencies
- **Convenience**: You can create and work with monetary amounts just as easily as you can with a built-in number type

Swift doesn't have any built-in types for working with money, but it does give us everything we need to build one for ourselves.

Representing Amounts with Integers

If you ask a random programmer how to represent money, they may tell you to use an `Int`. They'll say that Integers, unlike conventional floating-point types, can precisely represent monetary amounts. And they'd be right — they certainly can. The only adjustment you'd

need to make is to express the price in terms of their *minor unit*. In the case of US Dollars, that means multiplying by 100 to express the amount in cents ($1 = 100¢).[1]

Where you start to run into problems is when you try to multiply or divide by fractional amounts, such as when you're calculating tax. Every time you perform one of these operations, you round any remainder into the final result. Intermediate rounding like this can cause error to accumulate and cause results to be off by a little bit.

This all may be more or less acceptable depending on your use case. However, the most compelling argument against using integers to represent money is that it's reinventing the wheel.

There's already a standard Swift type that has what you're looking for: `Decimal`.

Representing Amounts with Decimals

A *decimal* is a number type that can precisely represent base-10 numbers. In Swift, the Foundation framework provides the `Decimal` type, which is bridged to the `NSDecimalNumber` class (which itself wraps the `NSDecimal` structure and its associated functions).

Unlike `Double` or `Float` values, you don't have to worry about `Decimal` values not adding up the way you expected.

A floating-point decimal number can be expressed by the signed product of a significand and a power of 10:

$$(-1)^{\text{sign}} \times \text{significand} \times 10^{\text{exponent}}$$

Decimal floating-point numbers are often represented as a sequence of *binary-coded decimals*, or BCD. There are a variety of different encoding methods, the easiest to understand being "natural

1. Some currencies, like US Dollars and Euros, have minor units whereas others, like Japanese Yen, don't. The 2015 update to the ISO 4217 standard lists this information about currencies in addition to their code and name.

BCD" (NBCD), in which each decimal digit is represented by its corresponding 4-bit unsigned integer value.[2] 4 bits can encode 16 states, and we can map the decimal digits 0 – 9 to the first ten of those:[3]

Digit	0	1	2	3	4	5	6	7	8	9
NBCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

For example, the fractional component of the number 0.123456 can be represented by 6 decimal digits that are encoded using NBCD into 6 groups of 4 bits, or 3 bytes if we use store a digit in both the low and high bits:

```
(0b0001_0010, 0b0011_0100, 0b0101, 0b0110)
```

The IEEE 754 specification was amended in 2008 to add decimal floating-point formats (32-, 64-, and 128-bit). However, `NSDecimal` dates back to the original Mac OS X SDKs, circa 2000 — predating the IEEE decimal standard by a decade. While it's unclear exactly how `NSDecimal` works (its implementation is proprietary to Apple), the important thing is that it does, in fact, work.[4]

---

2. Also known as "8421 encoding", in reference to the respective values of each binary digit.

3. NBCD is a rather inefficient encoding method, using only 10 of the 16 (62.5%) possible states in 4-bit. One way to improve on this approach would be to encode three decimal digits at a time in 10-bit *declets*. This has significantly better storage characteristics because 2^{10} (1024) is quite close to 10^3 (1000).

4. Why hasn't Apple updated `NSDecimalNumber` to follow the new standards? Truth is, there's little incentive to throw out a working solution, especially when replacing existing functionality can break software that depends on a particular behavior. The most plausible way forward would be for Swift to add native decimal types to its standard library in a future release.

Defining a Currency Type

A *currency* is a unit of monetary value. Currencies are standardized by the ISO 4217 specification, which defines a unique identifying code for each.

Option 1: Currency as an Enumeration

Given a fixed, well-defined set of possible values, an enumeration is a natural candidate for modeling currencies. You can define a case for each currency by its ISO 4217 code and provide the remaining information in computed properties, like so:

```
enum Currency: String {
    case USD
    // ...

    var code: String { return self.rawValue }
    var symbol: String {
        switch self {
            case .USD: return "$"
            // ...
        }
    }
}
```

This approach is viable but suboptimal. When you use enumerations, everything eventually turns into a `switch` statement. As the number of cases increases, so does the inconvenience of implementation. And with several hundred currencies listed in the ISO 4217 specification, the implementation becomes really tedious.

Option 2: Currency as a Structure

In contrast to an enumeration, where information is disbursed across switch statements in computed properties, structures allow information to stay localized as fields in a single object.

```
struct Currency {
    let code: String
    let symbol: String
    let name: String
}
```

We can define constant type properties for all of the standard currencies. The three-letter ISO 4217 identifiers are convenient as they are unique and valid Swift identifiers.

```
extension Currency {
    static let USD = Currency(code: "USD",
                              symbol: "$",
                              name: "US Dollar")

    static let CNY = Currency(code: "CNY",
                              symbol: "¥",
                              name: "Chinese Yuan Renminbi")
}
```

Let's give this structure-based approach a test-drive with a simple `Money` type.

```
struct Money {
    var amount: Decimal
    let currency: Currency
}

let priceInUSD = Money(amount: 14.00, currency: .USD)
let priceInCNY = Money(amount: 29.9, currency: .CNY)
```

So far, so good. The `Decimal` type gives us the precision we need, and a `currency` property provides a meaningful unit for the monetary amounts.

Unfortunately, this approach doesn't meet our criteria for type-safety, which is problematic if you accidentally try mix-and-match amounts in different currencies:

```
priceInUSD == priceInUSD // true
priceInUSD == priceInCNY // false
priceInUSD < priceInCNY // well...
priceInUSD + priceInCNY // hmm...
```

Monetary amounts can be compared (greater than, less than) and added together, but *only* if they have the same currency. Such an implementation would require error handling or assertions to deal with invalid operations at run time.

Again, this is a viable approach, but we can still do better.

Option 3: Currency as a Protocol

Instead of representing each currency as an instance, we can instead define a new type for each. Before, `code`, `symbol`, and `name` were instance properties of a `Currency` structure, but now they're properties of a type conforming to a `CurrencyType` protocol:

```
protocol CurrencyType {
    static var code: String { get }
    static var symbol: String { get }
    static var name: String { get }
}

enum USD: CurrencyType {
    static var code: String { return "USD" }
    static var symbol: String { return "$" }
    static var name: String { return "US Dollar" }
}
```

Why are we defining currencies with enumerations?

In Swift, enumerations are different from structures and classes because they don't require initializers. Without any initializers, a type can't be constructed — which is exactly the behavior we want.

```
USD() // 'USD' cannot be constructed because it has no
accessible initializers
```

And without any cases, the only accessible members on these are the type properties `code`, `symbol`, and `name`.

```
USD.code // USD
USD.symbol // $
USD.name // US Dollar
```

By having currencies as types, we can use them as constraints in a generic `Money` structure:

```
struct Money<C: CurrencyType> {
    typealias C = Currency

    var amount: Decimal

    init(_ amount: Decimal) {
        self.amount = amount
    }
}

let priceInUSD = Money<USD>(amount: 14.00)
let priceInCNY = Money<CNY>(amount: 29.9)
```

Tip: Naming the protocol `CurrencyType` rather than `Currency` allows us to use that name for the associated type in `Money` without conflicting with the generic constraint.

