

O'REILLY®

Head First GO

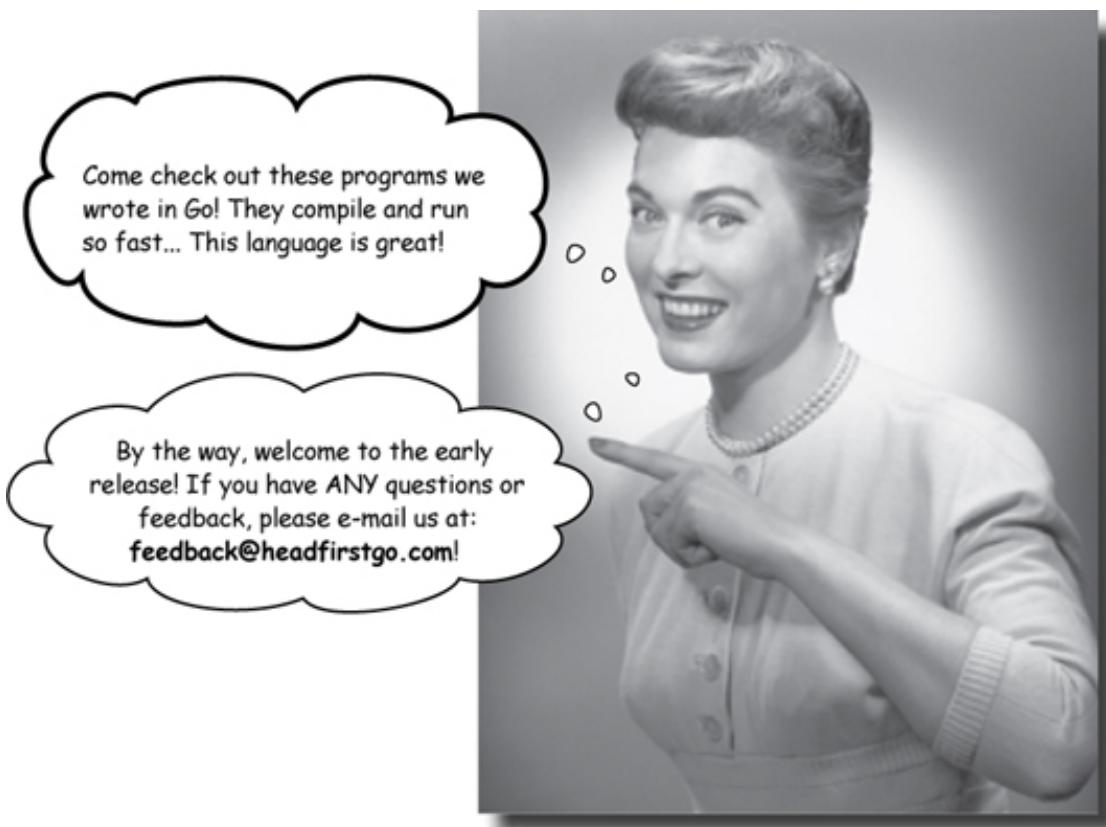
A Brain-Friendly Guide



Jay McGavren

1 syntax basics

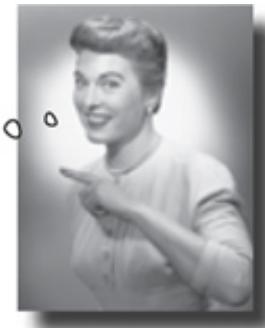
Let's Get Going



Are you ready to turbo-charge your software? Do you want a **simple** programming language that **compiles fast**? That **runs fast**? That makes it **easy to distribute** your work to users? Then **you're ready for Go!**

Go is a programming language that focuses on **simplicity** and **speed**. It's much less complex than other languages, so it's quicker to learn. And it lets you make full use of today's multi-core computer processors, so your programs run faster. This chapter will show you all the Go features that will make **your life as a developer easier**, and

No, really, we mean **any** feedback! If something in this book confuses you, other readers will be confused too. We want to work this stuff out before printing! E-mail us at:
feedback@headfirstgo.com
And thanks!



READY, SET, GO!

Back in 2007, the search engine Google had a problem. They had to maintain programs with millions of lines of code. Before they could test new changes, they had to compile the code into a runnable form, a process which at the time took the better part of an hour. Needless to say, this was bad for developer productivity.

So Google engineers Robert Griesemer, Rob Pike, and Ken Thompson sketched out some goals for a new language:

- Fast compilation
- Less cumbersome code
- Unused memory freed automatically (garbage collection)
- Easy to write software that does several operations simultaneously (concurrency)
- Good support for processors with multiple cores

After a couple years of work, Google had a language that was fast to write code for and produced programs that were fast to compile and run. The project switched to an open-source license in 2009. Go is now free for anyone to use. And you should use it! Go is rapidly gaining popularity thanks to its simplicity and power.

If you're writing a command-line tool, Go can produce executable files for Windows, Mac, and Linux, all from the same source code. If you're writing a web server, Go can help you handle many users connecting at once. And no matter *what* you're writing, it will help you ensure that your code is easier to maintain and add to.

Ready to learn more? Let's Go!



THE GO PLAYGROUND

The easiest way to try Go is to visit <https://play.golang.org> in your web browser. There, the Go team has set up a simple editor where you can enter Go code and run it on their servers. The result is displayed right there in your browser.

The Go Playground

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, Go!")
}
Hello, Go!
Program exited.
```

(Of course, this only works if you have a stable Internet connection. If you don't, see [page 25](#) to learn how to download and run the Go compiler directly on your computer. Then run the following examples using the compiler instead.)

Let's try it out now!



- ① Open <http://play.golang.org> in your browser. (Don't worry if what you see doesn't quite match the screenshot; it just means they've improved the site since this book was printed!)

- 2 Delete any code that's in the editing area, and type this instead:

```
package main
import "fmt"      Don't worry, we'll explain what all
                  this means on the next page!
func main() {
    fmt.Println("Hello, Go!")
}
```

- 3 Click the "Format" button, which will automatically reformat your code according to Go conventions.

- 4 Click the "Run" button.

You should see "Hello, Go!" displayed at the bottom of the screen. Congratulations, you've just run your first Go program!

Turn the page, and we'll explain what we just did...

Output → Hello, Go!

WHAT DOES IT ALL MEAN?

You've just run your first Go program! Now let's look at the code and figure out what it actually means...

Every Go file starts with a `package` declaration. A **package** is a group of code that all does similar things, like formatting strings or drawing images. The `package` declaration gives the name of the package that this file's code will become a part of. In this case, we use the special package `main`, which is required if this code is going to be run directly (usually from the terminal).

Next, Go files almost always have one or more `import` statements. Each file needs to **import** other packages before its code can use the code those other packages contain. Loading all the Go code on your computer at once would result in a big, slow program, so instead you specify only the packages you need by importing them.

```

This line says all the rest of the code in
this file belongs to the "main" package.

package main      This says we'll be using text formatting
import "fmt"       code from the "fmt" package.

func main() {      The "main" function is special; it gets run
    fmt.Println("Hello, Go!")      first when your program runs.

}

It does this by calling
the "Println" function
from the "fmt" package.

This line displays ("prints") "Hello, Go!"
in your terminal (or web browser, if
you're using the Go Playground).

```

The last part of every Go file is the actual code, which is often split up into one or more functions. A **function** is a group of one or more lines of code that you can **call** (run) from other places in your program. When a Go program is run, it looks for a function named `main` and runs that first, which is why we named this function `main`.



Don't worry if you don't understand all this right now!

We'll look at everything in more detail in the next few pages.

The typical Go file layout

You'll quickly get used to seeing these three sections, in this order, in almost every Go file you work with:

1. The package declaration
2. Any `import` statements
3. The actual code

```
The package declaration. {package main  
The imports section. {import "fmt"  
The actual code. {func main() {  
    fmt.Println("Hello, Go!")  
}}
```

The saying goes, "a place for everything, and everything in its place." Go is a very *consistent* language. This is a good thing: you'll often find you just *know* where to look in your project for a given piece of code, without having to think about it!

THERE ARE NO DUMB QUESTIONS

Q: My other programming language requires that each statement end with a semicolon. Doesn't Go?

A: You *can* use semicolons to separate statements in Go, but it's not required (in fact, it's generally frowned upon).

Q: Why did we run the automatic reformatting tool on our code?

A: Whenever you share your code, other Go developers will expect it to be in the standard Go format. That means that things like indentation and spacing will be formatted in a standard way, making it easier for everyone to read. Where other languages achieve this by relying on people manually reformatting their code to conform to a style guide, with Go all you have to do is run the standard formatting tool, and it will automatically fix everything for you. We ran the formatter on every example we created for this book, and you should run it on all your code, too!

WHAT IF SOMETHING GOES WRONG?

Go programs have to follow certain rules to avoid confusing the compiler. If we break one of these rules, we'll get an error message.

Suppose we forgot to add parentheses on our call to the `Println` function on line 6:

If we try to run this version of the program, we get an error:

```

Line 1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println "Hello, Go!"
7 }

```

Suppose we forgot the parentheses that used to be here...

Name of temporary file used by Go Playground.

Line number where the error occurred.

Description of the error.

```

tmp/sandbox254046298/main.go:6: syntax error: unexpected literal "Hello, Go!" at end of statement

```

Go tells us the name of the source code file and the line number where there's a problem. (The Go Playground saves the contents of the online editor to a temporary file before running it, which is where the "main.go" file name comes from.) Then it gives a description of the error. In this case, because we deleted the parenthesis, Go can't tell we're trying to call the `Println` function, so it can't understand why we're putting "Hello, Go" at the end of line 6.



BREAKING STUFF IS EDUCATIONAL!

We can get a feel for the rules Go programs have to follow by intentionally breaking our program in various ways. Take this code sample, try making one of the changes below, and run it. Then undo your change, and try the next one. See what happens!

```

package main
import "fmt"
func main() {
    fmt.Println("Hello, Go!")
}

```

Try breaking our code sample and see what happens!

If you do this...

...it will fail because...

Delete the package declaration... ~~package main~~

Every Go file has to begin with a package declaration.

Delete the import statement... <code>import "fmt"</code>	Every Go file has to import every package it references.
Import a second (unused) package... <code>import "fmt"</code> <code>import "strings"</code>	Go files must import <i>only</i> the packages they reference. (This helps keep your code compiling fast!)
Rename the <code>main</code> function... <code>func mainhello</code>	Your program looks for a function named <code>main</code> to run first.
Change the <code>Println</code> call to lower-case... <code>fmt.Pprintln("Hello, Go!")</code>	Everything in Go is case-sensitive, so although <code>fmt.Println</code> is valid, there's no such thing as <code>fmt.println</code> .
Delete the package name before <code>Println</code> ... <code>fmt.Println("Hello, Go!")</code>	The <code>Println</code> function isn't part of this package, so Go needs the package name before the function call.

Let's try the first one as an example...

Delete the package declaration... →

```
import "fmt"

func main() {
    fmt.Println("Hello, Go!")
}
```

You'll get errors! → can't load package: package main:
tmp/sandbox732822291/main.go:1:1: expected 'package', found 'import'

CALLING FUNCTIONS

Our example includes a call to the `fmt` package's `Println` function. To call a function, type the function name (`Println` in this case), and a pair of parenthesis.

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, Go!")
}
```

We'll explain this part shortly!

A call to the `Println` function.

fmt.**Println**()

Function name.

Parentheses.

Like many functions, `Println` can take one or more **arguments**: values you want the function to work with. The arguments appear in parenthesis after the method name.

Inside the parenthesis are one or more arguments, separated by commas.

fmt.Println("First argument", "Second argument")

Output → First argument Second argument

`Println` can be called with no arguments, or you can provide several arguments. When we look at other functions later, however, you'll find that most require a specific number of arguments. If you provide too few or too many, you'll get an error message saying how many arguments were expected, and you'll need to fix your code.

THE PRINTLN FUNCTION

Use the `Println` function when you need to see what your program is doing. Any arguments you pass to it will be printed (displayed) in your terminal, with each argument separated by a space.

After printing all its arguments, `Println` will skip to a new terminal line. (That's why "ln" is at the end of its name.)

```
fmt.Println("First argument", "Second argument")
fmt.Println("Another line")
```

Output → First argument Second argument
Another line

USING FUNCTIONS FROM OTHER PACKAGES

The code in our first program is all part of the `main` package, but the `Println` function is in the `fmt` package. To be able to call `Println`, we first have to import the package containing it.

```

package main
import "fmt"

func main() {
    fmt.Println("Hello, Go!")
}

```

We have to import the "fmt" package before we can access its `Println` function.

This specifies that we're calling a function that's part of the "fmt" package.

Once we've imported the package, we can access any functions it contains by typing the package name, a dot, and the name of the function we want.

Package name. A dot. Name of the function.

`fmt`.`Println`()

Here's a code sample that calls functions from a couple other packages. Because we need to import multiple packages, we switch to an alternate format for the `import` statement that lets you list multiple packages within parentheses, one package name per line.

```

package main
import (
    "math"           ← Import the "math" package so we can use math.Floor.
    "strings"        ← Import the "strings" package so we can use strings.Title.
)

```

Call the `Floor` function from the "math" package. → `func main() {`

Call the `Title` function from the "strings" package. → `math.Floor(2.75)`

→ `strings.Title("head first go")`

→ `}`

← This program has no output.
(We'll explain why in a moment!)

Once we've importing the `math` and `strings` packages, we can access the `math` package's `Floor` function with `math.Floor`, and the `strings` package's `Title` function with `strings.Title`.

You may have noticed that in spite of including those two function calls in our code, the above sample doesn't display any output. We'll look at how to fix that next.

FUNCTION RETURN VALUES

In our previous code sample, we tried calling the `math.Floor` and `strings.Title` functions, but they didn't produce any output:

```

package main

import (
    "math"
    "strings"
)

func main() {
    math.Floor(2.75)
    strings.Title("head first go")
}

```

This program produces
no output!

When we call the `fmt.Println` function, we don't need to communicate with it any further after that. We pass one or more values for `Println` to print, and we trust that it printed them. But sometimes a program needs to be able to call a function and get data back from it. For this reason, functions in most programming languages can have **return values**: a value that the function computes and returns to its caller.

The `math.Floor` and `strings.Title` functions are both examples of functions that use return values. The `math.Floor` function takes a floating-point number, rounds it down to the nearest whole number, and returns that whole number. And the `strings.Title` function takes a string, capitalizes each word it contains (converting it to "title case"), and returns the capitalized string.

To actually see the results of these function calls, we need to take their return values and pass those to `fmt.Println`:

```

package main

import (
    "fmt" ← Import the "fmt" package as well..
    "math"
    "strings"
)

Call fmt.Println with   func main() {
    the return value → fmt.Println(math.Floor(2.75))
    from math.Floor.   fmt.Println(strings.Title("head first go"))
}

```

Takes a number, rounds it down,
and returns that value.

Output

2
Head First Go

Call fmt.Println with the return
value from strings.Title.

Takes a string, and returns a new string
with each word capitalized.

Once this change is made, the return values get printed, and we can see the results.

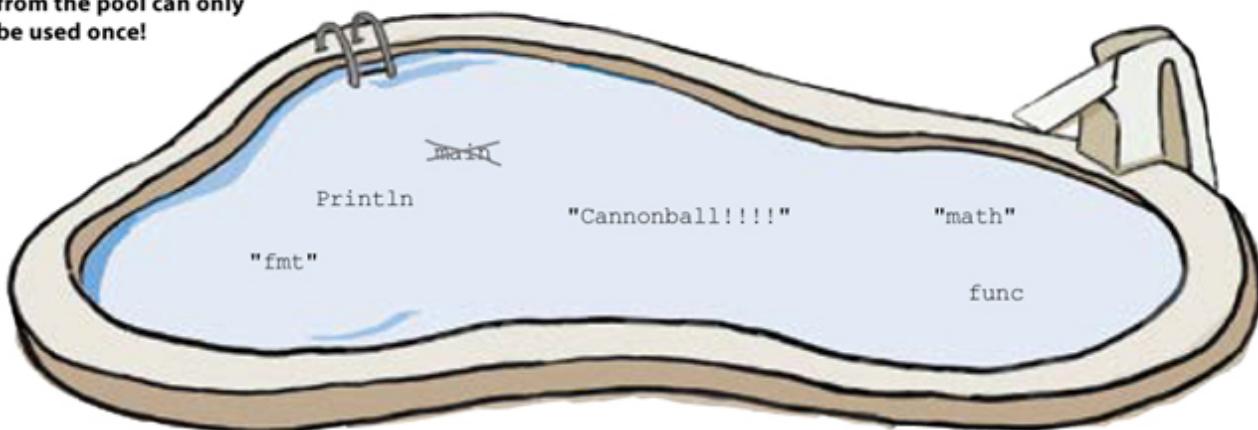


POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make code that will run and produce the output shown.

```
package main ← We've done the first  
import ( _____ )  
main() {  
    fmt.Println(_____)  
}  
Output  
Cannonball!!!!
```

Note: each snippet from the pool can only be used once!



A GO PROGRAM TEMPLATE

[Answers on page 29.](#)

For the code snippets that follow, just imagine inserting them into this full Go program:

```
package main  
import "fmt"  
func main() {  
    fmt.Println(_____)  
}
```

Insert your code here!

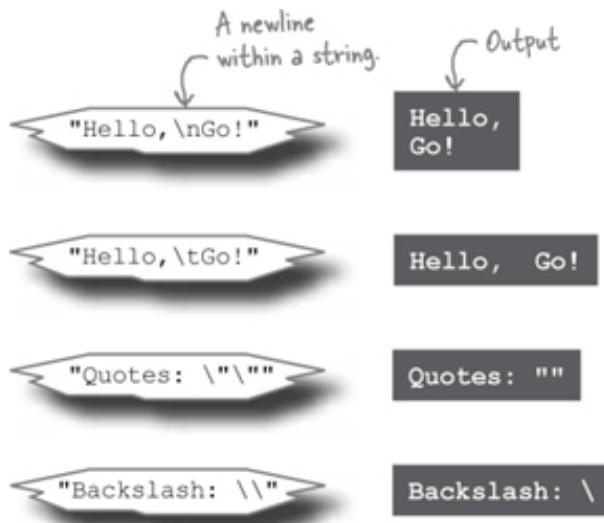
Better yet, try typing this program into the Go Playground, and then insert the snippets one at a time to see for yourself what they do!

STRINGS

We've been passing **strings** as arguments to `Println`. A string is a series of bytes that usually represent text characters. You can define strings directly within your code using **string literals**: text between double quotation marks that Go will treat as a string.



Within strings, characters like newlines, tabs, and other characters that would be hard to include in program code can be represented with **escape sequences**: a backslash followed by characters that represent another character. Escape sequence Value



Escape sequence	Value
\n	A newline character.
\t	A tab character.
\"	Double quotation marks.
\\\\"	A backslash.

RUNES

Whereas strings are usually used to represent a whole series of text characters, Go's **runes** are used to represent single characters.

```
package main  Here's our template again...
import "fmt"
func main() {
    fmt.Println(Insert your code here!)
}
```

String literals are written surrounded by double quotation marks ("'), but **rune literals** are written with single quotation marks ('').

Go programs can use any character from any language on earth, because Go uses the Unicode standard for storing runes. Runes are kept as numeric codes, not the characters themselves, and if you pass a rune to `fmt.Println`, you'll see that numeric code in the output, not the original character.



Just as with string literals, escape sequences can be used in a rune literal to represent characters that would be hard to include in program code:



BOOLEANS

Boolean values can be one of only two values: `true` or `false`. They're especially useful with conditional statements, which cause sections of code to run only if a condition is true or false. (We'll look at conditionals in the next chapter.)



NUMBERS

You can also define numbers directly within your code, and it's even simpler than string literals: just type the number.

```

package main  Here's our template again...
import "fmt"
func main() {
    fmt.Println(Insert your code here!)
}

```

A whole number by itself is called an **integer literal**:



A number with a decimal point is called a **floating-point literal**:



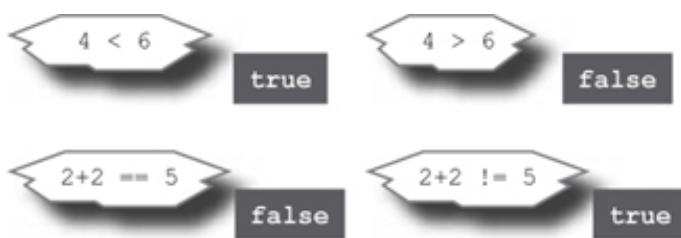
As we'll see shortly, Go treats integer and floating-point numbers as different types, so remember that a decimal point can be used to distinguish between their literal values.

MATH OPERATIONS AND COMPARISONS

Go's basic math operators work just like they do in most other languages. The `+` symbol is for addition, `-` for subtraction, `*` for multiplication, and `/` for division..



You can use `<` and `>` to compare two values and see if one is less than or greater than another. You can use `==` (that's *two* equals signs) to see if two values are equal, and `!=` (that's an exclamation point and an equals sign, read aloud as "not equal") to see if two values are not equal. The result of a comparison is a boolean value, either `true` or `false`.



TYPES

In a previous code sample, we saw the `math.Floor` function, which rounds a floating-point number down to the nearest whole number, and the `strings.Title` function, which converts a string to title case. It makes sense that you would pass a number as an argument to the `Floor` function, and a string as an argument to the `Title` function. But what would happen if you passed a string to `Floor` and a number to `Title`?

```

package main

import (
    "fmt"
    "math"
    "strings"
)

func main() {
    fmt.Println(math.Floor("head first go"))
    fmt.Println(strings.Title(2.75))
}

```

Errors

Normally takes a floating-point number!

Normally takes a string!

cannot use "head first go" (type string) as type float64 in argument to math.Floor
 cannot use 2.75 (type float64) as type string in argument to strings.Title

Go prints two error messages, one for each function call, and the program doesn't even run!

Things in the world around you can often be classified into different types based on what they can be used for. You don't eat a car or truck for breakfast (because they're vehicles), and you don't drive an omelet or bowl of cereal to work (because they're breakfast foods).

Likewise, values in Go are all classified into different **types**, which specify what the values can be used for. Integers can be used in math operations, but strings can't. Strings can be capitalized, but numbers can't. And so on.

Go is **statically typed**, which means that it knows what the types of your values are even before your program runs. Functions expect their arguments to be of particular types, and their return values have types as well (which may or may not be the same as the argument types). If you accidentally use the wrong type of value in the wrong place, Go will give you an error message. This is a good thing: it lets you find out there's a problem before your users do!

Go is statically typed. If you use the wrong type of value in the wrong place, Go will let you know.

You can view the type of any value by passing it to the `reflect` package's `TypeOf` method. Let's find out what the types are for some of the values we've already seen:

```

package main
import (
    "fmt"
    "reflect"
)
func main() {
    fmt.Println(reflect.TypeOf(42))
    fmt.Println(reflect.TypeOf(3.1415))
    fmt.Println(reflect.TypeOf(true))
    fmt.Println(reflect.TypeOf("Hello, Go!"))
}

```

Import "reflect" package so we can use its `TypeOf` method.

Returns the type of its argument.

Output

int
float64
bool
string

Here's what those types are used for:

Type	Description
<code>int</code>	An integer. Can be any whole number.
<code>float64</code>	A floating point number. Holds numbers with a fractional part. (The 64 in the type name is because up to 64 bits of data are used to hold the number. This means that <code>float64</code> values can be fairly, but not infinitely, precise before being rounded off.)
<code>bool</code>	A boolean value. Can only be <code>true</code> or <code>false</code> .
<code>string</code>	A string. A series of data that usually represents text characters.



Draw lines to match the values below to their types.

Some types will have more than one value that matches with them.

5.2 float64

1

false bool

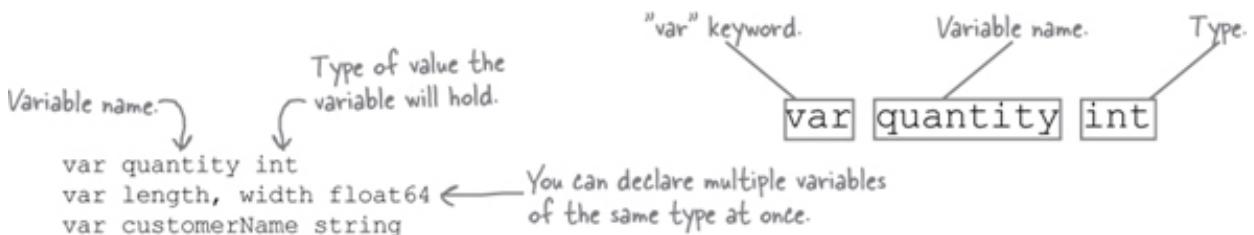
1.0

"hello" string

Answers on page 29.

DECLARING VARIABLES

In Go, a **variable** is a piece of storage containing a value. You can give a variable a name by using a **variable declaration**. Just use the `var` keyword followed by the desired name and the type of values the variable will hold.



Once you declare a variable, you can assign any value of that type to it with = (that's a *single* equals sign):

`quantity = 2`

`customerName = "Damon Cole"`

You can assign values to multiple variables in the same statement. Just place multiple variable names on the left side of the =, and the same number of values on the right side, separated with commas.

`length, width = 1.2, 2.4` ← Assigning multiple variables at once.

Once you've assigned values to variables, you can use them in any context where you would use the original values:

```

package main

import "fmt"

func main() {
    Declaring the variables. { var quantity int
                                var length, width float64
                                var customerName string

    Assigning values to the variables. { quantity = 4
                                         length, width = 1.2, 2.4
                                         customerName = "Damon Cole"

    Using the variables. { fmt.Println(customerName)
                           fmt.Println("has ordered", quantity, "sheets")
                           fmt.Println("each with an area of")
                           fmt.Println(length*width, "square meters")
}

```

Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters

If you know beforehand what a variable's value will be, you can declare variables and assign them values on the same line:

Just add an assignment onto the end.

Declaring variables { var quantity int = 4
AND assigning values { var length, width float64 = 1.2, 2.4 ← If you're declaring multiple variables, provide multiple values.
var customerName string = "Damon Cole"

You can assign new values to existing variables, but they need to be values of the same type. Go's static typing ensures you don't accidentally assign the wrong kind of value to a variable.

Assigned types don't match the declared types! { quantity = "Damon Cole"
customerName = 4

Errors

cannot use "Damon Cole" (type string) as type int in assignment
cannot use 4 (type int) as type string in assignment

Go also requires that every variable you declare get used somewhere in your program. If you don't, it will report an error, and you'll need to either use the variable or remove its declaration. This is a good thing, because the presence of an unused variable often indicates a bug! Go reports an error to help you find and fix the problem.

```

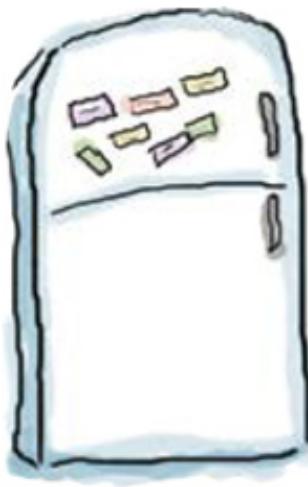
func main() {
    var depth float64
}

```

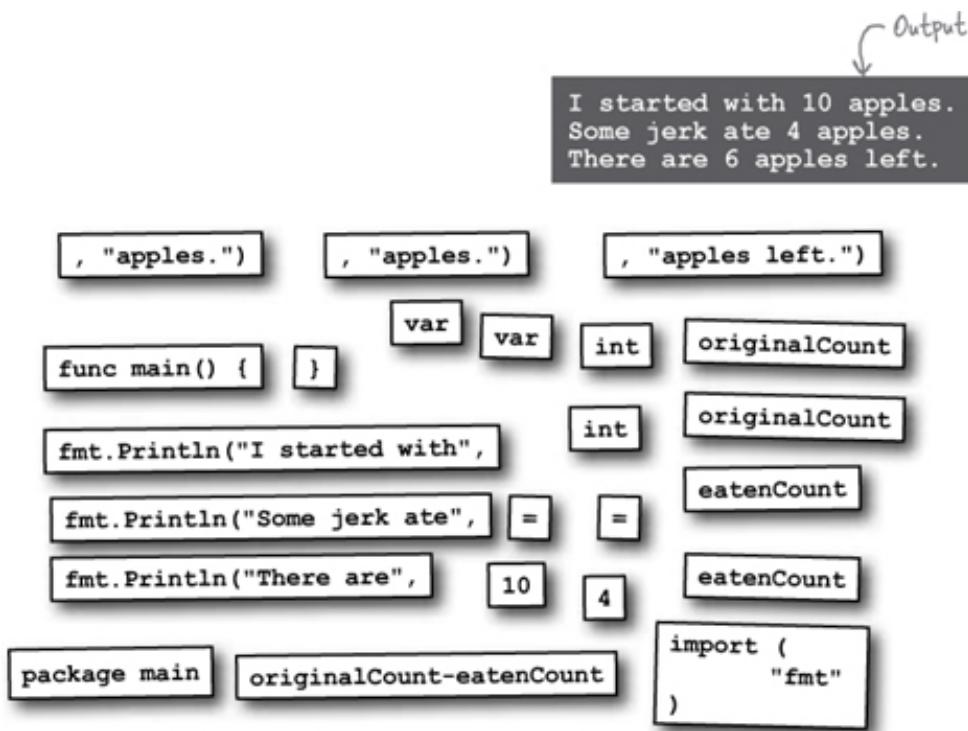
Error

depth declared and not used

CODE MAGNETS



A Go program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output?



SHORT VARIABLE DECLARATIONS

Answers on page 29.

We mentioned that you can declare variables and assign them values on the same line:

Declarign variables AND assigning values `{var quantity int = 4
var length, width float64 = 1.2, 2.4
var customerName string = "Damon Cole"}` Just add an assignment onto the end. If you're declaring multiple variables, provide multiple values.

But if you know what the initial value of a variable is going to be as soon as you declare it, it's more typical to use a **short variable declaration**. Instead of explicitly declaring the type of the variable and later assigning to it with `=`, you do both at once using `:=`.

Let's update the previous example to use short variable declarations:

```
package main

import "fmt"

func main() {
    quantity := 4
    length, width := 1.2, 2.4
    customerName := "Damon Cole"

    fmt.Println(customerName)
    fmt.Println("has ordered", quantity, "sheets")
    fmt.Println("each with an area of")
    fmt.Println(length*width, "square meters")
}
```

Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters

There's no need to explicitly declare the variable's type; the *type of the value assigned to the variable* becomes the *type of that variable*.

Because short variable declarations are so convenient and concise, they're used more often than regular declarations. You'll still see both forms occasionally, though, so it's important to be familiar with both.



BREAKING STUFF IS EDUCATIONAL!

Take our program that uses variables, try making one of the changes below, and run it. Then undo your change, and try the next one. See what happens!

```
package main

import "fmt"

func main() {
    quantity := 4
    length, width := 1.2, 2.4
    customerName := "Damon Cole"

    fmt.Println(customerName)
    fmt.Println("has ordered", quantity, "sheets")
    fmt.Println("each with an area of")
    fmt.Println(length*width, "square meters")
}
```

Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters

If you do this...

...it will fail because...

Add a second declaration for the

You can only declare a variable once. (Although you can assign new values to it as often as you want. You can also

<p>same variable</p> <pre>quantity := 4</pre> <pre>quantity := 4</pre>	declare other variables with the same name, as long as they're in a different scope. We'll learn about scopes later in the chapter.)
<p>Delete the : from a short variable declaration</p> <pre>quantity ÷ = 4</pre>	If you forget the :, it's treated as an assignment, not a declaration, and you can't assign to a variable that hasn't been declared.
<p>Assign a <code>string</code> to an <code>int</code> variable</p> <pre>quantity := 4</pre> <pre>quantity = "a"</pre>	Variables can only be assigned values of the same type.
<p>Mismatch number of variables and values</p> <pre>length, width := 1.2</pre>	You're required to provide a value for every variable you're assigning, and a variable for every value.
<p>Remove code that uses a variable</p> <pre>fmt.Println(customerName)</pre>	All declared variables must be used in your program. If you remove the code that uses a variable, you must also remove the declaration.

NAMING RULES

Go has one simple set of rules that apply to the names of variables, functions, and

types:

- A name must begin with a letter, and can have any number of additional letters and numbers.
- If the name of a variable, function, or type begins with a capital letter, it is considered **exported** and can be accessed from packages outside the current one. (This is why the "P" in `fmt.Println` is capitalized: so it can be used from the `main` package or any other.) If a variable/function/type name begins with a lower-case letter, it is considered **unexported** and can only be accessed within the current package.

The image shows handwritten notes comparing valid and invalid Go variable names. On the left, under 'OK', there are three examples: 'length', 'stack2', and 'sales.Total'. On the right, under 'Illegal', there are two examples: '2stack' and 'sales.total'. Handwritten annotations explain the rules: 'Can't start with a number!' points to '2stack', and 'Can't access anything in another package unless its name is capitalized!' points to 'sales.total'.

Those are the only rules enforced by the language. But the Go community follows some additional conventions as well:

- If a name consists of multiple words, each word after the first should be capitalized, and they should be attached together without spaces between them, like this: `topPrice`, `RetryConnection`, etc. (The first letter of the name should only be capitalized if you want to export it from the package.) This style is often called "camel case" because the capitalized letters look like the humps on a camel.
- When the meaning of a name is obvious from the context, the Go community's convention is to abbreviate it: to use `i` instead of `index`, `max` instead of `maximum`, and so on. (However, we at Head First believe that nothing is obvious when you're learning a new language, so we will *not* be following that convention in this book.)

The image shows handwritten notes on Go naming conventions. It compares 'OK' conventions (like `sheetLength`, `TotalUnits`, and `i`) with 'Breaks conventions' (like `sheetlength`, `Total_Units`, and `index`). Annotations explain: 'Subsequent words should be capitalized!' points to the camel-cased forms, 'This is legal, but words should be joined directly!' points to `Total_Units`, and 'Consider replacing with an abbreviation!' points to `index`.

CONVERSIONS

Math and comparison operations in Go require that the included values be of the same type. If they're not, you'll get an error when trying to run your code.

```

Set up a float64 variable → var length float64 = 1.2
Set up an int variable. → var width int = 2
fmt.Println("Area is", length*width) ← If we use both the
                                         float64 and the int in
                                         a math operation...
fmt.Println("length > width?", length > width) ← ...we'll get errors!
                                         Or a comparison...

```

Errors → invalid operation: length * width (mismatched types float64 and int)
invalid operation: length > width (mismatched types float64 and int)

The same is true when assigning new values to variables. If the type of value being assigned doesn't match the declared type of the variable, you'll get an error.

```

Set up a float64 variable → var length float64 = 1.2
Set up an int variable. → var width int = 2
length = width ← If we assign the int value
                  to the float64 variable...
fmt.Println(length)
...we'll get errors!

```

Error → cannot use width (type int) as type float64 in assignment

The solution is to use **conversions**, which let you convert a value from one type to another type. You just provide the type you want to convert a value to, immediately followed by the value you want to convert in parentheses.

Type to convert to Value to convert
float64(2)

The result is a new value of the desired type. Here's the result of running `TypeOf` on an integer, and again on that same integer after conversion to a `float64`:

```

fmt.Println(reflect.TypeOf(2)) ← Without a conversion...
fmt.Println(reflect.TypeOf(float64(2))) ← With a conversion...

```

int **float64** ← Type is changed.

Let's update our failing code example to convert the `int` value to a `float64` before using it in any math operations or comparisons with other `float64` values.

```

var length float64 = 1.2
var width int = 2
fmt.Println("Area is", length*float64(width)) ← Convert the int to a float64 before
                                         multiplying it with another float64.
fmt.Println("length > width?", length > float64(width)) ← Convert the int to a float64
                                         before comparing it with
                                         another float64.

```

**Area is 2.4
length > width? false**

The math operation and comparison both work correctly now!

Now let's try converting an `int` to a `float64` before assigning it to a `float64` variable:

```
var length float64 = 1.2
var width int = 2
length = float64(width) ← Convert the int to a
fmt.Println(length)      float64 before assigning it
                        to the float64 variable.
```

2

Again, with the conversion in place, the assignment is successful.

When making conversions, be aware of how they might change the resulting values. For example, `float64` values can store fractional values, but `int` values can't. When you convert a `float64` to an `int`, the fractional portion is truncated (rounded *down*) to the nearest whole number! This can throw off any operations you do with the resulting value.

```
var length float64 = 3.75
var width int = 5
width = int(length) ← This conversion causes the
fmt.Println(width)      fractional portion to be
                        truncated (rounded DOWN)!
```

3 ← The resulting value is 0.75 lower!

As long as you're cautious, though, you'll find conversions essential to working with Go. They allow otherwise-incompatible types to work together.



We've written the Go code below, to calculate a total price with tax and determine if we have enough funds to make a purchase. But we're getting errors when we try to include it in a full program!

```

var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = price * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = price + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total < availableFunds)

    ↘ Errors

```

```

invalid operation: price * taxRate (mismatched types int and float64)
invalid operation: price + tax (mismatched types int and float64)
invalid operation: total < availableFunds (mismatched types float64 and int)

```

Fill in the blanks below to update this code. Fix the errors so that it produces the expected output. (Hint: Before doing math operations or comparisons, you'll need to use conversions to make the types compatible.)

```

var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = _____
fmt.Println("Tax is", tax, "dollars.")

var total float64 = _____
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", _____)

```

Expected output

```

Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true

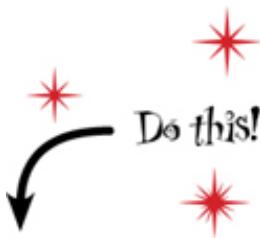
```

Answers on page 30.

INSTALLING GO ON YOUR COMPUTER

The Go Playground is a great way to try out the language. But its practical uses are limited. You can't use it to work with files, for example. And it doesn't have a way to take user input from the terminal, which we're going to need for an upcoming program.

So to wrap up this chapter, let's download and install Go on your computer. Don't worry, the Go team has made it really easy! On most operating systems, you just have to run an installer program, and you'll be done.



- ➊** Visit <https://golang.org> in your web browser.
- ➋** Click the download link.
- ➌** Select the binary distribution for your OS. The download should begin automatically.
- ➍** Visit the installation instructions page for your OS (you may be taken there automatically after the download starts), and follow the directions there. (Don't skip any steps, or your system may not be able to find the `go` command!)
- ➎** Restart your computer.
- ➏** Open a new terminal or command prompt window.
- ➐** Confirm Go was installed by typing "`go version`" (without quotes) at the prompt and hitting the Return or Enter key. You should see a message with the version of Go that's installed.



Web sites are always changing.

It's possible that `golang.org` or the Go installer will be updated after this book is published, and these directions will no longer be completely accurate. In that case, visit:

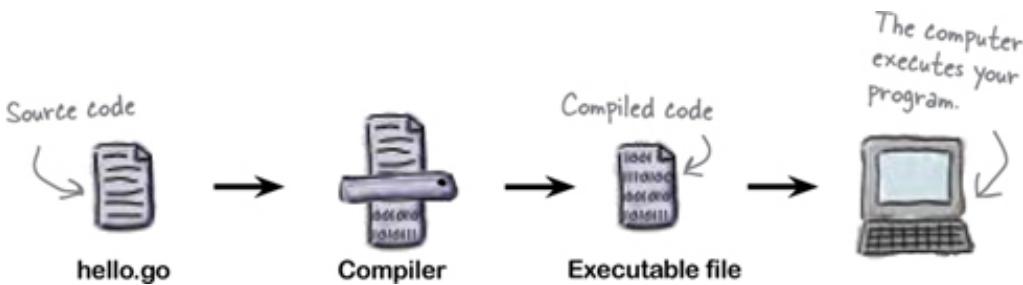
<http://headfirstgo.com>

for help and troubleshooting tips!

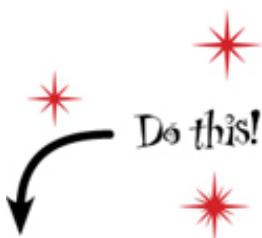
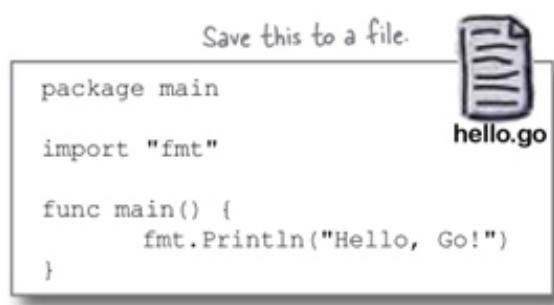
COMPILING GO CODE

Our interaction with The Go Playground has consisted of typing in code and having it mysteriously run. Now that we've actually installed Go on your computer, it's time to take a closer look at how this works.

Computers actually aren't capable of running Go code directly. Before that can happen, we need to take the source code file and **compile** it: convert it to a binary format that a CPU can execute.



Let's try using our new Go installation to compile and run our "Hello, Go!" example from earlier.



