CHAPTER 2

Predeclared Types and Declarations

Now that you have your development environment set up, it's time to start looking at Go's language features and how to best use them. When trying to figure out what "best" means, there is one overriding principle: write your programs in a way that makes your intentions clear. As I go through features and discuss the options, I'll explain why I find that a particular approach produces clearer code.

I'll start by looking at the types that are built into Go and how to declare variables of those types. While every programmer has experience with these concepts, Go does some things differently, and subtle differences exist between Go and other languages.

The Predeclared Types

Go has many types built into the language. These are called *predeclared* types. They are similar to types that are found in other languages: booleans, integers, floats, and strings. Using these types idiomatically is sometimes a challenge for developers who are transitioning from another language. You'll look at these types and see how they work best in Go. Before I review the types, let's cover some of the concepts that apply to all types.

The Zero Value

Go, like most modern languages, assigns a default *zero value* to any variable that is declared but not assigned a value. Having an explicit zero value makes code clearer and removes a source of bugs found in C and C++ programs. As I talk about each type, I will also cover the zero value for the type. You can find details on the zero value in The Go Programming Language Specification.

Literals

A Go *literal* is an explicitly specified number, character, or string. Go programs have four common kinds of literals. (I'll cover a rare fifth kind of literal when discussing complex numbers.)

An *integer literal* is a sequence of numbers. Integer literals are base 10 by default, but different prefixes are used to indicate other bases: 0b for binary (base 2), 0o for octal (base 8), or 0x for hexadecimal (base 16). You can use either upper- or lowercase letters for the prefix. A leading 0 with no letter after it is another way to represent an octal literal. Do not use it, as it is very confusing.

To make it easier to read longer integer literals, Go allows you to put underscores in the middle of your literal. This allows you to, for example, group by thousands in base 10 (1_234). These underscores have no effect on the value of the number. The only limitations on underscores are that they can't be at the beginning or end of numbers, and you can't have them next to each other. You could put an underscore between every digit in your literal (1_2_3_4), but don't. Use them to improve readability by breaking up base 10 numbers at the thousands place or to break up binary, octal, or hexadecimal numbers at 1-, 2-, or 4-byte boundaries.

A *floating-point literal* has a decimal point to indicate the fractional portion of the value. They can also have an exponent specified with the letter `e` and a positive or negative number (such as 6.03e23). You also have the option to write them in hexadecimal by using the 0x prefix and the letter `p` for indicating any exponent (0x12.34p5, which is equal to 582.5 in base 10). As integer literals, you can use underscores to format your floating-point literals.

A *rune literal* represents a character and is surrounded by single quotes. Unlike many other languages, in Go single quotes and double quotes are *not* interchangeable. Rune literals can be written as single Unicode characters (`'a'`), 8-bit octal numbers (`'\141'`), 8-bit hexadecimal numbers (`'\x61'`), 16-bit hexadecimal numbers (`'\u0061'`), or 32-bit Unicode numbers (`'\U00000061'`). There are also several backslash-escaped rune literals, with the most useful ones being newline (`'\n'`), tab (`'\t'`), single quote (`'\''`), and backslash (`'\\'`).

Practically speaking, use base 10 to represent your integer and floating-point literals. Octal representations are rare, mostly used to represent POSIX permission flag values (such as 0o777 for rwxrwxrwx). Hexadecimal and binary are sometimes used for bit filters or networking and infrastructure applications. Avoid using any of the numeric escapes for rune literals, unless the context makes your code clearer.

There are two ways to indicate *string literals*. Most of the time, you should use double quotes to create an *interpreted string literal* (e.g., type **`"Greetings and Salutations"`**). These contain zero or more rune literals. They are called "interpreted" because they interpret rune literals (both numeric and backslash escaped) into single characters.

One rune literal backslash escape is not legal in a string literal: the single quote escape. It is replaced by a backslash escape for double quotes.