Unlike the previous structure-based approach, this protocol-oriented solution allows the type system to do its job and enforce semantics at build time.

```
priceInUSD == priceInUSD // true
priceInUSD == priceInCNY // Binary operator '==' cannot be
applied...
priceInUSD < priceInCNY // Binary operator '<' cannot be
applied...
priceInUSD + priceInCNY // Binary operator '+' cannot be
applied...
```

This approach meets all of the criteria we established from the start for precision, type-safety, and convenience. Despite some caveats relating to dynamic type lookup and deserialization,[5] we think this one's a winner.

Generating Currency Declarations with GYB

Now that we've settled on our protocol-oriented approach, it's time to declare a `CurrencyType` for each of the codes listed in ISO 4217.

We could roll up our sleeves and start typing away, but that sounds about as fun as organizing our `~/Library/Developer/Xcode/DerivedData` directory.

Let's work smarter, not harder, by taking advantage of code generation.

GYB[6] is an internal build tool used to generate code in the Swift standard library. Without a formal macro system in the language, it's the closest we have to a community standard for the task.

---

5. Swift doesn't provide a mechanism for looking up a type by name at run time. For instance, if you're decoding a JSON response, you can't construct a `Money` value using the currency code in the payload. Instead, you can define the currency (or currencies) you support in code and validate that the currency code in the JSON matches your expectations.

6. GYB is an acronym for "Generate Your Boilerplate", a reference to ~~another Python tool GYP, or "Generate Your Projects".~~ the Haskell package SYB, or "Scrap Your Boilerplate".

GYB isn't part of the standard Xcode toolchain, so you won't find it with xcrun. Instead, you can install it using Homebrew:

```
$ brew install nshipster/formulae/gyb
```

GYB templates have a simple syntax, consisting of the following:

- `%{ ... }` evaluates a block of code
- `% ...`: followed by `%end` manages control flow
- `${ ... }` substitutes the result of an expression
- Any other text is inserted into the output

In our `Currency.swift.gyb` template, we import the CSV module and read a data file containing the ISO 4217 currency definitions.[7]

Code	Symbol	Name
AFN	؋	Afghani
ALL	Lek	Lek
ARS	$	Argentine Peso
AUD	$	Australian Dollar
AWG	ƒ	Aruban Florin
⋮	⋮	⋮
UZS	лв	Uzbekistan Sum
VND	₫	Vietnamese Dong
XCD	$	East Caribbean Dollar
YER	﷼	Yemeni Rial
ZAR	R	Rand

7. ISO 4217 defines only the 3 letter codes and official currency names. Currency symbols and localized names are provided by the Unicode Common Locale Data Repository (CLDR).

For each row in the file, we extract the fields into local variables and declare a new `Currency` enumeration using the ISO code as the type name and each field for its respective type property:

```
%{
  import csv
}%
% with open('iso4217.csv') as file:
    % for row in csv.DictReader(file):
%{
        code = row["Code"]
        name = row["Name"]
        symbol = row["Symbol"]
        variable = int(row["MinorUnit"]) \
                    if row["MinorUnit"]
                    else 0
}%
        % if code and name and symbol:

public enum ${code}: CurrencyType {
    public static var code: String {
        return "${code}"
    }

    public static var name: String {
        return "${name}"
    }

    public static var symbol: String {
        return "${symbol}"
    }

    public static var minorUnit: Int {
        return ${minorUnit}
    }
}
        %end
    %end
%end
```

To generate our file, we call the gyb command from the terminal like so:

```
$ ./gyb --line-directive '' -o Currency.swift
Currency.swift.gyb
```

Now let's open up `Currency.swift` and verify that everything generated correctly:

```
public enum AFN: CurrencyType {
    public static var code: String {
        return "AFN"
    }

    public static var name: String {
        return "Afghani"
    }

    public static var symbol: String {
        return "؋"
    }

    public static var minorUnit: Int {
        return 0
    }
}

// ...
```

Brilliant. And just as an extra pat on the back, let's see how much boilerplate code we didn't have to write ourselves:

```
$ wc -l ./Currency.swift
1880
```

Nearly two thousand LOC! That's a few hours saved from what is, essentially, data entry into Xcode.

Let's use that time to do something more productive, like take our `Money` type from before to completion.

Defining a Money Type

We define `Money` to be a generic structure that stores an `amount` of a `CurrencyType`:

```
import Foundation

struct Money<C: CurrencyType>: Equatable {
    typealias Currency: C

    var amount: Decimal

    init(_ amount: Decimal) {
        self.amount = amount
    }

    var currency: CurrencyType.Type {
        return Currency.self
    }
}
```

We conform to `Equatable` in the declaration because, as of Swift 4.1, the compiler can automatically synthesize conformance.

In addition to checking for equality, we'll want to be able to do less-than and greater-than comparisons as well. Conformance for `Comparable` can't be synthesized, but the implementation couldn't be simpler:

```
extension Money: Comparable {
  static func < <C: Currency>(lhs: Money<C>, rhs: Money<C>) ->
Bool {
    return lhs.amount < rhs.amount
  }
}
```

Implementing Literal Expressibility

As with any type relating to numbers, it would be convenient to initialize `Money` from a literal. If a currency has minor units, they're typically expressed with floating-point literals; if

not, integer literals make more sense. To support both, we'll adopt `ExpressibleByIntegerLiteral` and `ExpressibleByFloatLiteral` in an extension.

```
extension Money: ExpressibleByIntegerLiteral {
    public init(integerLiteral value: Int) {
        self.init(Decimal(integerLiteral: value))
    }
}

extension Money: ExpressibleByFloatLiteral {
    public init(floatLiteral value: Double) {
        self.init(Decimal(floatLiteral: value))
    }
}
```

Implementing Mathematical Operators

Monetary amounts with the same currency should be able to be added and subtracted. To support this, we define the + and - operators (in addition to their mutating counterparts) as static functions in an extension.

```
extension Money {
    static func + (lhs: Money<C>, rhs: Money<C>) -> Money<C> {
        return Money<C>(lhs.amount + rhs.amount)
    }

    static func += (lhs: inout Money<C>, rhs: Money<C>) {
        lhs.amount += rhs.amount
    }

    // repeat for subtraction
}
```

Implementing these operators unlocks powerful functionality, including summation over a collection of monetary amounts using the `reduce(_:_:)` method:

```
let prices: [Money<USD>] =
    [2.19, 5.39, 20.99, 2.99, 1.99, 1.99, 0.99]
let subtotal = prices.reduce(0.00, +)
```

Another convenient operator to have is multiplication by scalars. Again, we implement the * and *= operators in an extension:[8]

```
extension Money {
    static func * (lhs: Money<C>, rhs: Decimal) -> Money<C> {
        return Money<C>(lhs.amount * rhs)
    }

    static func * (lhs: Decimal, rhs: Money<C>) -> Money<C> {
        return Money<C>(lhs * rhs.amount)
    }

    static func *= (lhs: inout Money<C>, rhs: Decimal) {
        lhs.amount *= rhs
    }

    // repeat for other number types
}
```

Inevitably, multiplication will be used to calculate taxes:

```
let tax = 0.08 * subtotal
let total = subtotal + tax
```

To complement our binary operators, the prefix operator for negation (-) is a nice-to-have:

```
extension Money {
    public static prefix func - (value: Money<C>) -> Money<C> {
        return Money<C>(-value.amount)
    }
}
```

This can be helpful for accounting purposes.
After all: one balance sheet's credit is another one's debit.

```
let credit: Money<USD> = 1.00 // $1.00
let debit = -credit // ($1.00)
```

---

8. The `Decimal` type doesn't have generic initializers for other number types, so we need to implement these operators for other number types, like `Double` and `Int`, ourselves. This would be another good occasion to break out gyb for code generation.