- ➊ Using your favorite text editor, save our "Hello, Go!" code from earlier in a plain-text file named "hello.go".
- ➋ Open a new terminal or command prompt window.
- ➌ In the terminal, change to the directory where you saved "hello.go".
- ➍ Run "go fmt hello.go" to clean up the code formatting. (This step isn't required, but it's a good idea anyway.)

5 Run "go build hello.go" to compile the source code. This will add an executable file to the current directory. On Mac or Linux, the executable will be named just "hello". On Windows, the executable will be named "hello.exe".

6 Run the executable file. On Mac or Linux, do this by typing "./hello" (which means "run a program named hello in the current directory"). On Windows, just type "hello.exe".

Change to whatever directory you saved "hello.go" in.

Format code.

Compile code.

Run executable.

```
Shell Edit View Window Help
$ cd try_go
$ go fmt hello.go
$ go build hello.go
$ ./hello
Hello, Go!
$
```

Compiling and running hello.go on Mac or Linux

Change to whatever directory you saved "hello.go" in.

Format code.

Compile code.

Run executable.

```
Command Prompt
>cd try_go
>go fmt hello.go
>go build hello.go
>hello.exe
Hello, Go!
>
```

Compiling and running hello.go on Windows

GO TOOLS

When you install Go, it adds an executable named "go" to your command prompt. The go executable gives you access to various commands, including:

Command	Description
go build	Compiles source code files into executable files, which it will save in the current directory.
go run	Compiles and runs a source file, but without saving an executable file.
go fmt	Re-formats source files using Go standard formatting.

go version | Displays the current Go version.

We just tried the `go fmt` command, which re-formats your code in the standard Go format. It's equivalent to the "Format" button on the Go Playground site. We recommend running `go fmt` on every source file you create.

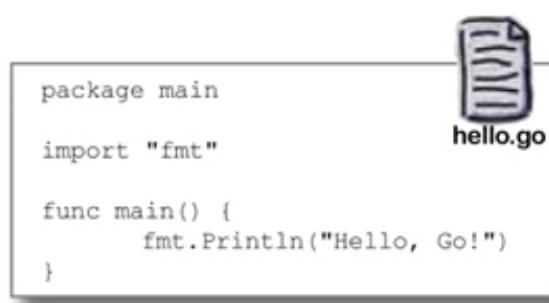
Most editors can be set up to automatically run "go fmt" every time you save a file! See: <https://blog.golang.org/go-fmt-your-code>

We also used the `go build` command, which compiles your code into executable files. You can distribute these files to users, and they'll be able to run them even if they don't have Go installed.

But we haven't tried the `go run` command yet. Let's do that now.

TRY OUT CODE QUICKLY WITH "GO RUN"

The `go run` command compiles and runs a source file, without saving an executable file to the current directory. It's great for quickly trying out simple programs. Let's use it to run our "hello.go" sample.



- ❶ Open a new terminal or command prompt window.
- ❷ In the terminal, change to the directory where you saved "hello.go".
- ❸ Type "`go run hello.go`" and hit Enter/Return. (The command is the same on all operating systems.)

You'll immediately see the program output. If you make changes to the source code,

there's no need to compile a new executable; just run it with `go run` and you'll be able to see the results right away. When you're working on small programs, `go run` is a handy tool to have!



YOUR GO TOOLBOX

That's it for Chapter 1! You've added function calls and types to your toolbox.

Functions

A function is a chunk of code that you can call from other places in your program.

When calling a function, you can use arguments to provide the function with data.

Go functions can provide data back to their callers using one or more return values.

Types

Values in Go are classified into different types, which specify what the values can be used for.

Math operations and comparisons between different types are not allowed, but you can convert a value to a new type if needed.

Go variables can only store values of their declared type.



BULLET POINTS

- A package is a group of related functions and other code.
- Before you can use a package's functions within a Go file, you need to import that package.
- A `string` is a series of bytes that usually represent text characters.
- A `rune` represents a single text character.
- Go's two most common numeric types are `int`, which holds integers, and `float64`, which holds floating-point decimal numbers.
- The `bool` type holds boolean values, which are either `true` or `false`.
- A variable is a piece of storage that contains values of a specified type.
- A variable, function, or type can only be accessed from code in other packages if its name begins with a capital letter.
- The `go fmt` command automatically reformats source files to use Go standard formatting. You should run `go fmt` on any code that you plan to share with others.
- The `go build` command compiles Go source code into a binary format that computers can execute.
- The `go run` command compiles and runs source code, but without saving a binary file.

POOL PUZZLE SOLUTION

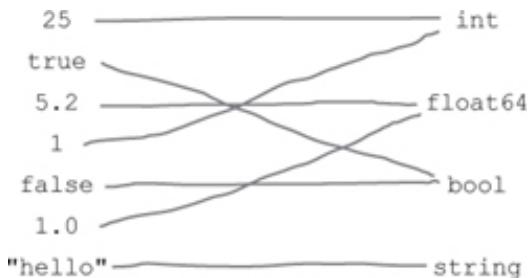
```
package main

import "fmt"
)

func main() {
    fmt.Println("Cannonball!!!!")
}
    ↓ Output
    Cannonball!!!!
```



Draw lines to match the values below to their types. Some types will have more than one value that matches with them.



CODE MAGNETS SOLUTION

A Go program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output?

```
package main

import (
    "fmt"
)

func main() {
    var originalCount int = 10
    fmt.Println("I started with", originalCount, "apples.")

    var eatenCount int = 4
    fmt.Println("Some jerk ate", eatenCount, "apples.")

    fmt.Println("There are", originalCount-eatenCount, "apples left.")
}
```

The code is reconstructed as follows:

```
package main

import (
    "fmt"
)

func main() {
    var originalCount int = 10
    fmt.Println("I started with", originalCount, "apples.")

    var eatenCount int = 4
    fmt.Println("Some jerk ate", eatenCount, "apples.")

    fmt.Println("There are", originalCount-eatenCount, "apples left.")
}
```

Output

I started with 10 apples.
Some jerk ate 4 apples.
There are 6 apples left.



Fill in the blanks below to update this code. Fix the errors so that it produces the expected output. (Hint: Before doing math operations or comparisons, you'll need to use conversions to make the types compatible.)

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = float64(price) * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = float64(price) + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total < float64(availableFunds))
```

Expected output
↓
Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true

2 conditionals and loops

Which Code Runs Next?



Every program has parts that only apply in certain situations.

"This code should run *if* there's an error. Otherwise, that other code should run."

Almost every program contains code that should be run only when a certain *condition* is true. So almost every programming language provides **conditional statements** that let you determine whether to run segments of code. Go is no exception.

You may also need some parts of your code to run *repeatedly*. Like most languages, Go provides **loops** that run sections of code more than once. We'll learn to use both conditionals and loops in this chapter!

CALLING METHODS

In Go, it's possible to define **methods**: functions that are associated with values of a given type. Go methods are kind of like the methods that you may have seen attached to "objects" in other languages, but they're much simpler.

We'll be taking a detailed look at how methods work in chapter TODO. But we need to use a couple methods to make our examples for this chapter work, so let's look at some brief examples of calling methods now.

The `time` package has a `Time` type that represents a date (year, month, and day) and time (hour, minute, second, etc.). Each `time.Time` value has a `Year` method that returns the year. The code below uses this method to print the current year:

```
package main
import (
    "fmt"
    "time" ← We need to import the
              "time" package so we can
              use the time.Time type.
)
func main() {
    var now time.Time = time.Now() ↓ time.Now returns a time.Time value
    var year int = now.Year() ← representing the current date and time.
    fmt.Println(year)
} 2018 (Or whatever year your
          computer's clock is set for.)
```

time.Time values have a Year method that returns the year.

The `time.Now` function returns a new `Time` value for the current date and time, which we store in the `now` variable. Then, we call the `Year` method on the value that `now` refers to:

Holds a time.Time value. ↘ now.Year() Call the Year method on the time.Time value.

The `Year` method returns an integer with the year, which we then print.

The `strings` package has a `Replacer` type that can search through a string for a substring, and replace each occurrence of that substring with another string. The code below replaces every `#` symbol in a string with the letter `"o"`:

```

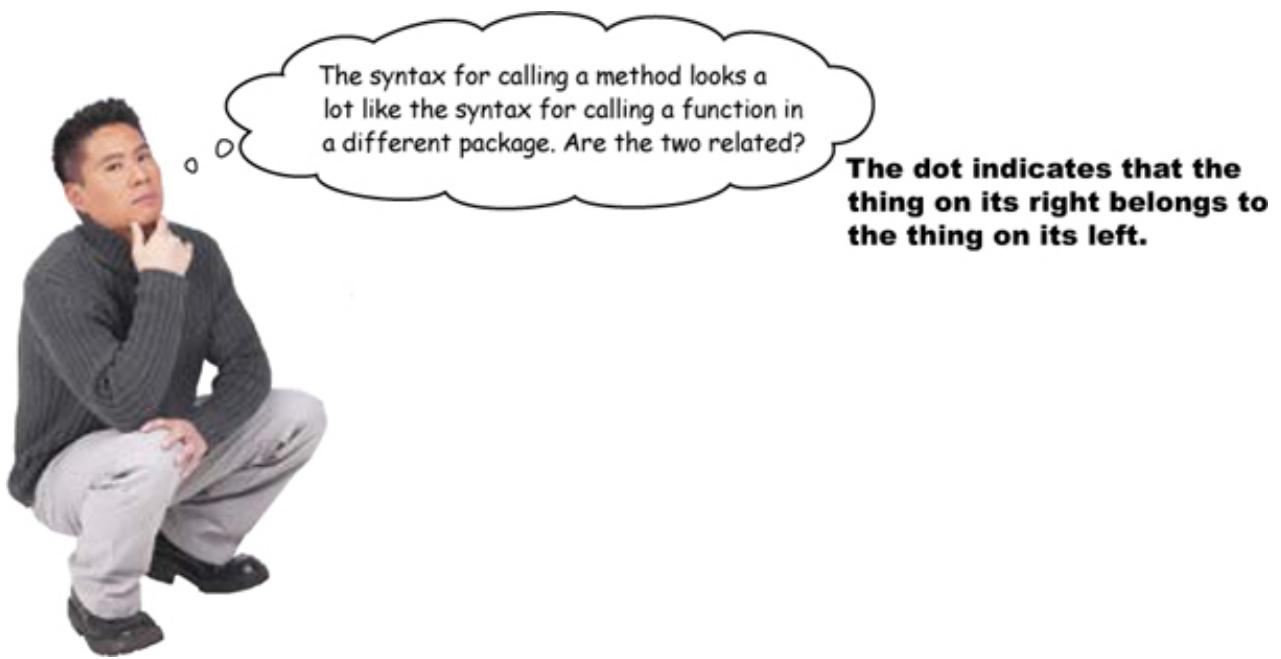
package main

Import packages used in the "main" function. { import (
    "fmt"
    "strings"
}

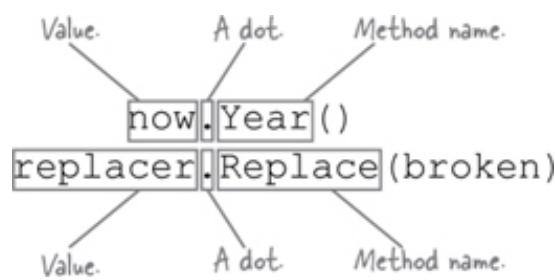
func main() {
    broken := "G# r#cks!"
    replacer := strings.NewReplacer("#", "o") ←
        This returns a strings.Replacer value that's set up to replace every "#" with "o".
    fixed := replacer.Replace(broken) ←
        Call the Replace method on the strings.Replacer, and pass it a string to do the replacements on.
    fmt.Println(fixed)
}
Print the string returned from the Replace method. → Go rocks!

```

The `strings.NewReplacer` function takes arguments with a string to replace ("#"), and a string to replace it with ("o"), and returns a `strings.Replacer` value. When we pass a string to the `Replacer` value's `Replace` method, it returns a string with those replacements made.



Whereas the functions we saw earlier belonged to a *package*, the methods belong to an individual *value*. That value is what appears to the left of the dot.



MAKING THE GRADE

In this chapter, we're going to look at features of Go that let you decide whether to run some code or not, based on a condition. Let's look at a situation where we might need

that ability...

We need to write a program that allows a student to type in their percentage grade, and tells them whether they passed or not. Passing or failing follows a simple formula: a grade of 60% or more is passing, less than 60% is failing. So our program will need to give one response if the entered percentage grade is 60 or greater, and a different response otherwise.

COMMENTS

Let's create a new file, "pass_fail.go", to hold our program. We're going to take care of a detail we omitted in our previous programs, and add a description of what the program does at the top.

```
Comment → // pass_fail reports whether a grade is passing or failing.  
Since this will be → package main  
another executable  
program, we use the  
"main" package.      func main() { ← As before, Go will look for  
                      }           a "main" function to run  
                                when the program starts.
```

Most Go programs include descriptions in their source code of what they do, intended for people maintaining the program to read. These **comments** are ignored by the compiler.

The most common form of comment is marked with two slash characters (//). Everything from the slashes to the end of the line is treated as part of the comment. A // comment can appear on a line by itself, or following a line of code.

```
// The total number of widgets in the system.  
var TotalCount int // Can only be a whole number.
```

The less frequently used form of comments, **block comments**, spans multiple lines. Block comments start with /*, end with */, and everything between those markers (including newlines) is part of the comment.

```
/*  
Package widget includes all the functions used  
for processing widgets.  
*/
```

GETTING A GRADE FROM THE USER

Now let's add some actual code to our `pass_fail.go` program. The first thing it needs to do is allow the user to input a percentage grade. We want them to type a number and press Enter, and we'll store the number they typed in a variable. Let's add code to handle this.

```
// pass_fail reports whether a grade is passing or failing.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    fmt.Println("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input := reader.ReadString('\n')
    fmt.Println(input)
}
```

Import packages used in the "main" function.

Prompt the user to enter a grade.

Set up a "buffered reader" that gets text from the keyboard.

Return everything the user has typed, up to where they pressed the Enter key.

Print what the user typed.

First, we need to let the user know to enter something, so we use the `fmt.Print` function to display a prompt. (Unlike the `Println` function, `Print` doesn't skip to a new terminal line after printing a message, which lets us keep the prompt and the user's entry on the same line.)

Next, we need a way to read (receive and store) input from the program's "standard input", which all keyboard input goes to. The line `reader := bufio.NewReader(os.Stdin)` stores a `bufio.Reader` value in the `reader` variable that can do that for us.

```
reader := bufio.NewReader(os.Stdin)
```

Returns a new `bufio.Reader` value.

The Reader will read from standard input (the keyboard.)

To actually get the user's input, we call the `ReadString` method on the `Reader`. `ReadString` requires an argument with a rune (character) that marks the end of the input. We want to read everything the user types up until they press Enter, so we give `ReadString` a newline rune.

```
input := reader.ReadString('\n')
```

Returns what the user typed, as a string.

Everything up until the newline rune will be read.

Once we have the user input, we simply print it.

That's the plan, anyway. But if we try to compile or run this program, we'll get an error:

Error → multiple-value
reader.ReadString()
in single-value context



Don't worry too much about the details of how `bufio.Reader` works.

All you really need to know at this point is that it lets us read input from the keyboard.

MULTIPLE RETURN VALUES FROM A FUNCTION OR METHOD

We're trying to read the user's keyboard input, but we're getting an error. The compiler is reporting a problem in this line of code:

```
input := reader.ReadString('\n')      Error → multiple-value reader.ReadString()  
                                         in single-value context
```

The problem is that the `ReadString` method is trying to return *two* values, and we've only provided *one* variable to assign a value to.

In most programming languages, functions and methods can only have a single return value, but in Go, they can return any number of values. The most common use of multiple return values in Go is to return an additional error value that can be consulted to find out if anything went wrong while the function or method was running. A few examples:



```
bool, error := strconv.ParseBool("true") ← Returns an error if the string  
file, error := os.Open("myfile.txt") ← Returns an error if the file can't be opened.  
response, error := http.Get("http://golang.org") ← Returns an error if the site can't be retrieved.
```

So what's the big deal? Just add a variable to hold that error, and then ignore it!

Go doesn't allow us to declare a variable unless we use it.

Go requires that every variable that gets *declared* must also get *used*, somewhere in your program. If we add an `error` variable and then don't check it, our code won't compile. Unused variables often indicate a bug, so this is an example of Go helping you detect and fix bugs!

```
// pass_fail reports whether a grade is...
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    fmt.Println("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input, error := reader.ReadString('\n')
    fmt.Println(input)
}

...we'll get an error! Error → error declared and not used
```

OPTION 1: IGNORE THE ERROR RETURN VALUE WITH THE BLANK IDENTIFIER

The `ReadString` method returns a second error value along with the user's input, and we need to do something with the second value. We've tried just adding a second variable and ignoring it, but our code still won't compile.

```
input, error := reader.ReadString('\n') Error → error declared and not used
```

When we have a value that would normally be assigned to a variable, but that we don't intend to use, we can use Go's **blank identifier**. Assigning a value to the blank identifier essentially discards it (while making it clear to others reading your code that you are doing so). To use the blank identifier, simply type a single underscore (_) character in an assignment statement, where you would normally type a variable name.

Let's try using the blank identifier in place of our old `error` variable:

```
// pass_fail reports whether a grade is passing or failing.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    fmt.Println("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input, _ := reader.ReadString('\n')
    fmt.Println(input)
}
```

Use the blank identifier as a placeholder for the error value.

Now to try the change out. In your terminal, change to the directory where you saved "`pass_fail.go`", and run the program with:

```
go run pass_fail.go
```

The terminal window shows the command `$ go run pass_fail.go` being entered, followed by the prompt `Enter a grade:` . The user types `100` and presses enter. The terminal then displays the echoed input `100`.

Annotations in the image:

- An arrow points from the text "Run `pass_fail.go`..." to the terminal command line.
- An arrow points from the text "Type a number, then press Enter." to the terminal prompt.
- An arrow points from the text "Your number will be printed out in response." to the terminal output line.

When you type a grade (or any other string) at the prompt and press enter, your entry will be echoed back to you. Our program is working!

OPTION 2: HANDLE THE ERROR



I don't know... doesn't
ignoring the error seem
kind of... sloppy?

That's true. If an error actually occurred, this program wouldn't tell us!

If we got an error back from the `ReadString` method, the blank identifier would just cause the error to be ignored, and our program would proceed anyway, possibly with invalid data.

```
func main() {
    fmt.Print("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input, _ := reader.ReadString('\n')
    fmt.Println(input)
}
```

Ignores any error return value!

Prints what may be an invalid value!

A code snippet showing a Go function named `main`. It prints a prompt, creates a `bufio.Reader` from `os.Stdin`, reads a string until a newline, and then prints the input. Handwritten annotations explain that the error return value from `ReadString` is ignored, and that the printed value might be invalid.

In this case, it would be more appropriate to alert the user and stop the program if there was an error.

The `log` package has a `Fatal` function that can do both of these operations for us at once: log a message to the terminal *and* stop the program. ("Fatal" in this context means reporting an error that "kills" your program.)

Let's get rid of the blank identifier, and replace it with an `error` variable so that we're recording the error again. Then, we'll use the `Fatal` function to log the error and halt the program.

```

// pass_fail reports whether a grade is passing or failing.
package main

import (
    "bufio"
    "fmt"
    "log" ← Add the "log" package.
    "os"
)

func main() {
    Go back to storing fmt.Println("Enter a grade: ")
    the error return reader := bufio.NewReader(os.Stdin)
    value in a variable. input, error := reader.ReadString('\n')
    log.Fatal(error) ← Report the error and stop the program.
    fmt.Println(input)
}

```

But if we try running this updated program, we'll see there's a new problem...

CONDITIONALS

If our program encounters a problem reading input from the keyboard, we've set it up to report the error and stop running. But now, it stops running even when everything's working correctly!

```

Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 100
2018/03/11 18:27:08 <nil>
exit status 1
$ ← The error value is "nil".

```

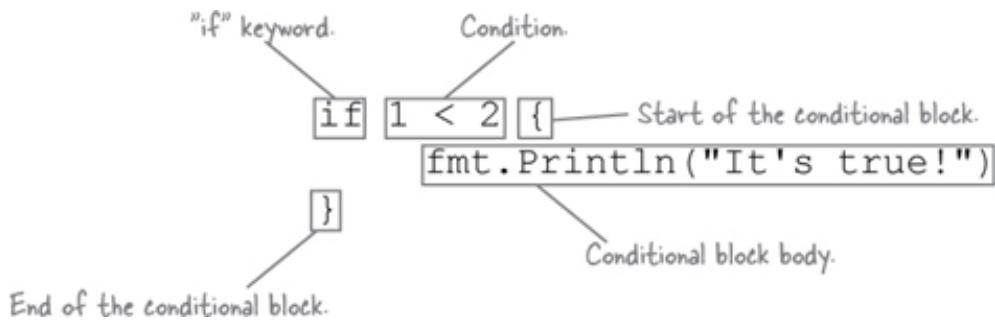
An error gets logged even if everything's working correctly! →

Store the error return value in a variable.

Log the error return value.

Functions and methods like `ReadString` return an error value of **nil**, which basically means "there's nothing there". In other words, if `error` is `nil`, it means there was no error. But our program is set up to simply report the `nil` error! What we *should* do is exit the program only if the `error` variable has a value other than `nil`.

We can do this using **conditionals**: statements that cause a block of code (one or more statements surrounded by {} braces) to be executed only if a condition is met.



An expression is evaluated, and if its result is `true`, the code in the conditional block body is executed. If it's `false`, the conditional block is skipped.

```
if true {           if false {
    fmt.Println("I'll be printed!")   fmt.Println("I won't!"
}                   }
```

As with most other languages, Go supports multiple branches in the condition. These statements take the form `if/else if/else`.

```
if score == 100 {
    fmt.Println("Perfect!")
} else if score >= 60 {
    fmt.Println("You pass.")
} else {
    fmt.Println("You fail!")
}
```

Conditionals rely on a boolean expression (one that evaluates to `true` or `false`) to decide whether the code they contain should be executed.

```
if 1 == 1 {           if 1 >= 2 {
    fmt.Println("I'll be printed!")   fmt.Println("I won't!")
}                   }

if 1 > 2 {           if 2 <= 2 {
    fmt.Println("I won't!")         fmt.Println("I'll be printed!")
}                   }

if 1 < 2 {           if 2 != 2 {
    fmt.Println("I'll be printed!")   fmt.Println("I won't!")
}                   }
```

When you need to execute code only if a condition is *false*, you can use `!`, the boolean

negation operator, which lets you take a `true` value and make it `false`, or a `false` value and make it `true`.

```
if !true {           if !false {  
    fmt.Println("I won't be printed!")      fmt.Println("I will!")  
}  
}
```

If you want to run some code only if two conditions are *both* true, you can use the `&&` ("and") operator. If you want it to run if *either* of two conditions is true, you can use the `||` ("or") operator.

```
if true && true {           if false || true {  
    fmt.Println("I'll be printed!")      fmt.Println("I'll be printed!")  
}  
}
```

```
if true && false {           if false || false {  
    fmt.Println("I won't!")        fmt.Println("I won't!")  
}  
}
```

THERE ARE NO DUMB QUESTIONS

Q: My other programming language requires that an `if` statement's condition be surrounded with parentheses. Doesn't Go?

A: No, and in fact the `go fmt` tool will remove any parentheses you add, unless you're using them to set order of operations.



Because they're in conditional blocks, only some of the `Println` calls in the below code will be executed. Write down what the output would be.

(We've done the first two lines for you.)

```

if true {
    fmt.Println("true")
}
if false {
    fmt.Println("false")
}
if !false {
    fmt.Println("!false")
}
if true {
    fmt.Println("if true")
} else {
    fmt.Println("else")
}
if false {
    fmt.Println("if false")
} else if true {
    fmt.Println("else if true")
}
if 12 == 12 {
    fmt.Println("12 == 12")
}
if 12 != 12 {
    fmt.Println("12 != 12")
}
if 12 > 12 {
    fmt.Println("12 > 12")
}
if 12 >= 12 {
    fmt.Println("12 >= 12")
}
if 12 == 12 && 5.9 == 5.9 {
    fmt.Println("12 == 12 && 5.9 == 5.9")
}
if 12 == 12 && 5.9 == 6.4 {
    fmt.Println("12 == 12 && 5.9 == 6.4")
}
if 12 == 12 || 5.9 == 6.4 {
    fmt.Println("12 == 12 || 5.9 == 6.4")
}

```

Output:

true

false

Answers on page 42.

LOGGING A FATAL ERROR, CONDITIONALLY

Our grading program is reporting an error and exiting, even if it reads input from the keyboard successfully.

Store the error return
value in a variable.

```

input, error := reader.ReadString('\n')
log.Fatal(error) ← Log the error return value.

```

An error gets logged even if
everything's working correctly!

```

Shell Edit View Window Help
$ go run pass fail.go
Enter a grade: 100
2018/03/11 18:27:08 <nil>
exit status 1
$ 

```

The error value is "nil".

We know that if the value in our `error` variable is `nil`, it means reading from the keyboard was successful. Now that we know about `if` statements, let's try updating our code to log an error and exit only if `error` is *not* `nil`.

```
// pass_fail reports whether a grade is passing or failing.
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
)

func main() {
    fmt.Print("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input, error := reader.ReadString('\n')
IF "error" is not nil...————→ if error != nil {
    log.Fatal(error) ← Report the error and
    }                                stop the program.
    fmt.Println(input)
}
```

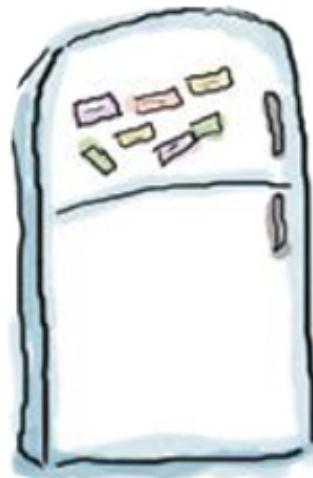
If we re-run our program, we'll see that it's working again. And now, if there are any errors when reading user input, we'll see those as well!

Run `pass_fail.go`... →

Your number will be printed out in response. →

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 100
100
$
```

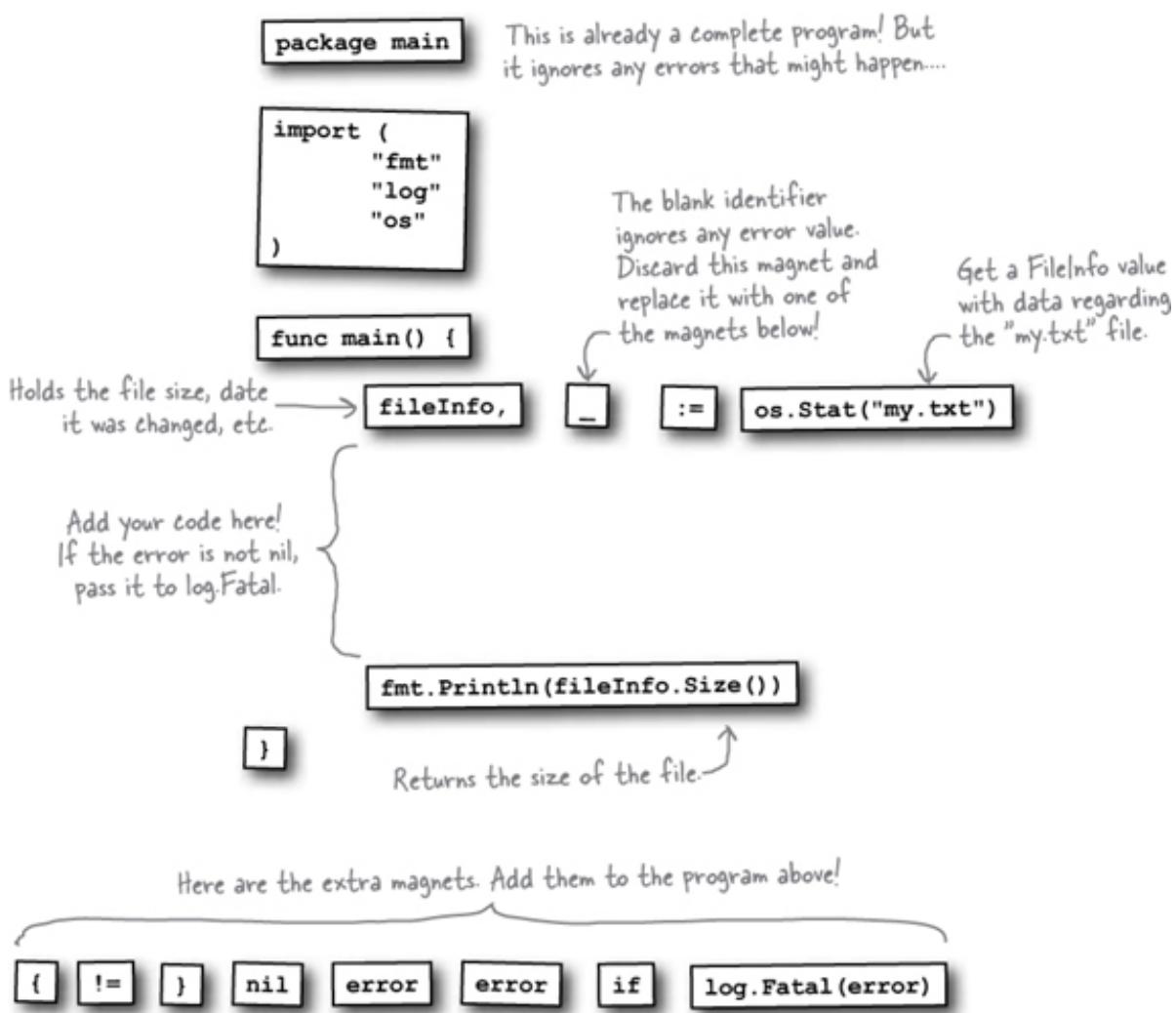
CODE MAGNETS



A Go program that prints the size of a file is on the fridge. It calls the `os.Stat` function, which returns an `os.FileInfo` value, and possibly an error value. Then it calls the `Size` method on the `FileInfo` value to get the file size.

But the original program uses the `_` blank identifier to ignore the error value from `os.Stat`. If an error occurs (which could happen if the file doesn't exist), this will cause the program to fail.

Reconstruct the extra code snippets to make a program that works just like the original one, but also checks for an error from `os.Stat`. If the error from `os.Stat` is not `nil`, the error should be reported, and the program should exit. Discard the magnet with the `_` blank identifier; it won't be used in the finished program.



CONVERTING STRINGS TO NUMBERS Answers on page 43.

Conditional statements will also let us evaluate the entered grade. Let's add an `if/else` statement to determine whether the grade is passing or failing. If the entered percentage grade is 60 or greater, we'll set the status to "passing". Otherwise, we'll set it to "failing".

```
// package and import statements omitted
func main() {
    fmt.Print("Enter a grade: ")
}
```

```

reader := bufio.NewReader(os.Stdin)
input, error := reader.ReadString('\n')
if error != nil {
    log.Fatal(error)
}

if input >= "60" {
    status := "passing"
} else {
    status := "failing"
}
}

```

In its current form, though, this gets us a compilation error.

Error → cannot convert 60 to type string
invalid operation: input >= 60 (mismatched types string and int)

Here's the problem: input from the keyboard is read in as a string. Go can only compare to other numbers; we can't compare a number with a string. And there's no direct type conversion from `string` to a number:

```

float64("200")
Error → cannot convert "200" (type string) to type float64

```

There are a pair of issues we'll need to address here:

- The `input` string still has a newline character on the end, from when the user pressed the Enter key while entering it. We need to strip that off.
- The remainder of the string needs to be converted to a floating-point number.

Removing the newline character from the end of the `input` string will be easy. The `strings` package has a `TrimSpace` function that will remove all whitespace characters (newlines, tabs, and regular spaces) from the start and end of a string.

```

s := "\t formerly surrounded by space \n"
fmt.Println(strings.TrimSpace(s))
                                formerly surrounded by space

```

So, we can get rid of the newline on `input` by passing it to `TrimSpace`, and assigning the return value back to the `input` variable.

```
input = strings.TrimSpace(input)
```

All that should remain in the `input` string now is the number the user entered. We can use the `strconv` package's `ParseFloat` function to convert it to a `float64` value.

Arguments are the string you want to convert...
grade, error := strconv.ParseFloat(input, 64)
...and the number of bits of precision for the result.
Return values are a float64....
...and possibly an error.

You pass `ParseFloat` a string that you want to convert to a number, as well as the number of bits of precision the result should have. Since we're converting to a `float64` value, we pass the number `64`. (In addition to `float64`, Go offers a less-precise `float32` type, but you shouldn't use that unless you have a good reason.)

`ParseFloat` converts the string to a number, and returns it as a `float64` value. Like `ReadString`, it also returns a second error value, which will be `nil` unless there was some problem converting the string. (For example, a string that *can't* be converted to a number. We don't know of a numeric equivalent to "hello"...)



This whole "bits of precision" thing isn't that important right now.

It's basically just a measure of how much computer memory a floating-point number takes up. As long as you know that you want a `float64`, and so you should pass `64` as the second argument to `ParseFloat`, you'll be fine.

Let's update `pass_fail.go` with calls to `TrimSpace` and `ParseFloat`:

```

// pass_fail reports whether a grade is passing or failing.
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "strings"           Add "strings" so we can use
    "strconv"          Add "strconv" so we can use
                        TrimSpace function.
                        ParseFloat
)

func main() {
    fmt.Println("Enter a grade: ")
    reader := bufio.NewReader(os.Stdin)
    input, error := reader.ReadString('\n')
    if error != nil {
        log.Fatal(error)
    }
    input = strings.TrimSpace(input)           Trim the newline character
                                            from the input string.
    grade, error := strconv.ParseFloat(input, 64)   Convert the string to a
                                                    float64 value.
    Just as with ReadString, report any error when converting. { if error != nil {
        log.Fatal(error)
    }

    if grade >= 60 {
        status := "passing"
    } else {
        status := "failing"
    }
}

```

First, we add the appropriate packages to the `import` section. We add code to remove the newline character from the `input` string. Then we pass `input` to `ParseFloat`, and store the resulting `float64` value in a new variable, `grade`.

Just as we did with `ReadString`, we test whether `ParseFloat` returns an error value. If it does, we report it and stop the program.

Finally, we update the conditional statement to test the number in `grade`, rather than the string in `input`. That should fix the error stemming from comparing a string to a number.

If we try to run the updated program, we no longer get the "mismatched types `string` and `int`" error. So it looks like we've fixed that issue. But we've got a couple more errors to address. We'll look at those next.

↓ Errors:

```

status declared
and not used
status declared
and not used

```

BLOCKS

We've converted the user's grade input to a `float64` value, and added it to a conditional to determine if it's passing or failing. But we're getting a couple more compile errors:

```
if grade >= 60 {  
    status := "passing"  
} else {  
    status := "failing"  
}
```

↓ Errors.
status declared
and not used
status declared
and not used

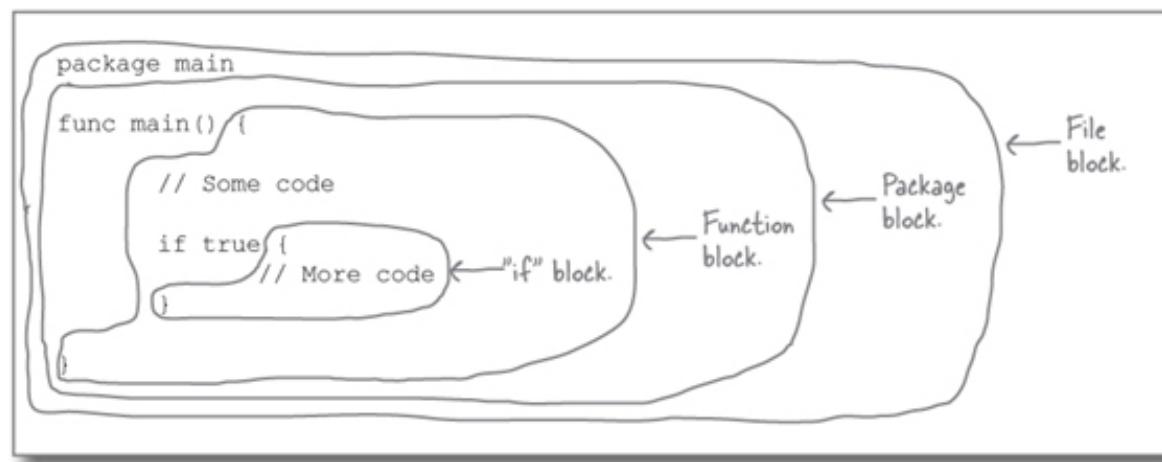
As we've seen previously, declaring a variable like `status` without using it afterwards is an error in Go. It seems a little strange that we're getting the error twice, but let's disregard that for now. We'll add a call to `Println` to print the percentage grade we were given, and the value of `status`.

```
func main {  
    // Omitting code up here...  
    if grade >= 60 {  
        status := "passing"  
    } else {  
        status := "failing"  
    }  
    fmt.Println("A grade of", grade, "is", status)  
}
```

↓ Print status
variable.
↓ Error
undefined: status

But now we get a *new* error, saying that the `status` variable is undefined when we attempt to use it in our `Println` statement! What's going on?

Go code can be divided up into **blocks**, segments of code. Blocks are usually surrounded by curly braces (`{}`), although there are also blocks at the source code file and package levels. Blocks can be nested inside one another.

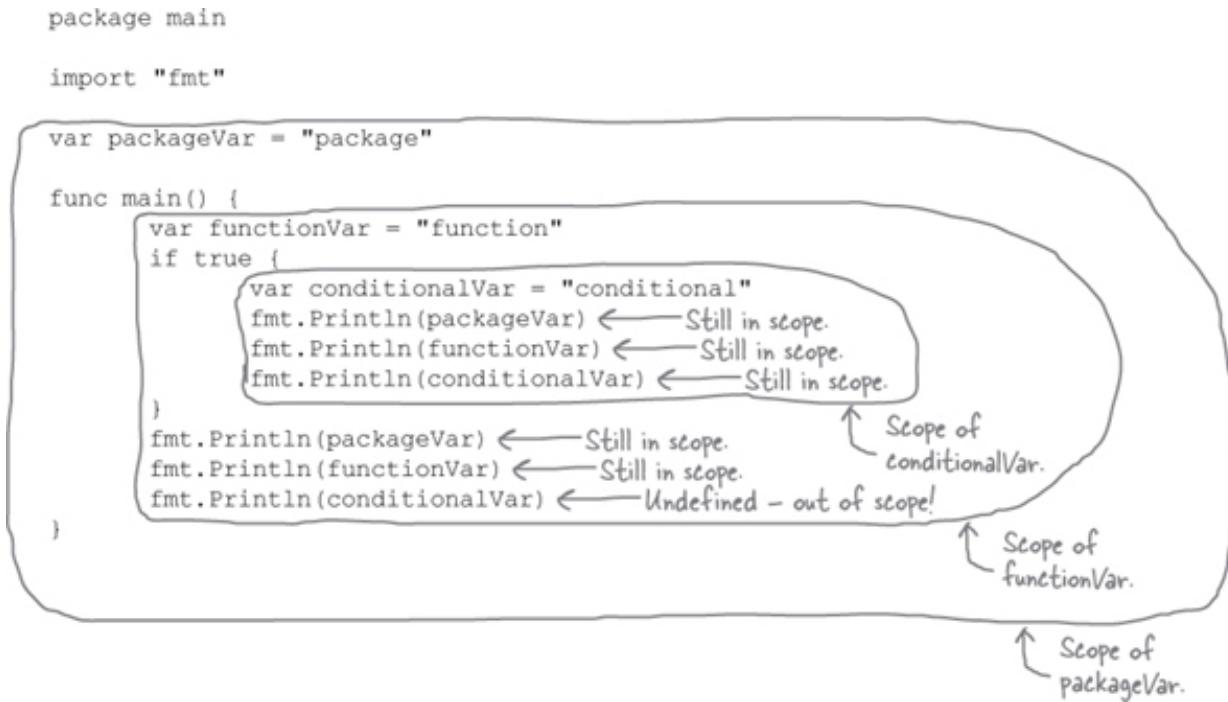


The bodies of functions and conditionals are both blocks as well. Understanding this will be key to solving our problem with the `status` variable...

BLOCKS AND VARIABLE SCOPE

Each variable you declare has a **scope**: a portion of your code that it's "visible" within. A declared variable can be accessed anywhere within its scope, but if you try to access it outside that scope, you'll get an error.

A variable's scope consists of the block it's declared in, and any blocks nested within that block.



Here are the scopes of the variables in the code above:

- The scope of `packageVar` is the entire `main` package. You can access `packageVar` anywhere within any function you define in the package.
- The scope of `functionVar` is the entire function it's declared in, including the `if` block nested within that function.
- The scope of `conditionalVar` is limited to the `if` block. When we try to access `conditionalVar` *after* the closing `}` brace of the `if` block, we'll get an error saying that `conditionalVar` is undefined!