The only characters that cannot appear in an interpreted string literal are unescaped backslashes, unescaped newlines, and unescaped double quotes. If you use an interpreted string literal and want your greetings on a different line from your salutations and want "Salutations" to appear in quotes, you need to type **`"Greetings and\n\"Salutations\""`**.

If you need to include backslashes, double quotes, or newlines in your string, using a *raw string literal* is easier. These are delimited with backquotes (`` ` ``) and can contain any character except a backquote. There's no escape character in a raw string literal; all characters are included as is. When using a raw string literal, you write a multiline greeting like so:

```
`Greetings and
"Salutations"`
```

Literals are considered *untyped*. I'll explore this concept more in "Literals Are Untyped" on page 27. As you will see in "var Versus :=" on page 28, there are situations in Go where the type isn't explicitly declared. In those cases, Go uses the *default type* for a literal; if there's nothing in the expression that makes clear what the type of the literal is, the literal defaults to a type. I will mention the default type for literals when discussing the different predeclared types.

Booleans

The `bool` type represents Boolean variables. Variables of `bool` type can have one of two values: `true` or `false`. The zero value for a `bool` is `false`:

```go
var flag bool // no value assigned, set to false
var isAwesome = true
```

It's hard to talk about variable types without showing a variable declaration, and vice versa. I'll use variable declarations first and describe them in "var Versus :=" on page 28.

Numeric Types

Go has a large number of numeric types: 12 types (and a few special names) that are grouped into three categories. If you are coming from a language like JavaScript that gets along with only a single numeric type, this might seem like a lot. And in fact, some types are used frequently while others are more esoteric. I'll start by looking at integer types before moving on to floating-point types and the very unusual complex type.

Integer types

Go provides both signed and unsigned integers in a variety of sizes, from one to eight bytes. They are shown in Table 2-1.

Table 2-1. The integer types in Go

Type name	Value range
`int8`	–128 to 127
`int16`	–32768 to 32767
`int32`	–2147483648 to 2147483647
`int64`	–9223372036854775808 to 9223372036854775807
`uint8`	0 to 255
`uint16`	0 to 65535
`uint32`	0 to 4294967295
`uint64`	0 to 18446744073709551615

It might be obvious from the name, but the zero value for all of the integer types is `0`.

The special integer types

Go does have some special names for integer types. A `byte` is an alias for `uint8`; it is legal to assign, compare, or perform mathematical operations between a `byte` and a `uint8`. However, you rarely see `uint8` used in Go code; just call it a `byte`.

The second special name is `int`. On a 32-bit CPU, `int` is a 32-bit signed integer like an `int32`. On most 64-bit CPUs, `int` is a 64-bit signed integer, just like an `int64`. Because `int` isn't consistent from platform to platform, it is a compile-time error to assign, compare, or perform mathematical operations between an `int` and an `int32` or `int64` without a type conversion (see "Explicit Type Conversion" on page 26 for more details). Integer literals default to being of `int` type.

Some uncommon 64-bit CPU architectures use a 32-bit signed integer for the `int` type. Go supports three of them: `amd64p32`, `mips64p32`, and `mips64p32le`.

The third special name is `uint`. It follows the same rules as `int`, only it is unsigned (the values are always 0 or positive).

There are two other special names for integer types, `rune` and `uintptr`. You looked at rune literals earlier and I'll discuss the `rune` type in "A Taste of Strings and Runes" on page 25 and `uintptr` in Chapter 16.

Choosing which integer to use

Go provides more integer types than some other languages. Given all these choices, you might wonder when you should use each of them. You should follow three simple rules:

- If you are working with a binary file format or network protocol that has an integer of a specific size or sign, use the corresponding integer type.
- If you are writing a library function that should work with any integer type, take advantage of Go's generics support and use a generic type parameter to represent any integer type (I talk more about functions and their parameters in Chapter 5 and more about generics in Chapter 8.)
- In all other cases, just use `int`.

You'll likely find legacy code where there's a pair of functions that do the same thing, but one has `int64` for the parameters and variables and the other has `uint64`. The reason is that the API was created before generics were added to Go. Without generics, you needed to write functions with slightly different names to implement the same algorithm with different types. Using `int64` and `uint64` meant that you could write the code once and let callers use type conversions to pass values in and convert data that's returned.