Tip: Money would be a good candidate for the `Numeric` and `SignedNumeric` protocols if it weren't for the requirement to support multiplication (although you can multiply money by a scalar, it doesn't make sense to multiply two amounts of money together).

Formatting Monetary Amounts

We can apply what we learned in Chapter 2 to our `Money` type and be confident in any representation of monetary amounts from here on out.

```
let price: Money<USD> = 14.00

let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.currencyCode = price.currency.code
formatter.string(for: price.amount) // $14.00
```

If we set a different `locale`, we can see that the format adjusts accordingly, without altering the amount of money being represented:

```
formatter.locale = Locale(identifier: "fr-FR")
formatter.string(for: price.amount) // 14,00 $US
```

We can use this approach to implement the `description` or `debugDescription` properties required by the `CustomStringConvertible` or `CustomDebugStringConvertible` protocols, or we can create a `MoneyFormatter` class that offers a more streamlined interface.

Converting Between Currencies

Money in one currency is incompatible with money in other currencies. However, you can convert from one currency to another through an exchange (for a fee). Exchange between currencies happens at a set rate, which fluctuates over time. This rate is defined as a ratio of how much 1 unit of a *fixed* currency will be exchanged for a *variable* currency. Two currencies that can be exchanged for one another at a particular rate are called a *currency pair*.

Note: Currency traders have nicknames for some of the most common pairs. For example, the exchange rate for Euros to British Pounds Sterling is sometimes called "Chunnel", in reference to undersea tunnel linking France with England.

We'll start by defining a `UnidirectionalCurrencyConverter` protocol, which defines the associated `Fixed` and `Variable` currency types and a `rate` property.

```
protocol UnidirectionalCurrencyConverter {
    associatedtype Fixed: CurrencyType
    associatedtype Variable: CurrencyType

    var rate: Decimal { get set }
}
```

By defining a `convert(_:)` from `Fixed` to `Variable` in a protocol extension, we provide a default implementation to all conforming types.

```
extension UnidirectionalCurrencyConverter {
    func convert(_ value: Money<Fixed>) -> Money<Variable> {
        return Money<Variable>(value.amount * rate)
    }
}
```

If an exchange supports conversion from one currency to another, there's a chance that it also supports conversion going the other way, too. To model this, we'll define a `BidirectionalCurrencyConverter` that inherits `UnidirectionalCurrencyConverter`.

```
protocol BidirectionalCurrencyConverter:
            UnidirectionalCurrencyConverter {}
```

Following the same pattern as before, we define a default implementation for a convert(_:) method that returns `Money<Fixed>` using the inverse of the defined `rate`:

```
extension BidirectionalCurrencyConverter {
    func convert(_ value: Money<Variable>) -> Money<Fixed> {
        return Money<Fixed>(value.amount / rate)
    }
}
```

Finally, we construct a concrete `CurrencyPair` type that conforms to `BidirectionalCurrencyConverter`:

```
struct CurrencyPair<Fixed, Variable>:
    BidirectionalCurrencyConverter
{
    var rate: Decimal

    init(rate: Decimal) {
        precondition(rate > 0)
        self.rate = rate
    }
}
```

Thanks to our implementation of the `convert(_:)` methods in protocol extensions, we don't need to do anything else to use `CurrencyPair`.

```
let EURtoUSD =
    CurrencyPair<EUR, USD>(rate: 1.17) // as of June 1st, 2018

let euroAmount: Money<EUR> = 123.45 // €123.45
let dollarAmount = EURtoUSD.convert(euroAmount) // $144.55
```

From here, you can extend your implementation with a variety of different features, including support for multi-currency exchanges or fetching the latest exchange rate from a web service. (If you do end up making a killing on the international money markets, remember who got you your start 🤑)

Recap

- A `Money` type encodes an amount and a currency.
- You should never represent monetary amounts using `Float` or `Double`. Use `Decimal` instead.
- Declaring currencies as types allows the type system to prevent invalid operations at compile time.
- Code generation with tools like GYB allows you to more easily implement correct and complete solutions.

Chapter 4:

Units and Measurements

Hop into the cockpit of a Cessna 172, and you'll be greeted by a dashboard packed with dials, buttons, switches, and knobs.

Directly above the yoke on the pilot's side of the aircraft are the six primary flight instruments arranged in a basic T arrangement:[1]

Along the top row, going left to right, you have the **airspeed indicator**, **attitude indicator**, and **altimeter**. Below that, you have the **turn coordinator**, **heading indicator**, and **vertical speed indicator**.

The airspeed indicator, attitude indicator, and vertical speed indicator get their readings from the *pitot-static system*. By measuring

1. Sometimes affectionately referred to as a "six-pack".

the dynamic pressure created when air flows over the plane during flight, you can determine the speed of the aircraft. By comparing the static pressure to the constant barometric pressure at sea level, you can determine altitude; vertical speed is derived from changes in altitude over time.

The attitude indicator, turn coordinator, and heading indicator get their readings from a *gyroscope*. When an aircraft changes its pitch (nose up or down), yaw (nose left or right), or roll (rotation about the axis from nose to tail), the resulting torque on the gyroscope can be measured to determine the plane's three-dimensional orientation.

Pilots rely on readings from these instruments to maintain course and ensure straight and level flight.

In the same way, users rely on apps for their day-to-day lives.

If your app works with physical units and measurements in any capacity — whether it's calculating the weight of a shipment, measuring the dimensions of a room with AR, or showing today's high temperature — you have a responsibility to provide accurate, meaningful information. This chapter will show you how you can use the Foundation framework to do just that.

Foundation Units and Measurements

Introduced in iOS 10 and macOS 10.12, Foundation's units and measurements APIs provide a robust, extensible way to work with and represent physical quantities.

`Unit` is the abstract base class for all unit types. `Dimension` is an abstract subclass of `Unit` that serves as the base class for all dimensional unit types.[2] `Measurement` is a generic structure that stores a value for a particular `Unit` type.

2. As opposed to dimensionless units, like count or ratio.

Each unit type is defined by a *base unit*, from which all other units for that type are defined.[3] For example, the base unit for `UnitLength` is `meters`, and other length units, like `yards`, `miles`, and `kilometers`, are defined in terms of of meters.

All `Unit` objects have a `symbol` — "m" for meters, "kg" for kilograms, and so on.