Now that we understand variable scope, we can explain why our `status` variable was undefined in the grading program. We declared `status` in our conditional blocks. (In fact, we declared it twice, since there are two separate blocks. That's why we got two "status declared and not used" errors.) But then we tried to access `status` *outside* those blocks, where it was no longer in scope.

```

func main() {
    // Omitting code up here...
    if grade >= 60 {
        status := "passing" ← "if" block.
    } else {
        status := "failing" ← "else" block.
    }
    fmt.Println("A grade of", grade, "is", status) ← NO "status" variable has been
}                                            defined in this scope!

```

Error → undefined: status

The solution is to move the declaration of the `status` variable out of the conditional blocks, and up to the function block. Once we do that, the `status` variable will be in scope both within the nested conditional blocks, *and* at the end of the function block.

```

func main() {
    // Omitting code up here...
    var status string ← Move declaration here.
    if grade >= 60 {
        status = "passing" ← Change these
    } else {                                to assignment
        status = "failing" ← statements.
    }
    fmt.Println("A grade of", grade, "is", status) ← Now, "status" will be in
}                                            scope at the end of the
                                                function block.

```



Watch it!

Don't forget to change the short variable declarations within the nested blocks to assignment statements!

If you don't change both occurrences of `:=` to `=`, you'll accidentally create new variables named `status` within the nested conditional blocks, which will then be out of scope at the end of the enclosing function block!

WE'VE FINISHED THE GRADING PROGRAM!

That was it! Our `pass_fail.go` program is ready for action! Let's take one more look at the complete code:

It's conventional to include a comment at the top describing what the program does.

```
// pass_fail reports whether a grade is passing or failing.
package main // Executable programs must go in the "main" package.

{ Import all required packages.
    import (
        "bufio"
        "fmt"
        "log"
        "os"
        "strings"
        "strconv" The "main" function gets invoked when the program launches.
    )

    func main() { Prompt the user to enter a percentage grade.
        fmt.Println("Enter a grade: ") Create a bufio.Reader, which lets us read keyboard input.
        reader := bufio.NewReader(os.Stdin)
        input, error := reader.ReadString('\n') If there's an error, print the message and exit.
        if error != nil {
            log.Fatal(error)
        }
        input = strings.TrimSpace(input) If there's an error, print the message and exit.
        grade, error := strconv.ParseFloat(input, 64) rest of the method.
        if error != nil {
            log.Fatal(error) Declare "status" variable here so it's in scope for the
        }
        status := "failing" Print the entered grade... ...and the pass/fail status.
        if grade >= 60 {
            status = "passing"
        }
        fmt.Println("A grade of", grade, "is", status)
    }
}
```

You can try running the finished program as many times as you like. Enter a percentage grade under 60, and it will report a failing status. Enter a grade over 60, and it will report that it's passing. Looks like everything's working!

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 56
A grade of 56 is failing
$ go run pass_fail.go
Enter a grade: 84.5
A grade of 84.5 is passing
$
```



Some of the lines of code below will result in a compile error, because they refer to a variable that is out of scope. Cross out the lines that have errors.

```
package main

import (
    "fmt"
)

var a = "a"

func main() {
    a = "a"
    b := "b"
    if true {
        c := "c"
        if true {
            d := "d"
            fmt.Println(a)
            fmt.Println(b)
            fmt.Println(c)
            fmt.Println(d)
        }
        fmt.Println(a)
        fmt.Println(b)
        fmt.Println(c)
        fmt.Println(d)
    }
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
}
```

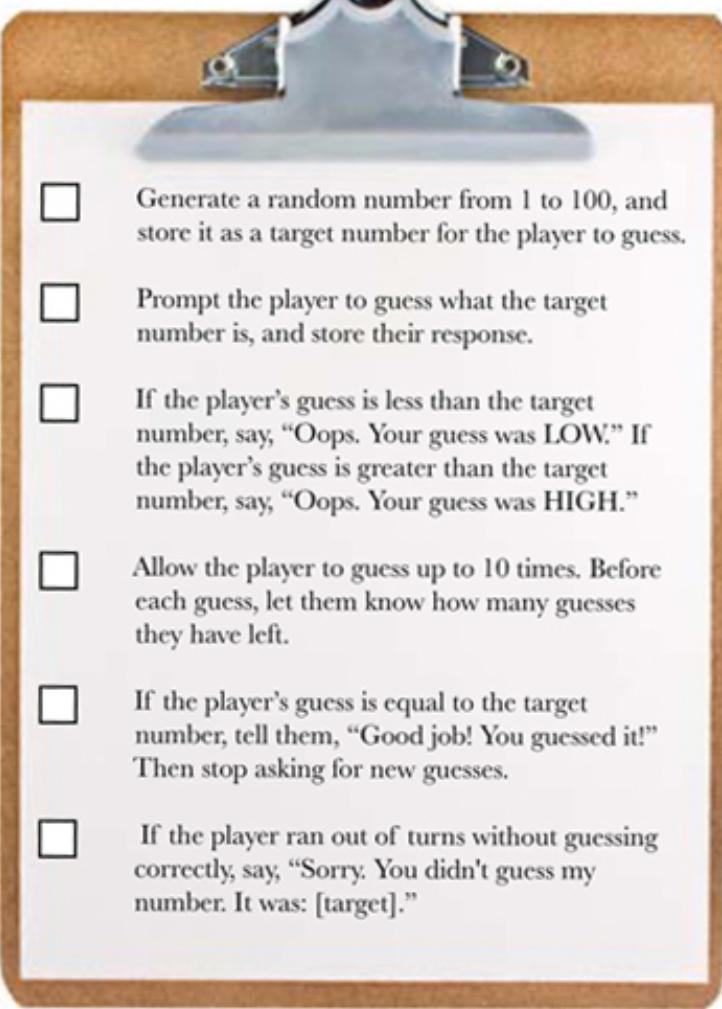
Answers on page 44.

LET'S BUILD A GAME

We're going to wrap up this chapter by building a simple game. If that sounds daunting, don't worry; you've already learned most of the skills you're going to need! Along the way, we'll learn about *loops*, which will allow the player to take multiple turns.

Let's look at everything we'll need to do:

This example debuted in Head First Ruby. (Another fine book that you should also buy!) It worked so well that we're using it again here.

- 
- Generate a random number from 1 to 100, and store it as a target number for the player to guess.
 - Prompt the player to guess what the target number is, and store their response.
 - If the player's guess is less than the target number, say, "Oops. Your guess was LOW." If the player's guess is greater than the target number, say, "Oops. Your guess was HIGH."
 - Allow the player to guess up to 10 times. Before each guess, let them know how many guesses they have left.
 - If the player's guess is equal to the target number, tell them, "Good job! You guessed it!" Then stop asking for new guesses.
 - If the player ran out of turns without guessing correctly, say, "Sorry. You didn't guess my number. It was: [target]."

I've put together
this list of
requirements for you.
Can you handle it?



Gary Richardott
Game Designer

Let's create a new source file, named "guess.go".

It looks like our first requirement is to generate a random number. Let's get started!

PACKAGE NAMES VS. IMPORT PATHS

The `math/rand` package has a `Intn` function that can generate a random number for us, so we'll need to import `math/rand`. Then we'll call `rand.Intn` to generate the random number.

```
package main

import (
    "fmt"
    "math/rand" ← Import the "math/rand"
)
func main() {           ↗ Call rand.Intn to generate a
    target := rand.Intn(100) + 1
    fmt.Println(target)
}
```

random number.



Hang on a second! You said `Intn` came from the `math/rand` package. So why are you just typing `rand.Intn` and not `math/rand.Intn`?

One is the package's import path, and the other is the package's name.

When we say `math/rand` we're referring to the package's *import path*, not its *name*. An **import path** is just a unique string that identifies a package, that you use in an `import` statement. Once you've imported the package, then you can refer to it by its package name.

For every package we've used so far, the import path has been identical to the package name. Here are a few examples:

Import Path	Package Name
<code>"fmt"</code>	<code>fmt</code>
<code>"log"</code>	<code>log</code>
<code>"strings"</code>	<code>strings</code>

But the import path and package name don't have to be identical. Many Go packages fall into similar categories, like compression or complex math. So they're grouped together under similar import path prefixes, such as `"archive/"` or `"math/"`. (Think of them as being similar to the paths of directories on your hard drive.)

Import Path	Package Name
"archive"	archive
"archive/tar"	tar
"archive/zip"	zip
"math"	math
"math/cmplx"	cmplx
"math/rand"	rand

The Go language doesn't require that a package name have anything to do with its import path. But by convention, the last (or only) segment of the import path is also used as the package name. So if the import path is "archive", the package name will be `archive`, and if the import path is "archive/zip", the package name will be `zip`.

Import Path	Package Name
" <u>archive</u> "	<u>archive</u>
"archive/ <u>tar</u> "	<u>tar</u>
"archive/ <u>zip</u> "	<u>zip</u>
" <u>math</u> "	<u>math</u>

"math/ <u>cmplx</u> "	<u>cmplx</u>
"math/ <u>rand</u> "	<u>rand</u>

So, that's why our `import` statement uses a path of "`math/rand`", but our `main` function just uses the package name: `rand`.

```
package main

import (
    "fmt"
    "math/rand" ← Use the full import path
    "math/rand" ← for "math/rand".
)
func main() { ↓ Use the package name: "rand".
    target := rand.Intn(100) + 1
    fmt.Println(target)
}
```

GENERATING A RANDOM NUMBER

Pass a number to `rand.Intn`, and it will return a random integer between 0 and the number you provided. In other words, if we pass an argument of `100`, we'll get a random number in the range 0-99. Since we need a number in the range 1-100, we'll just add 1 to whatever random value we get. We'll store the result in a variable, `target`. We'll do more with `target` later, but for now we'll just print it.

```
package main

import (
    "fmt"
    "math/rand"
)
func main() { ↓ Generate an integer
    target := rand.Intn(100) + 1 ← from 0-99. Add 1 to make
    fmt.Println(target)           ← it an integer from 1-100.
}
```

If we try running our program right now, we'll get a random number. But we just get the *same* random number over and over! The problem is, random numbers generated by computers aren't really that random. But there's a way increase that randomness...

```

Shell Edit View Window Help
$ go run guess.go
81
$ go run guess.go
81
$ go run guess.go
81
$ 

```

To get different random numbers, we're going to need to pass a value to the `rand.Seed` function. That will "seed" the random number generator, that is, give it a value that it will use to generate other random values. But if we keep giving it the same seed value, it will keep giving us the same random values, and we'll be back where we started.

We saw earlier that the `time.Now` function will give us a `Time` value representing the current date and time. We can use that to get a different seed value every time we run our program.

```

package main

import (
    "fmt"
    "math/rand" // Import the "time" package as well.
    "time"
)

func main() {
    seconds := time.Now().Unix() // Get the current date and time, as an integer.
    rand.Seed(seconds) // Seed the random number generator.
    target := rand.Intn(100) + 1
    fmt.Println("I've chosen a random number between 1 and 100.")
    fmt.Println("Can you guess it?")
    fmt.Println(target)
}

Now, the generated numbers should be different each time!

```

Get the current date and time, as an integer.

Seed the random number generator.

Let the player know we've chosen a number.

The `rand.Seed` function expects an integer, so we can't pass it a `Time` value directly. Instead, we call the `Unix` method on the `Time`, which will convert it to an integer. (Specifically, it will convert it to Unix time format, which is an integer with the number of seconds since January 1, 1970. But you don't really need to remember that.) We pass that integer to `rand.Seed`.

We also add a couple `Println` calls to let the user know we've chosen a random number. But aside from that, we can leave the rest of our code, including the call to `rand.Intn`, as-is. Seeding the generator should be the only change we need to make.

Now, each time we run our program, we'll see our message, along with a random number. It looks like our changes are successful!

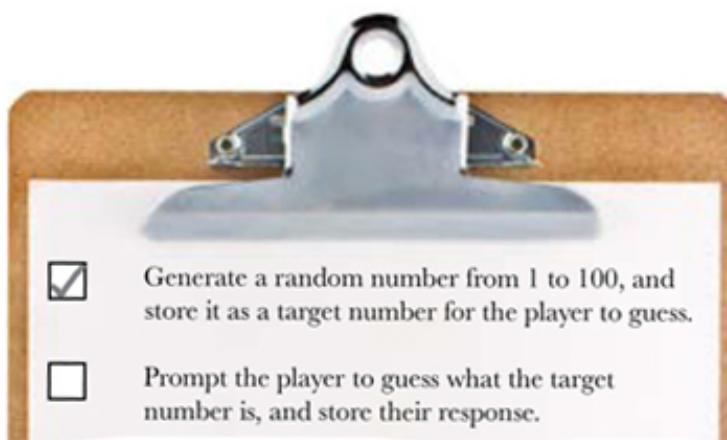
```
Shell Edit View Window Help
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
73
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
18
$
```

A different number each time we run the program!

GETTING AN INTEGER FROM THE KEYBOARD

Our first requirement is complete! Next we need to get the user's guess via the keyboard.

That should work in much the same way as when we read in a percentage grade from the keyboard for our grading program.



There will be only one difference: instead of converting the input to a `float64`, we need to convert it to an `int` (since our guessing game uses only whole numbers). So we'll pass the string read from the keyboard to the `strconv` package's `Atoi` (string to integer) function instead of its `ParseFloat` function. `Atoi` will give us an integer as its return value. (Just like `ParseFloat`, `Atoi` might also give us an error if it can't convert the string. If that happens, we again report the error and exit.)

```

package main

import (
    "bufio" ← Import these additional
    "fmt" ← packages. (We used all of these
    "log" ← in the grading program!)
    "math/rand"
    "os" ←
    "strconv" ← Create a bufio.Reader,
    "strings" ← which lets us read
    "time"
)

func main() {
    seconds := time.Now().Unix()
    rand.Seed(seconds)
    target := rand.Intn(100) + 1
    fmt.Println(target)
    fmt.Println("I've chosen a random number between 1 and 100.")
    fmt.Println("Can you guess it?")
    reader := bufio.NewReader(os.Stdin) ← keyboard input.

    fmt.Print("Make a guess: ") ← Ask for a number.
    input, error := reader.ReadString('\n') ← Read what the user types, up
    If there's an error, print the message and exit. { if error != nil {
        log.Fatal(error) } } until they press Enter.

    input = strings.TrimSpace(input) ← Remove the newline.
    guess, error := strconv.Atoi(input) ← Convert the input string
    If there's an error, print the message and exit. { if error != nil {
        log.Fatal(error) } } to an integer.
}

```

COMPARING THE GUESS TO THE TARGET

Another requirement finished. And this next one will be easy... We just need to compare the user's guess to the randomly generated number, and tell them whether it was higher or lower.

- Prompt the player to guess what the target number is, and store their response.
- If the player's guess is less than the target number, say, "Oops. Your guess was LOW." If the player's guess is greater than the target number, say, "Oops. Your guess was HIGH."

If `guess` is less than `target`, we need to print a message saying the guess was "low".

Otherwise, if `guess` is greater than `target`, we should print a message saying the guess was "high". Sounds like we need an `if/else if` statement. We'll add it below the other code in our `main` function.

```
// No changes to package and import statements; omitting

func main() {
    // No changes to previous code; omitting

    if guess < target {
        fmt.Println("Oops. Your guess was LOW.")
    } else if guess > target {
        fmt.Println("Oops. Your guess was HIGH.") ←
    }
}
```

If the player's guess was low, say so.

If the player's guess was high, say so.

Now try running our updated program from the terminal. It's still set up to print `target` each time it runs, which will be useful for debugging. Just enter a number lower than `target`, and you should be told your guess was "low". If you re-run the program, you'll get a new `target` value. Enter a number higher than that and you'll be told your guess was "high".

```
Shell Edit View Window Help
$ go run guess.go
81
I've chosen a random number between 1 and 100.
Can you guess it?
Make a guess: 1
Oops. Your guess was LOW.
$ go run guess.go
54
I've chosen a random number between 1 and 100.
Can you guess it?
Make a guess: 100
Oops. Your guess was HIGH.
$
```

LOOPS

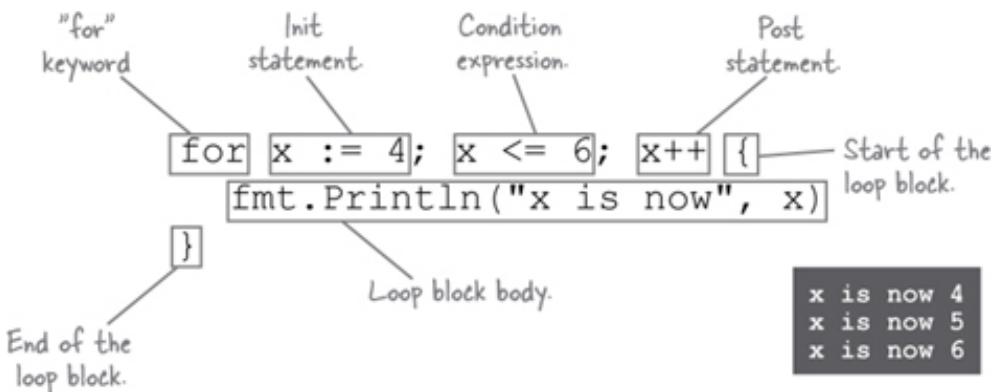
Another requirement down! Let's look at the next one.

Currently, the player only gets to guess once, but we need to allow them to guess up to 10 times.

- If the player's guess is less than the target number, say, "Oops. Your guess was LOW." If the player's guess is greater than the target number, say, "Oops. Your guess was HIGH."
- Allow the player to guess up to 10 times. Before each guess, let them know how many guesses they have left.

The code to prompt for a guess is already in place. We just need to run it *more than once*. We can use a **loop** to execute a block of code repeatedly. If you've used other programming languages, you've probably encountered loops. When you need one or

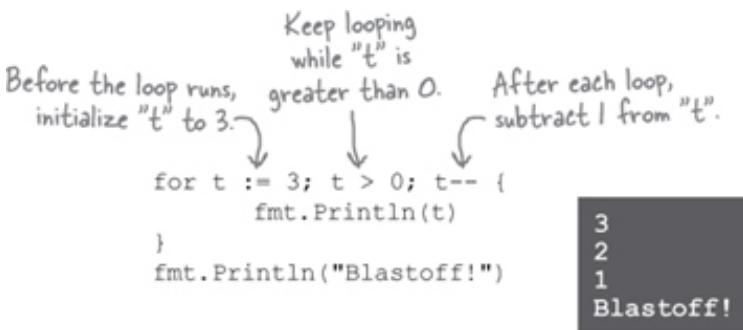
more statements executed over and over, you place them inside a loop.



Loops always begin with the `for` keyword. In one common kind of loop, `for` is followed by three segments of code that control the loop:

- An init statement that is usually used to initialize a variable
- A condition expression that determines when to break out of the loop
- A post statement that runs after each iteration of the loop

Often, the init statement is used to initialize a variable, the condition expression keeps the loop running until that variable reaches a certain value, and the post statement is used to update the value of that variable. For example, in this snippet, the `t` variable is initialized to 3, the condition keeps the loop going while `t > 0`, and the post statement subtracts 1 from `t` each time the loop runs. Eventually, `t` reaches 0 and the loop ends.



The `++` and `--` statements are frequently used in loop post statements. Each time they're evaluated, `++` adds 1 to a variable's value, and `--` subtracts 1.

```
x := 0
x++
fmt.Println(x)
x++
fmt.Println(x)
x--
fmt.Println(x)
```

1
2
1

Used in a loop, `++` and `--` are convenient for counting up or down.

```
for x := 1; x <= 3; x++ {  
    fmt.Println(x)  
}
```

1
2
3

```
for x := 3; x >= 1; x-- {  
    fmt.Println(x)  
}
```

3
2
1

Go also includes the assignment operators `+=` and `-=`. They take the value in a variable, add or subtract another value, and then assign the result back to the variable.

```
x := 0  
x += 2  
fmt.Println(x)  
x += 5  
fmt.Println(x)  
x -= 3  
fmt.Println(x)
```

2
7
4

`+=` and `-=` can be used in a loop to count in increments other than 1.

```
for x := 1; x <= 5; x += 2 {  
    fmt.Println(x)  
}
```

1
3
5

```
for x := 15; x >= 5; x -= 5 {  
    fmt.Println(x)  
}
```

15
10
5

When the loop finishes, execution will resume with whatever statement follows the loop block. But the loop will keep going as long as the condition expression evaluates to `true`. It's possible to abuse this; here are a loop that will run forever, and a loop that will never run at all:

```
for x := 1; true; x++ {  
    fmt.Println(x)  
}  
  
for x := 1; false; x++ {  
    fmt.Println(x)  
}
```

Infinite loop!

Loop that never runs!



Watch it!

It's possible for a loop to run forever, in which case your program will never stop on its own.

If this happens, with the terminal active, hold the Control key and press C to halt your program.

INIT AND POST STATEMENTS ARE OPTIONAL

If you want, you can leave out the init and post statements from a `for` loop, leaving only the condition expression. (Although you still need to make sure the condition eventually evaluates to `false`, or you could have an infinite loop on your hands.)

The diagram illustrates two examples of `for` loops:

Example 1: `x := 1` (Declare `x` in a separate statement) followed by `for x <= 3 {` (Use only the condition expression). Inside the loop, `fmt.Println(x)` and `x++` are shown. Below the loop, a vertical stack shows the values 1, 2, and 3.

Example 2: `x := 3` (Declare `x` in a separate statement) followed by `for x >= 1 {` (Use only the condition expression). Inside the loop, `fmt.Println(x)` and `x--` are shown. Below the loop, a vertical stack shows the values 3, 2, and 1.

LOOPS AND SCOPE

Just like with conditionals, the scope of any variables declared within a loop's block is limited to that block. (Although the init statement, condition expression, and post statement can be considered part of that scope as well.)

```
for x := 1; x <= 3; x++ {
    y := x + 1
    fmt.Println(y) ← Still in scope...
}
fmt.Println(y) ← Error: out of scope!
```

```
for x := 1; x <= 3; x++ {
    fmt.Println(x) ← Still in scope...
}
fmt.Println(x) ← Error: out of scope!
```

Also as with conditionals, any variable declared *before* the loop will still be in scope within the loop's control statements and block, *and* will still be in scope after the loop exits.

```
var x int ← Declared outside loop...
No need to declare x here, just assign to it!
for x = 1; x <= 3; x++ {
    ↑fmt.Println(x) ← Still in scope...
}
fmt.Println(x) ← Still in scope.
```



BREAKING STUFF IS EDUCATIONAL!

Here's a program that uses a loop to count to 3. Try making one of the changes below, and run it. Then undo your change, and try the next one. See what happens!

```
package main  
  
import "fmt"  
  
func main() {  
    for x := 1; x <= 3; x++ {  
        fmt.Println(x)  
    }  
}
```

1
2
3

If you do this...	...it will break because...
Add parentheses after the <code>for</code> keyword <code>for (x := 1; x <= 3; x++)</code>	Some other languages <i>require</i> parentheses around a <code>for</code> loop's control statements, but not only does Go not require them, it doesn't <i>allow</i> them.
Delete the <code>:</code> from the init statement <code>x /= 1</code>	Unless you're assigning to a variable that's already been declared in the enclosing scope (which you usually won't be), the init statement needs to be a <i>declaration</i> , not an <i>assignment</i> .
Remove the <code>=</code> from the condition expression <code>x < 3</code>	The expression <code>x < 3</code> becomes <code>false</code> when <code>x</code> reaches 3 (whereas <code>x <= 3</code> would still be <code>true</code>). So the loop would only count to 2.
Reverse the comparison in the condition expression <code>x > 3</code>	Because the condition is already false when the loop begins (<code>x</code> is initialized to 1, which is <i>less</i> than 3), the loop will never run.

<code>x >= 3</code>	
<code>x++</code> <code>x--</code>	The <code>x</code> variable will start counting <i>down</i> from 1 (1, 0, -1, -2, etc.), and since it will never be greater than 3, the loop will never end.
<code>fmt.Println(x)</code> Move the <code>fmt.Println(x)</code> statement outside the loop's block	Variables declared in the init statement or within the loop block are only in scope within the loop block.



Look carefully at the init statement, condition expression, and post statement for each of these loops. Then write what you think the output will be for each one.

(We've done the first one for you.)

```
for x := 1; x <= 3; x++ {
    fmt.Println(x)
}
```

123

```
for x := 3; x >= 1; x-- {
    fmt.Println(x)
}
```

```
for x := 2; x <= 3; x++ {
    fmt.Println(x)
}
```

```
for x := 1; x < 3; x++ {
    fmt.Println(x)
}
```

```
for x := 1; x <= 3; x+= 2 {
    fmt.Println(x)
}
```

```
for x := 1; x >= 3; x++ {
    fmt.Println(x)
}
```

USING A LOOP IN OUR GUESSING GAME

Our game still only prompts the user for a guess once. Let's add a loop around the code that prompts the user for a guess and tells them if it was "low" or "high", so that the user can guess 10 times.

We'll use an `int` variable named `guesses` to track the number of guesses the player has made. In our loop's init statement, we'll initialize `guesses` to `0`. We'll add `1` to `guesses` with each iteration of the loop, and we'll stop the loop when `guesses` reaches `10`.

We'll also add a `Println` statement at the top of the loop's block to tell the user how many guesses they have left.

```
// No changes to package and import statements; omitting

func main() {
    seconds := time.Now().Unix()
    rand.Seed(seconds)
    target := rand.Intn(100) + 1
    fmt.Println(target)
    fmt.Println("I've chosen a random number between 1 and 100.")
    fmt.Println("Can you guess it?")

    reader := bufio.NewReader(os.Stdin)

    for guesses := 0; guesses < 10; guesses++ { ←
        fmt.Println("You have", 10-guesses, "guesses left.") ←
            Use the "guesses"
            variable to track the
            number of guesses so far.

        fmt.Print("Make a guess: ") ←
            Subtract the number of guesses from 10 to
            tell the player how many they have left.

        input, error := reader.ReadString('\n')
        if error != nil {
            log.Fatal(error)
        }
        input = strings.TrimSpace(input)
        guess, error := strconv.Atoi(input)
        if error != nil {
            log.Fatal(error)
        }

        if guess < target {
            fmt.Println("Oops. Your guess was LOW.")
        } else if guess > target {
            fmt.Println("Oops. Your guess was HIGH.")
        }
    } ←
} ←
End of the for loop.
```

The existing code, which prompts the user for a guess and tells them if it's "low" or "high", will be run 10 times.

Now that our loop is in place, if we run our game again, we'll get asked 10 times what our guess is!

We're still set up to print the target number when the game starts.

Inside the loop, we say how many guesses are left, get the player's guess, and tell them if it was "low" or "high".

Right now, players don't get told when their guess is correct, and the loop doesn't stop.

```
Shell Edit View Window Help
$ go run temp.go
68
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 50
Oops. Your guess was LOW.
You have 9 guesses left.
Make a guess: 75
Oops. Your guess was HIGH.
You have 8 guesses left.
Make a guess: 68
You have 7 guesses left.
Make a guess:
```

Since the code to prompt for a guess and state whether it was "high" or "low" is inside the loop, it gets run repeatedly. After 10 guesses, the loop (and the game) will end.

But the loop always runs 10 times, even if the player guesses correctly! Fixing that will be our next requirement.

SKIPPING PARTS OF A LOOP WITH "CONTINUE" AND "BREAK"

The hard part is done! We only have a couple requirements left to go.

Right now, the loop that prompts the user for a guess always runs 10 times. Even if the player guesses correctly, we don't tell them so, and we don't stop the loop. Our next task is to fix that.

- Allow the player to guess up to 10 times. Before each guess, let them know how many guesses they have left.
- If the player's guess is equal to the target number, tell them, "Good job! You guessed it!" Then stop asking for new guesses.

Go provides two keywords that control the flow of a loop. The first, `continue`, immediately skips to the next iteration of a loop, without running any further code in the loop block.

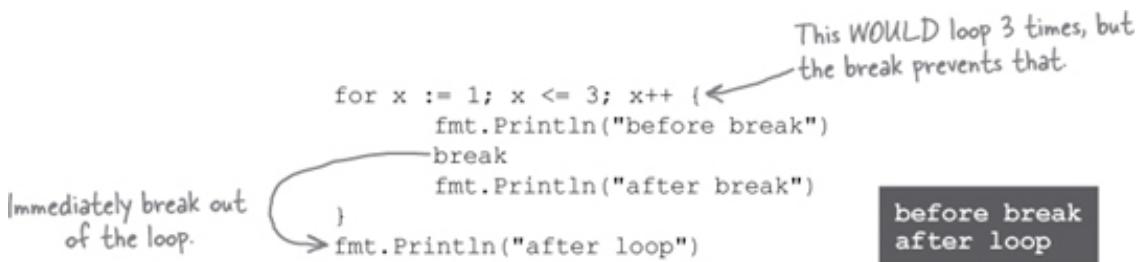
Skip directly back to the top of the loop.

```
for x := 1; x <= 3; x++ {
    fmt.Println("before continue")
    continue
    fmt.Println("after continue")
}
```

```
before continue
before continue
before continue
```

In the above example, the string "after continue" never gets printed, because the `continue` keyword always skips back to the top of the loop before the second call to `Println` can be run.

The second keyword, `break`, immediately breaks out of a loop. No further code within the loop block is executed, and no further iterations are run. Execution moves to the first statement following the loop.



```
for x := 1; x <= 3; x++ {  
    fmt.Println("before break")  
    break  
    fmt.Println("after break")  
}  
fmt.Println("after loop")
```

This diagram illustrates a `for` loop with annotations. A handwritten note on the left says "Immediately break out of the loop." An arrow points from this note to the `break` keyword. Another handwritten note at the top right says "This WOULD loop 3 times, but the break prevents that." An arrow points from this note to the `break` keyword. A small black box on the right contains the text "before break" and "after loop".

Here, in the first iteration of the loop, the string "before break" gets printed, but then the `break` statement immediately breaks out of the loop, without printing the "after break" string, and without running the loop again (even though it normally would have run two more times). Execution instead moves to the statement following the loop.

The `break` keyword seems like it would be applicable to our current problem: we need to break out of our loop when the player guesses correctly. Let's try using it in our game...

BREAKING OUT OF OUR GUESSING LOOP

We're using an `if/else if` conditional to tell the player the status of their guess. If the player guesses a number too "high" or too "low", we currently print a message telling them so.

It stands to reason that if the guess is neither too high *nor* too low, it must be correct. So let's add an `else` branch onto the conditional, that will run in the event of a correct guess. Inside the block for the `else` branch, we'll tell the player they were right, and then use the `break` statement to stop the guessing loop.

```

// No changes to package and import statements; omitting

func main() {
    // No changes to previous code; omitting

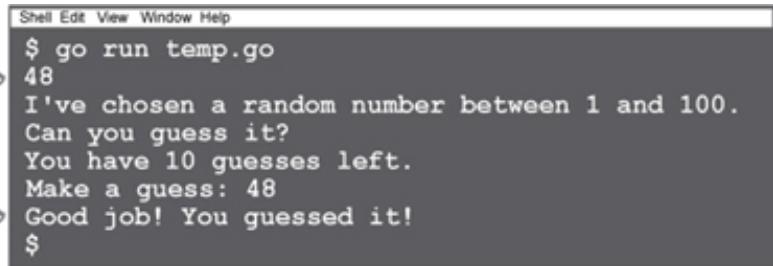
    for guesses := 0; guesses < 10; guesses++ {
        // No changes to previous code; omitting

        if guess < target {
            fmt.Println("Oops. Your guess was LOW.")
        } else if guess > target {
            fmt.Println("Oops. Your guess was HIGH.")
        } else {
            Congratulate the player. → fmt.Println("Good job! You guessed it!")
            break ←
        }
    }
}

```

Break out of the loop.

Now, when we make a correct guess, we'll see a congratulatory message, and the loop will exit without repeating the full 10 times.



The terminal window shows the following interaction:

```

Shell Edit View Window Help
$ go run temp.go
48
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 48
Good job! You guessed it!
$ 

```

Annotations with arrows point to specific parts of the output:

- An arrow points from the text "Here's the target; we'll cheat and make a correct guess immediately." to the number "48".
- An arrow points from the text "We get congratulated, and the loop exits!" to the message "Good job! You guessed it!".

That's another requirement complete!

REVEALING THE TARGET

We're so close! Just one more requirement left!

If the player makes 10 guesses without finding the target number, the loop will exit. In that event, we need to print a message saying they lost, and tell them what the target was.

- If the player's guess is equal to the target number, tell them, "Good job! You guessed it!" Then stop asking for new guesses.
- If the player ran out of turns without guessing correctly, say, "Sorry. You didn't guess my number. It was: [target]."

But we *also* exit the loop if the player guesses correctly. We don't want to say the player has lost when they've already won!

So, before our guessing loop, we'll declare a `success` variable that holds a `bool`. (We need to declare it *before* the loop so that it's still in scope after the loop ends.) We'll initialize `success` to a default value of `false`. Then, if the player guesses correctly, we'll set `success` to `true`, indicating we don't need to print the failure message.

```
// No changes to package and import statements; omitting

func main() {
    // No changes to previous code; omitting
    success := false
    for guesses := 0; guesses < 10; guesses++ {
        // No changes to previous code; omitting

        if guess < target {
            fmt.Println("Oops. Your guess was LOW.")
        } else if guess > target {
            fmt.Println("Oops. Your guess was HIGH.")
        } else {
            success = true
            fmt.Println("Good job! You guessed it!")
            break
        }
    }
    if !success {
        fmt.Println("Sorry, you didn't guess my number. It was:", target)
    }
}
```

Annotations on the right side of the code:

- `success := false`: *Declare "success" before the loop, so it's still in scope after the loop exits.*
- `success = true`: *If the player guesses correctly, indicate we don't need to print the failure message.*
- `if !success`: *If the player was NOT successful (if "success" is "false")...
...Print the failure message.*

After the loop, we add an `if` block that prints the failure message. But an `if` block only runs if its condition evaluates to `true`, and we only want to print the failure message if `success` is `false`. So we add the boolean negation operator, `!`. As we saw earlier, `!` turns `true` values `false` and `false` values `true`.

The result is that the failure message won't be printed if `success` is `false`, but *will* be printed if `success` is `true`.

THE FINISHING TOUCHES

Congratulations, that's the last requirement!

Let's take care of a couple final issues with our code, and then try out our game!



If the player ran out of turns without guessing correctly, say, “Sorry. You didn't guess my number. It was: [target].”

First, as we mentioned, it's typical to add a comment at the top of each Go program describing what it does. Let's add one now.

```
// guess challenges players to guess a random number.  
package main
```

Add a program description
comment, above the
package declaration.

Our program is also encouraging cheaters by printing the target number at the start of every game. Let's remove the `Println` call that does that.

```
target := rand.Intn(100) + 1  
fmt.Println(target) ← Don't reveal the target at the start of each game.  
fmt.Println("I've chosen a random number between 1 and 100.")
```

We should finally be ready to try running our complete code!

First, we'll run out of guesses on purpose to ensure the target number gets displayed...

The screenshot shows a terminal window with the following text:

```
Shell Edit View Window Help  
$ go run guess.go  
I've chosen a random number between 1 and 100.  
Can you guess it?  
You have 10 guesses left.  
Make a guess: 10  
Oops. Your guess was LOW.  
You have 9 guesses left.  
Make a guess: 20  
Oops. Your guess was LOW.  
...  
You have 1 guesses left.  
Make a guess: 62  
Oops. Your guess was LOW.  
Sorry, you didn't guess my number. It was: 63
```

Annotations on the left side of the terminal window:

- "Other incorrect guesses omitted..." → points to the ellipsis (...).
- "If we run out of guesses, the correct number is revealed." → points to the line "Sorry, you didn't guess my number. It was: 63".

Then we'll try guessing successfully. Our game is working great!

The screenshot shows a terminal window with the following text:

```
Shell Edit View Window Help Cheats  
$ go run guess.go  
I've chosen a random number between 1 and 100.  
Can you guess it?  
You have 10 guesses left.  
Make a guess: 50  
Oops. Your guess was HIGH.  
You have 9 guesses left.  
Make a guess: 40  
Oops. Your guess was LOW.  
You have 8 guesses left.  
Make a guess: 45  
Good job! You guessed it!
```

Annotation on the left side of the terminal window:

- "If we guess correctly, we see the victory message!" → points to the line "Good job! You guessed it!"

CONGRATULATIONS, YOUR GAME IS COMPLETE!



Using conditionals and loops, you've written a complete game in Go! Pour yourself a cold drink — you've earned it!

Here's our complete `guess.go` source code!

```

// guess challenges players to guess a random number.
package main

import (
    "bufio"
    "fmt"
    "log"
    "math/rand"
    "os"
    "strconv"
    "strings"
    "time"
)
}

func main() {
    seconds := time.Now().Unix() ← Get the current date
    rand.Seed(seconds) ← and time, as an integer.
    target := rand.Intn(100) + 1 ← Seed the random number generator.
    fmt.Println("I've chosen a random number between 1 and 100.")
    fmt.Println("Can you guess it?") ← Create a bufio.Reader, which lets us
                                    read keyboard input.

    reader := bufio.NewReader(os.Stdin) ← Set up to print a failure message by default.
    success := false ←
    for guesses := 0; guesses < 10; guesses++ {
        fmt.Println("You have", 10-guesses, "guesses left.") ←
        fmt.Print("Make a guess: ") ← Ask for a number.
        input, error := reader.ReadString('\n') ←
            If there's an error, print the message and exit. { if error != nil {
                log.Fatal(error) } } Read what the user types, up until they press Enter.

        input = strings.TrimSpace(input) ← Remove the newline.
        guess, error := strconv.Atoi(input) ←
            If there's an error, print the message and exit. { if error != nil {
                log.Fatal(error) } } Convert the input string to an integer.

        if guess < target { ← If the guess was too low, say so.
            fmt.Println("Oops. Your guess was LOW.")
        } else if guess > target { ← If the guess was too high, say so.
            fmt.Println("Oops. Your guess was HIGH.")
        } else { ← Otherwise, the guess must be correct...
            success = true ← Prevent the failure message from displaying.
            fmt.Println("Good job! You guessed it!")
            break ← Exit the loop.
        }
    }

    if !success { ← If "success" is false, tell player what the target was.
        fmt.Println("Sorry, you didn't guess my number. It was:", target)
    }
}

```



YOUR GO TOOLBOX

That's it for Chapter 2! You've added conditionals and loops to your toolbox.

Functions

Types

Conditionals

- Conditionals are statements that cause a block of code to be executed only if a condition is met
- An expression is evaluated, and if its result is true, the code in the conditional block body is executed.
- Go supports multiple branches in the condition. These statements take the form if/else if/else.

Loops

Loops cause a block of code to execute repeatedly.

One common kind of loop starts with the keyword "for", followed by an init statement that initializes a variable, a condition expression that determines when to break out of the loop, and a post statement that runs after each iteration of the loop.



BULLET POINTS

- A method is a kind of function that's associated with values of a given type.
- Go treats everything from a // marker to the end of the line as a comment—and ignores it.
- Multi-line comments start with /* and end with */. Everything in between, including newlines, is ignored.
- It's conventional to include a comment at the top of every program, explaining what it does.
- Unlike most programming languages, Go allows *multiple* return values from a function or method call.

- One common use of multiple return values is to return the function's main result, and then a second value indicating whether there was an error.
- To discard a value without using it, use the `_blank` identifier. The blank identifier can be used in place of any variable in any assignment statement.
- Functions, conditionals, and loops all have blocks of code that appear within `{}` braces.
- The code doesn't appear within `{}` braces, but files and packages also comprise blocks.
- The scope of a variable is limited to the block it is defined within, and all blocks nested within that block.
- In addition to a name, a package may have an import path that is required when importing it.
- The `continue` keyword skips to the next iteration of a loop.
- The `break` keyword exits out of a loop entirely.



Because they're in conditional blocks, only some of the `println` calls in the below code will be executed. Write down what the output would be.