You can see this pattern in the Go standard library within the functions `FormatInt` and `FormatUint` in the `strconv` package.

Integer operators

Go integers support the usual arithmetic operators: +, `-`, `*`, `/`, with `%` for modulus. The result of an integer division is an integer; if you want to get a floating-point result, you need to use a type conversion to make your integers into floating-point numbers.

Also, be careful not to divide an integer by 0; this causes a panic (I talk more about panics in "panic and recover" on page 218).

Integer division in Go follows truncation toward zero; see the Go spec's section on arithmetic operators for the full details.

You can combine any of the arithmetic operators with = to modify a variable: +=, -=, *=, /=, and %=. For example, the following code results in x having the value 20:

```go
var x int = 10
x *= 2
```

You compare integers with ==, !=, >, >=, <, and <=.

Go also has bit-manipulation operators for integers. You can bit shift left and right with << and >>, or do bit masks with & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), and &^ (bitwise AND NOT). As with the arithmetic operators, you can also combine all the bitwise operators with = to modify a variable: &=, |=, ^=, &^=, <<=, and >>=.

Floating-point types

Go has two floating-point types, as shown in Table 2-2.

Table 2-2. The floating-point types in Go

Type name	Largest absolute value	Smallest (nonzero) absolute value
`float32`	3.40282346638528859811704183484516925440e+38	1.401298464324817070923729583289916131280e-45
`float64`	1.797693134862315708145274237317043567981e+308	4.940656458412465441765687928682213723651e-324

Like the integer types, the zero value for the floating-point types is 0.

Floating point in Go is similar to floating-point math in other languages. Go uses the IEEE 754 specification, giving a large range and limited precision. Picking which floating-point type to use is straightforward: unless you have to be compatible with an existing format, use `float64`. Floating-point literals have a default type of `float64`, so always using `float64` is the simplest option. It also helps mitigate floating-point accuracy issues since a `float32` has only six- or seven-decimal digits of precision. Don't worry about the difference in memory size unless you have used the profiler to determine that it is a significant source of problems. (Testing and profiling are covered in Chapter 15.)

The bigger question is whether you should be using a floating-point number at all. In many cases, the answer is no. Just like other languages, Go floating-point numbers have a huge range, but they cannot store every value in that range; they store the nearest approximation. Because floats aren't exact, they can be used only in situations where inexact values are acceptable or the rules of floating point are well understood. That limits them to things like graphics, statistics, and scientific operations.

A floating-point number cannot represent a decimal value exactly. Do not use them to represent money or any other value that must have an exact decimal representation! You'll look at a third-party module for handling exact decimal values in "Importing Third-Party Code" on page 240.

IEEE 754

As mentioned earlier, Go (and most other programming languages) stores floating-point numbers using a specification called IEEE 754.

The actual rules are outside the scope of this book, and they aren't straightforward. You can learn more about IEEE 754 from The Floating Point Guide.

You can use all the standard mathematical and comparison operators with floats, except `%`. Floating-point division has a couple of interesting properties. Dividing a nonzero floating-point variable by 0 returns `+Inf` or `-Inf` (positive or negative infinity), depending on the sign of the number. Dividing a floating-point variable set to 0 by 0 returns `NaN` (Not a Number).

While Go lets you use `==` and `!=` to compare floats, don't do it. Because of the inexact nature of floats, two floating-point values might not be equal when you think they should be. Instead, define a maximum allowed variance and see if the difference between two floats is less than that. This value (sometimes called *epsilon*) depends on your accuracy needs; I can't give you a simple rule. If you aren't sure, consult your friendly local mathematician for advice. If you can't find one, The Floating Point Guide has a "Comparison" page that can help you out (or possibly convince you to avoid floating-point numbers unless absolutely necessary).

Complex types (you're probably not going to use these)

There is one more numeric type and it is pretty unusual. Go has first-class support for complex numbers. If you don't know what complex numbers are, you are not the target audience for this feature; feel free to skip ahead.