`Dimension` objects additionally have a `converter` property, which is used to convert to and from the base unit for that unit type. This can be any object that inherits from the abstract base class `UnitConverter`. In practice, this is almost always a `UnitConverterLinear` object, since most unit conversions for physical quantities can be expressed by a linear equation:

$$y = mx + b$$

The Foundation framework provides several built-in unit types:

3. Conventionally, these are SI base or derived units, rather than any other locale-specific customary units.

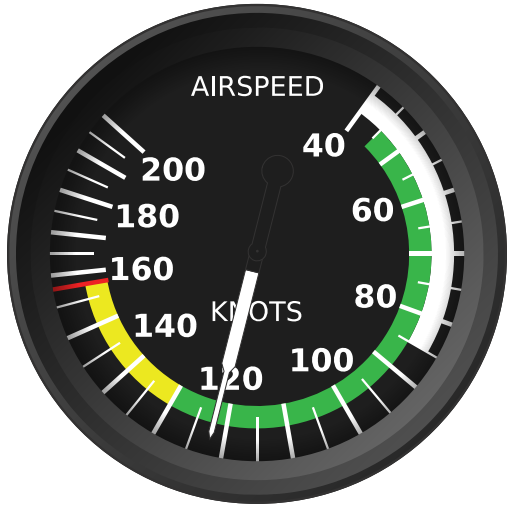
Unit class	Base unit
`UnitAcceleration`	meters per second squared (m/s²)
`UnitAngle`[4]	degrees (°)
`UnitArea`	square meters (m²)
`UnitConcentrationMass`	milligrams per deciliter (mg/dL)
`UnitDispersion`[5]	parts per million (ppm)
`UnitDuration`	seconds (s)
`UnitElectricCharge`	coulombs (C)
`UnitElectricCurrent`	amperes (A)
`UnitElectricPotentialDifference`	volts (V)
`UnitElectricResistance`	ohms (Ω)
`UnitEnergy`	joules (J)
`UnitFrequency`	hertz (Hz)
`UnitFuelEfficiency`	liters per 100 kilometers (L/100km)
`UnitIlluminance`	lux (lx)
`UnitLength`	meters (m)
`UnitMass`	kilograms (kg)
`UnitPower`	watts (W)
`UnitPressure`[6]	newtons per square meter (N/m²)
`UnitSpeed`	meters per second (m/s)
`UnitTemperature`	kelvin (K)
`UnitVolume`	liters (L)

Working with Measurements

Flight instruments traditionally express airspeed in *knots*, or nautical miles. The use of a naval unit in aviation may seem anachronistic,

4. `UnitAngle` defines its base unit as `degrees` rather than the SI unit, `radians`.
5. `UnitDispersion` is a subclass of `Dimension` despite its base (and only) unit, `partsPer Million` being dimensionless.
6. `UnitPressure` calls its base unit `newtonsPerMeterSquared` rather than `pascals` despite the pascal being one of the 22 named units derived from SI base units. Foundation doesn't even have a `UnitForce` for which the base unit would be `newtons`.

but knots are genuinely useful for navigation.[7] A nautical mile is equal to a minute (1/60th of a degree) of latitude, or about 1855 meters.[8]



To create a `Measurement` object for this airspeed, you pass the value and unit to the initializer.

```
let indicatedAirspeed =
    Measurement<UnitSpeed>(value: 123, unit: .knots)
```

Don't have an intuitive sense of how fast that is? You can use the `converted(to:)` method to express measurements in more familiar terms:

```
indicatedAirspeed
    .converted(to: .kilometersPerHour) // 227.80 km/h
```

---

7. Also, aircraft are more similar to naval vessels than, say, automobiles; both are propelled through a fluid, turned by rudders, and often piloted by crew with spiffy uniforms.

8. Because the earth is an oblate spheroid rather than a perfect sphere, the distance of one minute of latitude is about 1% greater at the equator compared to at either pole.

Measurements with different units can be added and subtracted so long as they all have the same unit type. For example, measurements expressed in kilograms, stones, and pounds can be added together because they all express quantities of mass.

```
let cessna172Weight =
    Measurement<UnitMass>(value: 660, unit: .kilograms)
let pilotWeight =
    Measurement<UnitMass>(value: 11, unit: .stones)
let fuelWeight =
    Measurement<UnitMass>(value: 300, unit: .pounds)

let takeoffWeight = (cessna172Weight +
                       pilotWeight +
                       fuelWeight
                     ).converted(to: .metricTons) // 0.86593079 t
```

Attempting to add together measurements with incompatible units (no matter how similar their pronunciation) results in a compiler error:

```
let length =
    Measurement<UnitLength>(value: 10, unit: .millimeters)
let volume =
    Measurement<UnitVolume>(value: 10, unit: .milliliters)

length + volume // Binary operator '+' cannot be applied...
```

You can, however, define binary operators for different units when they produce something meaningful, such as multiplying two lengths together to get area.

```
func *(lhs: Measurement<UnitLength>,
       rhs: Measurement<UnitLength>) -> Measurement<UnitArea> {
    return .init(value: lhs.value *
                        rhs.converted(to: lhs.unit).value,
                 unit: UnitArea.baseUnit())
}
```

We'll explore this more in Chapter 5.

Representing Measurements

If you want to display measurements to users, look no further than `MeasurementFormatter`. Simply create a formatter and call the `string(from:)` method:

```
let formatter = MeasurementFormatter()
formatter.string(from: indicatedAirspeed) // 141.546 mph
```

Configuring Scale

By default, `MeasurementFormatter` expresses measurements in terms of the preferred unit for the specified locale.[9] But even within a particular locale, our unit preferences may depend on the context. For example, the length of a room may be measured in feet or meters, whereas the distance to the airport may be measured in miles or kilometers.

```
let lengthOfRoom =
    Measurement<UnitLength>(value: 8, unit: .meters)
let distanceToAirport =
    Measurement<UnitLength>(value: 16, unit: .kilometers)
```

By setting the `unitOptions` property to include `naturalScale`, `MeasurementFormatter` chooses the most appropriate unit for the measurement value among those used in the specified locale.[10]

```
let formatter = MeasurementFormatter()
formatter.unitOptions = .naturalScale
formatter.string(from: lengthOfRoom) // 26.247 ft
formatter.string(from: distanceToAirport) // 9.942 mi
```

---

9. This information is provided by Unicode Common Locale Data Repository (CLDR). For more information, see TR 35 Part 6

10. The exact behavior for how units are selected isn't documented.

Displaying with the Provided Unit

When there's a strong convention for using a particular unit, you may want to override localization preferences. For example, the customary unit of mass for precious metals is troy ounces.

```
let ingotMass =
    Measurement<UnitMass>(value: 400, unit: .ouncesTroy)
```

By setting the `unitOptions` property to include the `providedUnit` option, `MeasurementFormatter` uses the unit provided by the measurement.

```
let formatter = MeasurementFormatter()
formatter.unitOptions = .providedUnit
formatter.string(from: ingotMass) // 400 oz t
```

Configuring Precision

If you're working with physical units in a scientific or engineering capacity, you may want the measurement to be represented with a particular level of precision. For example, let's say you take a reading from a mercury barometer with etchings at every 1/10 inch:

```
let barometerReading =
    Measurement<UnitPressure>(value: 29.9, unit:
.inchesOfMercury)
```