```

if true { ← "if" blocks run if the condition results in "true" (or if it IS "true").
    fmt.Println("true")
}
if false { ← If the condition is false, the block doesn't run.
    fmt.Println("false")
}
if !false { ← The boolean negation operator turns "false" into "true".
    fmt.Println("!false")
}
if true { ← The "if" branch runs...
    fmt.Println("if true")
} else { ← ...so the "else" branch doesn't.
    fmt.Println("else")
}
if false { ← The "if" branch doesn't run...
    fmt.Println("if false")
} else if true { ← ...So the "else if" branch MIGHT run.
    fmt.Println("else if true")
}
if 12 == 12 { ← 12 == 12 is "true".
    fmt.Println("12 == 12")
}
if 12 != 12 { ← The values ARE equal, so this is FALSE.
    fmt.Println("12 != 12")
}
if 12 > 12 { ← 12 is NOT greater than itself...
    fmt.Println("12 > 12")
}
if 12 >= 12 { ← ...But 12 IS equal to itself.
    fmt.Println("12 >= 12")
}
if 12 == 12 && 5.9 == 5.9 { ← The "&&" evaluates to "true" if BOTH expressions are "true".
    fmt.Println("12 == 12 && 5.9 == 5.9")
}
if 12 == 12 && 5.9 == 6.4 { ← One expression is "false".
    fmt.Println("12 == 12 && 5.9 == 6.4")
}
if 12 == 12 || 5.9 == 6.4 { ← The "||" evaluates to "true" if EITHER expression is "true".
    fmt.Println("12 == 12 || 5.9 == 6.4")
}

```

Output:

true

false

if true

else if true

12 == 12

12 >= 12

12 == 12 && 5.9 == 5.9

12 == 12 || 5.9 == 6.4

CODE MAGNETS SOLUTION

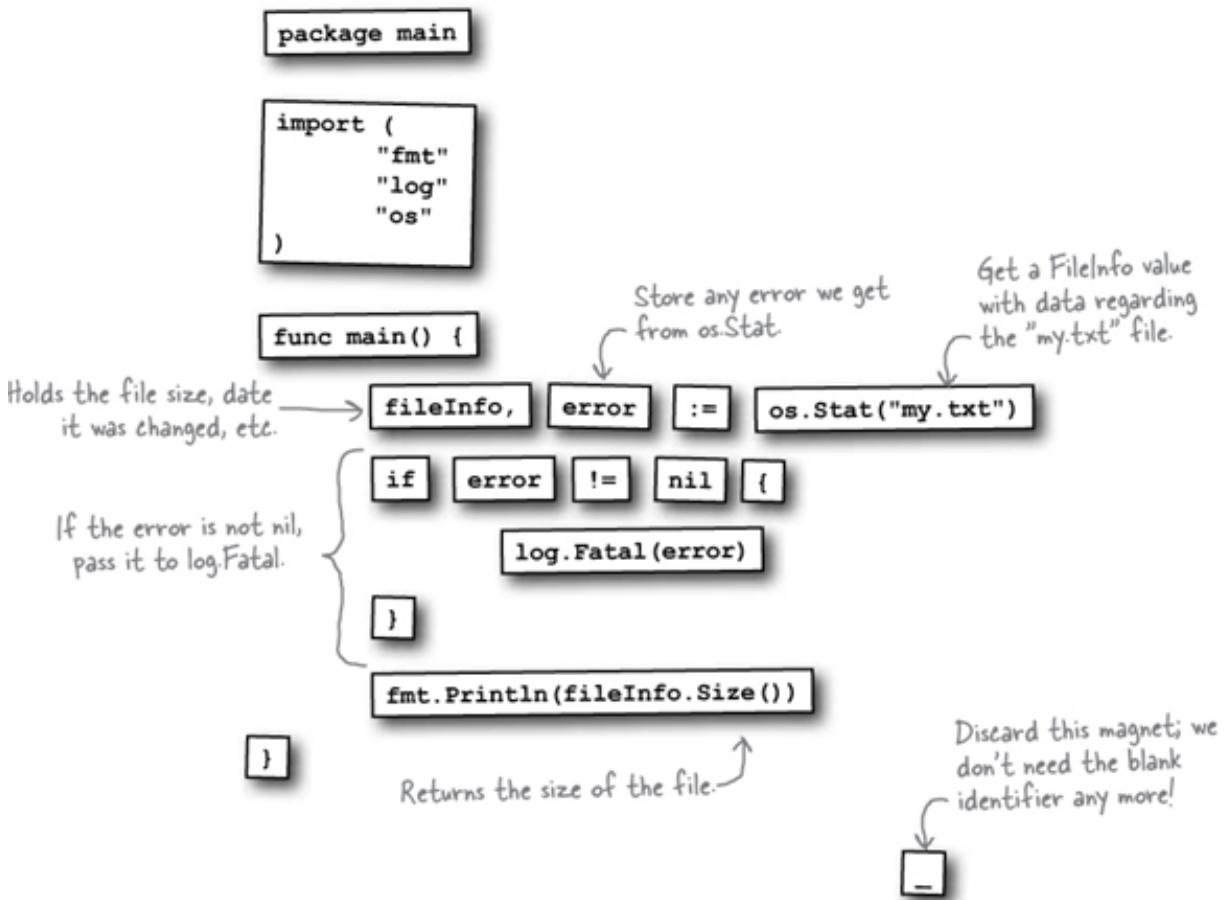


A Go program that prints the size of a file is on the fridge. It calls the `os.Stat` function, which returns an `os.FileInfo` value, and possibly an error. Then it calls the `Size` method

on the `FileInfo` value to get the file size.

The original program used the `_` blank identifier to ignore the error value from `os.Stat`. If an error occurred (which could happen if the file doesn't exist), this would cause the program to fail.

Your job was to reconstruct the extra code snippets to make a program that works just like the original one, but also checks for an error from `os.Stat`. If the error from `os.Stat` is not `nil`, the error should be reported, and the program should exit.



Some of the lines of code below will result in a compile error, because they refer to a variable that is out of scope. Cross out the lines that have errors.

```

package main

import (
    "fmt"
)

var a = "a"

func main() {
    a = "a"
    b := "b"
    if true {
        c := "c"
        if true {
            d := "d"
            fmt.Println(a)
            fmt.Println(b)
            fmt.Println(c)
            fmt.Println(d)
        }
        fmt.Println(a)
        fmt.Println(b)
        fmt.Println(c)
        fmt.Println(d)
    }
    fmt.Println(a)
    fmt.Println(b)
fmt.Println(c)
fmt.Println(d)
}

```



Look carefully at the init statement, condition expression, and post statement for each of these loops. Then write what you think the output will be for each one.

Start at 1. → Stop after 3. ↓ Count up.
`for x := 1; x <= 3; x++ {`
 `fmt.Println(x)`
`}`

123

Start at 3. → Stop after 1. ↓ Count down.
`for x := 3; x >= 1; x-- {`
 `fmt.Println(x)`
`}`

321

Start at 2. → Stop after 3. ↓ Count up.
`for x := 2; x <= 3; x++ {`
 `fmt.Println(x)`
`}`

23

Start at 1. → Stop AT 3. ↓ Count up.
`for x := 1; x < 3; x++ {`
 `fmt.Println(x)`
`}`

12

Start at 1. → Stop after 3. ↓ Count up 2 at a time.
`for x := 1; x <= 3; x+= 2 {`
 `fmt.Println(x)`
`}`

13

Start at 1. → Stop when x < 3 (i.e. immediately). ↓ Never runs!
`for x := 1; x >= 3; x++ {`
 `fmt.Println(x)`
`}`

No output; loop never runs!

3 functions

Call Me

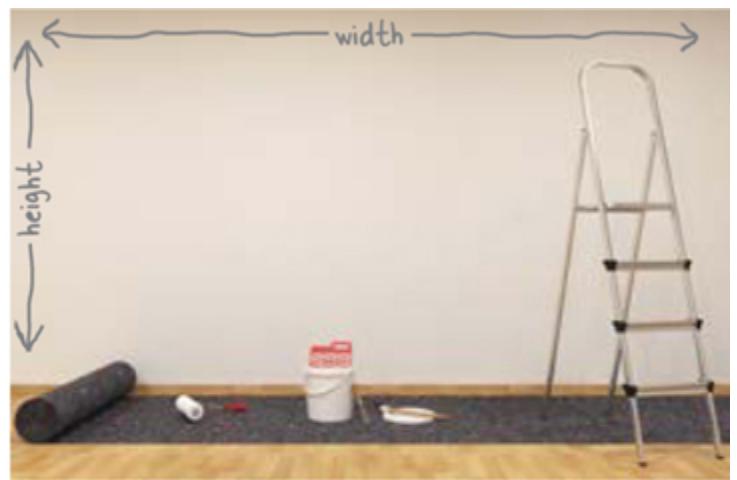


You've been missing out. You've been calling functions like a pro. But the only functions you could call were the ones Go defined for you. Now, it's your turn. We're going to show you how to create your own functions. We'll learn how to declare functions with and without parameters. We'll declare functions that return a single value, and we'll learn how to return multiple values so that we can indicate when there's been an error. And we'll learn about **pointers**, which allow us to make more memory-efficient function calls.

SOME REPETITIVE CODE

Suppose we need to calculate the amount of paint needed to cover several walls. The

manufacturer says each liter of paint covers 10 square meters. So, we'll need to multiply each wall's width (in meters) by its height to get its area, and then divide that by 10 to get the number of liters of paint needed.



```
// package and imports omitted
func main() {
    var width, height, area float64
    Calculate the amount for a first wall. { width = 4.2
    height = 3.0
    area = width * height ← Determine the area of the wall.
    fmt.Println(area/10.0, "liters needed") ← Calculate how much paint is needed for that area.
    Do the same for a second wall. { width = 5.2
    height = 3.5
    area = width * height ← Determine the area of the wall.
    fmt.Println(area/10.0, "liters needed") ← Calculate how much paint is needed for that area.
}
```

1.2600000000000002 liters needed
1.8199999999999998 liters needed

This works, but it has a couple problems:

- The calculations seem to be off by a tiny fraction, and are printing oddly precise floating-point values. We really only need a couple decimal places of precision.
- There's a fair amount of repeated code, even now. This will get worse as we add more walls.

Both items will take a little explanation to address, so let's just look at the first issue for now...

The calculations are slightly off because ordinary floating-point arithmetic on computers is ever-so-slightly inaccurate. (Usually by a few quadrillionths.) The reasons are a little too complicated to get into here, but this problem isn't exclusive to Go.

But as long as we round the numbers to a reasonable degree of precision before displaying them, we should be fine. Let's take a brief detour to look at a function that will help us do that.



Go on a Detour

FORMATTING OUTPUT WITH PRINTF AND SPRINTF



Floating-point numbers in Go are kept with a high degree of precision. This can be cumbersome when you want to display them:

```
fmt.Println("About one-third:", 1.0/3.0)
```

```
About one-third: 0.3333333333333333
```

That's a lot of decimal places!

To deal with these sorts of formatting issues, the `fmt` package provides the `Printf` function. `Printf` stands for "**p**rint, with **f**ormatting". It takes a string, and inserts one or more values into it, formatted in specific ways. Then it prints the resulting string.

```
fmt.Printf("About one-third: %0.2f\n", 1.0/3.0)
```

```
About one-third: 0.33
```

Much more readable!

The `Sprintf` function (also part of the `fmt` package) works just like `Printf`, except that it returns a formatted string instead of printing it.

```
resultString := fmt.Sprintf("About one-third: %0.2f\n", 1.0/3.0)
fmt.Println(resultString)
```

```
About one-third: 0.33
```

It looks like `Printf` and `Sprintf` *can* help us limit our displayed values to the correct number of places. The question is, *how*? To be able to use this method effectively, we'll need to learn about two features of `Printf`:

1. Formatting verbs (the `%0.2f` in the strings above is a verb)
2. Value widths (that's the `0.2` in the middle of the verb)



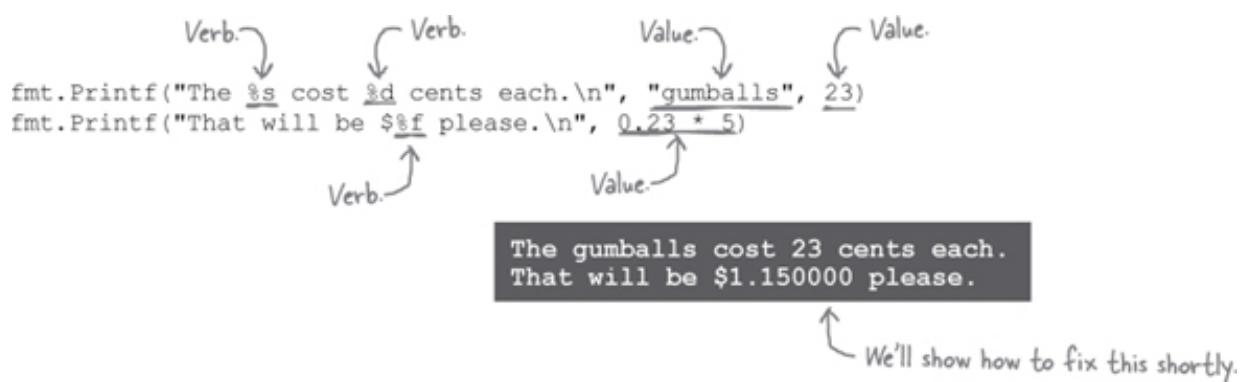
We'll explain exactly what those arguments to `Printf` mean on the next few pages.

We know, those method calls above look a little confusing. We'll show you a ton of examples that should clear that confusion up.

FORMATTING VERBS



The first argument to `Printf` is a string that will be used to format the output. Most of it is formatted exactly as it appears in the string. Any percent signs (%), however, will be treated as the start of a **formatting verb**, a section of the string that will be substituted with a value in a particular format. The remaining arguments are used as values with those verbs.



The letter following the percent sign indicates which verb to use. The most common verbs are:

Verb	Output
%f	Floating-point number

%d	Decimal integer
%s	String
%t	Boolean (<code>true</code> or <code>false</code>)
%v	Any value (chooses an appropriate format based on the supplied value's type)
%#v	Any value, formatted as it would appear in Go program code
%T	Type of the supplied value (<code>int</code> , <code>string</code> , etc.)
%%	A literal percent sign

```
fmt.Printf("A float: %f\n", 3.1415)
fmt.Printf("An integer: %d\n", 15)
fmt.Printf("A string: %s\n", "hello")
fmt.Printf("A boolean: %t\n", false)
fmt.Printf("Values: %v %v %v\n", 1.2, "\t", true)
fmt.Printf("Values: %#v %#v %#v\n", 1.2, "\t", true)
fmt.Printf("Types: %T %T %T\n", 1.2, "\t", true)
fmt.Printf("Percent sign: %%\n")
```

```
A float: 3.141500
An integer: 15
A string: hello
A boolean: false
Values: 1.2          true
Values: 1.2 "\t"    true
Types: float64 string bool
Percent sign: %
```

Notice, by the way, that we are making sure to add a newline at the end of each formatting string using the `\n` escape sequence. This is because unlike `Println`, `Printf` does not automatically add a newline for us.



We want to point out the `%#v` formatting verb in particular. Because it prints values the way they would appear in Go code, rather than how they normally appear, `%#v` can show you some values that would otherwise be hidden in your output. In this code, for example, `%#v` reveals an empty string, a tab character, and a newline, all of which were invisible when printed with `%v`. We'll use `%#v` more later in the book!

```
fmt.Printf("%v %v %v", "", "\t", "\n")
fmt.Printf("%#v %#v %#v", "", "\t", "\n")
```

%v prints all the values... →

But only with %#v can you actually see them!

→ ... "\t" "\n"

FORMATTING VALUE WIDTHS

So the %f formatting verb is for floating-point numbers. We can use %f in our program to format the amount of paint needed.

Insert a floating-point value. →

One of the values previously calculated by our program. →

fmt.Println("%f liters needed\n", 1.8199999999999998) → 1.820000 liters needed

→ Rounded, but still too many digits!

It looks like our value is being rounded to a reasonable number. But it's still showing six places after the decimal point, which is really too much for our current purpose.

For situations like this, formatting verbs let you specify the *width* of the formatted value.

Let's say we want to format some data in a plain-text table. We need to ensure the formatted value fills a minimum number of spaces, so that the columns align properly.

You can specify the minimum width after the percent sign in a format sequence. If the argument for that format sequence is shorter than the minimum width, it will be padded with spaces until the minimum width is reached.

The first field will have a minimum width of 12 characters. →

No minimum width for this second field. →

Print column headings. →

fmt.Println("-----") ← Print a heading divider.

Minimum width of 12 again. →

Minimum width of 2. →

fmt.Printf("%12s | %2i", "Stamps", 50)
fmt.Printf("%12s | %2i", "Paper Clips", 5)
fmt.Printf("%12s | %2i", "Tape", 99)

Padding! →

Product	Cost in Cents
Stamps	50
Paper Clips	5
Tape	99

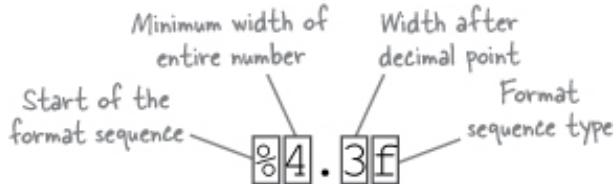
No padding; the value already fills the minimum width. →

Padding! →

FORMATTING FRACTIONAL NUMBER WIDTHS



And now we come to the part that's important for today's task: you can use value widths to specify the precision (the number of displayed digits) for floating-point numbers. Here's the format:



The minimum width of the entire number includes decimal places. If it's included, shorter numbers will be padded with spaces at the start until this width is reached. If it's omitted, no spaces will ever be added.

The width after the decimal point is the maximum number of digits to show. If a more precise number is given, it will be rounded (up or down) to fit in the given number of decimal places.

Here's a quick demonstration of various width values in action:

That last format, "%.*2f*", will let us take floating-point numbers of any precision and round them to two decimal places. (It also won't do any unnecessary padding.) Let's try it with the overly-precise values from our program to calculate paint volumes.

```
fmt.Printf("%.2f\n", 1.2600000000000002)
fmt.Printf("%.2f\n", 1.8199999999999998)
```

Rounded to two places!

That's much more readable. It looks like the `Printf` function can format our numbers for us. Let's get back to our paint calculator program, and apply what we've learned there.



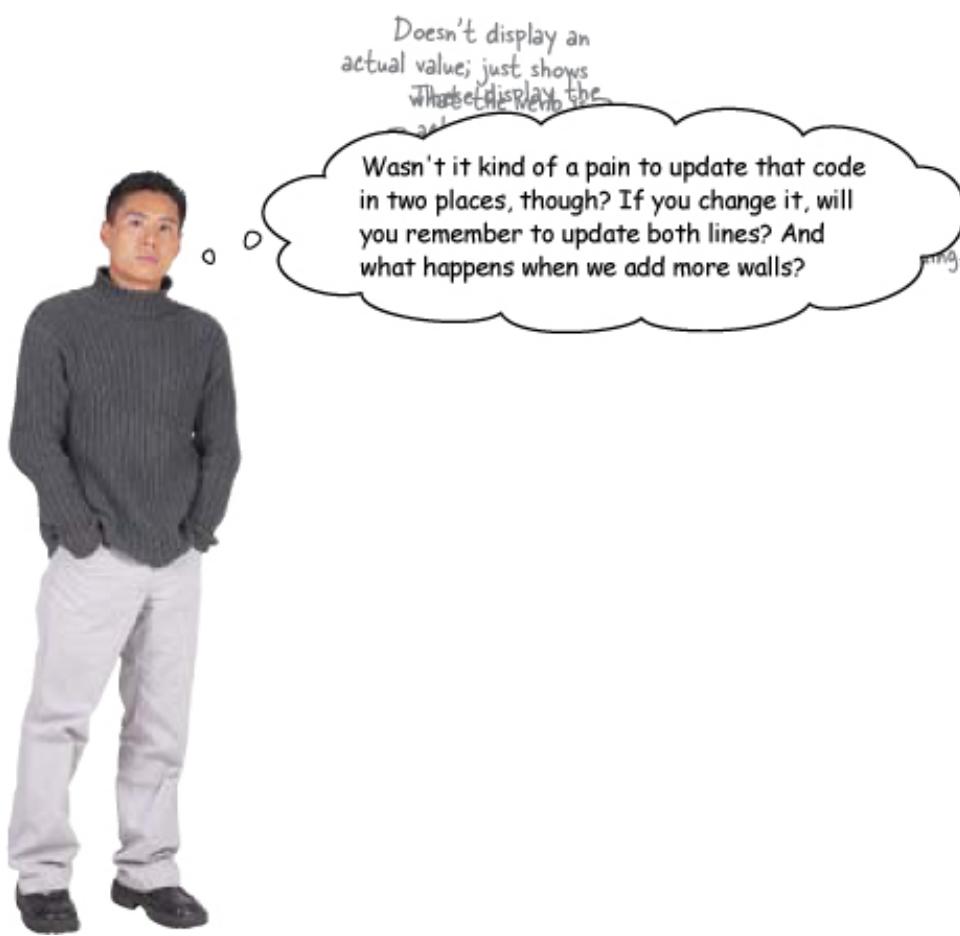
USING PRINTF IN OUR PAINT CALCULATOR

Now we have a `Printf` verb, "%.*2f*", that will let us round a floating-point number to two decimal places. Let's update our paint quantity calculation program to use it.

```
// package and imports omitted
func main() {
    var width, height, area float64
    width = 4.2
    height = 3.0
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0) ← Format the value and
    width = 5.2
    height = 3.5
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0) ← insert it into the string.
}

1.26 liters needed
1.82 liters needed
↑
Rounded to 2 places.
```

At last, we have reasonable-looking output! The tiny imprecisions introduced by floating-point arithmetic have been rounded away.



Good point. Go lets us declare our own functions, so perhaps we should move this code into a function.

As we mentioned way back at the start of [Chapter 1](#), a function is a group of one or more lines of code that you can call from other places in your program. And our program has two groups of lines that look very similar:

```

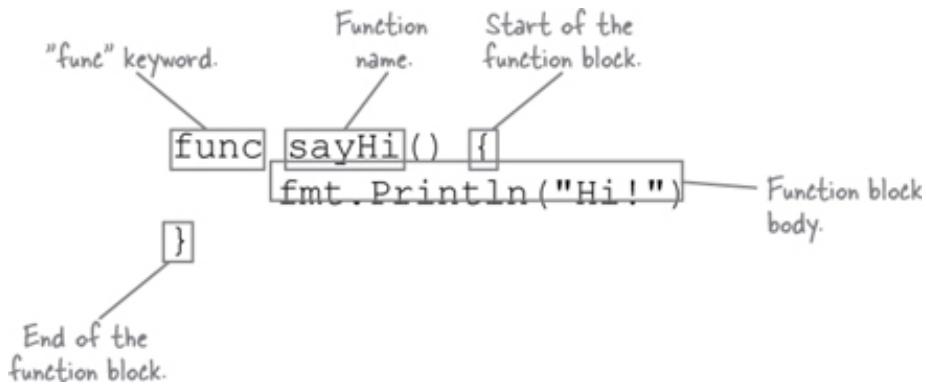
        var width, height, area float64
Calculate the paint needed for the first wall. {
    width = 4.2
    height = 3.0
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}
Calculate the paint needed for the second wall. {
    width = 5.2
    height = 3.5
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}

```

Let's see if we can convert these two sections of code into a single function.

DECLARING FUNCTIONS

A simple function declaration might look like this:



A declaration begins with the `func` keyword, followed by the name you want the function to have, a pair of parentheses `()`, and then a block containing the function's code.

Once you've declared a function, you can call it elsewhere in your package simply by typing its name, followed by a pair of parentheses. When you do, the code in the function's block will be run.

```

package main

import "fmt"

Declare a "sayHi" function {
    func sayHi() {
        fmt.Println("Hi!")
    }
}

func main() {
    sayHi()
}
Call "sayHi".

```

Notice that when we call `sayHi`, we're not typing the package name and a dot before the function name. When you call a function that's defined in the current package, you should not specify the package name. (Typing `main.sayHi()` would result in a compile error.)

The rules for function names are the same as the rules for variable names:

- A name must begin with a letter, followed by any number of additional letters and numbers. (You'll get a compile error if you break this rule.)

OK {
double
addPart
Publish

Use camelCase if there
are multiple words.
Capitalize the name if it will
be used by other packages.

- Functions whose name begins with a capital letter are *exported*, and can be used outside the current package. If you only need to use a function inside the current package, you should start its name with a lower-case letter.

Not OK {
2times
addpart
posts.publish

Illegal; can't begin with a number.
Breaks convention; should
use camelCase.
Illegal; can't access function
in another package unless
its name is capitalized.

- Names with multiple words should use

DECLARING FUNCTION PARAMETERS

If you want calls to your method to include arguments, you'll need to declare one or more parameters. A **parameter** is a variable local to a function, whose value is set when the function is called.

```
Parameter 1    Parameter 1    Parameter 2    Parameter 2
name.          type.        name.          type.
func repeatLine(line string, times int) {
    for i := 0; i < times; i++ {
        fmt.Println(line)
    }
}
```

You can declare one or more parameters between the parentheses in the function declaration, separated by commas. As with any variable, you'll need to provide a name followed by a type (`float64`, `bool`, etc.) for each parameter you declare.

If a function has parameters defined, then you'll need to pass a matching set of

arguments when calling it. When the function is run, each parameter will be set to a copy of the value in the corresponding argument. Those parameter values are then used within the code in the function block.

```
package main
import "fmt"

func main() {
    repeatLine("hello", 3)
}

func repeatLine(line string, times int) {
    for i := 0; i < times; i++ {
        fmt.Println(line)
    }
}
```

Passing arguments to the function...

...Sets the parameters...

...Which are then used when the function block runs.

hello
hello
hello

USING FUNCTIONS IN OUR PAINT CALCULATOR

Now that we know how to declare our own functions, let's see if we can get rid of the repetition in our paint calculator.

```
// package and imports omitted
func main() {
    var width, height, area float64
    width = 4.2
    height = 3.0
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)

    Repeated code! { width = 5.2
                    height = 3.5
                    area = width * height
                    fmt.Printf("%.2f liters needed\n", area/10.0)
    }
}
```

Repeated code!

Repeated code!

1.26 liters needed
1.82 liters needed

We'll move the code to calculate the amount of paint to a function, named `paintNeeded`. We'll get rid of the separate `width` and `height` variables, and instead take those as function parameters. Then, in our `main` function, we'll just call `paintNeeded` for each wall we need to paint.

```

package main
import "fmt"

Declare a
function named
"paintNeeded".
→ func paintNeeded(width float64, height float64) {
    width as a
    parameter.
    Take the wall
    height as another
    parameter.
    area := width * height ← Multiply width and height, as before.
    fmt.Printf("%.2f liters needed\n", area/10.0) ←
}
    Pass in the width.    Pass in the height.
    func main() {          →
        Call our new function. → paintNeeded(4.2, 3.0)
                                paintNeeded(5.2, 3.5)
                                paintNeeded(5.0, 3.3)
    }
    Painting more walls?
    Just add more calls!

```

Print the amount of
paint, as before.

1.26 liters needed
1.82 liters needed
1.65 liters needed

No more repeated code, and if we want to calculate the paint needed for additional walls, we just add more calls to `paintNeeded`. This is much cleaner!

FUNCTIONS AND VARIABLE SCOPE

Our `paintNeeded` function declares an `area` variable within its function block:

```

func paintNeeded(width float64, height float64) {
    area := width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}
Access the variable.

```

As with conditional and loop blocks, variables declared within a function block are only in scope within that function block. So if we were to try to access the `area` variable outside of the `paintNeeded` function, we'd get a compile error:

```

func paintNeeded(width float64, height float64) {
    area := width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}

func main() {
    paintNeeded(4.2, 3.0)
    fmt.Println(area)   Error → undefined: area
}
Out of scope!

```

But, also as with conditional and loop blocks, variables declared *outside* a function block will be in scope within that block. That means we can declare a variable at package level, and access it within any function in that package.

```

package main

import "fmt"                                If we defined a variable
                                                at package level...
var metersPerLiter float64←

func paintNeeded(width, height float64) float64 {
    area := width * height
    return area / metersPerLiter ← Still in scope here.
}

func main() {                               Still in scope here.
    metersPerLiter = 10.0←
    fmt.Printf("%.2f", paintNeeded(4.2, 3.0))  1.26
}

```



Exercise

Below is a program that declares several functions, then calls those functions within `main`. Write down what the program output would be.

(We've done the first line for you.)

```

package main

import "fmt"

func functionA(a int, b int) {
    fmt.Println(a + b)
}
func functionB(a int, b int) {
    fmt.Println(a * b)
}
func functionC(a bool) {
    fmt.Println(!a)
}
func functionD(a string, b int) {
    for i := 0; i < b; i++ {
        fmt.Print(a)
    }
    fmt.Println()
}

func main() {
    functionA(2, 3)
    functionB(2, 3)
    functionC(true)
    functionD("$", 4)
    functionA(5, 6)
    functionB(5, 6)
    functionC(false)
    functionD("ha", 3)
}

```

Output:

5

FUNCTION RETURN VALUES

Suppose we wanted to total the amount of paint needed for all the walls we're going to paint. We can't do that with our current `paintNeeded` function; it just prints the amount, and then discards it!

```
func paintNeeded(width float64, height float64) {  
    area := width * height  
    fmt.Printf("%.2f liters needed\n", area/10.0)  
}
```

Prints the amount of paint, but then we can't do anything further with it!

So instead, let's revise the `paintNeeded` function to return a value. Then, whoever calls it can print the amount, do additional calculations with it, or whatever else they need.

Functions always return values of a specific type (and only that type). To declare that a function returns a value, add the type of that return value following the parameters in the function declaration. Then use the `return` keyword in the function block, followed by the value you want to return.

```
func double(number float64) float64 {  
    Return keyword.--- return number * 2  
    }  
Value to return.
```

Return value type.

Callers of the function can then assign the return value to a variable, pass it directly to another function, or whatever else they need to do with it.

```
package main  
  
import "fmt"  
  
func double(number float64) float64 {  
    return number * 2  
}  
  
func main() {  
    dozen := double(6.0)  
    fmt.Println(dozen)  
    fmt.Println(double(4.2))  
}  
Assign return value to a variable.  
Pass return value to another function.  
12  
8.4
```

When a `return` statement runs, the function exits immediately, without running any code that follows it. You can use this together with an `if` statement to exit the function in conditions where there's no point in running the remaining code (due to an error or some other condition).

```

func status(grade float64) string {
    if grade < 60.0 { ← If grade is failing,
        return "failing" ← return immediately.
    }
    return "passing" ← Only runs if grade is >= 60.
}

func main() {
    fmt.Println(status(60.1))
    fmt.Println(status(59))
```

passing
failing

That means that it's possible to have code that never runs under any circumstances, if you include a return statement that isn't part of an `if` block. This almost certainly indicates a bug in the code, so Go helps you detect this situation by requiring that any function that declares a return type must end with a `return` statement. Ending with any other statement will cause a compile error.

```

func double(number float64) float64 {
    return number * 2 ← Function would always exit here...
    fmt.Println(number * 2) ←
}                                This line would never run!
```

Error → missing return at end of function

You'll also get a compile error if the type of your return value doesn't match the declared return type.

```

func double(number float64) float64 {
    return int(number * 2) ← ...Returns an integer!
}                                Expects a floating-point number...
```

Error → cannot use int(number * 2) (type int)
as type float64 in return argument

USING A RETURN VALUE IN OUR PAINT CALCULATOR

Now that we know how to use function return values, let's see if we can update our paint program to print the total amount of paint needed in addition to the amount needed for each wall.

We'll update the `paintNeeded` function to return the amount needed. We'll use that return value in the `main` function, both to print the amount for the current wall, and to add to a `total` variable that tracks the total amount of paint needed.

```

package main

import "fmt"

func paintNeeded(width float64, height float64) float64 {
    area := width * height
    return area / 10.0 ← Return the area instead
} ← of printing it.

func main() {
    var amount, total float64 ← Declare variables to hold the amount for the
    amount = paintNeeded(4.2, 3.0) ← current wall, as well as the total for all walls.
    fmt.Printf("%0.2f liters needed\n", amount) ← Call paintNeeded, and store the return value.
    total += amount ← Print the amount for this wall.
    Repeat the above steps for a second wall. amount = paintNeeded(5.2, 3.5)
    total += amount ← Add the amount for this wall to the total.
    fmt.Printf("%0.2f liters needed\n", amount) ←
    fmt.Printf("Total: %0.2f liters\n", total) ← Print the total for all walls.
}

```

1.26 liters needed
 1.82 liters needed
 Total: 3.08 liters

It works! Returning the value allowed our `main` function to decide what to do with the calculated amount, rather than relying on the `paintNeeded` function to print it.



BREAKING STUFF IS EDUCATIONAL!

Here's our updated version of the `paintNeeded` function that returns a value. Try making one of the changes below, and try to compile it. Then undo your change, and try the next one. See what happens!

```

area := width * height
return area / 10.0
}

```

If you do this...

...it will break because...

Remove the `return` statement:

```

func paintNeeded(width,
height float64) float64 {
    area := width *
height
}

```

If your function declares a return type, Go requires that it include a `return` statement.

```
    return area / 10.0  
}
```

Add a line *after* the `return` statement:

```
func paintNeeded(width,  
height float64) float64 {  
    area := width *  
    height  
    return area / 10.0  
    fmt.Println(area /  
10.0)  
}
```

If your function declares a return type, Go requires that its last statement be a `return` statement.

Remove the return type declaration:

```
func paintNeeded(width,  
height float64) float64 {  
    area := width *  
    height  
    return area / 10.0  
}
```

Go doesn't allow you to return a value you haven't declared.

Change the type of value being returned:

```
func paintNeeded(width,  
height float64) float64 {  
    area := width *  
    height  
    return int(area /  
10.0)  
}
```

Go requires that the type of the returned value match the declared type.

THE PAINTNEEDED FUNCTION NEEDS ERROR CHECKING



Your `paintNeeded` function works great, most of the time. But one of our users recently passed it a negative number by accident, and got a **negative** amount of paint back!

```
func main() {
    amount := paintNeeded(4.2, -3.0)
    fmt.Printf("%0.2f liters needed\n", amount)
}

func paintNeeded(width float64, height float64) float64 {
    area := width * height ← 4.2 * -3.0 is -12.6!
    return area / 10.0 ←
        -12.6 / 10.0 is -1.26!
```

-1.26 liters needed

It looks like the `paintNeeded` function had no idea the argument passed to it was invalid. It went right ahead and used that invalid argument in its calculations, and returned an invalid result. This is a problem — even if you knew a store where you could purchase a negative number of liters of paint, would you really want to apply that to your house? We need a way of detecting invalid arguments and reporting an error.

In [Chapter 2](#), we saw a couple different functions that, in addition to their main return value, also return a second value indicating whether there was an error. The `strconv.Atoi` function, for example, attempted to convert a string to an integer. If the conversion was successful, it returned an error value of `nil`, meaning our program could proceed. But if the error value *wasn't* `nil`, it meant the string couldn't be converted to a number. In that event, we chose to print the error value and exit the program.

```
If there was an error, print the message and exit. {   guess, error := strconv.Atoi(input) ←
    if error != nil {
        log.Fatal(error)
    }
}
```

Convert the input string to an integer.

If we want to do the same for the `paintNeeded` function, we're going to need two things:

- The ability to create a value representing an error
- The ability to return an additional value from `paintNeeded`

Let's get started figuring this out!

ERROR VALUES

Before we can return an error value from our `paintNeeded` function, we need an error value to return. An error value is basically any value with a method named `Error` that returns a string. The simplest way to create one is to pass a string to the `errors` package's `New` function, which will return a new error value. If you call the `Error` method on that error value, you'll get the string you passed to `errors.New`.

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    err := errors.New("height can't be negative") ← Create a new error value.
    fmt.Println(err.Error())
}
Returns the error message. ↑ height can't be negative
```

But if you're passing the error value to a function in the `fmt` or `log` packages, you probably don't need to call its `Error` method. Functions in `fmt` and `log` have been written to check whether the values passed to them have `Error` methods, and print the return value of `Error` if they do.

```
err := errors.New("height can't be negative")
fmt.Println(err) ← Prints the error message.
log.Fatal(err) ← Prints the error message,
then exits the program.

height can't be negative
2018/03/12 19:49:27 height can't be negative
```

If you need to format numbers or other values for use in your error message, you can use the `fmt.Errorf` function. It inserts values into a format string just like `fmt.Printf` or `fmt.Sprintf`, but instead of printing or returning a `string`, it returns an error value.

```
Returns an error value. → err := fmt.Errorf("a height of %0.2f is invalid", -2.33333)
Prints the error message. → fmt.Println(err.Error())
Also prints the error message. → fmt.Println(err)

Insert a floating-point
number, rounded to 2
decimal places. ↓
a height of -2.33 is invalid
a height of -2.33 is invalid
```

DECLARING MULTIPLE RETURN VALUES

Now we need a way to specify that our `paintNeeded` function will return an error value

along with the amount of paint needed.

To declare multiple return values for a function, place the return value types in a *second* set of parentheses in the function declaration (after the parentheses for the function parameters), separated with commas. (The parentheses around the return values are optional when there's only one return value, but are required if there's more than one return value.)

From then on, when calling that function, you'll need to account for the additional return values, usually by assigning them to additional variables.

```
package main
import "fmt"
func manyReturns() (int, bool, string) {
    return 1, true, "hello"
}

func main() {
    myInt, myBool, myString := manyReturns()
    fmt.Println(myInt, myBool, myString)
}
1 true hello
```

This function returns an integer, a boolean, and a string.

Store each return value in a variable.

If it makes the purpose of the return values clearer, you can supply names for each return value, similar to parameter names. The main purpose of named return values is as documentation for programmers reading the code.

```
package main
import (
    "fmt"
    "math"
)
func floatParts(number float64) (integerPart int, fractionalPart float64) {
    wholeNumber := math.Floor(number)
    return int(wholeNumber), number - wholeNumber
}

func main() {
    cans, remainder := floatParts(1.26)
    fmt.Println(cans, remainder)
}
1 0.26
```

Name for the first return value. Name for the second return value.

*Named return values can also be used for something called a "bare return", but this feature is controversial, and so we're not covering it in this book.

USING MULTIPLE RETURN VALUES WITH OUR PAINTNEEDED FUNCTION

As we saw on the previous page, it's possible to return multiple values of any type. But the most common use for multiple return values is to return a primary return value,

followed by an additional value indicating whether the function encountered an error. The additional value is usually set to `nil` if there were no problems, or an error value if an error occurred.

We'll follow that convention with our `paintNeeded` function as well. We'll declare that it returns two values, a `float64` and an `error`. (Error values have a type of `error`.) The first thing we'll do in the function block is to check whether the parameters are valid. If either the `width` or `height` parameters are less than `0`, we'll return a paint amount of `0` (which is meaningless, but we do have to return something), and an error value that we generate by calling `fmt.Errorf`. Checking for errors at the start of the function allows us to easily skip the rest of the function's code by calling `return` if there's a problem.

If there were no problems with the parameters, we proceed to calculate and return the paint amount just like before. The only other difference in the function code is that we return a second value of `nil` along with the paint amount, to indicate there were no errors.

```
package main
import "fmt"

func paintNeeded(width float64, height float64) (float64, error) {
    if width < 0 { ← If width is invalid, return 0 and an error.
        return 0, fmt.Errorf("a width of %0.2f is invalid", width)
    }
    if height < 0 { ← If height is invalid, return 0 and an error.
        return 0, fmt.Errorf("a width of %0.2f is invalid", height)
    }
    area := width * height
    return area / 10.0, nil ← Return the amount of paint, along with "nil", indicating there was no error.
}

func main() {
    amount, err := paintNeeded(4.2, -3.0)
    fmt.Println(err) ← Prints the error (or "nil" if there was none).
    fmt.Printf("%0.2f liters needed\n", amount)
}
```

Here's the return value with the amount of paint, just like before.

Here's a second return value that will indicate whether there were any errors.

Add a second variable to hold the second return value.

Prints the error (or "nil" if there was none).

a width of -3.00 is invalid
0.00 liters needed

In the `main` function, we add a second variable to record the error value from `paintNeeded`. We print the error (if any), and then print the paint amount.

If we pass an invalid argument to `paintNeeded`, we'll get an error return value, and print that error. But we also get `0` as the amount of paint. (As we said, this value is meaningless when there's an error, but we had to use *something* for the first return value.) So we wind up printing the message "0.00 liters needed"! We'll need to fix that...

ALWAYS CHECK FOR ERRORS!

When we pass an invalid argument to `paintNeeded`, we get an error value back, which we print for the user to see. But we also get an (invalid) amount of paint, which we print as well!

```
func main() {  
    amount, err := paintNeeded(4.2, -3.0)  
    fmt.Println(err) ← Prints the error.  
    fmt.Printf("%0.2f liters needed\n", amount) ←  
}  
a width of -3.00 is invalid  
0.00 liters needed
```

This gets set to an error value.
This gets set to 0 (a meaningless value).
Prints the meaningless value!

When a function returns an error value, it usually has to return a primary return value as well. But any other return values that accompany an error value should be considered unreliable, and ignored.

When you call a function that returns an error value, it's important to test whether that value is `nil` before proceeding. If it's anything other than `nil`, it means there's an error which must be handled.

How the error should be handled depends on the situation. In the case of our `paintNeeded` function, it might be best to simply skip the current calculation and proceed with the rest of the program:

```
func main() {  
    amount, err := paintNeeded(4.2, -3.0)  
    if err != nil { ← If the error value is not nil, there must be a problem...  
        fmt.Println(err) ← So print the error.  
    } else { ← Otherwise, the error value would be nil...  
        fmt.Printf("%0.2f liters needed\n", amount) ←  
    }  
    // Additional calculations here...  
}  
a width of -3.00 is invalid
```

So it would be okay to print the amount we got back.

But since this is such a short program, you could instead call `log.Fatal` to display the error message and exit the program.

```
func main() {  
    amount, err := paintNeeded(4.2, -3.0)  
    if err != nil { ← If the error value is not nil, there must be a problem...  
        log.Fatal(err) ← So print the error and exit the program.  
    }  
    fmt.Printf("%0.2f liters needed\n", amount) ←  
}
```

2018/03/12 19:49:27 a width of -3.00 is invalid

This code will never be reached if there's an error.