There isn't a lot to the complex number support in Go. Go defines two complex number types. `complex64` uses `float32` values to represent the real and imaginary part, and `complex128` uses `float64` values. Both are declared with the `complex` built-in function:

```
var complexNum = complex(20.3, 10.2)
```

Go uses a few rules to determine the type of the value returned by `complex`:

- If you use untyped constants or literals for both function parameters, you'll create an untyped complex literal, which has a default type of `complex128`.
- If both values passed into `complex` are of `float32` type, you'll create a `complex64`.
- If one value is a `float32` and the other value is an untyped constant or literal that can fit within a `float32`, you'll create a `complex64`.
- Otherwise, you'll create a `complex128`.

All the standard floating-point arithmetic operators work on complex numbers. Just as with floats, you can use `==` or `!=` to compare them, but they have the same precision limitations, so it's best to use the epsilon technique. You can extract the real and imaginary portions of a complex number with the `real` and `imag` built-in functions, respectively. The `math/cmplx` package has additional functions for manipulating `complex128` values.

The zero value for both types of complex numbers has 0 assigned to both the real and imaginary portions of the number.

Example 2-1 shows a simple program that demonstrates how complex numbers work. You can run it for yourself on The Go Playground or in the *sample_code/complex_numbers* directory in the Chapter 2 repository.

Example 2-1. Complex numbers

```
func main() {
    x := complex(2.5, 3.1)
    y := complex(10.2, 2)
    fmt.Println(x + y)
    fmt.Println(x - y)
    fmt.Println(x * y)
    fmt.Println(x / y)
    fmt.Println(real(x))
    fmt.Println(imag(x))
    fmt.Println(cmplx.Abs(x))
}
```

Running this code gives you the following:

```
(12.7+5.1i)
(-7.699999999999999+1.1i)
(19.3+36.62i)
(0.2934098482043688+0.24639022584228065i)
2.5
3.1
3.982461550347975
```

You can see floating-point imprecision on display here too.

In case you were wondering what the fifth kind of primitive literal was, Go supports imaginary literals to represent the imaginary portion of a complex number. They look just like floating-point literals, but they have an `i` for a suffix.

Despite having complex numbers as a predeclared type, Go is not a popular language for numerical computing. Adoption has been limited because other features (like matrix support) are not part of the language and libraries have to use inefficient replacements, like slices of slices. (You'll look at slices in Chapter 3 and how they are implemented in Chapter 6.) But if you need to calculate a Mandelbrot set as part of a larger program, or implement a quadratic equation solver, complex number support is there for you.

You might be wondering why Go includes complex numbers. The answer is simple: Ken Thompson, one of the creators of Go (and Unix), thought they would be interesting. There has been discussion about removing complex numbers from a future version of Go, but it's easier to just ignore the feature.

If you do want to write numerical computing applications in Go, you can use the third-party Gonum package. It takes advantage of complex numbers and provides useful libraries for things like linear algebra, matrices, integration, and statistics. But you should consider other languages first.

A Taste of Strings and Runes

This brings us to strings. Like most modern languages, Go includes strings as a built-in type. The zero value for a string is the empty string. Go supports Unicode; as I showed "Literals" on page 18, you can put any Unicode character into a string. Like integers and floats, strings are compared for equality using ==, difference with !=, or ordering with >, >=, <, or <=. They are concatenated by using the + operator.

Strings in Go are immutable; you can reassign the value of a string variable, but you cannot change the value of the string that is assigned to it.

Go also has a type that represents a single code point. The *rune* type is an alias for the `int32` type, just as `byte` is an alias for `uint8`. As you could probably guess, a rune literal's default type is a rune, and a string literal's default type is a string.

If you are referring to a character, use the rune type, not the `int32` type. They might be the same to the compiler, but you want to use the type that clarifies the intent of your code:

```
var myFirstInitial rune = 'J' // good - the type name matches the usage
var myLastInitial int32 = 'B' // bad - legal but confusing
```

I am going to talk a lot more about strings in the next chapter, covering some implementation details, relationships with bytes and runes, as well as advanced features and pitfalls.