If you were to represent this quantity in terms of millibars using `MeasurementFormatter`, the result would show more significant figures of precision than our original reading.

```
let pressureInMillibars =
    barometerReading.converted(to: .millibars)

let formatter = MeasurementFormatter()
formatter.unitOptions = .providedUnit
formatter.string(from: pressureInMillibars) // 1,012.531 mb
```

You can correct this behavior by configuring the significant digits properties on the underlying `numberFormatter` of the measurement formatter.

```
formatter.numberFormatter.usesSignificantDigits = true
formatter.numberFormatter.maximumSignificantDigits = 3
formatter.string(from: pressureInMillibars) // 1,010 mbar
```

Configuring Temperature Display

`MeasurementFormatter` gives special dispensation to how temperature is displayed. By default, temperature is expressed using the scale preferred by the user ("°F" in US locales, "°C" most everywhere else):

```
let temperatureInF =
    Measurement<UnitTemperature>(value: 72, unit: .fahrenheit)
let temperatureInC =
    Measurement<UnitTemperature>(value: 20.5, unit: .celsius)

let formatter = MeasurementFormatter()
formatter.locale = Locale(identifier: "en-US")
formatter.string(from: temperatureInF) // 72°F
formatter.string(from: temperatureInC) // 68.9°F

formatter.locale = Locale(identifier: "fr-FR")
formatter.string(from: temperatureInF) // 22,222 °C
formatter.string(from: temperatureInC) // 20,5 °C
```

Temperature is something that a lot of people feel strongly about, so much so that it's customary to omit the scale when displaying it. As if to say, *"Well of course it's 30° outside. How could that possibly be misinterpreted?"*

If you wish to be so brazen, you can pass the `temperatureWithoutUnit` unit option to — like it says on the tin — display the temperature without a unit.

However, be forewarned! This has the unexpected side effect of implicitly setting the `providedUnit` unit option:

```
formatter.unitOptions = .temperatureWithoutUnit

formatter.locale = Locale(identifier: "en-US")
formatter.string(from: temperatureInF) // 72°
formatter.string(from: temperatureInC) // 20.5° (!)

formatter.locale = Locale(identifier: "fr-FR")
formatter.string(from: temperatureInF) // 72° (!)
formatter.string(from: temperatureInC) // 20,5°
```

Warning: Setting the `temperatureWithoutUnit` unit option on `MeasurementFormatter` results in misleading representations of temperatures if you don't explicitly convert to the locale's preferred scale.

Unfortunately, there's no easy way to determine which temperature scale a locale uses. As far as we can tell, the only sanctioned way is to do the following:

- Create a separate measurement formatter
- Get the representation of a temperature
- Search for the `celsius` or `fahrenheit` unit symbol in the representation to determine the scale

...or you could just skip `temperatureWithoutUnit` entirely and remove the unit with some string post-processing. Your pick.

Defining a Custom Unit

On December 17, 1903, two brothers flew a glider 852 feet in 59 seconds across the sand dunes at Kittyhawk, North Carolina.

Aviation has been stuck with imperial units ever since.

Note: The International Civil Aviation Organization, or ICAO, has long wanted to transition everything over to metric, but getting rid of knots and feet will be difficult and expensive. For now, the official status of imperial units is "Termination date: not established."

Altitude is often expressed in feet, and vertical airspeed in feet per second. This unit doesn't come standard in `UnitSpeed`, but we can create it ourselves. All we need is the conversion rate and a symbol:

```
let converter = UnitConverterLinear(coefficient: 0.00508)
let feetPerMinute = UnitSpeed(symbol: "ft/s",
                              converter: converter)
```

Unless your custom unit is a one-off, it usually makes sense to expose it through a computed class property:

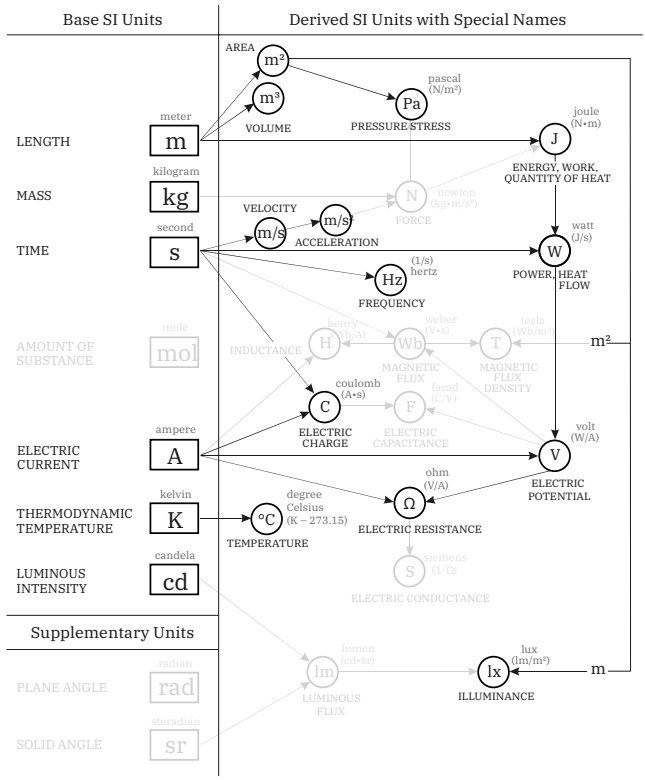
```
extension UnitSpeed {
  class var feetPerMinute: UnitSpeed {
    let converter = UnitConverterLinear(coefficient: 0.00508)
    return .init(symbol: "ft/s", converter: converter)
  }
}
```

Tip: We can use `.init` as instead of the more verbose constructor syntax here because the type can be inferred from the property declaration.

Subclassing Dimension

Although Foundation boasts an impressive number of built-in units,[11] you may find a lot of what you need to be missing. Consider the standard unit types found in aviation (missing units are grayed out):

11. Foundation has built-in support for 163 units across 21 unit types.



Adapted from Attachment B of "Units of Measurement to be Used in Air and Ground Operations" by the International Civil Aviation Organization

One of the more surprising omissions is the lack of a built-in unit for Force.[12] Fortunately, implementing `UnitForce` for ourselves requires very little force multiplied by time (i.e. work):

```
class UnitForce: Dimension {
  class var newtons: UnitForce {
    let baseUnitConverter = UnitConverterLinear(coefficient: 1)
    return .init(symbol: "N", converter: baseUnitConverter)
  }

  class var kiloNewtons: UnitForce {
    let converter = UnitConverterLinear(coefficient: 1000)
    return .init(symbol: "kN", converter: converter)
  }

  class var pounds: UnitForce {
    let converter = UnitConverterLinear(coefficient: 4.44822)
    return .init(symbol: "lbf", converter: converter)
  }

  override class func baseUnit() -> UnitForce {
    return .newtons
  }
}
```

Now if we wanted to represent, say, the thrust of a Pratt & Whitney PW4164 turbofan engine, we can do just that:

```
let pw4164Thrust =
    Measurement<UnitForce>(value: 64_500, unit: .pounds)
pw4164Thrust.converted(to: .kiloNewtons) // 286.91019 kN
```

Note: Pounds are actually a measure of weight, not mass, and weight is actually just another name for force. When pounds are used as a unit for force, they're often abbreviated as "lbf" instead of "lb".