The important thing to remember is that you should always check the return values to

see whether there *is* an error. What you do with the error at that point is up to you!



BREAKING STUFF IS EDUCATIONAL!

Here's a program that calculates the square root of a number. But if a negative number is passed to the `squareRoot` function, it will return an error value. Make one of the changes below, and try to compile it. Then undo your change, and try the next one. See what happens!

```
package main
```

```
import (
    "fmt"
    "math"
)
```

```
func squareRoot(number float64) (float64, error) {
    if number < 0 {
        return 0, fmt.Errorf("can't get square root of negative number")
    }
    return math.Sqrt(number), nil
}
```

```
func main() {
    root, err := squareRoot(-9.3)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%0.3f", root)
    }
}
```

If you do this...

...it will break because...

Remove one of the

The number of arguments to `return` must always

<p>arguments to return:</p> <pre><code>return math.Sqrt(number), nil</code></pre>	<p>match the number of return values in the function declaration.</p>
<p>Remove one of the variables the return values are assigned to:</p> <pre><code>root, err := squareRoot(-9.3)</code></pre>	<p>If you use any of the return values from a function, Go requires you to use all of them.</p>
<p>Remove the code that uses one of the return values:</p> <pre><code>root, err := squareRoot(-9.3) if err != nil { fmt.Println(err) } else { fmt.Printf("%0.3f", root) }</code></pre>	<p>Go requires that you use every variable you declare. This is actually a really useful feature when it comes to error return values, because it helps keep you from accidentally ignoring an error.</p>



POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make code that will run and produce the output shown.

```

package main

import (
    "errors"
    "fmt"
)

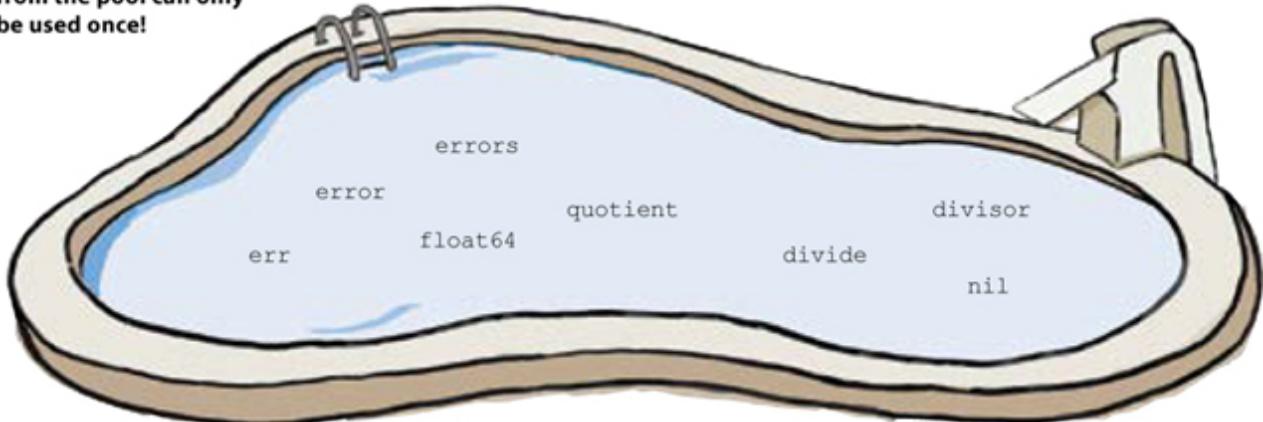
func divide(dividend float64, divisor float64) (float64, _____) {
    if divisor == 0.0 {
        return 0, _____.
    }
    return dividend / divisor, _____
}

func main() {
    _____, _____ := divide(5.6, 0.0)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%0.2f\n", quotient)
    }
}

```

Output
can't divide by 0

Note: each snippet from the pool can only be used once!



FUNCTION PARAMETERS RECEIVE COPIES OF THE ARGUMENTS

Answers on page 34.

As we mentioned, when you call a function that has parameters declared, you need to provide arguments to the call. The value in each argument is *copied* to the corresponding parameter variable. (Programming languages that do this are sometimes called "pass-by-value".)

Go is a "pass-by-value" language; function parameters receive a copy of the arguments from the function call.

This is fine in most cases. But if you want to pass a variable's value to a function and have it *change* the value in some way, you'll run into trouble. The function can only change the *copy* of the value in its parameter, not the original. So any changes you make within the function won't be visible outside it!

Here's an updated version of the `double` function we showed earlier. It takes a number,

multiplies it by 2, and prints the result.

```
package main

import "fmt"

func main() {
    amount := 6
    double(amount) ←
} ← Parameter is set to a copy of the argument
func double(number int) {
    number *= 2
    fmt.Println(number)
} ← Prints the doubled amount
```

Pass an argument to the function.

Suppose we wanted to move the statement that prints the doubled value from the `double` function back to the function that calls it, though. It won't work, because `double` only alters its *copy* of the value. Back in the calling function, when we try to print, we'll get the original value, not the doubled one!

```
func main() {
    amount := 6
    double(amount) ←
} ← Parameter is set to a copy of the argument
func double(number int) {
    number *= 2
} ← Alters the copied value,
      not the original!
} ← Prints the unchanged amount!
```

Pass an argument to the function.

Prints the original value!

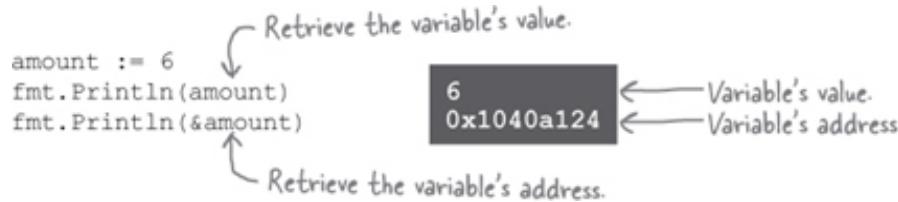
We need a way to allow a function to alter the original value a variable holds, rather than a copy. To learn how to do that, we'll need to make one more detour away from functions, to learn about *pointers*. **Go on a Detour**



POINTERS



You can get the *address* of a variable using & (an ampersand), which is Go's "address of" operator. For example, this code initializes a variable, prints its value, and then prints the variable's address...



We can get addresses for variables of any type. Notice that the address differs for each variable.

```
var myInt int
fmt.Println(&myInt)
var myFloat float64
fmt.Println(&myFloat)
var myBool bool
fmt.Println(&myBool)
```

0x1040a128
0x1040a140
0x1040a148

And what are these "addresses", exactly? Well, if you want to find a particular house in a crowded city, you use its address...



Just like a city, the memory your computer sets aside for your program is a crowded place. It's full of variable values: booleans, integers, strings, and more. Just like the address of a house, if you have the address of a variable, you can use it to find the value that variable contains.



Values that represent the address of a variable are known as **pointers**, because they *point* to the location where the variable can be found.

POINTER TYPES



The type of a pointer is written with a * symbol, followed by the type of the variable the pointer points to. The type of a pointer to an `int` variable, for example, would be written `*int` (you can read that aloud as "pointer to int").

We can use the `reflect.TypeOf` function to show us the types of our pointers from the previous program:

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var myInt int
    fmt.Println(reflect.TypeOf(&myInt)) ← Get a pointer to myInt and print the pointer's type.

    var myFloat float64
    fmt.Println(reflect.TypeOf(&myFloat)) ← Get a pointer to myFloat and print the pointer's type.

    var myBool bool
    fmt.Println(reflect.TypeOf(&myBool)) ← Get a pointer to myBool and print the pointer's type.
}

Here are the pointer types. → *int
                                         *float64
                                         *bool
```

We can declare variables that hold pointers. A pointer variable can only hold pointers to one type of value, so a variable might only hold `*int` pointers, only `*float64` pointers, and so on.

```
var myInt int
var myIntPtr *int ← Declare a variable that holds a pointer to an int.
myIntPtr = &myInt ← Assign a pointer to the variable.

fmt.Println(myIntPtr)

var myFloat float64
var myFloatPtr *float64 ← Declare a variable that holds a pointer to a float64.
myFloatPtr = &myFloat ← Assign a pointer to the variable.

fmt.Println(myFloatPtr)
```

0x1040a128
0x1040a140

As with other types, if you'll be assigning a value to the pointer variable right away, you can use a short variable declaration instead:

```
var myBool bool
myBoolPointer := &myBool ← A short declaration for a pointer variable.
fmt.Println(myBoolPointer)
```

0x1040a148

GETTING OR CHANGING THE VALUE AT A POINTER



You can get the value of the variable a pointer refers to by typing the `*` operator right before the pointer in your code. To get the value at `myIntPtr`, for example, you'd type `*myIntPtr`. (There's no official consensus on how to read `*` aloud, but we like to pronounce it as "value at": `*myIntPtr` is "value at `myIntPtr`".)

```
myInt := 4
myIntPtr := &myInt
fmt.Println(myIntPtr) ← Print the pointer itself.
fmt.Println(*myIntPtr) ← Print the value at the pointer.
```

```
myFloat := 98.6
myFloatPointer := &myFloat
fmt.Println(myFloatPointer) ← Print the pointer itself.
fmt.Println(*myFloatPointer) ← Print the value at the pointer.
```

```
myBool := true
myBoolPointer := &myBool
fmt.Println(myBoolPointer) ← Print the pointer itself.
fmt.Println(*myBoolPointer) ← Print the value at the pointer.
```

```
0x1040a124
4
0x1040a140
98.6
0x1040a150
true
```

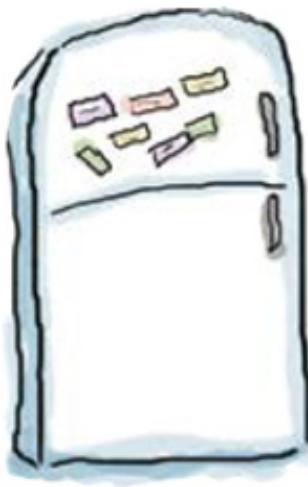
The `*` operator can also be used to update the value at a pointer:

```
myInt := 4
fmt.Println(myInt)      Assign a new value to the
myIntPtr := &myInt    variable at the pointer (myInt).
*myIntPtr = 8          ← Print the value of the
fmt.Println(*myIntPtr) ← variable at the pointer.
fmt.Println(myInt)    ← Print the variable's value directly.
```

```
4 ← Initial value of myInt
8 ← Result of updating
8 ← *myIntPtr.
8 ← Updated value of myInt
(same as *myIntPtr).
```

In the code above, `*myIntPtr = 8` accesses the variable at `myIntPtr` (that is, the `myInt` variable) and assigns a new value to it. So not only is the value of `*myIntPtr` updated, but `myInt` is as well.

CODE MAGNETS



A Go program that uses a pointer variable is scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output?

The program should declare `myInt` as an integer variable, and `myIntPtr` as a variable that holds an integer pointer. Then it should assign a value to `myInt`, and assign a pointer to `myInt` as the value of `myIntPtr`. Finally, it should print the value at `myIntPtr`.

```
package main

import "fmt"

func main() {
    Add your code here!
}

Here are the extra magnets. Add them to the program above!
```

Add your code here!

}

Output
42

Here are the extra magnets. Add them to the program above!

var	var	myInt	myInt	myInt	42
int	int	myIntPtr	myIntPtr	myIntPtr	
=	=	&	*	*	fmt.Println()

USING POINTERS WITH FUNCTIONS Answers on page 34.



It's possible to return pointers from functions; just declare that the function's return type is a pointer type.

```
func createPointer() *float64 {  
    var myFloat = 98.5  
    return &myFloat ← Return a pointer of the  
}                                         specified type.  
  
func main() {  
    var myFloatPointer *float64 = createPointer() ← Assign the returned  
    fmt.Println(*myFloatPointer) ← pointer to a variable.  
}  
                                         Print the value at  
                                         the pointer. 98.5
```

(By the way, unlike some other languages, it's okay to return a pointer to a variable that's local to a function. Even though that variable is no longer in scope, as long as you still have the pointer, Go will ensure you can still access the value.)

You can also pass pointers to functions as arguments. Just specify that the type of one or more parameters should be a pointer.

```
func printPointer(myBoolPointer *bool) {  
    fmt.Println(*myBoolPointer) ← Print the value at the pointer that gets passed in.  
}  
  
func main() {  
    var myBool bool = true  
    printPointer(&myBool) true  
}  
                                         Pass a pointer to the function.
```

Make sure you only use pointers as arguments, if that's what the function declares it will take. If you try to pass a value directly to a function that's expecting a pointer, you'll get a compile error.

```
func main() {  
    var myBool bool = true  
    printPointer(myBool)  
}
```

Error → cannot use myBool (type bool)
as type *bool in argument
to printPointer

Now you know the basics of using pointers in Go. We're ready to end our detour, and get back to fixing our `double` function!



FIXING OUR "DOUBLE" FUNCTION USING POINTERS

We have a `double` function that takes an `int` value and multiplies it by 2. We want to be able to pass a value in, and have that value doubled. But, as we learned, Go is a pass-by-value language, meaning that function parameters receive a *copy* of any arguments from the caller. Our function is doubling its copy of the value, and leaving the original untouched!

```
func main() {
    amount := 6
    double(amount)           ← Pass an argument to the function.
    fmt.Println(amount)       ← Prints the original value!
}
                           ← Parameter is set to a copy of the argument

func double(number int) {
    number *= 2
}
                           ← Alters the copied value, not the original!
                           6 ← Prints the unchanged amount!
```

Here's where our detour to learn about pointers is going to be useful. If we pass a pointer to the function, and then alter the value at that pointer, the changes will still be effective outside the function!

We only need to make a few small changes to get this working. In the `double` function, we need to update the type of the `number` parameter to take a `*int` rather than an `int`. Then we'll need to change the function code to update the value at the `number` pointer, rather than updating a variable directly. Finally, in the `main` function, we just need to update our call to `double` to pass a pointer rather than a direct value.

```
func main() {
    amount := 6
    double(&amount)           ← Pass a pointer instead of the variable value.
    fmt.Println(amount)
}
                           ← Accept a pointer instead of an int value.

func double(number *int) {
    *number *= 2
}
                           ← Update the value at the pointer.
                           12 ← Prints the doubled amount.
```

When we run this updated code, a pointer to the `amount` variable will be passed to the `double` function. The `double` function will take the value at that pointer, and double it, thereby changing the value in the `amount` variable. When we return to the `main` function

and print the `amount` variable, we'll see our doubled value!

You've learned a lot about writing your own functions in this chapter. The benefits of some of these features may not be clear right now. Don't worry, as our programs get more complex in later chapters, we'll be making good use of everything you've learned!



We've written the `negate` function below, which is *supposed* to update the value of the `truth` variable to its opposite (`false`), and update the value of the `lies` variable to its opposite (`true`). But when we call `negate` on the `truth` and `lies` variables and then print their values, we see that they're unchanged!

```
package main

import "fmt"

func negate(myBoolean bool) bool {
    return !myBoolean
}

func main() {
    truth := true
    negate(truth)
    fmt.Println(truth)
    lies := false
    negate(lies)
    fmt.Println(lies)
}
```

Actual output

true
false

Fill in the blanks below so that `negate` takes a pointer to a boolean value instead of taking a boolean value directly, then updates the value at that pointer to the opposite value. Be sure to change the calls to `negate` to pass a pointer instead of passing the value directly!

```
package main

import "fmt"

func negate(myBoolean _____) {
    _____ = !_____
}

func main() {
    truth := true
    negate(<_____>)
    fmt.Println(truth)
    lies := false
    negate(<_____>)
    fmt.Println(lies)
}
```

Output we want

false
true



YOUR GO TOOLBOX

That's it for **Chapter 3!** You've added function declarations and pointers to your toolbox.

Functions

Types

Conditionals

Loops

Function Declarations

- You can declare your own functions,
- and then call them elsewhere in
- the same package by typing the
- a function name, followed by a pair of
- parentheses containing the arguments
- the function requires (if any).
- You can declare that a function will
- return one or more values to its
- caller.

Pointers

You can get a pointer to a variable by typing Go's "address of" operator (&) right before the variable name:
`&myVariable`

Pointer types are written with a * followed by the type of value the pointer points to: *int, *bool, etc.



BULLET POINTS

- The `fmt.Printf` and `fmt.Sprintf` functions print values with formatting. The first

argument should be a formatting string containing **verbs** (%d, %f, %s, etc.) that values will be substituted for.

- Within a formatting verb, you can include a **width**: a minimum number of characters the formatted value will take up. For example, %12s results in a 12-character string (padded with spaces), %02d results in a 2-character integer (padded with zeros), and %0.3f results in a floating point number rounded to 3 decimal places.
- If you want calls to your function to accept arguments, you must declare one or more parameters, including types for each, in the function declaration. The number and type of arguments must always match the number and type of parameters, or you'll get a compile error.
- If you want your function to return one or more values, you must declare the return value types in the function declaration.
- You can't access a variable declared within a function outside that function. But you can access a variable declared outside a function (usually at package level) within that function.
- When a function returns multiple values, the last value usually has a type of `error`. Error values have an `Error()` method that returns a string describing the error.
- By convention, functions return an error value of `nil` to indicate there are no errors.
- You can access the value a pointer holds by putting a * right before it: `*myPointer`
- If a function receives a pointer as a parameter, and it updates the value at that pointer, then the updated value will still be visible outside the function.



Below is a program that declares several functions, then calls those functions within `main`. Write down what the program output would be.

```

package main

import "fmt"

func functionA(a int, b int) {
    fmt.Println(a + b)
}
func functionB(a int, b int) {
    fmt.Println(a * b)
}
func functionC(a bool) {
    fmt.Println(!a)
}
func functionD(a string, b int) {
    for i := 0; i < b; i++ {
        fmt.Print(a)
    }
    fmt.Println()
}

func main() {
    functionA(2, 3)
    functionB(2, 3)
    functionC(true)
    functionD("$", 4)
    functionA(5, 6)
    functionB(5, 6)
    functionC(false)
    functionD("ha", 3)
}

```

Output:

5

6

false

\$\$\$\$

||

30

true

hahaha

POOL PUZZLE SOLUTION

```

package main

import (
    "errors"
    "fmt"
)

func divide(dividend float64, divisor float64) (float64, error) {
    if divisor == 0.0 {
        return 0, errors.New("can't divide by 0")
    }
    return dividend / divisor, nil
}

func main() {
    quotient, err := divide(5.6, 0.0)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Printf("%0.2f\n", quotient)
    }
}

```

CODE MAGNETS SOLUTION

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    var myInt int
```

```
    var myIntPtr *int
```

```
    myInt = 42
```

```
    myIntPtr = &myInt
```

```
    fmt.Println(*myIntPtr)
```

```
}
```

Output

42



```
package main
```

```
import "fmt"
```

```
func negate(myBoolean *bool) {
    *myBoolean = !*myBoolean
}
```

```
func main() {
    truth := true
    negate(&truth)
    fmt.Println(truth)
    lies := false
    negate(&lies)
    fmt.Println(lies)
}
```

4 packages

Bundles of Code



It's time to get organized. So far, we've been throwing all our code together in a single file. As our programs grow bigger and more complex, that's going to quickly become a mess.

In this chapter, we'll show you how to create your own **packages** to help keep related code together in one place. But packages are good for more than just organization. Packages are an easy way to *share code between your programs*. And they're an easy way to *share code with other developers*.

DIFFERENT PROGRAMS, SAME FUNCTION

We've written two programs, each with an identical copy of a function, and it's becoming a maintenance headache...

On this page, we've got a new version of our `pass_fail.go` program from [Chapter 2](#). The code that reads a grade from the keyboard has been moved to a new `getFloat` function. `getFloat` returns the floating-point number the user typed, unless there's an error, in which case it returns `0` and an error value. If an error is returned, the program reports it and exists, otherwise it reports whether the grade is passing or failing, as before.

```
// pass_fail reports whether a grade is passing or failing.  
package main
```

```
import (  
    "bufio"  
    "fmt"  
    "log"  
    "os"  
    "strconv"  
    "strings"  
)
```

```
func getFloat() (float64, error) {  
    reader := bufio.NewReader(os.Stdin)  
    input, err := reader.ReadString('\n')  
    if err != nil {  
        return 0, err  
    }  
    input = strings.TrimSpace(input)  
    number, err := strconv.ParseFloat(input, 64)  
    if err != nil {  
        return 0, err  
    }  
    return number, nil  
}
```

```
func main() {  
    fmt.Print("Enter a grade: ")  
    grade, err := getFloat() ← We call getFloat to get  
    if err != nil {  
        log.Fatal(err) ← a grade...  
    }  
    var status string  
    if grade >= 60 {  
        status = "passing"  
    } else {  
        status = "failing"  
    }  
    fmt.Println("A grade of", grade, "is", status)  
}
```



`pass_fail.go`

Identical to the `getFloat` function on the next page!

Almost identical to the code in Chapter 2, except...

...if there's an error reading input, we return it from the function.

We also return any error converting the string to a `float64`.

Unchanged from Chapter 2 code.

Enter a grade: 89.7
A grade of 89.7 is passing

On this page, we've got a new `tocelsius.go` program, that lets the user type a temperature in the Fahrenheit measurement system and converts it to the Celsius system.

Notice that the `getFloat` function in `tocelsius.go` is identical to the `getFloat` function in `pass_fail.go`.

```
// tocelsius converts a temperature from Fahrenheit to Celsius.  
package main
```

```
import (  
    "bufio"  
    "fmt"  
    "log"  
    "os"  
    "strconv"  
    "strings"  
)
```

```
func getFloat() (float64, error) {  
    reader := bufio.NewReader(os.Stdin)  
    input, err := reader.ReadString('\n')  
    if err != nil {  
        return 0, err  
    }  
  
    input = strings.TrimSpace(input)  
    number, err := strconv.ParseFloat(input, 64)  
    if err != nil {  
        return 0, err  
    }  
    return number, nil  
}
```



Identical to the
getFloat function on
the previous page!

```
func main() {  
    fmt.Print("Enter a temperature in Fahrenheit: ")  
    fahrenheit, err := getFloat() ← We call getFloat to get a temperature...  
    if err != nil {  
        log.Fatal(err) ← If an error is returned, we log it and exit.  
    }  
    celsius := (fahrenheit - 32) * 5 / 9 ← Convert temperature to Celsius...  
    fmt.Printf("%0.2f degrees Celsius\n", celsius) ←  
        ...and print it with 2 decimal  
        places of precision.
```

```
Enter a temperature in Fahrenheit: 98.6  
37.00 degrees Celsius
```

SHARING CODE BETWEEN PROGRAMS USING PACKAGES



More repeated code... If we ever discover a bug in the `getFloat` function, it'll be a pain to fix it in two places. These are two different programs, though, so I guess it can't be helped...

```
func getFloat() (float64, error) {
    reader := bufio.NewReader(os.Stdin)
    input, err := reader.ReadString('\n')
    if err != nil {
        return 0, err
    }

    input = strings.TrimSpace(input)
    number, err := strconv.ParseFloat(input, 64)
    if err != nil {
        return 0, err
    }
    return number, nil
}
```

Actually, there is something we can do — we can move the shared function to a new package!

Go allows us to define our own packages. As we discussed back in [Chapter 1](#), a package is a group of code that all does similar things. The `fmt` package formats output, the `math` package works with numbers, the `strings` package works with strings, and so on. We've used the functions from each of these packages in multiple programs already.

Being able to use the same code between programs is one of the major reasons packages exist. If parts of your code are shared between multiple programs, you should consider moving them into packages.

THE GO WORKSPACE DIRECTORY HOLDS PACKAGE CODE

Go tools look for package code in a special directory (folder) on your computer called the **workspace**. By default, the workspace is a directory named `go` in the current user's home directory.

The workspace directory contains three subdirectories:

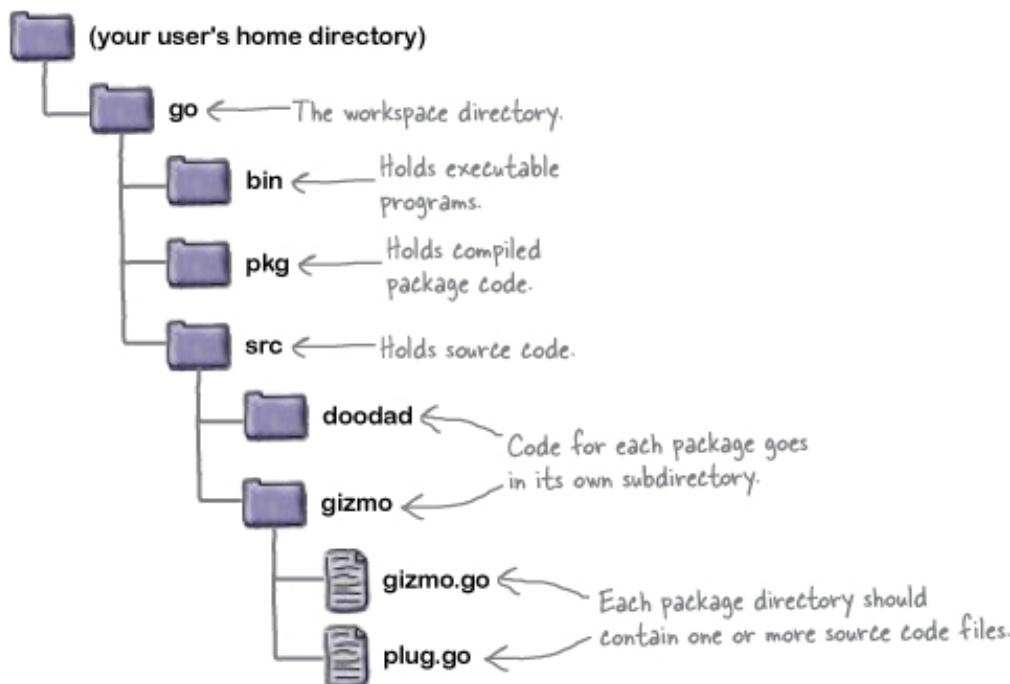
- `bin`, which holds compiled binary executable programs. (We'll talk about `bin` more later in the chapter.)
- `pkg`, which holds compiled binary package files. (We'll also talk about `pkg` more later in

the chapter.)

- `src`, which holds Go source code.

Within `src`, code for each package lives in its own separate subdirectory. By convention, the subdirectory name should be the same as the package name (so code for a `gizmo` package would go in a `gizmo` subdirectory).

Each package directory should contain one or more source code files. The file names don't matter, but they should end in a `.go` extension.



THERE ARE NO DUMB QUESTIONS

Q: You said a package folder can contain multiple files. What should go in each file?

A: Whatever you want! You can keep all of a package's code in one file, or split it between multiple files. Either way, it will all become part of the same package.

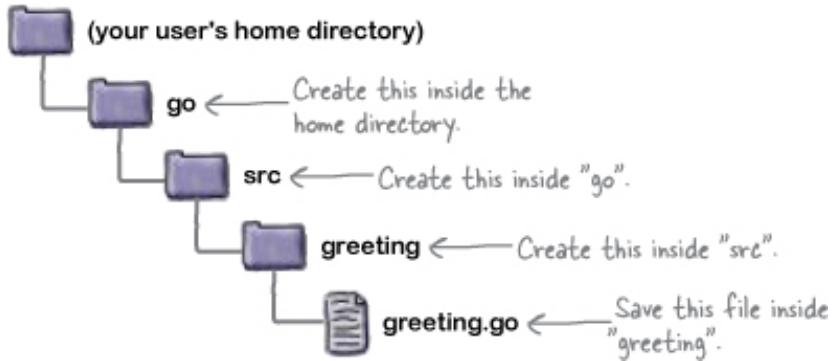
CREATING A NEW PACKAGE

Let's try setting up a package of our own in the workspace. We'll make a simple package named `greeting` that prints greetings in various languages.

The workspace directory isn't created by default when Go is installed, so you'll need to create it yourself. Start by going to your home directory. (The path is `C:\Users\yourname` on

most Windows systems, `/Users/yourname` on Macs, and `/home/yourname` on most Linux systems.) Within the home directory, create a directory named `go` — this will be our new workspace directory. Within the `go` directory, create a directory named `src`.

Finally, we need a directory to hold our package code. By convention, a package's directory should have the same name as a package. Since our package will be named `greeting`, that's the name you should use for the directory.



We know, that seems like a lot of nested directories (and actually, we'll be nesting them even deeper shortly). But trust us, once you've built up a collection of packages of your own as well as packages from others, this structure will help you keep your code organized.

And more importantly, this structure helps Go tools find the code. Because it's always in the `src` directory, Go tools know exactly where to look to find code for the packages you're importing.

Your next step is to create a file within the `greeting` directory, and name it `greeting.go`. The file should include the code below. We'll talk about it more shortly, but for now there's just a couple things we want you to notice...

Like all of our Go source code files thus far, this file starts with a `package` line. But unlike the others, this code isn't part of the `main` package; it's part of a package named `greeting`.

Also notice the two function definitions. They aren't much different from other functions we've seen so far. But because we want these functions to be accessible outside the `greeting` package, notice that we capitalize the first letter of their names so the functions are exported.

```

package greeting ← The package isn't "main", it's "greeting"!

import "fmt"
func Hello() {
    fmt.Println("Hello!")
}

func Hi() {
    fmt.Println("Hi!")
}

```

First letters capitalized so that functions are exported!

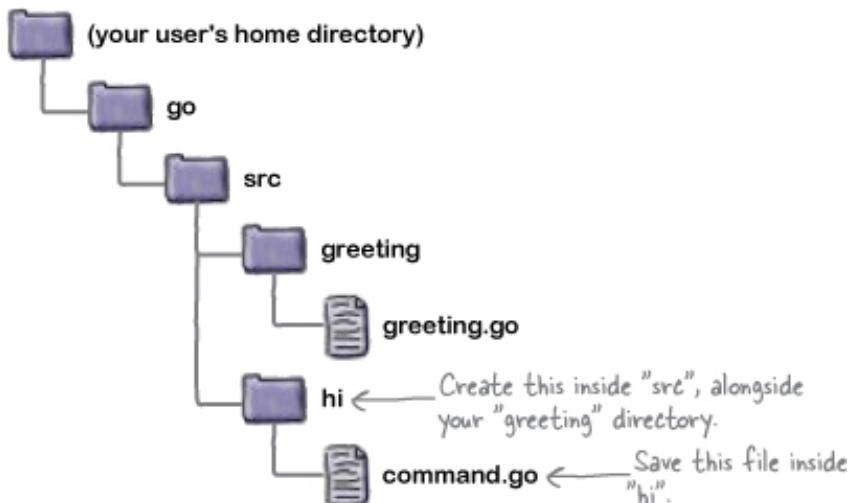


greeting.go

IMPORTING OUR PACKAGE INTO A PROGRAM

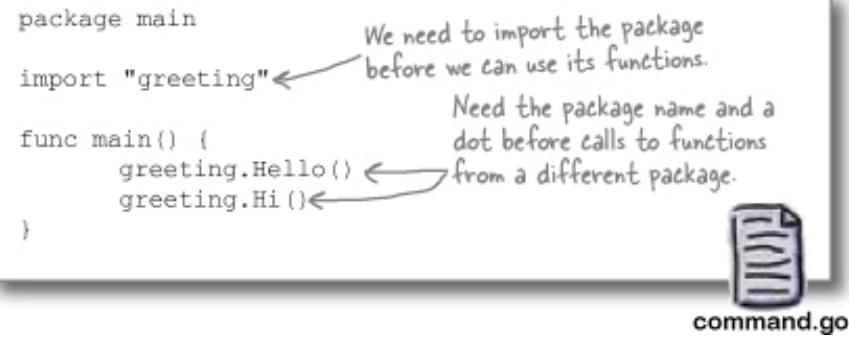
Now let's try using our new package within a program.

In your workspace directory, within the `src` subdirectory, create another subdirectory named `hi`. (We don't *have* to store code for executable programs within the workspace, but it's a good idea.)



Then, within your new `hi` directory, we need to create another source file. We can name the file anything we want, as long as it ends with a `.go` extension, but since this is going to be an executable command, we'll name it `command.go`. Save the code below within the file.

Like in every Go source code file, this code starts with a `package` line. But because we intend this to be an executable command, we need to use a package name of `main`. Generally, the package name should match the name of the directory it's kept in, but the `main` package is an exception to that rule.



Next we need to import the `greeting` package so we can use its functions. Go tools look for package code in a folder within the workspace's `src` directory whose name matches the name in the `import` statement. To tell Go to look for code in the `src/greeting` directory within the workspace, we use `import "greeting"`.

Finally, because this is code for an executable, we need a `main` function that will be called when the program runs. In `main` we call both functions that are defined in the `greeting` package. Both calls are preceded by the package name and a dot, so that Go knows which package the functions are a part of.

We're all set; let's try running the program. In your terminal or command prompt window, use the `cd` command to change to the `src/hi` directory within your workspace directory. (The path will vary based on the location of your home directory.) Then, use `go run command.go` to run the program.

Functions from the package are called! {

```

Shell Edit View Window Help
$ cd /Users/jay/go/src/hi
$ go run command.go
Hello
Hi
$ 

```

When it sees the `import "greeting"` line, Go will look in the `greeting` directory in your workspace's `src` directory for the package source code. That code gets compiled and imported, and we're able to call the `greeting` package's methods!

PACKAGES USE THE SAME FILE LAYOUT

Remember back in Chapter 1, we talked about the three sections every Go source code file has?

You'll quickly get used to seeing these three sections, in this order, in almost every Go file you work with:

1. The package declaration
2. Any import statements
3. The actual code

The package declaration. `{package main`
The imports section. `{import "fmt"`
The actual code. `{func main() {`
 `fmt.Println("Hello, Go!")`
 `}`

That rule holds true for the `main` package in our `command.go` file, of course. In our code, you can see a package declaration, followed by an imports section, followed by the actual code for our package.

The package declaration. `{package main`
The imports section. `{import "greeting"`
The actual code. `{func main() {`
 `greeting.Hello()`
 `greeting.Hi()`
 `}`

Packages other than `main` follow the same format. You can see that our `greeting.go` file also has a package declaration, imports section, and the actual package code at the end.

The package declaration. `{package greeting`
The imports section. `{import "fmt"`
The actual code. `{func Hello() {`
 `fmt.Println("Hello!")`
 `}`
 `func Hi() {`
 `fmt.Println("Hi!")`
 `}`



BREAKING STUFF IS EDUCATIONAL!

Take our code for the `greeting` package, as well as the code for the program that imports it. Try making one of the changes below, and run it. Then undo your change, and try the next one. See what happens!



```

package greeting

import "fmt"

func Hello() {
    fmt.Println("Hello!")
}

func Hi() {
    fmt.Println("Hi!")
}

```



```

package main

import "greeting"

func main() {
    greeting.Hello()
    greeting.Hi()
}

```

If you do this...

...it will fail because...

Change the name
on the greeting
directory



The Go tools use the name in the import path as the name of the directory to load the package source code from. If they don't match, the code won't load.

Change the name
on the package line
of `greeting.go`

```
package salutation
```

The contents of the `greeting` directory *will* actually load, as a package named `salutation`. Since the function calls in `command.go` still reference the `greeting` package, though, we'll get errors.

Change the
function names in
`greeting.go` and
`command.go` to all
lower-case

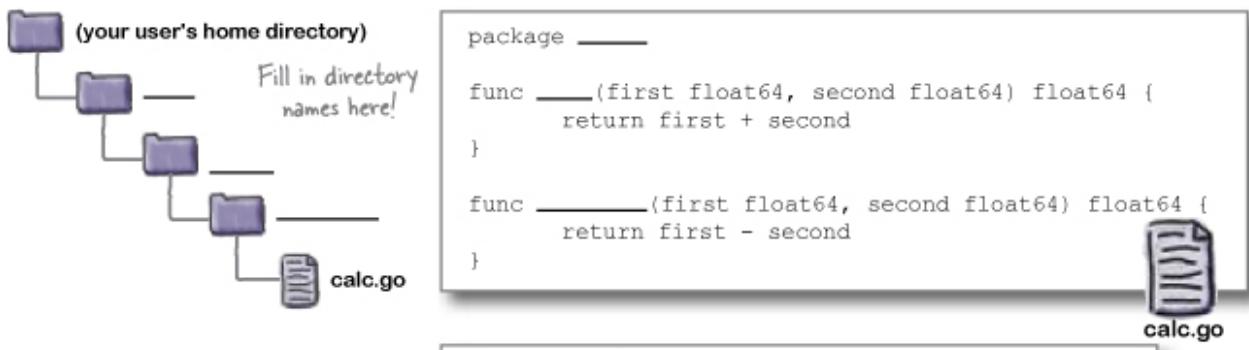
```
func Hhello()
func
```

Functions whose names begin with a lower-case letter are unexported, meaning they can only be used within their own package. To use a function from a different package, its name must begin with a capital letter, so it's exported.

```
hi()  
  
greeting.hello()  
greeting.hi()
```

POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines.
Don't use the same snippet more than once, and you won't need to use all the snippets.
Your **goal** is to set up a `calc` package within a Go workspace so `calc`'s functions can be used within `command.go`.

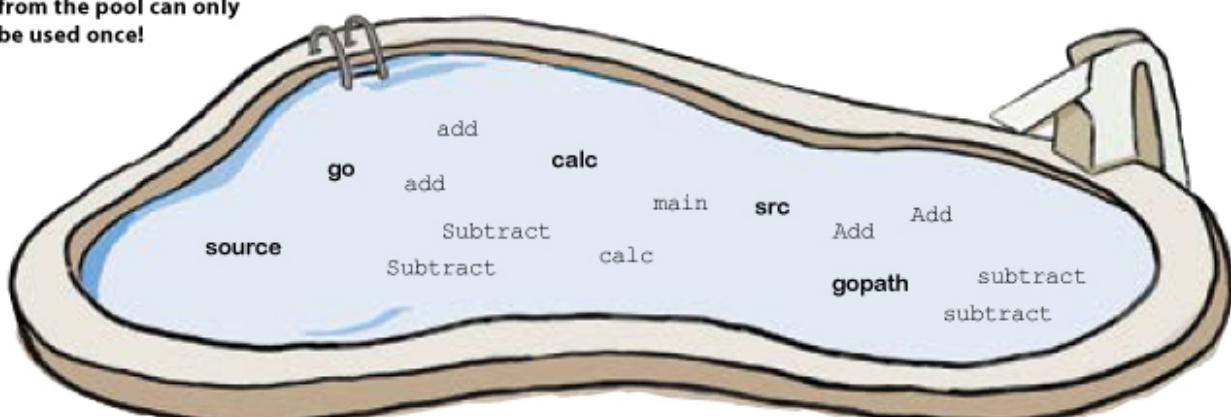


The 'command.go' file contains the following code:

```
package main  
  
import (  
    "calc"  
    "fmt"  
)  
  
func main() {  
    fmt.Println(calc.__(1, 2))  
    fmt.Println(calc.__(7, 3))  
}
```

An arrow labeled 'Output' points to the right, leading to a document icon. To the right of the document icon is a small box containing the numbers '3' and '4'. The document icon is associated with the line 'fmt.Println(calc.__(7, 3))'.

Note: each snippet from the pool can only be used once!



Developers using a package are going to need to type its name each and every time they call a function from that package. (Think of `fmt.Sprintf`, `fmt.Println`, `fmt.Print`, etc.) To make that as painless as possible, there are a few rules package names should follow:

- A package name should be all lower-case.
- The name should be abbreviated if the meaning is fairly obvious (such as `fmt`).
- It should be one word, if possible. If two words are needed, they should *not* be separated by underscores, and the second word should *not* be capitalized. (The `strconv` package is one example.)
- Imported package names can conflict with local variable names, so don't use a name that package users are likely to want to use as well. (For example, if the `strings` package were named `string`, no one who imported that package would be able to name a local variable `string`).

PACKAGE QUALIFIERS

When accessing a function, variable, etc. that's exported from a different package, you need to qualify the name of the function or variable by typing the package name before it. When you access a function or variable that's defined in the *current* package, however, you should *not* qualify the package name.

In our `command.go` file, since our code is in the `main` package, we need to specify that the `Hello` and `Hi` functions are from the `greeting` package, by typing `greeting.Hello` and `greeting.Hi`.

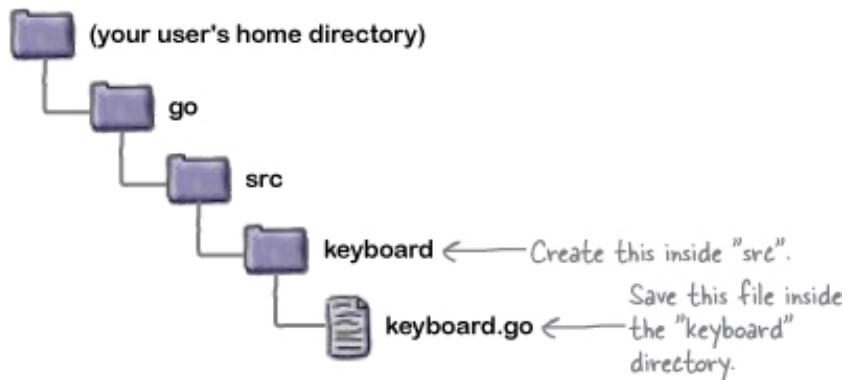


Suppose that we called the `Hello` and `Hi` methods from another function in the `greeting` package, though. There, we would just type `Hello` and `Hi` (without the package name qualifier) because we are calling the functions from the same package where they're defined.