Explicit Type Conversion

Most languages that have multiple numeric types automatically convert from one to another when needed. This is called *automatic type promotion*, and while it seems very convenient, it turns out that the rules to properly convert one type to another can get complicated and produce unexpected results. As a language that values clarity of intent and readability, Go doesn't allow automatic type promotion between variables. You must use a *type conversion* when variable types do not match. Even different-sized integers and floats must be converted to the same type to interact. This makes it clear exactly what type you want without having to memorize any type conversion rules (see Example 2-2).

Example 2-2. Type conversions

```
var x int = 10
var y float64 = 30.2
var sum1 float64 = float64(x) + y
var sum2 int = x + int(y)
fmt.Println(sum1, sum2)
```

In this sample code, you define four variables. `x` is an `int` with the value 10, and `y` is a `float64` with the value 30.2. Since these are not identical types, you need to convert them to add them together. For `sum1`, you convert `x` to a `float64` using a `float64` type conversion, and for `sum2`, you convert `y` to an `int` using an `int` type conversion. When you run this code, it prints out 40.2 40.

The same behavior applies with different-sized integer types (see Example 2-3).

Example 2-3. Integer type conversions

```go
var x int = 10
var b byte = 100
var sum3 int = x + int(b)
var sum4 byte = byte(x) + b
fmt.Println(sum3, sum4)
```

You can run these examples on The Go Playground or in the *sample_code/type_conversion* directory in the Chapter 2 repository.

This strictness around types has other implications. Since all type conversions in Go are explicit, you cannot treat another Go type as a boolean. In many languages, a nonzero number or a nonempty string can be interpreted as a boolean `true`. Just like automatic type promotion, the rules for "truthy" values vary from language to language and can be confusing. Unsurprisingly, Go doesn't allow truthiness. In fact, *no other type can be converted to a bool, implicitly or explicitly*. If you want to convert from another data type to boolean, you must use one of the comparison operators (`==`, `!=`, `>`, `<`, `<=`, or `>=`). For example, to check if variable `x` is equal to 0, the code would be `x == 0`. If you want to check if string `s` is empty, use `s == ""`.

Type conversions are one of the places where Go chooses to add a little verbosity in exchange for a great deal of simplicity and clarity. You'll see this trade-off multiple times. Idiomatic Go values comprehensibility over conciseness.

Literals Are Untyped

While you can't add two integer variables together if they are declared to be of different types of integers, Go lets you use an integer literal in floating-point expressions or even assign an integer literal to a floating-point variable:

```go
var x float64 = 10
var y float64 = 200.3 * 5
```

This is because, as I mentioned earlier, literals in Go are untyped. Go is a practical language, and it makes sense to avoid forcing a type until the developer specifies one. This means they can be used with any variable whose type is compatible with the literal. When you look at user-defined types in Chapter 7, you'll see that you can even use literals with user-defined types based on predefined types. Being untyped goes only so far; you can't assign a literal string to a variable with a numeric type or a literal number to a string variable, nor can you assign a float literal to an `int`. These are all flagged by the compiler as errors. Size limitations also exist; while you can write numeric literals that are larger than any integer can hold, it is a compile-time error to try to assign a literal whose value overflows the specified variable, such as trying to assign the literal 1000 to a variable of type `byte`.

var Versus :=

For a small language, Go has a lot of ways to declare variables. There's a reason for this: each declaration style communicates something about how the variable is used. Let's go through the ways you can declare a variable in Go and see when each is appropriate.