12. Didn't Apple *invent* the Newton?

Interoperability with Other Units and Measurement APIs

`Unit`, `Dimension`, and `Measurement` may now be the standard, but just like the metric system, you still have to deal with inferior, competing standards. The good news is that a kilogram in one API is the same as a kilogram in another, so bridging between the two is straightforward.

Foundation Date and Calendar APIs

Although they usually go hand in hand, it's important to distinguish between time and calendar dates within the context of units.

`Date`, contrary to its name, represents an exact moment in time. You can express the distance between two `Date` values in absolute terms using seconds, minutes, or hours. Other units, like days, weeks, months, or years, are relative to a particular calendar and can fluctuate in length.

TimeInterval

`NSTimeInterval`, or `TimeInterval` as it likes to be called now, dates all the way back to `distantPast`. It's an alias to `Double`, defined to represent a number of seconds.

```
typealias TimeInterval = Double // seconds
```

To use `TimeInterval` for measurements, create a conditional extension on `Measurement` types that have a `UnitType` of `UnitDuration` and declare an initializer that takes a `TimeInterval` argument.

```
extension Measurement where UnitType == UnitDuration {
    init(timeInterval: TimeInterval) {
        self.init(value: timeInterval, unit: .seconds)
    }
}
```

DateInterval

The `DateInterval` type, like `Date`, has a misleading name — it too represents moments in time rather than calendar dates. To convert date intervals into measurements, you can build on the approach we used for `TimeInterval`.

```
extension Measurement where UnitType == UnitDuration {
    init(_ dateInterval: DateInterval) {
        self.init(timeInterval: dateInterval.duration)
    }
}
```

DateComponents

`DateComponents` suffers from semantic overload in its design. It can represent either the calendar components for a moment in time (e.g. “day 1 of month 6 of year 2018”) or the distance between two moments in time relative to a calendar system (e.g. “4 days, 22 hours, and 3 minutes”).

Therefore, we have to be picky when converting `DateComponents` to a measure of duration. If there are any components other than nanoseconds, seconds, minutes, or hours, it's impossible to determine an absolute duration. We can enforce this by throwing an error or by simply making the initializer optional.

```
extension Measurement where UnitType == UnitDuration {
  init?(_ dateComponents: DateComponents) {
    guard dateComponents.era == nil,
        dateComponents.year == nil,
        dateComponents.month == nil,
        dateComponents.day == nil,
        dateComponents.weekday == nil,
        dateComponents.weekdayOrdinal == nil,
        dateComponents.weekOfYear == nil,
        dateComponents.yearForWeekOfYear == nil
    else {
        return nil
    }

    let nanoseconds = Measurement<UnitDuration>(
        value: Double(dateComponents.nanosecond ?? 0),
        unit: .nanoseconds
    )

    let seconds = Measurement<UnitDuration>(
        value: Double(dateComponents.second ?? 0),
        unit: .seconds
    )

    let minutes = Measurement<UnitDuration>(
        value: Double(dateComponents.minute ?? 0),
        unit: .minutes
    )

    let hours = Measurement<UnitDuration>(
        value: Double(dateComponents.minute ?? 0),
        unit: .hours
    )

    let total = nanoseconds + seconds + minutes + hours

    self.init(value: total.value, unit: total.unit)
  }
}
```

The only other step is to extend `UnitDuration` to provide a missing `nanoseconds` unit:

```
extension UnitDuration {
  class var nanoseconds: UnitDuration {
    let converter =
        UnitConverterLinear(coefficient: 0.0000000001)
    return .init(symbol: "ns", converter: converter)
  }
}
```

DateComponentsFormatter

Calendar units like years, months, and days may not have a place in science or engineering, but they're a convenient way to express relative time. `MeasurementFormatter` isn't equipped to do this, but you can extend `DateComponentsFormatter` to handle measurements:

```
extension DateComponentsFormatter {
  func string(from duration: Measurement<UnitDuration>)
        -> String?
  {
    let (seconds, fractionsOfASeconds) =
        modf(duration.converted(to: .seconds).value)
    let nanoseconds = fractionsOfASeconds * 1000000000
    let dateComponents =
        DateComponents(second: Int(seconds),
                       nanosecond: Int(nanoseconds))
    return string(from: dateComponents)
  }
}
```

Tip: The `modf` function returns a tuple containing the integer and fraction components of a floating point number.

Now we can express a duration measurement in more conventional terms:

```
let loggedFlyingTime =
    Measurement<UnitDuration>(value: 220, unit: .hours)

let formatter = DateComponentsFormatter()
formatter.allowedUnits = [.day]
formatter.unitsStyle = .full
formatter.includesApproximationPhrase = true
formatter.string(from: loggedFlyingTime) // About 9 days
```

Legacy Foundation Formatters

The first iteration of Foundation's units and measurement APIs came in iOS 8 and macOS 10.10, with `LengthFormatter`, `MassFormatter`, and `EnergyFormatter`.

They were useful in their time, but `MeasurementFormatter` has since rendered them obsolete. If you have code that uses these legacy APIs, bridging the formatter unit types to the corresponding `Dimension` is as simple as writing a `switch` statement:

```
extension UnitMass {
    convenience init(_ massFormatterUnit: MassFormatter.Unit) {
        var unit: UnitMass
        switch massFormatterUnit {
          case .gram: unit = .grams
          case .kilogram: unit = .kilograms
          case .ounce: unit = .ounces
          case .pound: unit = .pounds
          case .stone: unit = .stones
        }

        self.init(symbol: unit.symbol,
                  converter: unit.converter)
    }
}

// repeat for LengthFormatter and EnergyFormatter
```

Core Location

The Core Location framework takes the same lightweight `typealias` approach to units as `TimeInterval`:

```
typealias CLLocationDegrees = Double // degrees
typealias CLLocationDirection = Double // degrees
typealias CLLocationSpeed = Double // meters per second
typealias CLLocationDistance = Double // meters
typealias CLLocationAccuracy = Double // meters
```

All quantities in Core Location are expressed in terms of degrees, meters, or meters per second; the responsibility of localization is delegated to view controllers.

You can create a conditional extension on `Measurement` types with `UnitAngle`, `UnitLength`, or `UnitSpeed` unit types and declare an initializer that takes one of the corresponding CL-prefixed types:

```
extension Measurement where UnitType == UnitSpeed {
    init(clLocationSpeed: CLLocationSpeed) {
        self.init(value: clLocationSpeed,
                  unit: .metersPerSecond)
    }
}

// repeat for CLLocationDegrees, CLLocationDirection,
CLLocationDistance, and CLLocationAccuracy
```

HealthKit

HealthKit was introduced in iOS 8, at the same time as the aforementioned Foundation unit formatters. Unlike those formatters, though, HealthKit's unit APIs are very much still in use, and are unlikely to be replaced anytime soon.