MOVING OUR SHARED CODE TO A PACKAGE

Now that we understand how to add packages to the Go workspace, we're finally ready to move our `getFloat` function to a package that our `pass_fail.go` and `tocelsius.go` programs can both use.

Let's name our package `keyboard`, since it reads user input from the keyboard. We'll start by creating a new directory named `keyboard` inside our workspace's `src` directory.



Next, we'll create a source code file within the `keyboard` directory. We can name it anything we want, but we'll just name it after the package: `keyboard.go`.

At the top of the file, we'll need a package declaration with the package name: `keyboard`.

Then, because this is a separate file, we'll need an `import` statement with all the packages used in our code: `bufio`, `os`, `strconv`, and `strings`. (We need to leave out the `fmt` and `log` packages, as those are only used in the `pass_fail.go` and `toCelsius.go` files.)

Finally, we can copy the code from the old `getFloat` function as-is. But we need to be sure to rename the function to `GetFloat`, because it won't be exported unless the first letter of its name is capitalized.

```
package keyboard ← Add a package declaration.  
import (  
    "bufio"  
    "os"  
    "strconv"  
    "strings"  
) ← Capitalize the function name, so it's exported.  
func GetFloat() (float64, error) {  
    reader := bufio.NewReader(os.Stdin)  
    input, err := reader.ReadString('\n')  
    if err != nil {  
        return 0, err  
    }  
    input = strings.TrimSpace(input)  
    number, err := strconv.ParseFloat(input, 64)  
    if err != nil {  
        return 0, err  
    }  
    return number, nil  
}  
  
This code is identical to the old duplicated function code.
```


keyboard.go

Now the `pass_fail.go` program can be updated to use our new `keyboard` package.

Because we're removing the old `getFloat` function, we need to remove the unused `bufio`, `os`, `strconv`, and `strings` imports. In their place, we'll import the new `keyboard` package.

In our `main` function, in place of the old call to `getFloat`, we'll call the new `keyboard.GetFloat` function. The rest of the code is unchanged.

```

// pass_fail reports whether a grade is passing or failing.
package main

import (
    "fmt"
    "keyboard"
    "log"
)

```

Import only the packages used in this file.

We can remove the getFloat function that was here.

```

func main() {
    fmt.Println("Enter a grade: ")
    grade, err := keyboard.GetFloat()
    if err != nil {
        log.Fatal(err)
    }

    var status string
    if grade >= 60 {
        status = "passing"
    } else {
        status = "failing"
    }
    fmt.Println("A grade of", grade, "is", status)
}

```

Be sure to import our new package.

Call the "keyboard" package's function instead.

Enter a grade: 89.7
 A grade of 89.7 is passing

If we run the updated program, we'll see the same output as before.

We can make the same updates to the `tocelsius.go` program.

We update the imports, remove the old `getFloat`, and call `keyboard.GetFloat` instead.

And again, if we run the updated program, we'll get the same output as before. But this time, instead of relying on redundant function code, we're using the shared function in our new package!

```

// tocelsius converts a temperature...
package main

import (
    "fmt"
    "keyboard"
    "log"
)

```

Import only the packages used in this file.

We can remove the getFloat function that was here.

```

func main() {
    fmt.Println("Enter a temperature in Fahrenheit: ")
    fahrenheit, err := keyboard.GetFloat()
    if err != nil {
        log.Fatal(err)
    }
    celsius := (fahrenheit - 32) * 5 / 9
    fmt.Printf("%0.2f degrees Celsius\n", celsius)
}

```

Be sure to import our new package.

Call the "keyboard" package's function instead.

Enter a temperature in Fahrenheit: 98.6
 37.00 degrees Celsius

NESTED PACKAGE DIRECTORIES AND IMPORT

PATHS

When you're working with the packages that come with Go, like `fmt` and `strconv`, the package name is usually the same as its import path (the string you use in an `import` statement to import the package). But as we saw in [Chapter 2](#), that's not always the case...

But the import path and package name don't have to be identical. Many Go packages fall into similar categories, like compression or complex math. So they're grouped together under similar import path prefixes, such as `"archive/"` or `"math/"`. (Think of them as being similar to the paths of directories on your hard drive.)

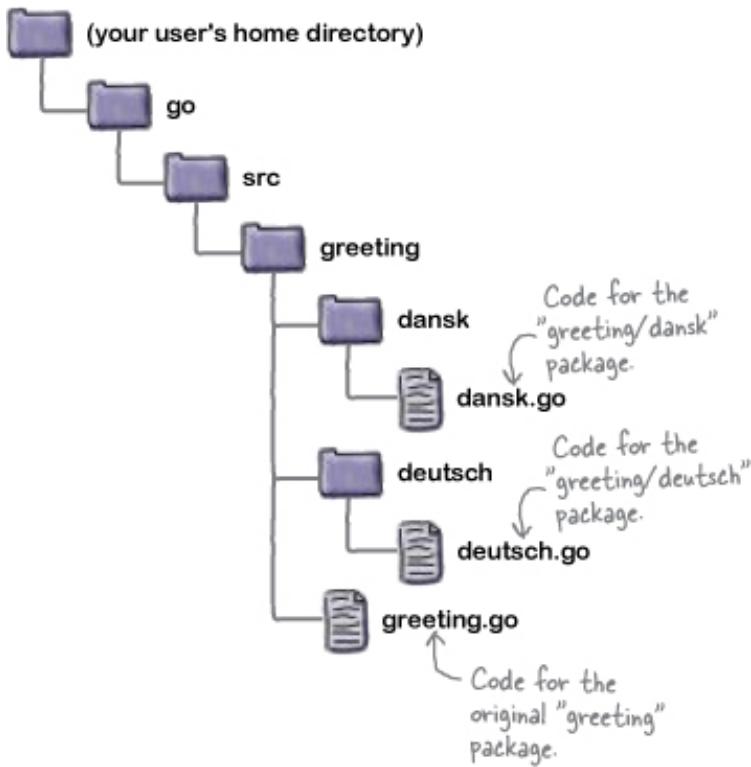
Import Path	Package Name
<code>"archive"</code>	<code>archive</code>
<code>"archive/tar"</code>	<code>tar</code>
<code>"archive/zip"</code>	<code>zip</code>
<code>"math"</code>	<code>math</code>
<code>"math/cmplx"</code>	<code>cmplx</code>
<code>"math/rand"</code>	<code>rand</code>

Some sets of packages are grouped together by import path prefixes like `"archive/"` and `"math/"`. We said to think of these prefixes as being similar to the paths of directories on your hard drive... And that wasn't a coincidence. These import path prefixes *are* created using directories!

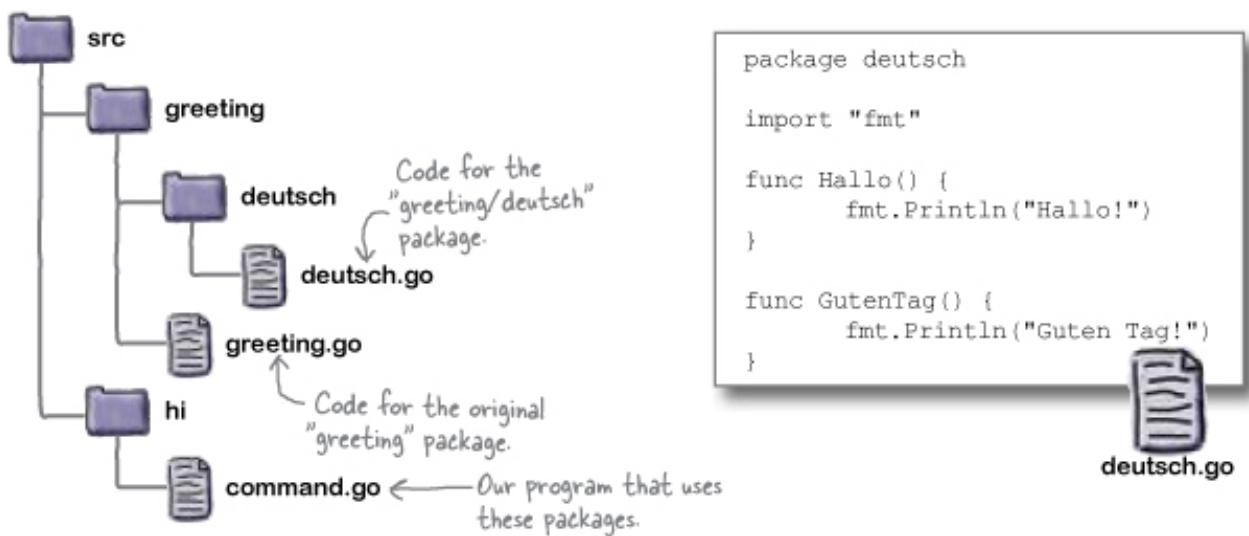
You can nest groups of similar packages together in a directory in your Go workspace. That directory then becomes part of the import path for all the packages it contains.

Suppose, for example, that we wanted to add packages for greetings in additional languages. That would quickly become a mess if we placed them all directly in the `src` directory. But if we place the new packages under the `greeting` directory, they'll all be grouped neatly together.

And placing the packages under the `greeting` directory affects their import path, too. If the `dansk` package were stored directly under `src`, its import path would be `"dansk"`. But place it within the `greeting` directory, and its import path becomes `"greeting/dansk"`. Move the `deutsch` package under the `greeting` directory, and its import path becomes `"greeting/deutsch"`. The original `greeting` package will still be available at an import path of `"greeting"`, as long as its source code file is stored directly under the `greeting` directory (not a subdirectory).



Suppose that we had a `deutsch` package nested under our `greeting` package directory, and that its code looked like this:



```

package deutsch

import "fmt"

func Hallo() {
    fmt.Println("Hallo!")
}

func GutenTag() {
    fmt.Println("Guten Tag!")
}

```

Let's update our `hi/command.go` code to use the `deutsch` package as well. Since it's nested under the `greeting` directory, we'll need to use an import path of `"greeting/deutsch"`. But once it's imported, we'll be using just the package name to refer to it: `deutsch`.

```

package main
import (
    "greeting"           Import the "greeting"
    "greeting/deutsch"   package, as before.
)
                                Import the
                                "deutsch" package
                                as well.

func main() {
    greeting.Hello()      Add calls to the new
    greeting.Hi()          package's functions.
    deutsch.Hallo()        ←
    deutsch.GutenTag()    ←
}

```



As before, we can run our code by using the `cd` command to change to the `src/hi` directory within your workspace directory. Then, we can use `go run command.go` to run the program. We'll see the results of our calls to the `deutsch` package methods in the output.

Here's the output from the `"deutsch"` package.

```

Shell Edit View Window Help
$ cd /Users/jay/go/src/hi
$ go run command.go
Hello!
Hi!
Hallo!
Guten Tag!

```

INSTALLING PACKAGES WITH "GO INSTALL"

When we use `go run`, Go has to find and compile all the packages a program depends on before it can execute the program. But it throws those compiled packages away when it's done.

Since Go's compilation is so fast, this isn't much of a problem on small projects. But once you find yourself writing many packages for a single project, you may find yourself waiting many seconds for everything to compile. This can add up when you're just testing small changes.

You can speed up this process with the `go install` command. The `go install` command saves compiled code as binary files (that is, files that aren't readable by humans but are easy for a computer to read) in your Go workspace. Once a package is installed with `go install`, it won't be compiled again unless you change its source code.

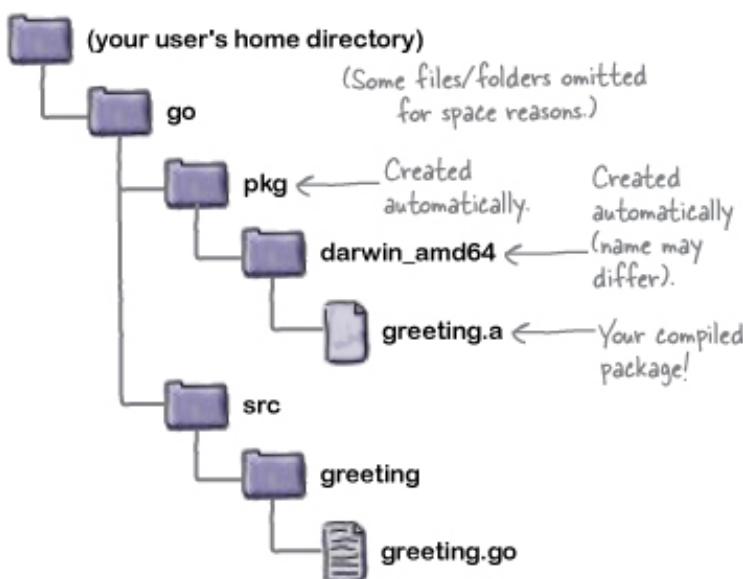
Let's try installing the `greeting` package now. From a terminal, type `go install`, followed by a space, then the import path of the package you want to install (`greeting`). It doesn't matter what directory you do this from; the `go` tool will look up the package source code within your workspace based on its import path.

```
Shell Edit View Window Help  
$ go install greeting  
$
```

If it doesn't show any error messages, and just returns you to your system prompt, that means everything worked correctly.

Now, if you look in your workspace directory, you'll see a new `pkg` directory has been created. Inside that will be a folder for indicating your operating system and type of CPU. And inside *that* will be a binary file for your compiled package.

There's no need to manually change the `pkg` folder or any of its contents; the `go` tool will manage them for you. And the next time you compile or run a program that uses your installed package, the pre-compiled version will be used automatically!



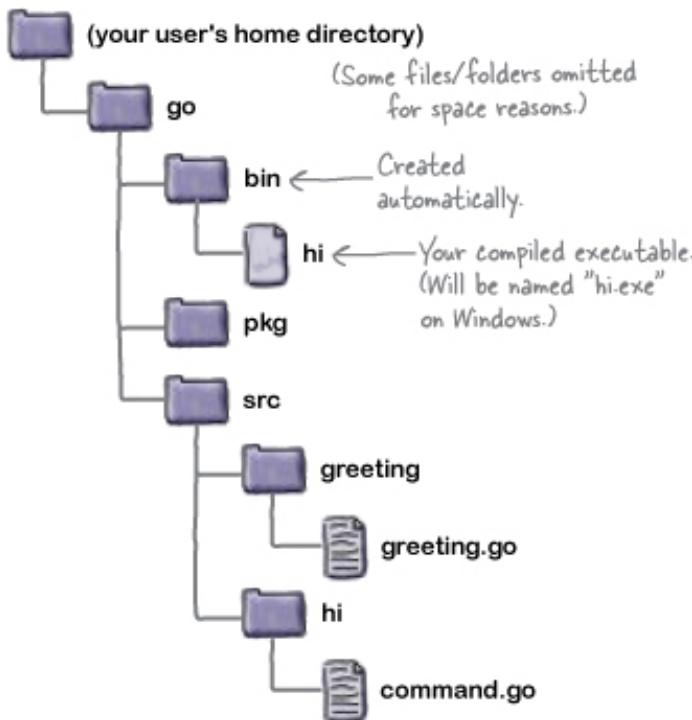
INSTALLING PROGRAM EXECUTABLES WITH "GO INSTALL"

The `go install` command can save compiled versions of executable programs, too. Just give it the name of a directory within `src` that contains code for an executable program (that is, `.go` files that begin with `package main`). The program will be compiled and an executable (a file you can execute even without Go installed) will be stored in a `bin` directory in your Go workspace.

Let's try installing an executable for our `hi/command.go` program. As before, from a terminal, we type `go install`, a space, and the name of a folder within our `src` directory (`hi`). Again, it doesn't matter what directory you do this from; the `go` tool will look the directory up within the `src` directory.

```
Shell Edit View Window Help  
$ go install hi  
$
```

When Go sees that the file inside the `hi` directory contains a `package main` declaration, it will know this is code for an executable program. First, if the packages the program depends on aren't already installed to the `pkg` directory, it will compile and install them. Then it will compile an executable file for the program itself, storing it in a directory named `bin` in the Go workspace. (The `bin` directory will be created automatically if it doesn't already exist.)



Now, you can use the `cd` command to change to the `bin` directory within your Go workspace. Once you're in `bin`, you can run the executable by typing `./hi` (or `hi.exe` on Windows).

```
Shell Edit View Window Help  
$ cd /Users/jay/go/bin  
$ ./hi  
Hello!  
Hi!  
Hallo!  
Guten Tag!
```

CHANGING WORKSPACES WITH THE GOPATH ENVIRONMENT VARIABLE

You may see developers on various websites talking about "setting your `GOPATH`" when discussing the Go workspace. `GOPATH` is an environment variable that Go tools consult to find the location of your workspace. You can use `GOPATH` to move your workspace to a

different directory, if you like.

An **environment variable** lets you store and retrieve values, kind of like a Go variable, but it's maintained by the operating system, not by Go. You can configure some programs by setting environment variables, and that includes the Go tool.

Suppose that, instead of in your home directory, you had set up your `greeting` package inside a directory named `code` in the root of your hard drive. And now you want to run your `command.go` file, which depends on `greeting`.



But you're getting an error saying the `greeting` package can't be found, because the `go` tool is still looking in the `go` directory in your home directory:

```
Shell Edit View Window Help
$ go run command.go
command.go:3:8: cannot find package "greeting" in any of:
/usr/local/go/libexec/src/greeting (from $GOROOT)
/Users/jay/go/src/greeting (from $GOPATH)
```

SETTING GOPATH

If your code is stored in a directory other than the default, you'll need to configure the `go` tool to look in the right place. You can do that by setting the `GOPATH` environment variable. How you'll do that depends on your operating system.

On Mac or Linux systems:

You'll need to use the `export` command to set the environment variable. At a terminal prompt, type:

```
export GOPATH="/code"
```

For a directory named `code` in the root of your hard drive, you'll want to use a path of `"/code"`. You can substitute a different path if your code is in a different location.

On Windows systems:

You'll need to use the `set` command to set the environment variable. At a command prompt, type:

```
set GOPATH="C:\code"
```

For a directory named `code` in the root of your hard drive, you'll want to use a path of "`C:\code`". You can substitute a different path if your code is in a different location.

Once that's done, `go run` (and other Go tools) should immediately begin using the directory you specified as their workspace. That means the `greeting` library will be found, and the program will run!

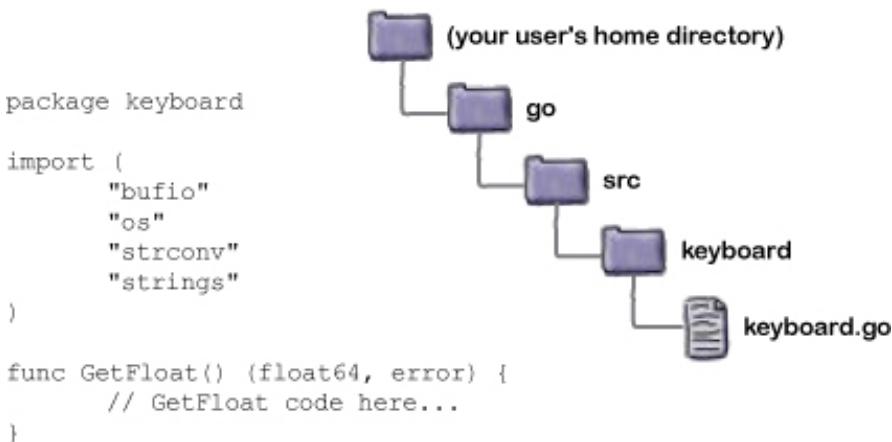
```
$ export GOPATH="/code"
$ go run command.go
Hello!
Hi!
```

```
C:\Users\jay>set GOPATH="C:\code"
C:\Users\jay>go run command.go
Hello!
Hi!
```

Note that the methods above will only set `GOPATH` for the current terminal/command prompt window. You'll need to set it again for each new window you open. But there are ways to set an environment variable permanently, if you want. The methods differ for each operating system, so we won't have space to go into them here. If you type "environment variables" followed by the name of your OS into your favorite search engine, the results should include helpful instructions.

PUBLISHING PACKAGES

We're getting so much use out of our `keyboard` package, we wonder if others might find it useful, too.



Let's create a repository to hold our code on GitHub, a popular code sharing website. That way, other developers can download it and use it in their own projects! Our GitHub username is `headfirstgo`, and we'll name the repository `keyboard`, so its URL will be:

<https://github.com/headfirstgo/keyboard>

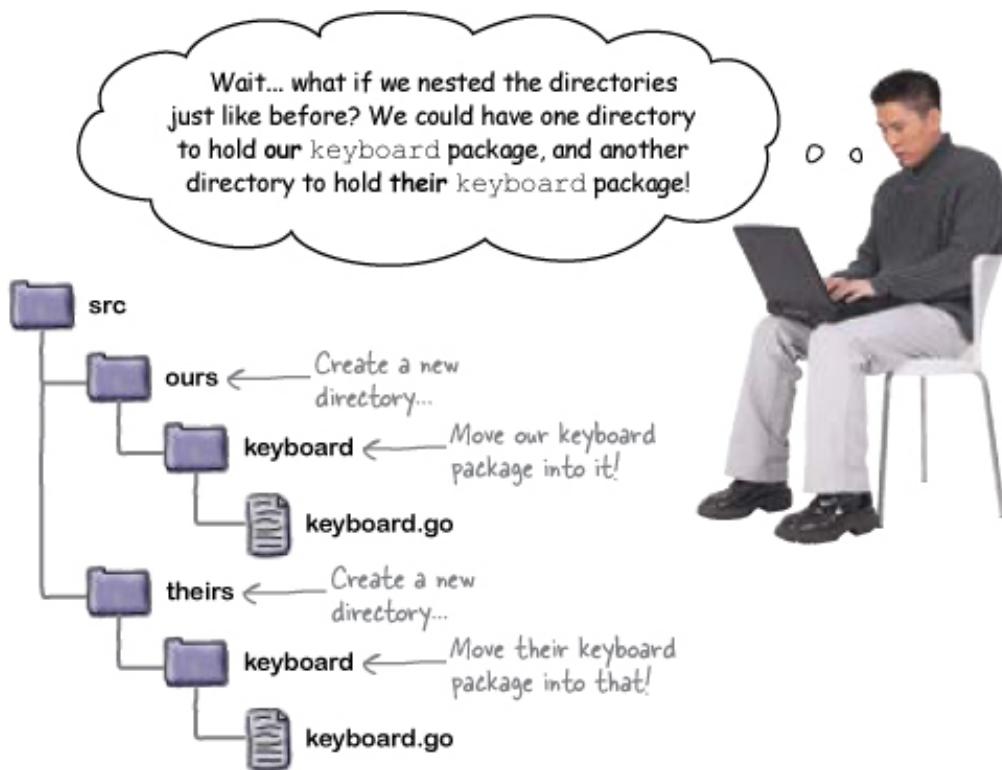
We'll upload the `keyboard.go` file to the repository, without

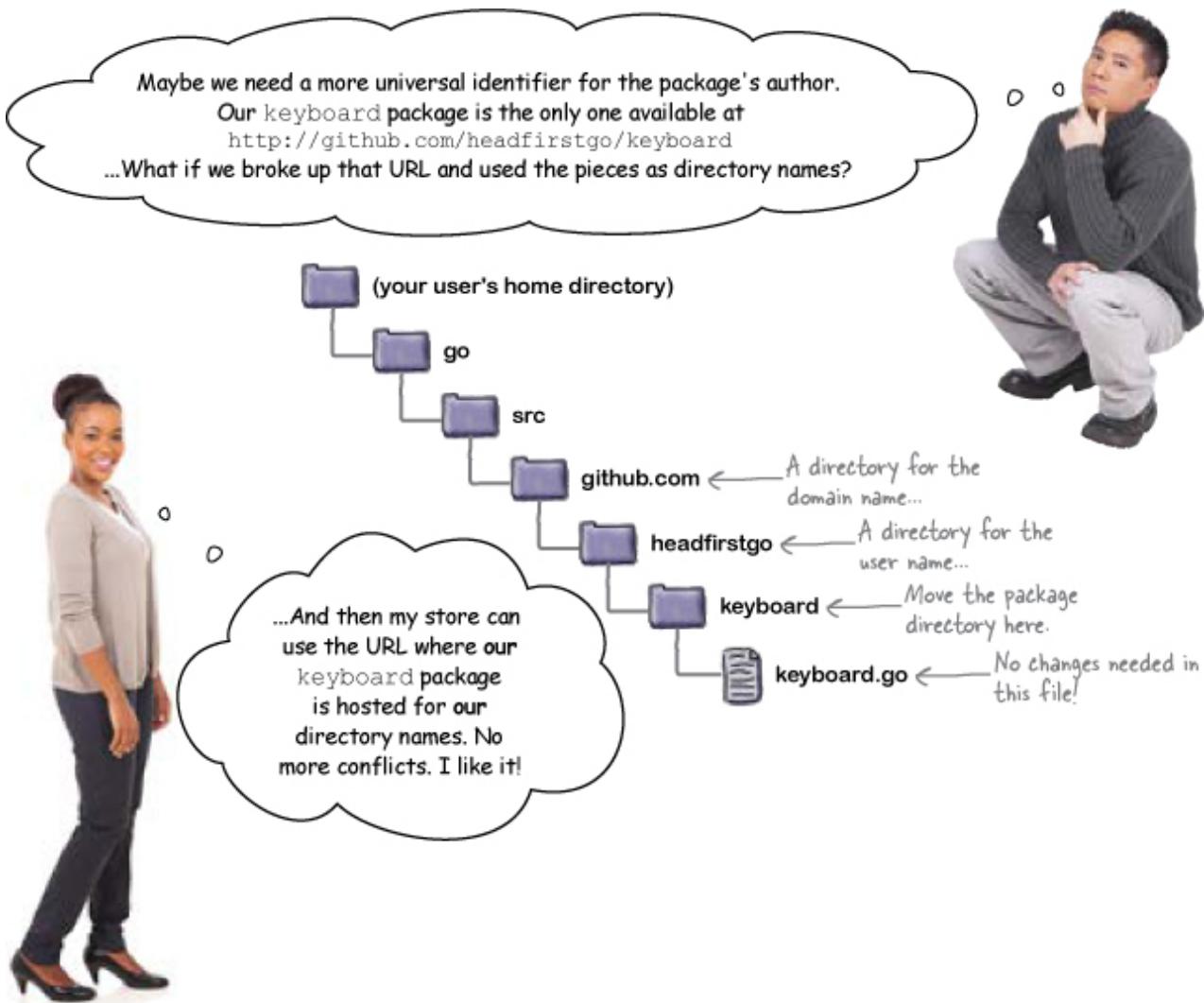
The screenshot shows a GitHub repository page for 'headfirstgo/keyboard'. The URL in the address bar is <https://github.com/headfirstgo/keyboard>. The repository name 'headfirstgo / keyboard' is displayed prominently. Below the name, it says 'A Go package for reading keyboard input.' There is one commit and one branch. A pull request from 'jaymcgavren' titled 'Add keyboard package.' has been merged. A note at the bottom states 'We uploaded just the source file, without any directories.' Handwritten annotations include: 'Our GitHub username is "headfirstgo"' pointing to the repository name; 'We named the repository "keyboard", the same as the package.' pointing to the repository name; and 'Here's the repository's URL.' pointing to the address bar.



Hmm, that's a valid concern. There can only be one `keyboard` directory in the Go

workspace's `src` directory, and so it *looks* like we can only have one package named `keyboard`!





Let's try that: we'll move our package into a directory structure that represents the URL where it's hosted. Inside our `src` directory, we'll create another directory named `github.com`. Inside that, we'll create a directory named after the next segment of the URL, `headfirstgo`. And then we'll move our `keyboard` package directory from the `src` directory into the `headfirstgo` directory.

Although moving the package into a new subdirectory will change its *import path*, it won't change the package *name*. And since the package itself only contains references to the name, we don't have to make any changes to the package code!

```

package keyboard
import (
    "bufio"
    "os"
    "strconv"
    "strings"
)
// More keyboard.go code here...

```

Package name is unchanged,
 so we don't have to change
 the package code.

`keyboard.go`

We *will* need to update the programs that rely on our package, though, because the

package import path has changed. Because we named each subdirectory after part of the URL where the package is hosted, our new import path looks a lot like that URL:

```
github.com/headfirstgo/keyboard
```

We only need to update the `import` statement in each program. Because the package name is the same, references to the package in the rest of the code will be unchanged.

With those changes made, all the programs that rely on our `keyboard` package should resume working normally.

```
// pass_fail reports whether a grade is passing or failing.  
package main  
  
import (  
    "fmt"  
    "github.com/headfirstgo/keyboard" ← Update the  
    "log"  
)  
  
func main() {  
    fmt.Println("Enter a grade: ")  
    grade, err := keyboard.GetFloat()  
    if err != nil {  
        log.Fatal(err) ← No change needed: package  
    }                                name is the same.  
    // More code here...  
}  
  
Enter a grade: 89.7  
A grade of 89.7 is passing
```

```
// tocelsius converts a temperature...  
package main  
  
import (  
    "fmt"  
    "github.com/headfirstgo/keyboard" ← Update the  
    "log"  
)  
  
func main() {  
    fmt.Println("Enter a temperature in Fahrenheit: ")  
    fahrenheit, err := keyboard.GetFloat()  
    if err != nil {  
        log.Fatal(err) ← No change needed: package  
    }                                name is the same.  
    // More code here...  
}  
  
Enter a temperature in Fahrenheit: 98.6  
37.00 degrees Celsius
```

DOWNLOADING AND INSTALLING PACKAGES WITH "GO GET"

Using a package's hosting URL as an import path has another benefit. The `go` tool has another subcommand named `go get` that can automatically download and install

packages for you.

We've set up a Git repository with the `greeting` package that we showed you previously at this URL:

```
https://github.com/headfirstgo/greeting
```

That means that from any computer with Go installed, you can type this in a terminal:

```
go get github.com/headfirstgo/greeting
```

The `go` tool will connect to `github.com`, download the Git repository at the `/headfirstgo/greeting` path, and save it in your Go workspace's `src` directory (Note: if the system doesn't have Git installed, you'll be prompted to install it when you run the `go get` command. Just follow the instructions on your screen. The `go get` command can also work with Subversion, Mercurial, and Bazaar repositories.)

The `go get` command will automatically create whatever subdirectories are needed to set up the appropriate import path. (A `github.com` directory, a `headfirstgo` directory, etc.) The packages saved in the `src` directory will look like this:

With the packages saved in the Go workspace, they're ready for use in programs. You can use the `greeting`, `dansk`, and `deutsch` packages in a program with an `import` statement like this:

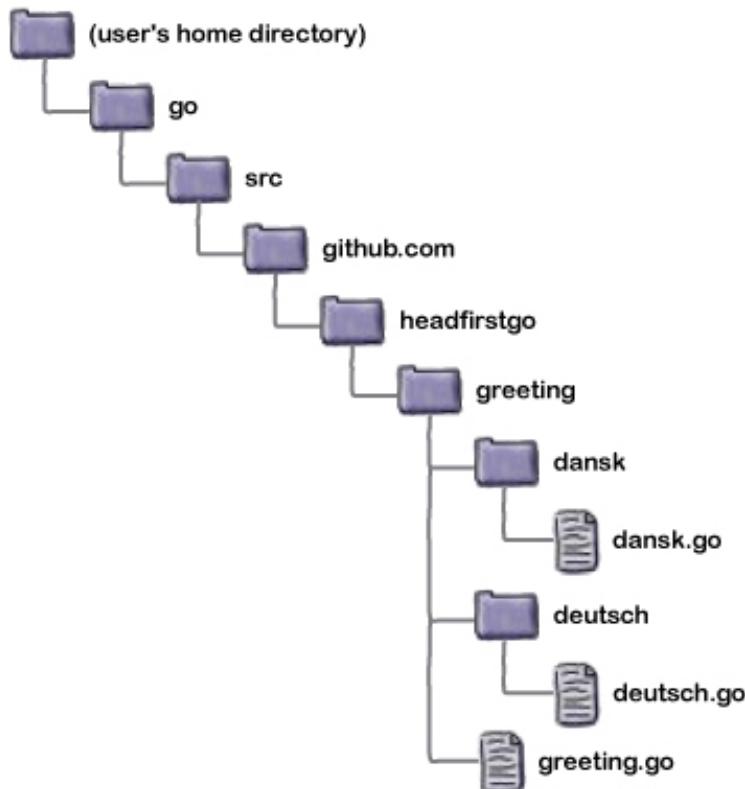
```
import (
    "github.com/headfirstgo/greeting"
    "github.com/headfirstgo/greeting/dansk"
    "github.com/headfirstgo/greeting/deutsch"
)
```

The `go get` command works for other packages, too. This command would install the `keyboard` package we showed you previously:

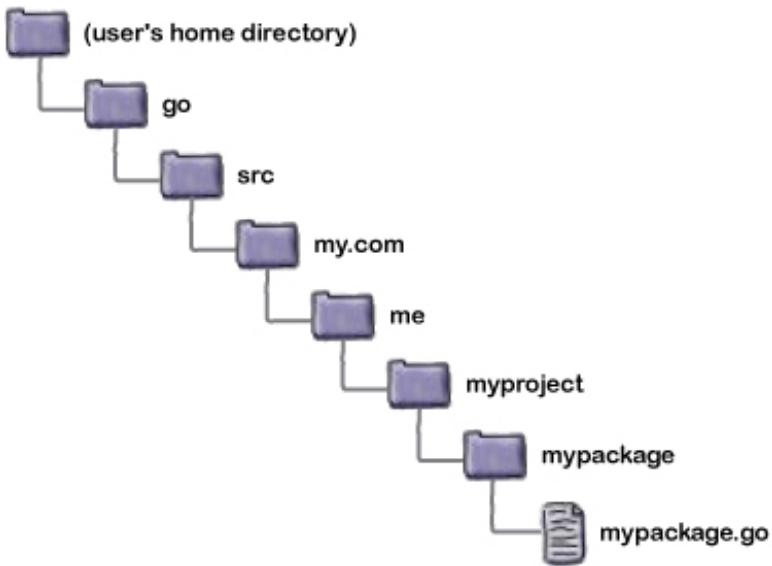
```
go get github.com/headfirstgo/keyboard
```

In fact, the `go get` command works for any package that has been set up properly on a hosting service, no matter who the author is. All you'll need to do is run `go get` and give it the package import path. The tool will look at the part of the path that corresponds to the host address, connect to that host, and download the package at the URL represented by the rest of the import path. It makes using other developers' code really

easy!



We've set up a Go workspace with a simple package named `mypackage`. Complete the program below to import `mypackage` and call its `MyFunction` function.



```
package mypackage  
func MyFunction() {  
}
```



Your code here:

```

package main

import _____

func main() {
    _____
}

```

Answers on page 25.

READING PACKAGE DOCUMENTATION WITH "GO DOC"



You can use the `go doc` command to display documentation on any package or function.

You can get a documentation for a package by passing its import path to `go doc`. For example, we can get info on the `strconv` package by running `go doc strconv`.

Get documentation for
strconv package

Package name and
import path.

Package description.

Included functions.

(Some output omitted to save space.)

```

Shell Edit View Window Help
$ go doc strconv
package strconv // import "strconv"

Package strconv implements conversions to and from
string representations of basic data types.

Numeric Conversions

The most common numeric conversions are Atoi (string
to int) and Itoa (int to string).

    i, err := strconv.Atoi("-42")
    s := strconv.Itoa(-42)

    [...Further description of the package here...]

    [...Function names...]
    func Itoa(i int) string
    func ParseBool(str string) (bool, error)
    func ParseFloat(s string, bitSize int) (float64, error)
    [...More function names...]

```

The output includes the package name and import path (which are one and the same in this case), a description of the package as a whole, and a list of all the functions the package includes.

You can also use `go doc` to get detailed info on specific functions by providing a function name following the package name. Suppose we saw the `ParseFloat` function in the list of the `strconv` package's functions and we wanted to know more about it. We could bring up its documentation with `go doc strconv.Parsefloat`.

You'll get back a description of the function and what it does:

Get documentation for `strconv.ParseFloat`

Function name, parameters, and return values.

Function description.

```
Shell Edit View Window Help
$ go doc strconv.ParseFloat
func ParseFloat(s string, bitSize int) (float64, error)
    ParseFloat converts the string s to a floating-point
    number with the precision specified by bitSize: 32
    for float32, or 64 for float64. When bitSize=32, the
    result still has type float64, but it will be
    convertible to float32 without changing its value.
```

The first line looks just like a function declaration would look like in code. It includes the function name, followed by parentheses containing the names and types of the parameters it takes (if any). If there are any return values, those will appear after the parameters.

This is followed by a detailed description of what the function does, along with any other information developers need in order to use it.

We can get documentation for our `keyboard` package in the same way, by providing its import path to `go doc`. Let's see if there's anything there that will help our would-be user. From a terminal, run:

```
go doc github.com/headfirstgo/keyboard
```

The `go doc` tool is able to derive basic information like the package name and import path from the code. But there's no package description, so it's not that helpful.

Get documentation for "keyboard" package.

Package name and import path.

No package description!

Package functions.

```
Shell Edit View Window Help
$ go doc github.com/headfirstgo/keyboard
package keyboard // import "github.com/headfirstgo/keyboard"
func GetFloat() (float64, error)
```

Requesting info on the `GetFloat` function doesn't get us a description either:

Get documentation for
GetFloat function.
No function description!

```
Shell Edit View Window Help
$ go doc github.com/headfirstgo/keyboard GetFloat
func GetFloat() (float64, error)
```

DOCUMENTING YOUR PACKAGES WITH DOC COMMENTS

The `go doc` tool works hard to add useful info to its output based on examining the code. Package names and import paths are added for you. So are function names, parameters, and return values.

But `go doc` isn't magic. If you want your users to see documentation of a package or function's intent, you'll need to add it yourself.

Fortunately, that's easy to do: you simply add **doc comments** to your code. Ordinary Go comments that appear immediately before a package or function declaration are treated as doc comments, and will be displayed in `go doc`'s output.

Let's try adding doc comments for the `keyboard` package. At the top of the `keyboard.go` file, immediately before the `package` line, we'll add a comment describing what the package does. And immediately before the declaration of `GetFloat`, we'll add a couple comment lines describing that function.

```
// Package keyboard reads user input from the keyboard.
package keyboard

import (
    "bufio"
    "os"
    "strconv"
    "strings"
)

// GetFloat reads a floating-point number from the keyboard.
// It returns the number read and any error encountered.
func GetFloat() (float64, error) {
    // No changes to GetFloat code
}
```

The next time we run `go doc` for the package, it will find the comment before the `package` line and convert it to a package description. And when we run `go doc` for the `GetFloat` function, we'll see a description based on the comment lines we added above `GetFloat`'s declaration.

```

File Edit Window Help
$ go doc github.com/headfirstgo/keyboard
package keyboard // import "github.com/headfirstgo/keyboard"
Package description. → Package keyboard reads user input from the keyboard.

func GetFloat() (float64, error)

File Edit Window Help
$ go doc github.com/headfirstgo/keyboard GetFloat
func GetFloat() (float64, error)
Function description. { GetFloat reads a floating-point number from the
keybaord. It returns the number read and any error
encountered.

```

Being able to display documentation via `go doc` makes developers that install a package happy.



And doc comments make developers who work on a package's code happy, too! They're ordinary comments, so they're easy to add. And you can easily refer to them while making changes to the code.

```

Package comment. → // Package keyboard reads user input from the keyboard.
package keyboard

import (
    "bufio"
    "os"
    "strconv"
    "strings"
)

Function comment. { // GetFloat reads a floating-point number from the keyboard.
// It returns the number read and any error encountered.
func GetFloat() (float64, error) {
    // GetFloat code here
}

```

There are a few conventions to follow when adding doc comments:

- Comments should be complete sentences.

- Package comments should begin with "Package" followed by the package name.

```
// Package mypackage enables widget management.
```

- Function comments should begin with the name of the function they describe.

```
// MyFunction converts widgets to gizmos.
```

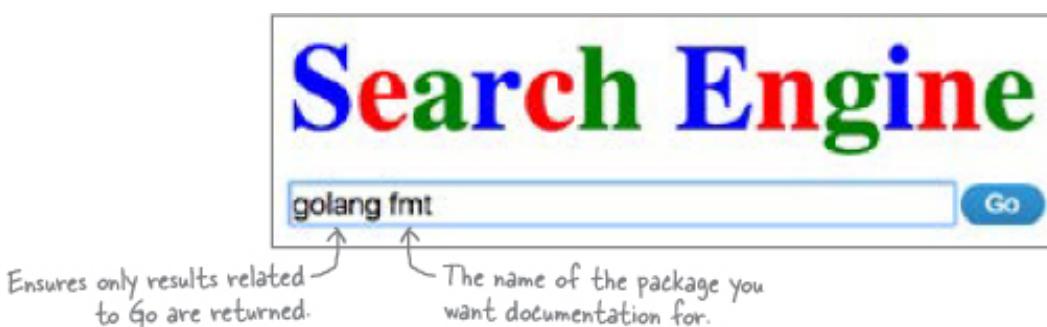
- You can include code examples in your comments by indenting them.