The most verbose way to declare a variable in Go uses the `var` keyword, an explicit type, and an assignment. It looks like this:

```go
var x int = 10
```

If the type on the righthand side of the = is the expected type of your variable, you can leave off the type from the left side of the =. Since the default type of an integer literal is `int`, the following declares `x` to be a variable of type `int`:

```go
var x = 10
```

Conversely, if you want to declare a variable and assign it the zero value, you can keep the type and drop the = on the righthand side:

```go
var x int
```

You can declare multiple variables at once with `var`, and they can be of the same type:

```go
var x, y int = 10, 20
```

You can declare all zero values of the same type:

```go
var x, y int
```

or of different types:

```go
var x, y = 10, "hello"
```

There's one more way to use `var`. If you are declaring multiple variables at once, you can wrap them in a *declaration list*:

```go
var (
    x    int
    y        = 20
    z    int = 30
    d, e     = 40, "hello"
    f, g string
)
```

Go also supports a short declaration and assignment format. When you are within a function, you can use the `:=` operator to replace a `var` declaration that uses type inference. The following two statements do exactly the same thing—they declare `x` to be an `int` with the value of 10:

```
var x = 10
x := 10
```

As with `var`, you can declare multiple variables at once using `:=`. These two lines both assign 10 to `x` and "hello" to `y`:

```
var x, y = 10, "hello"
x, y := 10, "hello"
```

The `:=` operator can do one trick that you cannot do with `var`: it allows you to assign values to existing variables too. As long as at least one new variable is on the lefthand side of the `:=`, any of the other variables can already exist:

```
x := 10
x, y := 30, "hello"
```

Using `:=` has one limitation. If you are declaring a variable at the package level, you must use `var` because `:=` is not legal outside of functions.

How do you know which style to use? As always, choose what makes your intent clearest. The most common declaration style within functions is `:=`. Outside of a function, use declaration lists on the rare occasions when you are declaring multiple package-level variables.

In some situations within functions, you should avoid `:=`:

- When initializing a variable to its zero value, use `var x int`. This makes it clear that the zero value is intended.
- When assigning an untyped constant or a literal to a variable and the default type for the constant or literal isn't the type you want for the variable, use the long `var` form with the type specified. While it is legal to use a type conversion to specify the type of the value and use `:=` to write `x := byte(20)`, it is idiomatic to write `var x byte = 20`.
- Because `:=` allows you to assign to both new and existing variables, it sometimes creates new variables when you think you are reusing existing ones (see "Shadowing Variables" on page 68 for details). In those situations, explicitly declare all your new variables with `var` to make it clear which variables are new, and then use the assignment operator (`=`) to assign values to both new and old variables.

While `var` and `:=` allow you to declare multiple variables on the same line, use this style only when assigning multiple values returned from a function or the comma ok idiom (see Chapter 5 and "The comma ok Idiom" on page 58).

You should rarely declare variables outside of functions, in what's called the *package block* (see "Blocks" on page 67). Package-level variables whose values change are a bad idea. When you have a variable outside of a function, it can be difficult to track the changes made to it, which makes it hard to understand how data is flowing through your program. This can lead to subtle bugs. As a general rule, you should only declare variables in the package block that are effectively immutable.

Avoid declaring variables outside of functions because they complicate data flow analysis.

You might be wondering: does Go provide a way to *ensure* that a value is immutable? It does, but it is a bit different from what you may have seen in other programming languages. It's time to learn about `const`.

Using const

Many languages have a way to declare a value as immutable. In Go, this is done with the `const` keyword. At first glance, it seems to work exactly as it would in other languages. Try out the code in Example 2-4 on The Go Playground or in the *sample_code/const_declaration* directory in the Chapter 2 repository.

Example 2-4. `const` declarations

```go
package main

import "fmt"

const x int64 = 10

const (
    idKey   = "id"
    nameKey = "name"
)

const z = 20 * 10

func main() {
    const y = "hello"

    fmt.Println(x)
    fmt.Println(y)

    x = x + 1 // this will not compile!
    y = "bye" // this will not compile!
    fmt.Println(x)
    fmt.Println(y)
}
```

When you run this code, compilation fails with the following error messages:

```
./prog.go:20:2: cannot assign to x (constant 10 of type int64)
./prog.go:21:2: cannot assign to y (untyped string constant "hello")
```

As you see, you declare a constant at the package level or within a function. Just as with `var`, you can (and should) declare a group of related constants within a set of parentheses.