`HKUnit` has a few things that Foundation doesn't. For one, `HKUnit` supports SI prefixes like giga-, kilo-, and milli-. It also has a few unit types not found in Foundation, including Siemens for electrical

conductance, and the pharmacology international unit (IU). But the most interesting feature of all is that `HKUnit` supports complex units — you can multiply and divide and take the reciprocal of any unit:

```
let meters = HKUnit.meterUnit()
let seconds = HKUnit.secondUnit()
let squaredSeconds = seconds.unitRaisedToPower(2)
let metersPerSecondSquared =
    meters.unitDividedByUnit(squaredSeconds)
```

Due to certain limitations of Swift's type system, `Unit` and `Dimension` don't support this functionality directly. However, as we'll explore in Chapter 5, we can get pretty far with a bit of elbow grease.

Recap

- Use `MeasurementFormatter` to create localized representations of units and measurements.
- By default, `MeasurementFormatter` expresses measurements in terms of the base unit for the unit type. To override this, set the `unitOptions` property to include the `providedUnit` option.
- Don’t specify the `temperatureWithoutUnit` without explicitly converting the temperature to the user’s preferred scale.
- You can use the `numberFormatter` property on `MeasurementFormatter` to configure the format and precision of the representation.

Chapter 5:

Using Playgrounds as an Interactive Calculator

Cochon des Cieux LLC offers a range of agricultural aviation services to Vermilion parish and the surrounding region. For years, the sight of their pilot, Bubba, flying his signature bright yellow biplane across the Bayou has been a fixture of the growing season.

Things get especially busy from mid-March through the end of April when farmers flood their paddy fields and plant the season's rice crop. Around these parts, it's mostly long-grain varieties — Cocodrie, LaKast, Presidio and the like.

The wet terrain of the Bayou in early spring make fields inaccessible to larger machinery. Many farmers, having learned their lesson the hard way, have come to prefer aerial application for seeding.[1] In fact, the latest job to come in was from a farmer who lost a tractor in the mud last season.

Bubba reckons that he should be able to pull off the job in a single load, though it's close enough that he'll want to make sure the numbers add up.

You see, the main challenge in agricultural aviation is getting the plane off the ground in the first place. You want to load up your hopper without exceeding the maximum takeoff weight. What makes this tough is that you need to factor in how much fuel you need and how much you can accomplish in a single load.

---

1. The term "cropdusting" typically only applies when spraying pesticides, and has fallen out of favor in general.

To determine our takeoff weight, we'll need to calculate the weight of our payload and fuel.

```
let emptyPlaneWeight: Measurement<UnitMass> // ?
let payloadWeight: Measurement<UnitMass> // ?
let fuelWeight: Measurement<UnitMass> // ?
let pilotWeight: Measurement<UnitMass> // ?
let maximumTakeoffWeight: Measurement<UnitMass> //?

let takeoffWeight = emptyPlaneWeight +
                    payloadWeight +
                    fuelWeight +
                    pilotWeight

let canTakeOff = takeoffWeight < maximumTakeoffWeight // ?!
```

We're going to expand on what we learned in Chapter 4 to perform preflight calculations with real-world measurements. The interactive coding environment of a Playground provides a unique way to experiment and visualize our results.

Defining our Known Quantities

The Grumman G-164 Ag Cat is a single-engine agricultural biplane with the following specifications:

Crew	1
Payload Capacity	400 US gal (1,514 liters)
Fuel Tank Capacity	46 US gal (174.12 liters)
Length	27 ft 7¼ in (8.41 m)
Wingspan	42 ft 4½ in (12.92 m)
Wing Area	392.7 ft² (36.48 m²)
Height	12 ft 1 in (3.68 m)
Empty Weight	3,150 lb (1,429 kg)
Maximum Takeoff Weight	7,020 lb (3,184 kg)

Right off the bat, we know our empty plane weight and maximum takeoff weight.

```
let emptyPlaneWeight =
    Measurement<UnitMass>(value: 3150, unit: .pounds)
let maximumTakeoffWeight =
    Measurement<UnitMass>(value: 7020, unit: .pounds)
```

Our pilot is a warthog 🐗, weighing in from tusk to tail at about 75 kilograms. Adding clothing, personal effects, and fib factor, we'll round it up to 100 kg (just to be safe).

```
let pilotWeight =
    Measurement<UnitMass>(value: 100, unit: .kilograms)
```

Calculating Unknown Quantities

Sure, we could formalize our problem in terms of a system of equations and use a constraint solver to run the numbers for us… but that sounds like a lot of work. Too fancy for a plain-spoken warthog, certainly. Besides, most farmers prefer to have everything written out as a sequence of calculations. It's just easier to reason about things that way.

Deriving Compound Unit Measurements

One of the first calculations we'll want to make is to see how big of a field we're working with. The farm encompasses an area 400 yards on its long edge and 250 yards on its short edge.

```
let fieldLength =
    Measurement<UnitLength>(value: 400, unit: .yards)
let fieldWidth =
    Measurement<UnitLength>(value: 250, unit: .yards)
```

As mentioned in Chapter 4, Foundation's unit and measurement APIs don't support proper unit composition. However, we can get close enough by defining operators for the combinations of units that we're interested in. Like calculating area as the product of two lengths:

```
func *(lhs: Measurement<UnitLength>,
       rhs: Measurement<UnitLength>) -> Measurement<UnitArea> {
   return .init(value: lhs.converted(to: .meters).value *
                       rhs.converted(to: .meters).value,
                unit: UnitArea.squareMeters)
}
```

We normalize both lengths into meters and multiply the values together to arrive at the product in square meters.

To express the result in different terms, we simply call the `converted(to:)` method on the result.

```
let fieldArea =
    (fieldLength* fieldWidth).converted(to: .acres) // 20.66 ac
```

Canceling Units

Agricultural flying is closer to a dogfight than piloting a commercial airliner. The low-altitude runs and precision turns involved are equal parts thrilling and tedious, requiring a pilot with both skill and patience.

A typical airspeed for aerial application is 100 knots. This might not seem very fast, but it sure feels like it when you're only a few feet off the ground.

```
let speed =
    Measurement<UnitSpeed>(value: 100, unit: .knots)
```

But before you can get to work, you have to travel to the job site first. This is called *ferrying*.

```
let ferryLength =
    Measurement<UnitLength>(value: 10, unit: .miles)
let ferryDuration = (ferryLength / speed) * 2
```

Speed is a unit of length over time. When you divide length by speed the length units cancel each other out, resulting in a time measurement. Therefore, we can calculate the ferry duration by taking the distance we need to travel and dividing by the speed of the aircraft.

Once they arrive, a pilot always flies around the perimeter at least once (no matter how familiar they are with a field) This allows them to check for obstacles like poles, wires, and trees.

```
let flyAroundLength = ((fieldLength * 2) + (fieldWidth * 2))
let flyAroundDuration = (flyAroundLength / speed)
```

Calculating Scalar Quantities

After getting a lay of the land, the pilot lines up to make their first run. They find a reference point on the horizon and align themselves to be perpendicular with the leading edge of the field and facing away from the sun. Whenever possible, flight lines are cross-wind, which helps make the coverage more uniform.

Note: Agricultural pilots should set the trim tabs to be slightly down, such that a neutral yoke yields a gentle climb. This makes it less likely for a pilot crash to the ground should they look off and get distracted.