- Other than indentation for code samples, don't add extra punctuation characters for emphasis or formatting. Doc comments will be displayed as plain text, and should be formatted that way.

VIEWING DOCUMENTATION IN A WEB BROWSER

If you're more comfortable in a web browser than a terminal, there are other ways to view package documentation.

The simplest is to type the word "golang" followed by the name of the package you want into your favorite search engine. ("Golang" is commonly used for web searches regarding the Go language because "go" is too common a word to be useful for filtering out irrelevant results.) If we wanted documentation for the `fmt` package, we could search for "golang fmt":



The results should include sites that offer Go documentation in HTML format. If you're searching for a package in the Go standard library (like `fmt`), one of the top results will probably be from `golang.org`, a site run by the Go development team. The documentation will have much the same contents as the output of the `go doc` tool, with package names, import paths, and descriptions.

The screenshot shows the 'Package fmt' page from golang.org. At the top, it says 'Secure | https://golang.org/pkg/fmt/'. Below that, the package name 'fmt' is highlighted in blue. A callout arrow points to the word 'fmt' with the text 'Package name.'. Underneath, the import path 'import "fmt"' is shown, with a callout arrow pointing to 'fmt' and the text 'Import path.'. There are links for 'Overview' and 'Index'. A 'Overview' dropdown menu is open, showing a sub-menu item 'Println'. A callout arrow points to 'Println' with the text 'Package description.'. Below this, a paragraph of text describes the package's implementation of formatted I/O. At the bottom of the screenshot, there is a link 'Printing'.

One major advantage of the HTML documentation is that each function name in the list of the package's functions will be a handy clickable link leading to the function documentation.

The screenshot shows the 'func Println' page from golang.org. At the top, it says 'Secure | https://golang.org/pkg/fmt/#Println'. Below that, the function name 'Println' is highlighted in blue. A callout arrow points to 'Println' with the text 'Function name.'. Below the function name, the function signature 'func Println(a ...interface{}) (n int, err error)' is shown, with a callout arrow pointing to it and the text 'Function parameters and return values.'. At the bottom of the page, a brief description of the function is provided, with a callout arrow pointing to it and the text 'Function description.'.

But the content is just the same as what you'd see when running `go doc` in your terminal. It's all based on the same simple doc comments in the code.

SERVING HTML DOCUMENTATION TO YOURSELF WITH "GODOC"

The same software that powers the `golang.org` site's documentation section is actually available on *your* computer, too. It's a tool called `godoc` (not to be confused with the "`go doc`" command), and it's automatically installed along with Go. The `godoc` tool generates HTML documentation based on the code in your main Go installation and your workspace, and includes a web server that can share the resulting pages with browsers. (Don't worry, with its default settings `godoc` won't accept connections from any computer other than your own.)

To run `godoc` in web server mode, we'll type the `godoc` command (again, don't confuse that with "`go doc`") in a terminal, followed by a special option: `-http=:6060`.

Run the godoc web server.

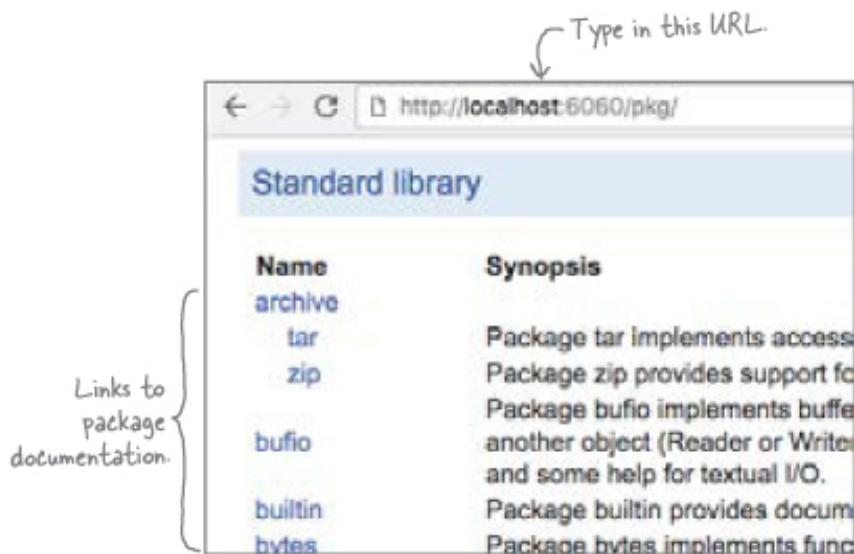


```
File Edit Window Help
$ godoc -http=:6060
```

Then with `godoc` running, you can type the URL:

`http://localhost:6060/pkg`

...into your web browser's address bar and press Enter. Your browser will connect to your own computer, and the `godoc` server will respond with an HTML page. You'll be presented with a list of all the packages installed on your machine.



Each package name in the list is a link to that package's documentation. Click it, and you'll see the same package docs that you'd see on golang.org.



Package bufio ← Package name.

import "bufio" ← Import path.

Overview
Index
Examples

Overview ▾ ← Package description.

Package bufio implements buffered I/O. It wraps (Reader or Writer) that also implements the Inter...

THE "GODOC" SERVER INCLUDES YOUR PACKAGES!

If we scroll further through our local `godoc` server's list of packages, we'll see something interesting: our `keyboard` package!

A screenshot of a web browser window displaying the `godoc` documentation for the `pkg` package. The URL in the address bar is `localhost:6060/pkg/`. The page lists several packages: `mag`, `fmt`, `github.com`, `headfirstgo`, `→ keyboard`, `go`, and `net`. A callout bubble points to the `keyboard` package with the text "Hey, look! It's our 'keyboard' package!". The `keyboard` package has a detailed description: "Package keyboard reads user input from the keyboard." Below the package list, there is a note: "Packages act declare the types used to represent syntax."

In addition to packages from Go's standard library, the `godoc` tool also builds HTML documentation for any packages in your Go workspace. These could be third-party packages you've installed, or packages you've written yourself.

Click the `keyboard` link, and you'll be taken to the package's documentation. The docs will include any doc comments from our code!

A screenshot of the `godoc` documentation for the `keyboard` package. On the left, under the "Overview" section, it says "Package keyboard reads user input from the keyboard." A callout bubble points to this with the text "Package doc comment.". On the right, under the "func GetFloat" section, it shows the function signature "func GetFloat() (float64, error)" and a detailed description: "GetFloat reads a floating-point number from the keyboard. It returns the float64 value and an error if any occurs." A callout bubble points to this with the text "Function doc comment.".

When you're ready to stop the `godoc` server, return to your terminal window, then hold the Ctrl key and press C. You'll be returned to your system prompt.

A screenshot of a terminal window. The title bar says "File Edit Window Help". The command entered is `$ godoc -http=:6060`. A callout bubble points to the command with the text "Press Ctrl-C to stop 'godoc'." When the command is run, the terminal shows the output `^C` and a new prompt `$`.

Go makes it easy to document your packages, which makes packages easier to share, which makes them easier for other developers to use. It's just one more feature that makes packages a great way to share code!



YOUR GO TOOLBOX

That's it for Chapter 4! You've added packages to your toolbox.

Functions

Types

Conditionals

Loops

Function Declarations

Pointers

Packages

The Go workspace is a special directory on your computer that holds Go code.

You can set up a package for your programs to use by creating a directory in the workspace that contains one or more source code files.



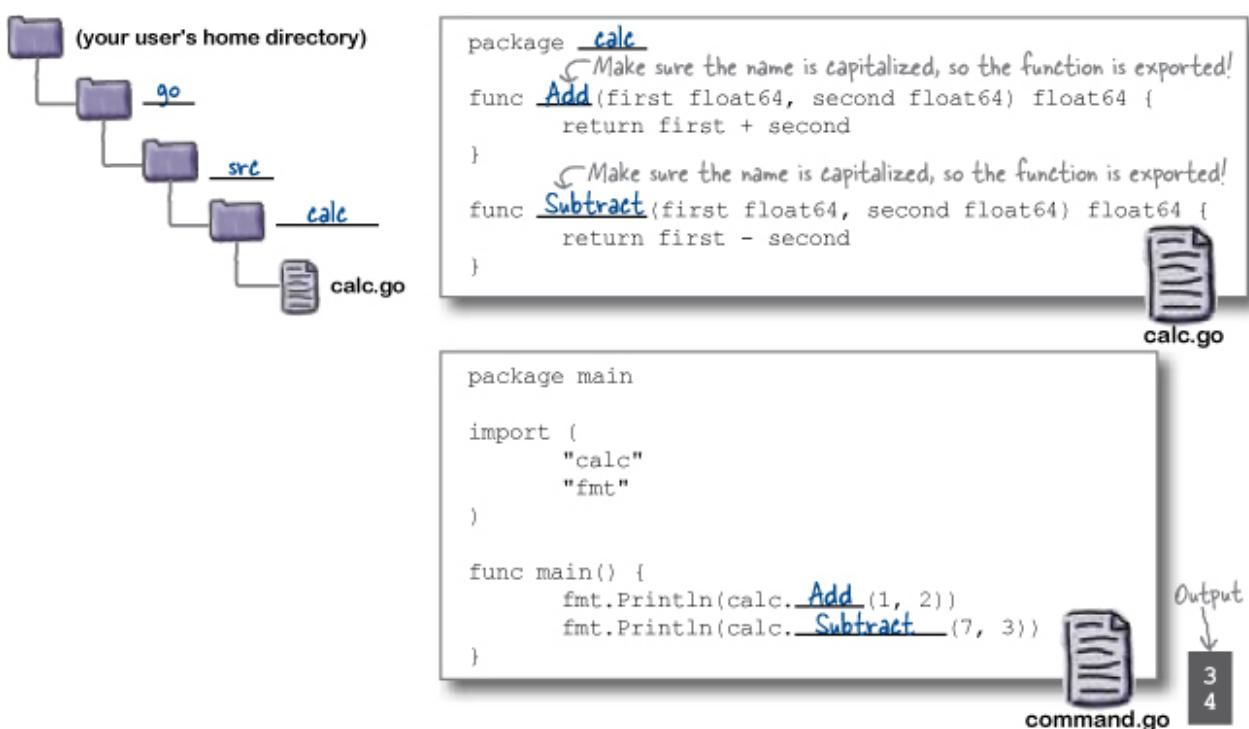
BULLET POINTS

- By default, the workspace directory is a directory named `go` within your user's home directory.
- You can use another directory as your workspace by setting the `GOPATH` environment variable.
- Go uses three subdirectories within the workspace: the `bin` directory holds compiled executable programs, the `pkg` directory holds compiled package code, and the `src` directory holds Go source code.
- The names of the directories within the `src` directory are used to form a package's import path. Names of nested directories are separated by / characters in the import path.
- The package's name is determined by the `package` declarations at the top of the source code files within the package directory. Except for the `main` package, the package name should be the same as the name of the directory that contains it.

- Package names should be all lower-case, and ideally consist of a single word.
- A package's functions can only be called from outside that package if they're exported. A function is exported if its name begins with a capital letter.
- The `go install` command compiles a package's code and stores it in the `pkg` directory for general packages, or the `bin` directory for executable programs.
- A common convention is to use the URL where a package is hosted as its import path. This allows the `go get` tool to find, download, and install packages given only their import path.
- The `go doc` tool displays documentation for packages. Doc comments within the code are included in `go doc`'s output.

POOL PUZZLE SOLUTION

Your **job** is to take code snippets from the pool and place them into the blank lines. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to set up a `calc` package within a Go workspace so `calc`'s functions can be used within `command.go`.





We've set up a Go workspace with a simple package named `mypackage`. Complete the program below to import `mypackage` and call its `MyFunction` function.

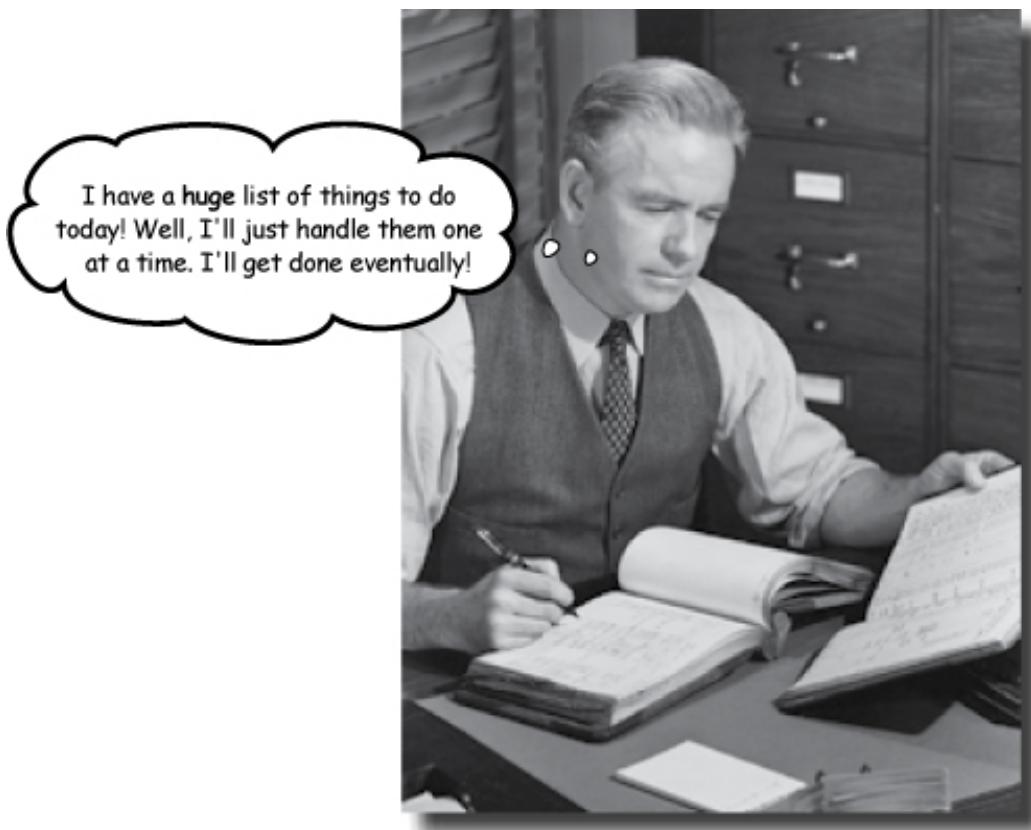
```
package mypackage  
func MyFunction() {  
}
```



```
package main  
import "my.com/me/myproject/mypackage"  
func main() {  
    mypackage.MyFunction()  
}
```

5 arrays

On the List



A whole lot of programs deal with lists of things. Lists of addresses. Lists of phone numbers. Lists of products. Go has *two* ways of storing lists built-in. This

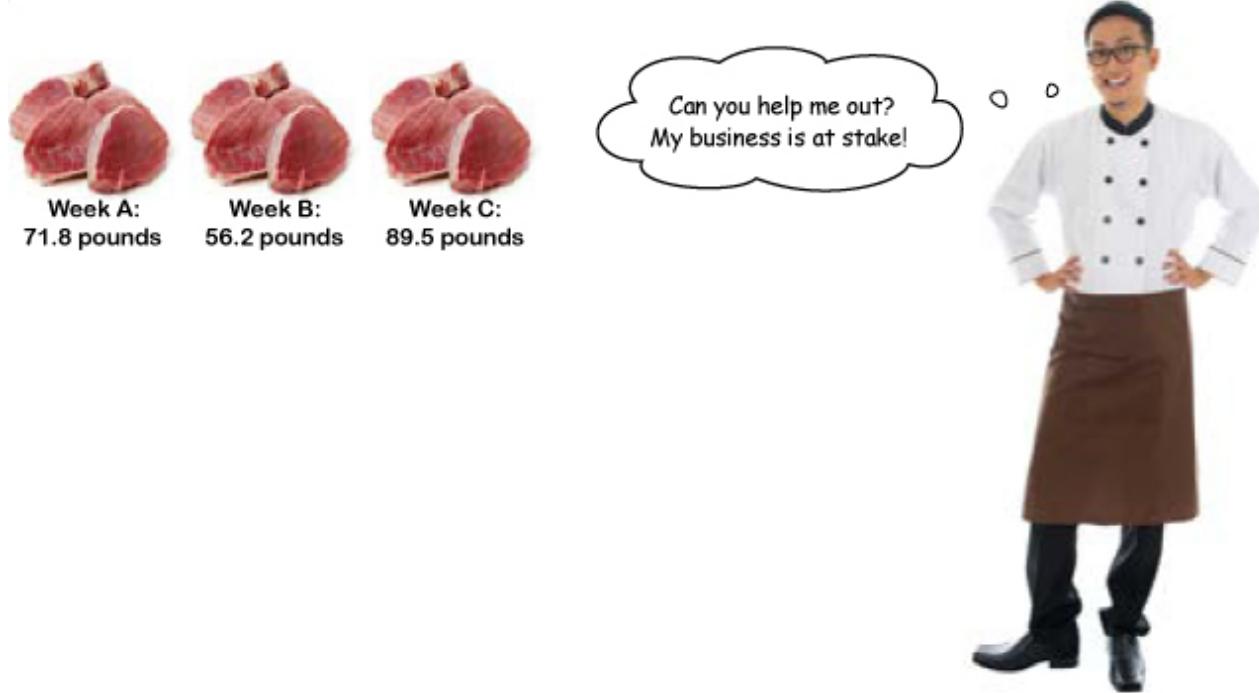
chapter will introduce the first: **arrays**. You'll learn about how to create arrays, how to fill them with data, and how to get that data back out again. Then you'll learn about processing all the elements in array, first the *hard* way with `for` loops, and then the *easy* way with `for ... range` loops.

ARRAYS HOLD COLLECTIONS OF VALUES

A local restaurant owner has a problem. He needs to know how much beef to order for

the upcoming week. If he orders too much, the excess will go to waste. If he doesn't order enough, he'll have to tell his customers that he can't make their favorite dishes.

He keeps data on how much meat was used the previous three weeks. He needs a program that will give him some idea how much to order.



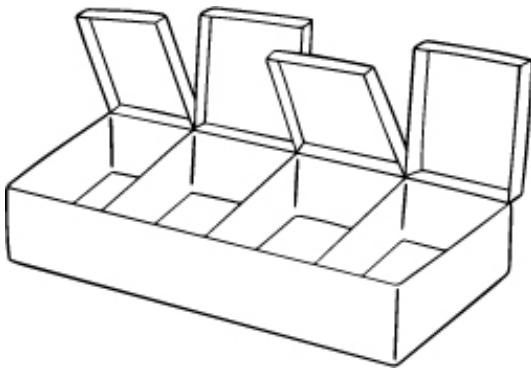
This should be simple enough: we can calculate an average by taking the three amounts, adding them together, and dividing by 3. The average should offer a good estimate of how much to order.

$$(week\ A + week\ B + week\ C) \div 3 = \text{average}$$

The first issue is going to be storing the sample values. It would be a pain to declare three separate variables, and even more so if we wanted to average more values together later. But, like most programming languages, Go offers a data structure that's perfect for this sort of situation...

An **array** is a collection of values that all share the same type. Think of it like one of those pill boxes with compartments — you can store and retrieve pills from each compartment separately, but it's also easy to transport the container as a whole.

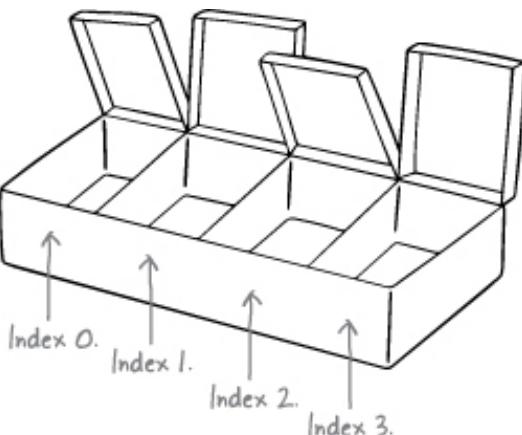
The values an array holds are called its **elements**. You can have an array of strings, an array of booleans, or an array of any other Go type (even an array of arrays). You can store an entire array in a single variable, and then access any element within the array that you need.



An array holds a specific number of elements, and it cannot grow or shrink. To declare a variable that holds an array, you need to specify the number of elements it holds in square brackets ([]), followed by the type of elements the array holds.

```
Number of  
elements array  
will hold.  
Type of  
elements array  
will hold.  
var myarray [4]string
```

To set the array elements' values or to retrieve values later, you'll need a way to specify which element you mean. Elements in an array are numbered, starting with 0. An element's number is called its **index**.



If you wanted to make an array with the names of notes on a musical scale, for example, the first note would be assigned to index 0. The second note would be at index 1. And so forth. The index is specified in square brackets.

```
Create an array of 7 strings.  
var notes [7]string  
notes[0] = "do" ← Assign a value to the first element.  
notes[1] = "re" ← Assign a value to the second element.  
notes[2] = "mi" ← Assign a value to the third element.  
fmt.Println(notes[0]) ← Print the first element.  
fmt.Println(notes[1]) ← Print the second element.  
do  
re
```

Here's an array of integers:

```
var primes [5]int
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0])
```

Create an array of 5 integers.
Assign a value to the first element.
Assign a value to the second element.
Print the first element.

2

And an array of `time.Time` values:

```
var dates [3]time.Time
dates[0] = time.Unix(1257894000, 0)
dates[1] = time.Unix(1447920000, 0)
dates[2] = time.Unix(1508632200, 0)
fmt.Println(dates[1])
```

Create an array of 3 Time values.
Assign a value to the first element.
Assign a value to the second element.
Assign a value to the third element.
Print the second element.

2015-11-19 08:00:00 +0000 UTC

ZERO VALUES IN ARRAYS

As with variables, when an array is created, all the values it contains are initialized to the zero value for the type that array holds. So an array of `int` values is filled with zeros by default:

```
Print an explicitly-
assigned element → var primes [5]int
primes[0] = 2
fmt.Println(primes[0])
Print elements that have not
had values explicitly assigned. {fmt.Println(primes[2])
{fmt.Println(primes[4])}
```

2 ← Explicitly-assigned value.
0 ← Zero value.
0 ← Zero value.

The zero value for strings, however, is an empty string, so an array of `string` values is filled with empty strings by default:

```
Print elements that have not
had values explicitly assigned. {fmt.Println(notes[3])
{fmt.Println(notes[6])}
Print an explicitly-
assigned element. → fmt.Println(notes[0])
```

do ← Zero value (empty string).
do ← Zero value (empty string).
do ← Explicitly-assigned value.

As long as you know what you're doing, zero values make it safe to manipulate an array element even if you haven't explicitly assigned a value to it. For example, here we have an array of integer counters. We can increment any of them without explicitly assigning a value first, because we know they will all start from 0.

```

var counters [3]int
counters[0]++ ← Increment the first element from 0 to 1.
counters[0]++ ← Increment the first element from 1 to 2.
counters[2]++ ← Increment the third element from 0 to 1.
fmt.Println(counters[0], counters[1], counters[2])

```



ARRAY LITERALS

If you know in advance what values an array should hold, you can initialize the array with those values using an **array literal**. An array literal starts just like an array type, with the number of elements it will hold in square brackets, followed by the type of its elements. This is followed by a list in curly braces of the initial values each element should have. The element values should be separated by commas.

Number of elements array will hold.
Type of elements array will hold.
Comma-separated list of array values.

[3]	int	{9, 18, 27}
-----	-----	-------------

These examples are just like the previous ones we showed, except that instead of assigning values to the array elements one by one, the entire array is initialized using array literals.

```

var notes [7]string = [7]string{"do", "re", "mi", "fa", "so", "la", "ti"} ← Assign values
fmt.Println(notes[3], notes[6], notes[0])                                using an array
var primes [5]int = [5]int{2, 3, 5, 7, 11} ← Assign values using an
fmt.Println(primes[0], primes[2], primes[4])                                array literal.

```

fa	ti	do
2	5	11

Using an array literal also allows you to do short variable declarations with `:=`.

Short variable declaration.
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
primes := [5]int{2, 3, 5, 7, 11}
Short variable declaration.



Below is a program that declares a couple arrays and prints out their elements. Write down what the program output would be.

```
package main

import "fmt"

func main() {
    var numbers [3]int
    numbers[0] = 42
    numbers[2] = 108
    var letters = [3]string{"a", "b", "c"}  
  
    Output:  
  
    fmt.Println(numbers[0]) .....  
    fmt.Println(numbers[1]) .....  
    fmt.Println(numbers[2]) .....  
    fmt.Println(letters[2]) .....  
    fmt.Println(letters[0]) .....  
    fmt.Println(letters[1]) .....
```

Answers on page 25.

ACCESSING ARRAY ELEMENTS WITHIN A LOOP

You don't have to explicitly write the integer index of the array element you're accessing in your code. You can also use the value in an integer variable as the array index.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
index := 1
fmt.Println(index, notes[index]) ← Print the array element at index 1.
index = 3
fmt.Println(index, notes[index]) ← Print the array element at index 3.
```

1	re
3	fa

That means you can do things like process elements of an array using a `for` loop. You loop through indexes in the array, and use the loop variable to access the element at the current index.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
for i := 0; i <= 2; i++ { ← Loop through indexes 0, 1, and 2.
    fmt.Println(i, notes[i])
}  
  
Print the element at  
the current index.
```

0	do
1	re
2	mi

When accessing array elements using a variable, you need to be careful which index values you use. As we mentioned, arrays hold a specific number of elements. Trying to

access an index that is outside the array will cause a **panic**, an error that occurs while your program is running (as opposed to when it's compiling).

```
The array only has 7 elements.  
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
for i := 0; i <= 7; i++ { ← Loops up through index 7 (the  
    fmt.Println(i, notes[i]) eighth element), which doesn't exist!  
}
```

Normally, a panic causes your program to crash and an error message to be shown to the user. Needless to say, panics should be avoided whenever possible.

```
Access indexes 0 through 6. {  
0 do  
1 re  
2 mi  
3 fa  
4 so  
5 la  
6 ti  
Accessing index 7 → causes a panic!  
panic: runtime error: index out of range  
goroutine 1 [running]:  
main.main()  
/tmp/sandbox732328648/main.go:8 +0x140
```

CHECKING ARRAY LENGTH WITH THE "LEN" FUNCTION

Writing loops that only access valid array indexes can be somewhat error-prone. Fortunately, there are a couple ways to make the process easier.

The first is to check the actual number of elements in the array before accessing it. You can do this with the built-in `len` function, which returns the length of the array (the number of elements it contains).

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
fmt.Println(len(notes)) ← Print the length of the "notes" array.  
primes := [5]int{2, 3, 5, 7, 11}  
fmt.Println(len(primes)) ← Print the length of the "primes" array.
```

7
5

When setting up a loop to process an entire array, you can use `len` to determine which indexes are safe to access.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
```

The highest value the "i" variable will reach is 6. ↘ Returns the length of the array, 7.

```
for i := 0; i < len(notes); i++ {  
    fmt.Println(i, notes[i])  
}
```

```
0 do  
1 re  
2 mi  
3 fa  
4 so  
5 la  
6 ti
```

This still has the potential for mistakes, though. If `len(notes)` returns 7, the highest index you can access is 6 (because array indexes start at 0, not 1). If you try to access index 7, you'll get a panic.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
```

The highest value the "i" variable will reach is 7! ↘ Returns the length of the array, 7.

```
for i := 0; i <= len(notes); i++ {  
    fmt.Println(i, notes[i])  
}
```

Accessing index 7 causes a panic! →

```
0 do  
1 re  
2 mi  
3 fa  
4 so  
5 la  
6 ti  
panic: runtime error: index out of range  
goroutine 1 [running]:  
main.main()  
/tmp/sandbox094804331/main.go:11 +0x140
```

LOOPING OVER ARRAYS SAFELY WITH "FOR ... RANGE"

An even safer way to process each element of an array is to use the special `for ... range` loop. In the `range` form, you provide a variable that will hold the integer index of each element, another variable that will hold the value of the element itself, and the array you want to loop over. The loop will run once for each element in the array, assigning the element's index to your first variable and the element's value to your second variable. You can add code to the loop block to process those values.

```
Variable that will hold each element's index.  
Variable that will hold each element's value.  
"range" keyword.  
The array being processed.  
for index, value := range myArray {  
    // Loop block here.  
}
```

This form of the `for` loop has no messy init, condition and post expressions. And

because the element value is automatically assigned to a variable for you, there's no risk that you'll accidentally access an invalid array index. Because it's safer and easier to read, you'll see the `for` loop's `range` form used most often when working with arrays and other collections.

Here's our previous code that prints each value in our array of musical notes, updated to use a `for ... range` loop:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
Variable to hold  
each index.    Variable to hold  
each string.    Process each value in the array.  
for index, note := range notes {  
    fmt.Println(index, note)  
}
```

0	do
1	re
2	mi
3	fa
4	so
5	la
6	ti

The loop runs seven times, once for each element of the `notes` array. For each element, the `index` variable gets set to the element's index, and the `note` variable gets set to the element's value. Then we print the index and value.

USING THE BLANK IDENTIFIER WITH "FOR ... RANGE" LOOPS

As always, Go requires that you use every variable you declare. If we stop using the `index` variable from our `for ... range` loop, we'll get a compile error:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
  
for index, note := range notes {  
    fmt.Println(note)  
}  
The "index" variable has been  
removed from the output
```

Compile error.
index declared and not used

And the same would be true if we didn't use the variable that holds the element value:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
  
for index, note := range notes {  
    fmt.Println(index)  
}  
Doesn't use the  
"note" variable.
```

Compile error.
note declared and not used

Remember in [Chapter 2](#), when we were calling a function with multiple return values, and we wanted to ignore one of them? We assigned that value to the blank identifier (`_`)

), which causes Go to discard that value, without giving a compiler error...

We can do the same with values from `for ... range` loops. If we don't need the index for each array element, we can just assign it to the blank identifier:

Use the blank identifier as a placeholder for the index value.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
for _, note := range notes {  
    fmt.Println(note)  
}  
Use only the "note" variable.
```

do
re
mi
fa
so
la
ti

And if we don't need the value variable, we can assign that to the blank identifier instead:

Use the blank identifier as a placeholder for the element value.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}  
for index, _ := range notes {  
    fmt.Println(index)  
}  
Use only the "note" variable.
```

0
1
2
3
4
5
6

GETTING A SUM OF THE NUMBERS IN AN ARRAY

OK, OK, got it. Arrays hold a collection of values. Use `for ... range` loops to process array elements. Now can we finally write this program to help me figure out how much beef to order?



We finally know everything we need to create an array of `float64` values and calculate their average. Let's take the amounts of beef that were used in previous weeks, and incorporate them into a program, named `average`.

The first thing we'll need to do is set up a program file. In your Go workspace directory (which is a `go` directory within your user's home directory, unless you've set the `GOPATH` environment variable), create the following nested directories (if they don't already exist). Within the innermost directory, `average`, save a file named `command.go`.



Now let's write our program code within the `command.go` file. Since this will be an executable program, our code will be part of the `main` package, and our code will reside in the `main` function.

We'll start by just calculating a total for the three sample values; we can go back later to calculate an average. We use an array literal to create an array of three `float64` values, pre-populated with the sample values from prior weeks. We declare a `float64` variable named `sum` to hold the total, starting with a value of `0`.

Then we use a `for ... range` loop to process each number. We don't need the element indexes, so we discard them using the `_` blank identifier. We add each number to the value in `sum`. After we've totaled all the values, we print `sum` before exiting.

```
// average calculates the average of several numbers.
package main ← This will be an executable program, so we use the "main" package.

import "fmt"

func main() {
    numbers := [3]float64{71.8, 56.2, 89.5} ← Use an array literal to create
                                                an array with the 3 float64
                                                values we're averaging.
    var sum float64 = 0 ← Declare a float64 variable to hold the sum of the 3 numbers.
    Discard the for _, number := range numbers { ← Loop through each number in the array.
                                                ↑ sum += number ← Add the current number to the total.
    }
    fmt.Println(sum)
}
```

Let's try compiling and running our program. We'll use the `go install` command to create an executable. We're going to need to provide our executable's import path to `go install`. If we used this directory structure...



...That means the import path for our package will be `github.com/headfirstgo/average`. So, from your terminal, type:

```
go install github.com/headfirstgo/average
```

You can do so from within any directory. The `go` tool will look for a

`github.com/headfirstgo/average` directory within your workspace's `src` directory, and compile any `.go` files it contains. The resulting executable will be named `average`, and will be stored in the `bin` directory within your Go workspace.

Then, you can use the `cd` command to change to the `bin` directory within your Go workspace. Once you're in `bin`, you can run the executable by typing `./average` (or `average.exe` on Windows).

Compile the contents of the "average" directory, and install the resulting executable.

Change to the "bin" directory within your workspace..

Run the executable.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
217.5
$
```

The program will print the total of the three values from our array, and exit.

GETTING THE AVERAGE OF THE NUMBERS IN AN ARRAY

We've got our `average` program printing a total of the array's values, now let's update it to print an actual average. To do that, we'll divide the total by the array's length.

Passing the array to the `len` method returns an `int` value with the array length. But since the total in the `sum` variable is a `float64` value, we'll need to convert the length to a `float64` as well so we can use them together in a math operation. We store the result in the `sampleCount` variable. Once that's done, all we have to do is divide `sum` by `sampleCount`, and print the result.

```
// average calculates the average of several numbers.
package main

import "fmt"

func main() {
    numbers := [3]float64{71.8, 56.2, 89.5}
    var sum float64 = 0
    for _, number := range numbers {
        sum += number
    }
    sampleCount := float64(len(numbers)) ← Get the array length as an int
    fmt.Printf("Average: %0.2f\n", sum/sampleCount) ← and convert it to a float64.
}
Divide the total of the array's values by the array length to get the average.
```

Once the code is updated, we can repeat the previous steps to see the new result: run `go install` to recompile the code, change to the `bin` directory, and run the updated `average`

executable. Instead of the sum of the values in the array, we'll see the average.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
Average: 72.50
$
```

The average of the array values.



POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in this code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a program that will print the index and value of all the array elements that fall between 10 and 20 (it should match the output shown).

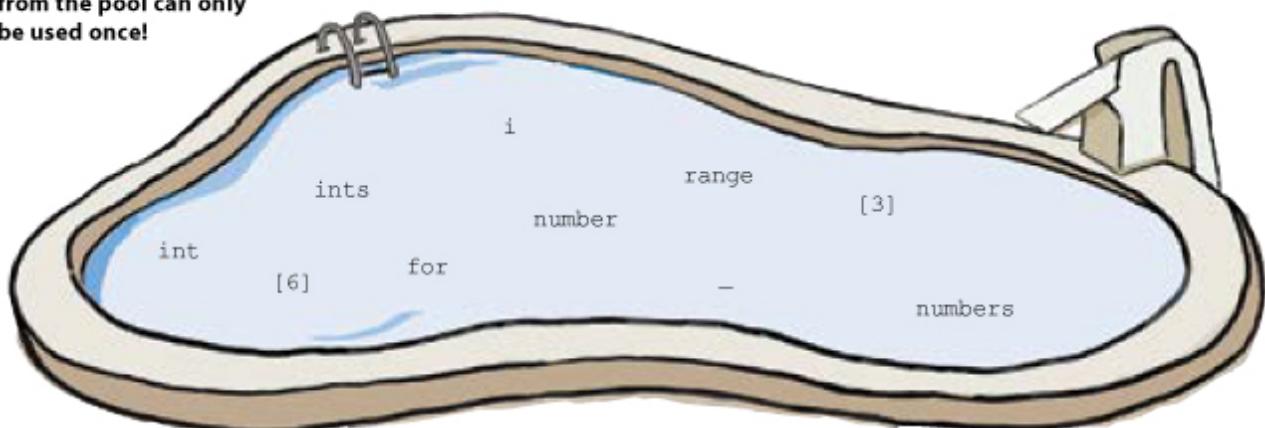
```
package main

import "fmt"

func main() {
    _____ := _____int{3, 16, -2, 10, 23, 12}
    for i, _____ := _____ numbers {
        if number >= 10 && number <= 20 {
            fmt.Println(_____, number)
        }
    }
}
```

Output
1 16
3 10
5 12

Note: each snippet from the pool can only be used once!





That's great, but your program only tells me how much to order for this week. What should I do when I have data for more weeks? I can't edit the code to change the array values; I don't even have Go installed!

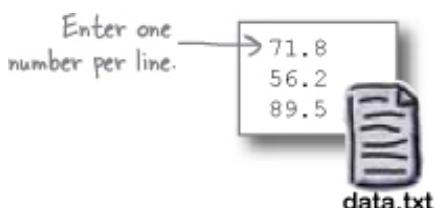
That's true, a program where users have to edit and compile the source code themselves isn't very user-friendly.

Previously, we've used the standard library's `os` and `bufio` packages to read data a line at a time from the keyboard. We can use the same packages to read data a line at a time from text files. Let's go on a brief detour to learn how to do that.

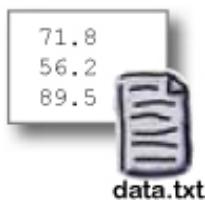
Then, we'll come back and update the `average` program to read its numbers in from a text file.

In your favorite text editor, create a new file named `data.txt`. Save it somewhere *outside* of your Go workspace directory for now.

Within the file, enter our three floating-point sample values, one number per line.



Before we can update our program to average numbers from a text file, we need to be able to read the file's contents. To start, let's write a program that only reads the file, and then we'll incorporate what we learn into our averaging program.



In the same directory as `data.txt`, create a new program named `readfile.go`. We'll just be running `readfile.go` with `go run`, rather than installing it, so it's okay to save it outside of your Go workspace directory. Save the following code in `readfile.go`. (We'll take a closer look at how this code works on the next page.)

```
package main
import (
    "bufio"
    "fmt"
    "log"
    "os"
)
func main() {
    file, err := os.Open("data.txt")
    if err != nil {
        log.Fatal(err)
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    err = file.Close()
    if err != nil {
        log.Fatal(err)
    }
}
If there was an error opening the file, report it and exit.
Loops until the end of the file is reached and scanner.Scan returns false.
If there was an error closing the file, report it and exit.
If there was an error scanning the file, report it and exit.
```

Annotations on the code:

- An arrow points to the line `file, err := os.Open("data.txt")` with the text "Open the data file for reading."
- An arrow points to the line `scanner := bufio.NewScanner(file)` with the text "Create a new Scanner for the file."
- An arrow points to the line `for scanner.Scan()` with the text "Read a line from the file."
- An arrow points to the line `fmt.Println(scanner.Text())` with the text "Print the line."
- An arrow points to the line `err = file.Close()` with the text "Close the file, to free resources."

Then, from your terminal, change to the directory where you saved the two files, and run `go run readfile.go`. The program will read the contents of `data.txt`, and print them out.

Change to the directory you saved "data.txt" and "readfile.go" in.
Run "readfile.go".
The contents of "data.txt" will be printed.

```
Shell Edit View Window Help
$ cd /Users/jay/code
$ go run readfile.go
71.8
56.2
89.5
```



Our test `readfile.go` program is successfully reading the lines of the `data.txt` file and printing them out. Let's take a closer look at how the program works.

We start by passing a string with the name of the file we want to open to the `os.Open` function. Two values are returned from `os.Open`: a pointer to an `os.File` value representing the opened file, and an `error` value. As we've seen with so many other functions, if the `error` value is `nil` it means the file was opened successfully, but any other value means there was an error. (This could happen if the file is missing or unreadable.) If that's the case, we log the error message and exit the program.

```
file, err := os.Open("data.txt")
If there was an error opening the file, report it and exit { if err != nil {
    log.Fatal(err)
}}
```

Open the data file for reading.

Then we pass the `os.File` value to the `bufio.NewScanner` function. That will return a `bufio.Scanner` value which reads from the file.

```
scanner := bufio.NewScanner(file)
```

Create a new Scanner for the file.

The `Scan` method on `bufio.Scanner` is designed to be used as part of a `for` loop. It will read a single line of text from the file, returning `true` if it read data successfully and `false` if it did not. If `Scan` is used as the condition on a `for` loop, the loop will continue running as long as there is more data to be read. Once the end of the file is reached (or there's an error), `Scan` will return `false`, and the loop will exit.

After calling the `Scan` method on the `bufio.Scanner`, calling the `Text` method returns a string with the data that was read. For this program, we simply call `Println` within the loop to print each line out.

```
Loops until the end of the file is reached and scanner.Scan { for scanner.Scan() {
    fmt.Println(scanner.Text())
    } }
```

Read a line from the file.

Print the line.