Be aware that `const` in Go is very limited. Constants in Go are a way to give names to literals. They can only hold values that the compiler can figure out at compile time. This means that they can be assigned:

- Numeric literals
- `true` and `false`
- Strings
- Runes
- The values returned by the built-in functions `complex`, `real`, `imag`, `len`, and `cap`
- Expressions that consist of operators and the preceding values

I'll cover the `len` and `cap` functions in the next chapter. Another value that can be used with `const` is called `iota`. I'll talk about `iota` when I discuss creating your own types in Chapter 7.

Go doesn't provide a way to specify that a value calculated at runtime is immutable. For example, the following code will fail to compile with the error `x + y (value of type int) is not constant`:

```
x := 5
y := 10
const z = x + y // this won't compile!
```

As you'll see in the next chapter, there are no immutable arrays, slices, maps, or structs, and there's no way to declare that a field in a struct is immutable. This is less limiting than it sounds. Within a function, it is clear if a variable is being modified, so immutability is less important. In "Go Is Call by Value" on page 114, you'll see how Go prevents modifications to variables that are passed as parameters to functions.

Constants in Go are a way to give names to literals. There is *no* way in Go to declare that a variable is immutable.

Typed and Untyped Constants

Constants can be typed or untyped. An untyped constant works exactly like a literal; it has no type of its own but does have a default type that is used when no other type can be inferred. A typed constant can be directly assigned only to a variable of that type.

Whether to make a constant typed depends on why the constant was declared. If you are giving a name to a mathematical constant that could be used with multiple numeric types, keep the constant untyped. In general, leaving a constant untyped gives you more flexibility. In certain situations, you'll want a constant to enforce a type. You'll see a use for typed constants when I cover enumerations with `iota` in "iota Is for Enumerations—Sometimes" on page 152.

Here's what an untyped constant declaration looks like:

```
const x = 10
```

All of the following assignments are legal:

```
var y int = x
var z float64 = x
var d byte = x
```

Here's what a typed constant declaration looks like:

```
const typedX int = 10
```

This constant can be assigned directly only to an `int`. Assigning it to any other type produces a compile-time error like this:

```
cannot use typedX (type int) as type float64 in assignment
```

Unused Variables

One of the goals for Go is to make it easier for large teams to collaborate on programs. To do so, Go has some rules that are unique among programming languages. In Chapter 1, you saw that Go programs need to be formatted in a specific way with `go fmt` to make it easier to write code-manipulation tools and to provide coding standards. Another Go requirement is that *every declared local variable must be read*. It is a *compile-time error* to declare a local variable and to not read its value.

The compiler's unused variable check is not exhaustive. As long as a variable is read once, the compiler won't complain, even if there are writes to the variable that are never read. The following is a valid Go program that you can run on The Go Playground or in the *sample_code/assignments_not_read* directory in the Chapter 2 repository:

```
func main() {
    x := 10 // this assignment isn't read!
    x = 20
    fmt.Println(x)
    x = 30 // this assignment isn't read!
}
```

While the compiler and `go vet` do not catch the unused assignments of 10 and 30 to `x`, third-party tools can detect them. I'll talk about these tools in "Using Code-Quality Scanners" on page 267.

The Go compiler won't stop you from creating unread package-level variables. This is one more reason you should avoid creating package-level variables.

Unused Constants

Perhaps surprisingly, the Go compiler allows you to create unread constants with `const`. This is because constants in Go are calculated at compile time and cannot have any side effects. This makes them easy to eliminate: if a constant isn't used, it is simply not included in the compiled binary.

Naming Variables and Constants

There is a difference between Go's rules for naming variables and the patterns that Go developers follow when naming their variables and constants. Like most languages, Go requires identifier names to start with a letter or underscore, and the name can contain numbers, underscores, and letters. Go's definition of "letter" and "number" is a bit broader than many languages. Any Unicode character considered a letter or digit is allowed. This makes all the variable definitions in Example 2-5 perfectly valid Go.