For an eccentric rectangular plot like this one, the direction of the prevailing winds can make quite a difference in how many runs are necessary to cover the field.

```
var applicationLength: Measurement<UnitLength>
var applicationWidth: Measurement<UnitLength>

(applicationLength, applicationWidth) =
    (fieldWidth, fieldLength) // best-case

(applicationLength, applicationWidth) =
    (fieldLength, fieldWidth) // worst-case
```

The best case is for the plane to fly parallel to the longest edge, which minimizes the number of passes. However, hope is not a strategy, and luck shouldn't be a factor for our success, so we'll make our calculations based on the maximum expected flying time.

By dividing the field width by the swath width of the plane (40 ft.) and rounding up, we can get the smallest whole number of passes needed to cover the field.

First, we need to extend `Measurement` to implement a division operator for measurements with the same unit type. Units cancel out, and the result is a scalar — in this case, a `Double`:

```
extension Measurement {
    public static func / <T>(lhs: Measurement<T>, rhs:
Measurement<T>) -> Double where T: Dimension {
        return lhs.value / rhs.converted(to: lhs.unit).value
    }
}
```

Our aircraft is configured with a swath width of 40 ft, which is standard for aerial seeding. Performing the calculation, we see that the minimum number of passes is 30.

```
let swathWidth = Measurement<UnitLength>(value: 40, unit: .feet)
let swathPasses = ceil(applicationWidth / swathWidth) // 30
```

Factoring in trim passes along the edges and cleanup passes, we bump the estimate up to 34.

```
let trimPasses = 2.0
let cleanupPasses = ceil(0.05 * swathPasses) // 5% fudge factor

let totalPasses = swathPasses + trimPasses + cleanupPasses // 34
```

Tip: The numbers of passes are integers, but we're representing them with `Double` instead of `Int` to avoid conversions later on.

Once a pilot finishes a pass, they pull up, turn down-wind, and set up their next run. Turning is the most common maneuver during aerial applications. It's also the most dangerous. That's why it's important for pilots to exercise caution and resist the urge to make steep banking turns.

In aviation, the standard rate turn is 3° per second. Each turn is a full 180°, which takes one minute.

```
let standardTurnDuration =
    Measurement<UnitDuration>(value: 1.0, unit: .minutes)
```

Compared to the amount of time spent flying over a field, you can see why pilots might be tempted to cut corners:

```
let passDuration = (applicationLength / speed) // 4.4 s
```

We can calculate the total amount of time in the air by multiplying the number of passes by the duration of each pass and factoring in the time spent en route:

```
let totalDuration =
    (totalPasses *
        (passDuration + standardTurnDuration) +
      ferryDuration +
      flyAroundDuration
    ).converted(to: .minutes) // 47.24 min
```

Calculating Quantities from Rates

Why are we so interested in the flight time? Because that'll tell us how much fuel we need. Unlike cars and trucks, plane fuel efficiency is rated in gallons per hour.

Speaking of automobiles, `UnitFuelEfficiency` defines units for miles per gallon and liters per 100 kilometers, which is how fuel efficiency is most commonly expressed in the United States and the rest of the world. Unfortunately, these compound units are mutually incompatible: the former is length over volume and the latter is volume over length. We can't reuse our trick from before and define a custom multiplication operator, because the units won't work out consistently — especially not if we add volume over time to our definition of fuel efficiency.

Instead, let's declare a new `Rate` type to encapsulate these kinds of conversion rates:

```
struct Rate<Numerator, Denominator>
    where Numerator: Unit, Denominator: Unit
{
    var value: Double
    var numeratorUnit: Numerator
    var denominatorUnit: Denominator
}
```

A `Rate` structure has associated `Numerator` and `Denominator` unit types, and is initialized with a constant factor and the specific units.

Here's where it gets interesting:

```
public func * <T, U>(lhs: Rate<T,U>,
                     rhs: Measurement<U>) -> Measurement<T>
    where T: Dimension, U: Dimension
{
    return .init(value: Measurement(value: lhs.value,
                                     unit: rhs.unit)
                            .converted(to: lhs.denominatorUnit)
                            .value *
                        rhs.value,
                 unit: lhs.numeratorUnit)
}
```

Multiplying a measurement with one unit type by a rate whose denominator is that same unit type causes those types to cancel out, resulting in a measurement with the numerator type. For example, distance over time multiplied by time yields distance. A generic function allows us to implement this for all permutations of rates and measurements for which this holds.

Combined with our manual overrides for known derivations of compound units, we have pretty much everything that we'd get from a more formalized solution. It's not going to impress Haskell or OCaml enthusiasts, but hey — it works for us!

Let's kick the tires by using `Rate` to calculate our fuel consumption:

```
let fuelConsumptionRate =
    Rate<UnitVolume, UnitDuration>(value: 50,
                                   unit: .gallons,
                                   per: .hours)

var volumeOfFuel = totalDuration *
                   fuelConsumptionRate // 39.12 gal
```

Wait, how much can our tank hold, again?

```
let fuelTankCapacity =
    Measurement<UnitVolume>(value: 46, unit: .gallons)
requiredFuelVolume < fuelTankCapacity // true
```

Alright, we're good in terms of fuel capacity. Heck, we're close enough that we might as well top it off.

```
fuelVolume = fuelTankCapacity
```

How much does that contribute to our weight limit? Avgas weighs about 6 pounds per gallon, so...

```
let fuelWeightRate =
    Rate<UnitMass, UnitVolume>(value: 6,
                               unit: .pounds,
                               per: .gallons)

let fuelWeight = fuelWeightRate * fuelVolume // 276.0 lb
```

Our last step is to calculate how much payload we can carry. That seems pretty important; should we have maybe started with that instead? Yeah, probably. (Oh well!)

Some rice farmers use a slurry of wet seeds in their aerial application, but farmers around these parts have had success with dry seeding. One advantage is that dry is lot lighter than wet, so you can reduce the seeding rates to as low as 120 pounds per acre.

```
let payloadWeightRate =
    Rate<UnitMass, UnitArea>(value: 120,
                             unit: .pounds,
                             per: .acres)
```

To determine the weight of our payload, all we need to do is multiply the rate by the acreage calculated at the start:

```
let payloadWeight = payloadWeightRate * fieldArea // 2479.33 lb
```

With all of our unknowns now known, now is the moment of truth: will we be able to take off?

```
let takeoffWeight = (
    weightOfFuel +
    payloadWeight +
    pilotWeight +
    planeWeight).converted(to: .pounds) // 6125.79 lb

takeoffWeight < maximumTakeoffWeight // true
```

Huzzah! So there you have it. By our calculations, this pig can fly — with a healthy margin of error. 

The sample code for this chapter includes definitions for a dozen new unit types and operators for pretty much every unit conversion you can think of. So feel free to play around with these!

If you're looking for a challenge, here are some problems that you can try to work out:

- Approximately how many seed granules are in the hopper?
- How many seeds are dispersed per second during application?
- How fast does the propeller need to go in order to generate the force necessary for takeoff?
- How much G-force is acting on the plane when it makes its final turn into the first swath pass over the field, assuming a coordinated standard rate turn?

Recap

- Swift is limited in its ability to express composite unit relationships in the type system. However, we can closely approximate much of this functionality by defining operators for known combinations of unit types.
- Xcode playgrounds can be useful for performing dimensional calculations — especially when related functionality will eventually be implemented in code.