Example 2-5. Variable names you should never use

```
_0 := 0_0
_𝟙 := 20
π := 3
ａ := "hello" // Unicode U+FF41
```

```
__ := "double underscore" // two underscores
fmt.Println(_0)
fmt.Println(_𝟙)
fmt.Println(π)
fmt.Println(ａ)
fmt.Println(__)
```

You can test out this awful code on The Go Playground. While it works, *do not* use variable names like this. These names are considered nonidiomatic because they break the fundamental rule of making sure that your code communicates what it is doing. These names are confusing or difficult to type on many keyboards. Look-alike Unicode code points are the most insidious, because even if they appear to be the same character, they represent entirely different variables. You can run the code shown in Example 2-6 on The Go Playground or in the *sample_code/look_alike_code_points* directory in the Chapter 2 repository.

Example 2-6. Using look-alike code points for variable names

```
func main() {
    ａ := "hello"   // Unicode U+FF41
    a := "goodbye" // standard lowercase a (Unicode U+0061)
    fmt.Println(ａ)
    fmt.Println(a)
}
```

When you run this program, you get:

```
hello
goodbye
```

Even though the underscore is a valid character in a variable name, it is rarely used, because idiomatic Go doesn't use snake case (names like `index_counter` or `number_tries`). Instead, idiomatic Go uses camel case (names like `indexCounter` or `numberTries`) when an identifier name consists of multiple words.

> An underscore by itself (_) is a special identifier name in Go; I'll talk more about it when I cover functions in Chapter 5.

In many languages, constants are always written in all uppercase letters, with words separated by underscores (names like `INDEX_COUNTER` or `NUMBER_TRIES`). Go does not follow this pattern. This is because Go uses the case of the first letter in the name of a package-level declaration to determine if the item is accessible outside the package. I will revisit this when I talk about packages in Chapter 10.

Within a function, favor short variable names. *The smaller the scope for a variable, the shorter the name that's used for it*. It is common in Go to see single-letter variable names used with `for` loops. For example, the names `k` and `v` (short for *key* and *value*) are used as the variable names in a `for-range` loop. If you are using a standard `for` loop, `i` and `j` are common names for the index variable. There are other idiomatic ways to name variables of common types; I will mention them as I cover more parts of the standard library.

Some languages with weaker type systems encourage developers to include the expected type of the variable in the variable's name. Since Go is strongly typed, you don't need to do this to keep track of the underlying type. However, you may see Go code where the first letter of a type is used as the variable name (for example, `i` for integers or `f` for floats). When you define your own types, similar patterns apply, especially when naming receiver variables (which are covered in "Methods" on page 144).

These short names serve two purposes. The first is that they eliminate repetitive typing, keeping your code shorter. Second, they serve as a check on how complicated your code is. If you find it hard to keep track of your short-named variables, your block of code is likely doing too much.

When naming variables and constants in the package block, use more descriptive names. The type should still be excluded from the name, but since the scope is wider, you need a more complete name to clarify what the value represents.

For more discussion of Go naming recommendations, read the Naming section of Google's Go Style Decisions.

Exercises

These exercises demonstrate the concepts discussed in the chapter. Solutions to these exercises, along with the programs in this chapter, are in the Chapter 2 repository.

1. Write a program that declares an integer variable called `i` with the value 20. Assign `i` to a floating-point variable named `f`. Print out `i` and `f`.
2. Write a program that declares a constant called `value` that can be assigned to both an integer and a floating-point variable. Assign it to an integer called `i` and a floating-point variable called `f`. Print out `i` and `f`.
3. Write a program with three variables, one named `b` of type `byte`, one named `smallI` of type `int32`, and one named `bigI` of type `uint64`. Assign each variable the maximum legal value for its type; then add `1` to each variable. Print out their values.

Wrapping Up

You've covered a lot of ground here, understanding how to use the predeclared types, declare variables, and work with assignments and operators. In the next chapter, we will look at the composite types in Go: arrays, slices, maps, and structs. We will also take another look at strings and runes and how they interact with character encodings.