Once the loop exits, we're done with the file. Keeping files open consumes resources from the operating system, so files should always be closed when a program is done with them. Calling the `close` method on the `os.File` will accomplish this. Like the `open` function, the `close` method returns an `error` value, which will be `nil` unless there was a problem. (Unlike `open`, `close` returns only a *single* value, as there is no useful value for it to return other than the error.)

```
err = file.Close() ← Close the file, to free resources.  
If there was an error closing {  
    if err != nil {  
        log.Fatal(err)  
    }  
    the file, report it and exit }
```

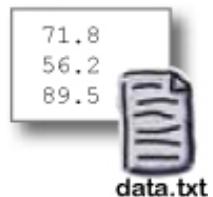
It's also possible that the `bufio.Scanner` encountered an error while scanning through the file. If it did, calling the `Err` method on the scanner will return that error, which we log before exiting.

```
If there was an error scanning {  
    if scanner.Err() != nil {  
        log.Fatal(scanner.Err())  
    }  
    the file, report it and exit }
```



READING A TEXT FILE INTO AN ARRAY

Our `readfile.go` program worked great — we were able to read the lines from of our `data.txt` file in as strings, and print them out. Now we need to convert those strings to numbers and store them in an array. Let's create a package named `datafile` that will do this for us.



In your Go workspace directory, create a `datafile` directory within the `headfirstgo` directory. Within the `datafile` directory, save a file named `floats.go`. (We name it `floats.go` because this file will contain code that reads floating-point numbers from files.)



Within `floats.go`, save the following code. A lot of this is based on code from our test `readfile.go` program; we've grayed out the parts where the code is identical. We'll explain the new code in detail on the next page.

```

// Package datafile allows reading data samples from files.
package datafile

import (
    "bufio"
    "os"
    "strconv"

Take the file name to read as an argument. // GetFloats reads a float64 from each line of a file.
func GetFloats(fileName string) ([3]float64, error) {
    var numbers [3]float64 ← Declare the array we'll be returning.
    file, err := os.Open(fileName) ← Open the provided file name.

If there was an error opening the file, return it { if err != nil {
    return numbers, err
}

i := 0 ← This variable will track which array index we should assign to.
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)

If there was an error converting the line to a number, return it { if err != nil {
    return numbers, err
}

    i += 1 ← Move to the next array index.
}
err = file.Close()
If there was an error closing the file, return it { if err != nil {
    return numbers, err
}

If there was an error scanning the file, return it { if scanner.Err() != nil {
    return numbers, scanner.Err()
}

return numbers, nil ← If we got this far, there were no errors, so return the array of numbers and a "nil" error.
}

```

The function will return an array of numbers and any error encountered.

Convert the file line string to a float64.

We want to be able to read from files other than `data.txt`, so we accept the name of the file we should open as a parameter. We set the function up to return two values, an array of `float64` values and an `error` value. Like most functions that return an error, the other return value should only be considered usable if the error value is `nil`.

Take the file name to read as an argument. func GetFloats(fileName string) ([3]float64, error) {

The function will return an array of numbers and any error encountered.

Next we declare an array of 3 `float64` values that will hold the numbers we read from the file.

`var numbers [3]float64` ← Declare the array we'll be returning.

Just like in `readfile.go`, we open the file for reading. The difference is that instead of a hard-coded string of "`data.txt`", we open whatever file name was passed to the function. If an error is encountered, we need to return an array along with the error value, so we just return the `numbers` array (even though nothing has been assigned to it yet).

`file, err := os.Open(fileName)` ← Open the provided file name.

If there was an error opening the file, return it { if err != nil {
 return numbers, err
}

We need to know which array element to assign each line to, so we create a variable to track the current index.

`i := 0` ← This variable will track which array index we should assign to.

The code to set up a `bufio.Scanner` and loop over the file's lines is identical to the code from `readfile.go`. The code within the loop is different, however: we need to call `strconv.Parsefloat` on the string read from the file to convert it to a `float64`, and assign the result to the array. If `Parsefloat` results in an error, we need to return that. And if the parsing is successful, we need to increment `i` so that the next number is assigned to the next array element.

```
numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
If there was an error converting the line to a number, return it. { if err != nil {
    return numbers, err
}
i += 1 ← Move to the next array index.
```

Convert the file line string to a float64.

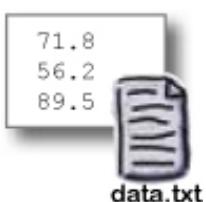
Our code to close the file and report any errors is identical to `readfile.go`, except that we return any errors instead of exiting the program directly. If there are no errors, the end of the `GetFloats` function will be reached, and the array of `float64` values will be returned along with a `nil` error.

```
If there was an error scanning the file, return it. { if scanner.Err() != nil {
    return numbers, scanner.Err()
}
return numbers, nil ← If we got this far, there were no errors, so return the array of numbers and a "nil" error.
```

UPDATING OUR "AVERAGE" PROGRAM TO READ A TEXT FILE

We're ready to replace the hard-coded array in our `average` program with an array read in from the `data.txt` file!

Writing our `datafile` package was the hard part. Here in the main program, we only need to do three things:



- Update our `import` declaration to include the `datafile` package, and remove unused

imports that are no longer used in this file.

- Replace our array of hard-coded numbers with a call to `datafile.GetFloats("data.txt")`.
- Check whether we got an error back from `GetFloats`, and log it and exit if so.

All the remaining code will be exactly the same.



```
// average calculates the average of several numbers.
package main

import (
    "fmt"
    "github.com/headfirstgo/datafile"
    "log"
)

func main() {
    numbers, err := datafile.GetFloats("data.txt")
    If there was an error, { if err != nil {
        report it and exit } log.Fatal(err)
    }
    var sum float64 = 0
    for _, number := range numbers {
        sum += number
    }
    sampleCount := float64(len(numbers))
    fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

Annotations:

- Import our package.
- Load "data.txt", parse the numbers it contains, and store the array.

We can compile the program using the same terminal command as before:

```
go install github.com/headfirstgo/average
```

Since our program imports the `datafile` package, that will automatically be compiled as well.

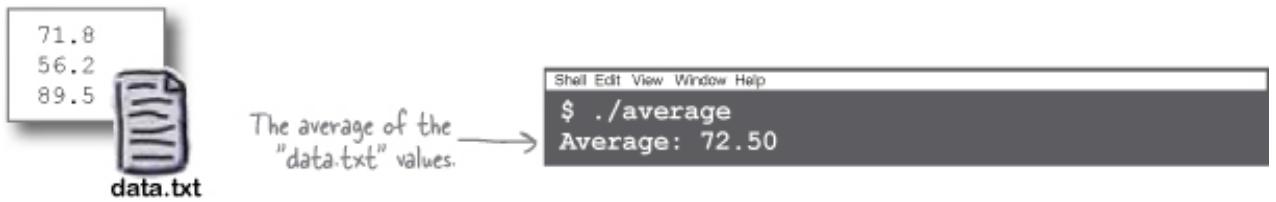
Compiles both the "average" program and the "datafile" package it depends on. → 

We'll need to move the `data.txt` file to the `bin` subdirectory of the Go workspace. That's because we'll be running the `average` executable from that directory, and it will look for `data.txt` in the same directory. Once you've moved `data.txt`, change into that `bin` subdirectory.

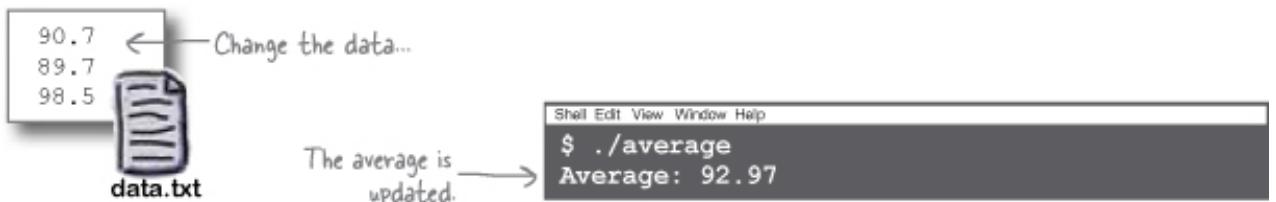
Move the "data.txt" file to the "bin" subdirectory of the workspace. (Use the appropriate command for your system, or re-save it using your text editor.) → 

Change to the "bin" subdirectory.

When we run the `average` executable, it will load the values from `data.txt` into an array, and use them to calculate an average.

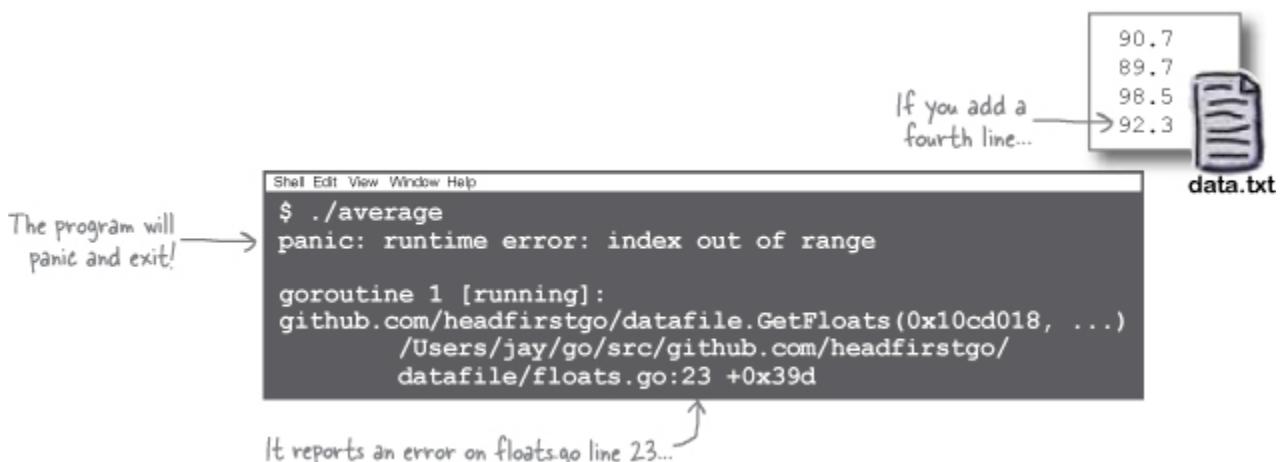


If we change the values in `data.txt`, the average will change as well.



OUR PROGRAM CAN ONLY PROCESS THREE VALUES!

But there's a problem — the `average` program only works if there are three or fewer lines in `data.txt`. If there are four or more, `average` will panic and exit when it's run!



When a Go program panics, it outputs a report with information on the line of code where the problem occurred. In this case, it looks like the problem is on line 23 of the `floats.go` file.

If we look at line 23 of `floats.go`, we'll see that it's the part of the `GetFloats` function where numbers from the file get added to the array!

```

// ...Preceding code omitted...
func GetFloats(fileName string) ([3]float64, error) {
    var numbers [3]float64
    file, err := os.Open(fileName)
    if err != nil {
        return numbers, err
    }
    i := 0
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        Here's line 23, where a number is → numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
        assigned to the array!           if err != nil {
                                         return numbers, err
        }
        i += 1
    }
    // ...Rest of GetFloats code omitted...
}

```

Remember when a mistake in a previous code sample led a program to attempt to access an eighth element of a seven-element array? That program panicked and exited, too.

The array only has 7 elements.

```

notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for i := 0; i <= 7; i++ { ← Loops up through index 7 (the
    fmt.Println(i, notes[i]) eighth element), which doesn't exist!
}

```

Access indexes 0 through b. {

Accessing index 7 causes a panic! →

0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti

panic: runtime error: index out of range

The same problem is happening in our `GetFloats` function. Because we declared that the `numbers` array holds 3 elements, that's *all* it can hold. When the fourth line of the `data.txt` file is reached, it attempts to assign to a *fourth* element of `numbers`, which results in a panic.

```

func GetFloats(fileName string) ([3]float64, error) {
    var numbers [3]float64 ← The only valid indexes are
    file, err := os.Open(fileName)   numbers[0] through numbers [2]...
    if err != nil {
        return numbers, err
    }
    i := 0
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        → numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return numbers, err
        }
        i += 1
    }
    // ...Rest of GetFloats code omitted...
}

```

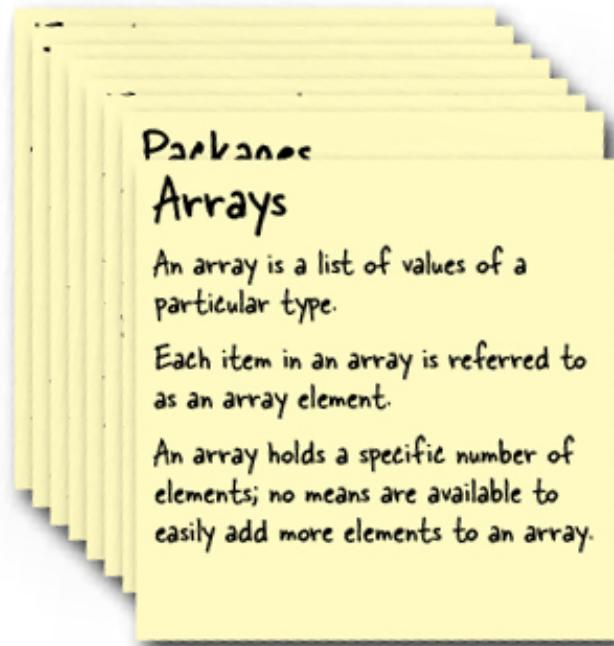
This attempts to assign to numbers [3], which causes a panic!

Go arrays are fixed in size; they can't grow or shrink. But the `data.txt` file can have as many lines as the user wants to add. It looks like we can't use an array to store the numbers from the file, after all!



YOUR GO TOOLBOX

That's it for **Chapter 5!** You've added arrays to your toolbox.



BULLET POINTS

- To declare an array variable, include the array length in square brackets and the type of elements it will hold:

```
var myarray [3]int
```

- To assign or access an element of an array, provide its index in square brackets. Indexes start at 0, so the first element of `myarray` is `myarray[0]`.

- As with variables, the default value for all array elements is the zero value for that element's type.

- You can set element values at the time an array is created using an array literal:

```
[3]int{4, 9, 6}
```

- If you store an index in a variable that is not valid for an array, and then try to access an array element using that variable as an index, you will get a panic — a runtime error.

- You can get the number of elements in an array with the built-in `len` function.

- You can conveniently process all the elements of an array using the special `for ... range` loop syntax, which loops through each element and assigns its index and value to variables you provide.

- When using a `for ... range` loop, you can ignore either the index or value for each element by assigning it to the `_` blank identifier.

- The `os.Open` function opens a file. It returns a pointer to an `os.File` value representing that opened file.

- Passing an `os.File` value to `bufio.NewScanner` returns a `bufio.Scanner` value whose `Scan` and `Text` methods can be used to read a line at a time from the file as strings.



Below is a program that declares a couple arrays and prints out their elements. Write down what the program output would be.

```

package main

import "fmt"

func main() {
    var numbers [3]int
    numbers[0] = 42
    numbers[2] = 108
    var letters = [3]string{"a", "b", "c"}

    Output

    fmt.Println(numbers[0]) 42
    fmt.Println(numbers[1]) 0
    fmt.Println(numbers[2]) 108
    fmt.Println(letters[2]) c
    fmt.Println(letters[0]) a
    fmt.Println(letters[1]) b
}

```

POOL PUZZLE SOLUTION

```

package main

import "fmt"

func main() {
    numbers := [6]int{3, 16, -2, 10, 23, 12}
    for i, number := range numbers {
        if number >= 10 && number <= 20 {
            fmt.Println(i, number)
        }
    }
}

```

Output

1	16
3	10
5	12

6 slices

Appending Issue



We've learned we can't add more elements to an array. That's a real problem for our program, because we don't know in advance how many pieces of data our file contains. But that's where Go **slices** come in. Slices are a collection type that can grow to hold additional items — just the thing to fix our current program! We'll also see how slices can provide an easier way for users to provide data to *all* your programs, and how they can help you write functions that are more convenient to call.

SLICES

There actually *is* a Go data structure that we can add more values to — it's called a

slice. Like arrays, slices are made up of multiple elements, all of the same type. *Unlike* arrays, functions are available for slices that allow us to add extra elements onto the end.

To declare a variable that holds a slice, you type an empty pair of square brackets, followed by the type of elements the slice will hold.

The diagram shows the declaration of a slice variable 'myslice'. It consists of the keyword 'var' followed by the variable name 'myslice', then an empty pair of square brackets '[]', and finally the type 'string'. Two arrows point from handwritten text to specific parts of the code: one arrow points to the empty brackets with the label 'Empty pair of square brackets.', and another arrow points to the word 'string' with the label 'Type of elements slice will hold.'

```
var myslice []string
```

This is just like the syntax for declaring an array variable, except that you don't specify the size.

The diagram compares two declarations: 'myarray' and 'myslice'. 'myarray' is shown with the value '[5]int', with an arrow pointing to the '[5]' labeled 'An array - note the size.'. 'myslice' is shown with the value '[]int', with an arrow pointing to it labeled 'A slice - no size specified.'

```
var myarray [5]int
var myslice []int
```

Declaring a slice variable doesn't automatically create a slice. For that, you can call the built-in `make` function. You pass `make` the type of the slice you want to create (which should be the same as the type of the variable you're going to assign it to), and the length of slice it should create.

The diagram shows the creation of a slice variable 'notes'. It starts with 'var notes []string', then 'notes = make([]string, 7)'. An arrow points from the handwritten note 'Declare a slice variable.' to the first line. Another arrow points from the handwritten note 'Create a slice with 7 strings.' to the second line.

```
var notes []string
notes = make([]string, 7)
```

Once the slice is created, you assign and retrieve its elements using the same syntax you would for an array.

The diagram shows assignments and prints for the slice 'notes'. It includes the following code:
`notes[0] = "do"`
`notes[1] = "re"`
`notes[2] = "mi"`
`fmt.Println(notes[0])`
`fmt.Println(notes[1])`
Below the code, there is a small box containing 'do' and 're'. Arrows point from the handwritten notes to the code:

- 'Assign a value to the first element.' points to 'notes[0] = "do"'.
- 'Assign a value to the second element.' points to 'notes[1] = "re"'.
- 'Assign a value to the third element.' points to 'notes[2] = "mi"'.
- 'Print the first element.' points to 'fmt.Println(notes[0])'.
- 'Print the second element.' points to 'fmt.Println(notes[1])'.

```
notes[0] = "do"
notes[1] = "re"
notes[2] = "mi"
fmt.Println(notes[0])
fmt.Println(notes[1])
```

You don't have to declare the variable and create the slice in separate steps; using `make` with a short variable declaration will infer the variable's type for you.

```

Create a slice with 5 integers,
and set up a variable to hold it.

primes := make([]int, 5)
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0])

```

2

The built-in `len` function works the same way with slices as it does with arrays. Just pass `len` a slice, and its length will be returned as an integer.

```

notes := make([]string, 7)
primes := make([]int, 5)
fmt.Println(len(notes))
fmt.Println(len(primes))

```

7
5

Both `for` and `for ... range` loops work just the same with slices as they do with arrays, too:

```

letters := []string{"a", "b", "c"}
for i := 0; i < len(letters); i++ {
    fmt.Println(letters[i])
}
for _, letter := range letters {
    fmt.Println(letter)
}

```

a
b
c
a
b
c

SLICE LITERALS

Just like with arrays, if you know in advance what values a slice will start with, you can initialize the slice with those values using a **slice literal**. A slice literal looks a lot like an array literal, but where an array literal has the length of the array in square brackets, a slice literal's square brackets are empty. The empty brackets are then followed by the type of elements the slice will hold, and a list in curly braces of the initial values each element will have.

There's no need to call the `make` function; using a slice literal in your code will create the slice *and* pre-populate it.

Empty pair of square brackets.
Type of elements slice will hold.
Comma-separated list of slice values.

`[]int{9, 18, 27}`

These examples are just like the previous ones we showed, except that instead of assigning values to the slice elements one by one, the entire slice is initialized using slice literals.

```

notes := []string("do", "re", "mi", "fa", "so", "la", "ti") ← Assign values using a slice literal.
fmt.Println(notes[3], notes[6], notes[0])
primes := []int{2, 3, 5, 7, 11} ← Assign values using a slice literal.
fmt.Println(primes[0], primes[2], primes[4])

```

fa	ti	do
2	5	11



POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in this code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a program that will run and produce the output shown..

```

package main

import "fmt"

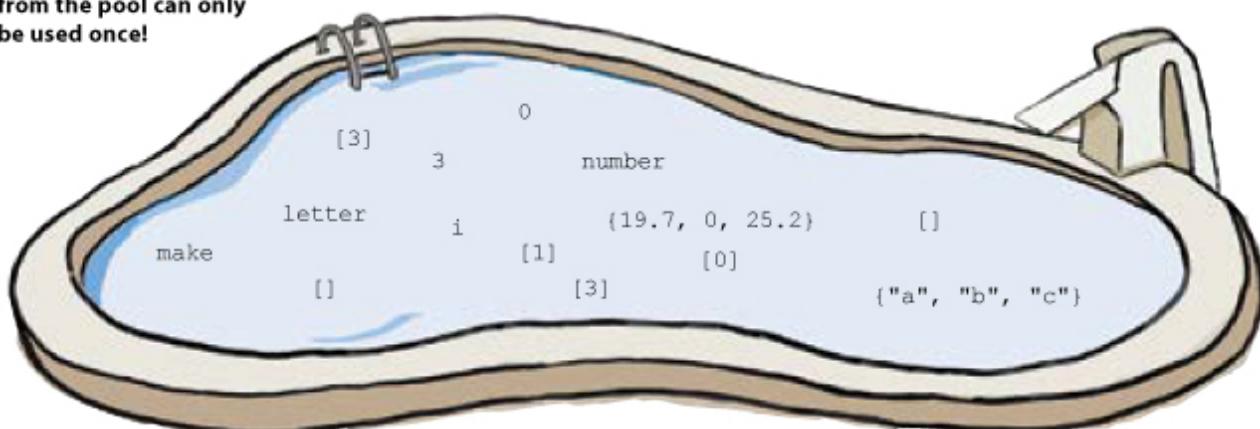
func main() {
    numbers := _____(_____float64, ____)
    numbers____ = 19.7
    numbers[2] = 25.2
    for ___, _____ := range numbers {
        fmt.Println(i, number)
    }
    var letters = _____string_____
    for i, letter := range letters {
        fmt.Println(i, _____)
    }
}

```

Output

0	19.7
1	0
2	25.2
0	a
1	b
2	c

Note: each snippet from the pool can only be used once!



Answers on page 28.



Hold up! It looks like slices can do everything arrays can do, and you say we can add additional values to them! Why didn't you just show us slices, and skip that array nonsense?

Because slices are built on top of arrays. You can't understand how slices work without understanding arrays. Here, we'll show you why...

THE SLICE OPERATOR

Every slice is built on top of an **underlying array**. It's the underlying array that actually holds the slice's data; the slice is merely a view into some (or all) of the array's elements.

When you use the `make` function or a slice literal to create a slice, the underlying array is created for you automatically (and you can't access it, except through the slice). But you can also create the array yourself, and then create a slice based on it with the **slice operator**.

Index of array where slice should start. Index of array slice should stop before.

```
myslice := myarray[1:3]
```

The slice operator looks similar to the syntax for accessing an individual element or slice of an array, except that it has two indexes, the index of the array where the slice should start, and the index of the array that the slice should stop before.

```

Index 0 - slice will start here. Index 3 - slice will stop before here.
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice1 := underlyingArray[0:3]
fmt.Println(slice1)

```

[a b c]

Elements 0 through 2 of underlyingArray.

Notice that we emphasize that the second index is the index the slice will stop before. That is, the slice should include the elements up to, but *not* including, the second index. If you use `underlyingArray[i:j]` as a slice operator, the resulting slice will actually contain the elements `underlyingArray[i]` through `underlyingArray[j-1]`.

(We know, it's counterintuitive. But a similar notation has been used in the Python programming language for over 20 years, and it seems to work OK.)

```

Index 1 - slice will start here. Index 4 - slice will stop before here.
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
i, j := 1, 4
slice2 := underlyingArray[i:j]
fmt.Println(slice2)

```

[b c d]

Elements 1 through 3 of underlyingArray.

If you want a slice to include the last element of an underlying array, you actually specify a second index that's one *beyond* the end of the array in your slice operator.

```

Index 2 - slice will start here. There is no index 5, but slice will stop before here.
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice3 := underlyingArray[2:5]
fmt.Println(slice3)

```

[c d e]

Elements 2 through 4 of underlyingArray.

Make sure you don't go any farther than that, though, or you'll get an error:

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}  
slice3 := underlyingArray[2:6]
```

```
invalid slice index 6 (out of bounds for 5-element array)
```

The slice operator has defaults for both the start and stop indexes. If you omit the start index, a value of `0` (the first element of the array) will be used.

Index 0 – slice will start here. Index 3 – slice will stop before here.

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}  
slice4 := underlyingArray[:3]  
fmt.Println(slice4)
```

[a b c]

Elements 0 through 2 of underlyingArray.

And if you omit the stop index, everything from the start index to the end of the underlying array will be included in the resulting slice.

Index 1 – slice will start here. End of the array – slice will end here.

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}  
slice5 := underlyingArray[1:]  
fmt.Println(slice5)
```

[b c d e]

Elements 1 through the end of underlyingArray.

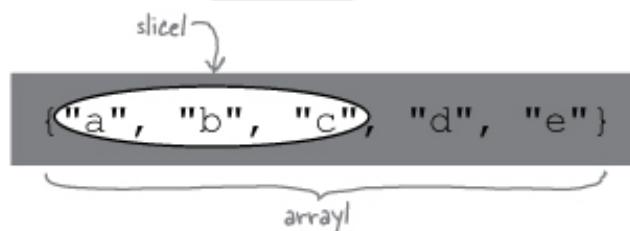
UNDERLYING ARRAYS

As we mentioned, a slice doesn't hold any data itself; it's merely a view into the elements of an underlying array. You can think of a slice as a microscope, focusing on a particular portion of the contents of a slide (the underlying array).

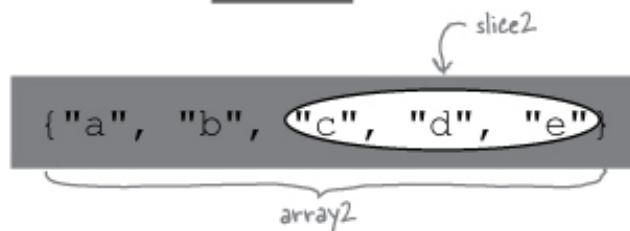
When you take a slice of an underlying array, you can only "see" the portion of the array's elements that are visible through the slice.



```
array1 := [5]string{"a", "b", "c", "d", "e"}  
slice1 := array1[0:3]  
fmt.Println(slice1)      [a b c]
```



```
array2 := [5]string{"a", "b", "c", "d", "e"}  
slice2 := array2[2:5]  
fmt.Println(slice2)      [c d e]
```

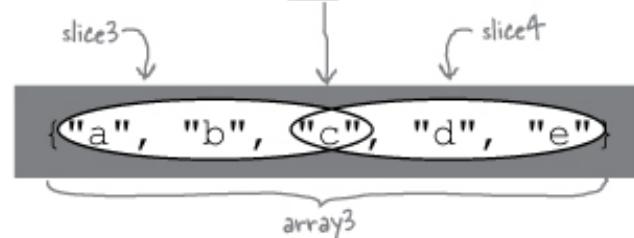


It's even possible to have multiple slices point to the same underlying array. Each slice will then be a view into its own subset of the array's elements. The slices can even overlap!

```
array3 := [5]string{"a", "b", "c", "d", "e"}  
slice3 := array3[0:3]  
slice4 := array3[2:5]  
fmt.Println(slice3, slice4)
```

[a b c] [c d e]

The array element
at index 2 appears
in both slices.

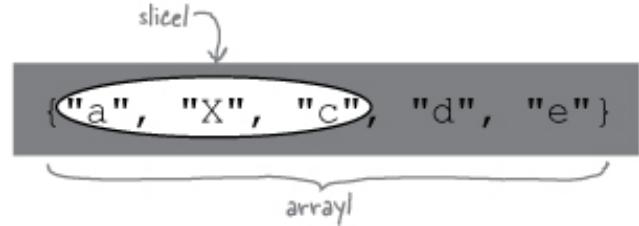


CHANGE THE UNDERLYING ARRAY, CHANGE THE SLICE

Now, here's something to be careful about: because a slice is just a view into the contents of an array, if you change the underlying array, those changes will *also* be visible within the slice!

```
array1 := [5]string{"a", "b", "c", "d", "e"}  
slice1 := array1[0:3]  
array1[1] = "X"  
fmt.Println(array1)  
fmt.Println(slice1)
```

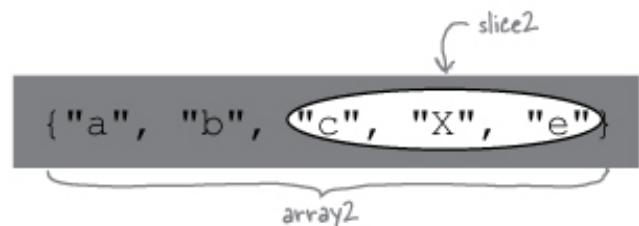
```
[a X c d e]  
[a X c]
```



Assigning a new value to a slice element will change the corresponding element in the underlying array.

```
array2 := [5]string{"a", "b", "c", "d", "e"}  
slice2 := array2[2:5]  
slice2[1] = "X"  
fmt.Println(array2)  
fmt.Println(slice2)
```

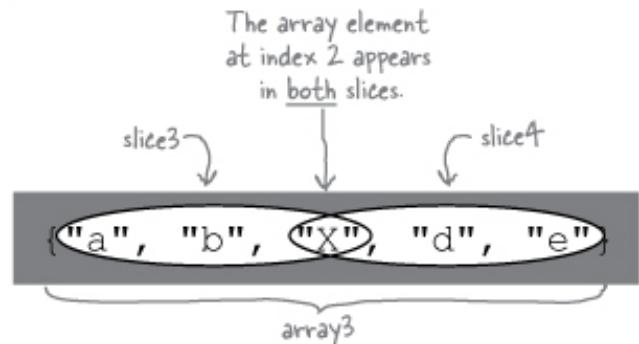
```
[a b c X e]  
[c X e]
```



If multiple slices point to the same underlying array, a change to the array's elements will be visible in *all* the slices.

```
array3 := [5]string{"a", "b", "c", "d", "e"}  
slice3 := array3[0:3]  
slice4 := array3[2:5]  
array3[2] = "X"  
fmt.Println(array3)  
fmt.Println(slice3, slice4)
```

```
[a b X d e]  
[a b X] [X d e]
```



Because of these potential issues, you may find it's generally better to create slices using `make` or a slice literal, rather than creating an array and using a slice operator on it. With `make` and with slice literals, you never have to work with the underlying array.

ADD ONTO A SLICE WITH THE "APPEND" FUNCTION

Look, all of this info on slices is just super. Really. But I'm still stuck with a program that can only read three lines from a text file, because it uses an array. You said we can add more values onto a slice – I want to hear about that!



Go offers a built-in `append` function that takes a slice, and one or more values you want to append to the end of that slice. It returns a new, larger slice with all the same elements as the original slice, plus the new elements added onto the end.

```
Assign the return value of "append" back to the same slice variable. slice := []string{"a", "b"} ← Create a slice.  
fmt.Println(slice, len(slice))  
slice = append(slice, "c") ← Append an element to the end of the slice.  
fmt.Println(slice, len(slice))  
slice = append(slice, "d", "e") ← Append two elements to the end of the slice.  
fmt.Println(slice, len(slice))  
  
Has 1 more element, and the length is increased by 1. [a b] 2  
Has 2 more elements, and the length is increased by 2. [a b c] 3  
[a b c d e] 5
```

You don't have to keep track of what index you want to assign new values to, or anything! Just call `append` with your slice and the value(s) you want added to the end, and you'll get a new, longer slice back. It's really that easy!

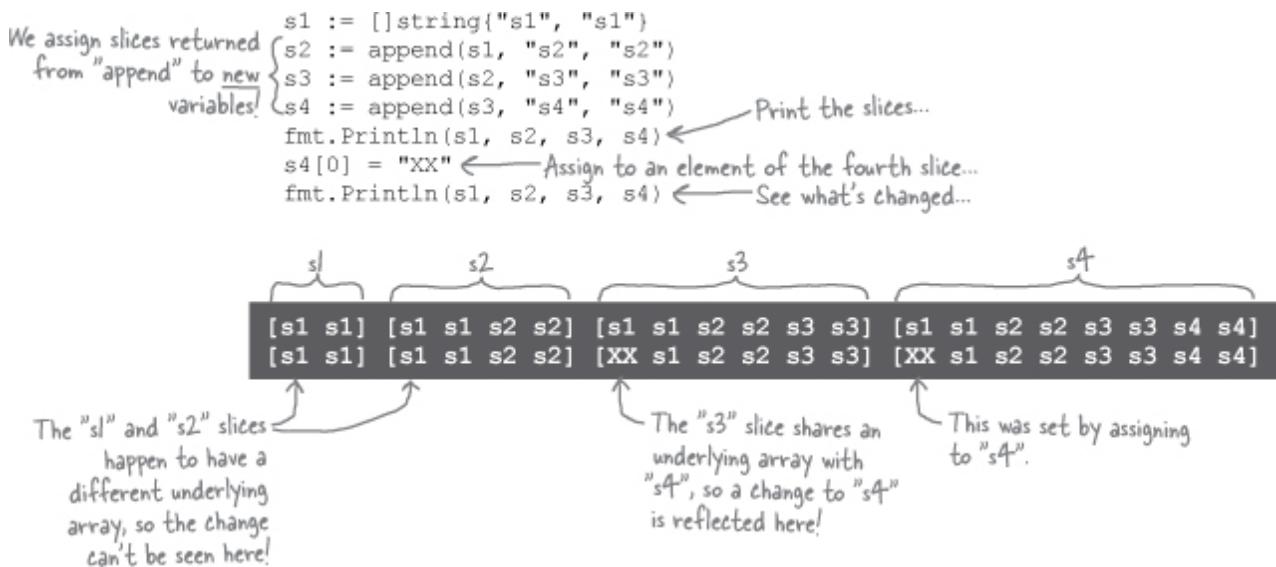
Well, with one caution...

Notice that we're making sure to assign the return value of `append` back to the *same* slice variable we passed to `append`. This is to avoid some potentially inconsistent behavior in the slices returned from `append`.

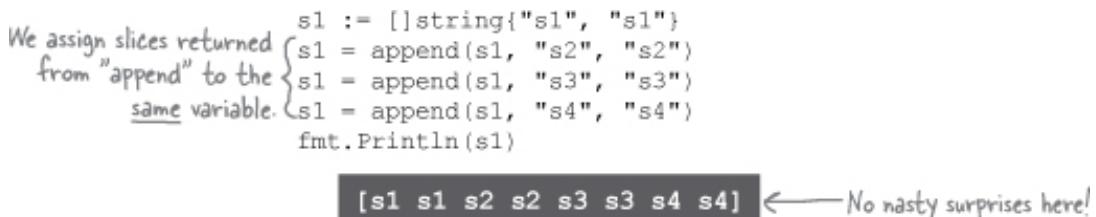
A slice's underlying array can't grow in size. If there isn't room to add elements to the array, all its elements will be copied to a new, larger array, and the old array will be discarded. But since all this happens behind the scenes in the `append` function, there's no easy way to tell whether the slice returned from `append` has the *same* underlying array as

the slice you passed in, or a *different* underlying array. If you keep both slices, this can lead to some unpredictable behavior.

Below, for example, we have four slices, the last three created by calls to `append`. Here we are *not* following the convention of assigning `append`'s return value back to the same variable. When we assign a value to an element of the `s4` slice, we can see the change reflected in `s3`, because `s4` and `s3` happen to share the same underlying array. But the change is *not* reflected in `s2` or `s1`, because they have a *different* underlying array.

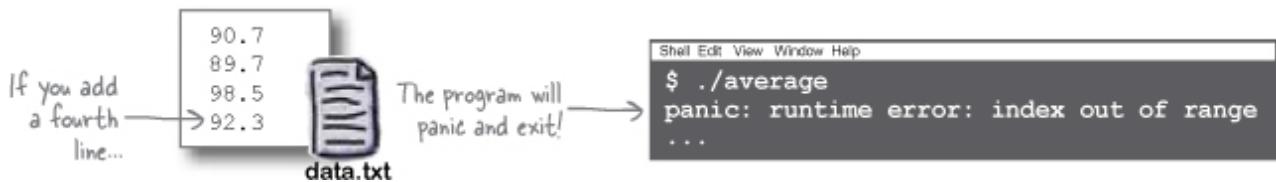


So when calling `append`, it's conventional to just assign the return value back to the same slice variable you passed to `append`. You don't need to worry about whether two slices have the same underlying array if you're only storing one slice!



READING ADDITIONAL FILE LINES USING SLICES AND "APPEND"

Now that we know about slices and the `append` function, we can finally fix our `average` program! Remember, `average` was failing as soon as we added a fourth line to the `data.txt` file it reads from:



We traced the problem back to our `datafile` package, which stores the file lines in an array that can't grow beyond 3 elements:

your workspace > src > github.com > headfirstgo > datafile > floats.go

```
// Package datafile allows reading data samples from files.
package datafile

import (
    "bufio"
    "os"
    "strconv"
)

// GetFloats reads a float64 from each line of a file.
func GetFloats(fileName string) ([3]float64, error) {
    var numbers [3]float64 ← The only valid indexes are
    file, err := os.Open(fileName)   numbers[0] through numbers[2]...
    if err != nil {
        return numbers, err
    }
    i := 0
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        numbers[i], err = strconv.ParseFloat(scanner.Text(), 64) ← The function
        if err != nil {                                returns an array of
            return numbers, err                         float64 values.
        }
        i += 1
    }
    err = file.Close()
    if err != nil {
        return numbers, err
    }
    if scanner.Err() != nil {
        return numbers, scanner.Err()
    }
    return numbers, nil
}
```

This attempts to assign to numbers [3], which causes a panic!

Most of our work with slices has just centered around understanding them. Now that we do, updating the `GetFloats` method to use a slice instead of an array doesn't involve much effort.

First, we update the method declaration to return a slice of `float64` values instead of an array. Previously, we stored the array in a variable called `numbers`; we'll just use that same variable name to hold the slice. We create the slice using an empty slice literal (although we could have instead called `make([]float64, 0)` if we wanted). After that, much of the rest of the code in `GetFloat` can remain the same — the slice is basically a drop-in replacement for the array.

The big difference is that instead of assigning values read from the file to a specific

array index, we can just call `append` to extend the slice and add new values. That means we can get rid of the code to create and update the `i` variable that tracks the index. We assign the `float64` value returned from `ParseFloat` to a new temporary variable, just to hold it while we check for any errors in parsing. Then we pass the `numbers` slice and the new value from the file to `append`, making sure to assign the return value back to the `numbers` variable.

```

// ...Preceding code omitted...
func GetFloats(fileName string) ([]float64, error) {
    numbers := []float64{} ← Use an empty slice literal to
                           file, err := os.Open(fileName) create the slice.
    if err != nil {
        return numbers, err
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        number, err := strconv.ParseFloat(scanner.Text(), 64) ← Convert the string to a float64
        if err != nil {
            return numbers, err
        }
        numbers = append(numbers, number) ← Append the new number
                                         to the slice.
    }
    err = file.Close() ← No changes needed here, either.
    if err != nil {
        return numbers, err
    }
    if scanner.Err() != nil {
        return numbers, scanner.Err()
    }
    return numbers, nil
}

```

All the rest of the code in the `GetFloats` function can remain as it is.

TRYING OUR IMPROVED PROGRAM

The slice returned from the `GetFloats` function works like a drop-in replacement for an array in our main `average` program, too. In fact, we don't have to make *any* changes to the main program!

Because we used a `:=` short variable declaration to assign the `GetFloats` return value to a variable, the `numbers` variable automatically switches from an inferred type of `[3]float64` (an array) to a type of `[]float64` (a slice). And because the `for ... range` loop and the `len` functions work the same way with a slice as they do with an array, no changes are needed to that code, either!

```
// average calculates the average of several numbers.
package main

import (
    "fmt"
    "github.com/headfirstgo/datafile"
    "log"
)

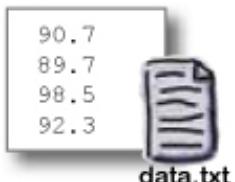
func main() {
    numbers, err := datafile.GetFloats("data.txt")
    if err != nil {
        log.Fatal(err)
    }
    var sum float64 = 0
    for _, number := range numbers { No changes needed anywhere!
        sum += number
    }
    sampleCount := float64(len(numbers))
    fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

Automatically gets a type of []float64 instead of [3]float64.

Works the same with a slice as it did with an array.

for _, number := range numbers { Also works the same with a slice.

That means we're ready to try the changes out! Ensure the `data.txt` file is still saved in your Go workspace's `bin` subdirectory, and then compile and run the code using the same commands as before. It will read all the lines of `data.txt`, and display their average. Then try updating `data.txt` to have more lines, or fewer; it will still work regardless!



Compiles the updated "datafile" package, because "average" depends on it →

Change to the "bin" subdirectory →

Run the program →

The average of the numbers from all 4 lines of the file!

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
Average: 92.80
```



Below is a program that takes a slice of an array and then appends elements to the slice. Write down what the program output would be.

```
package main

import "fmt"

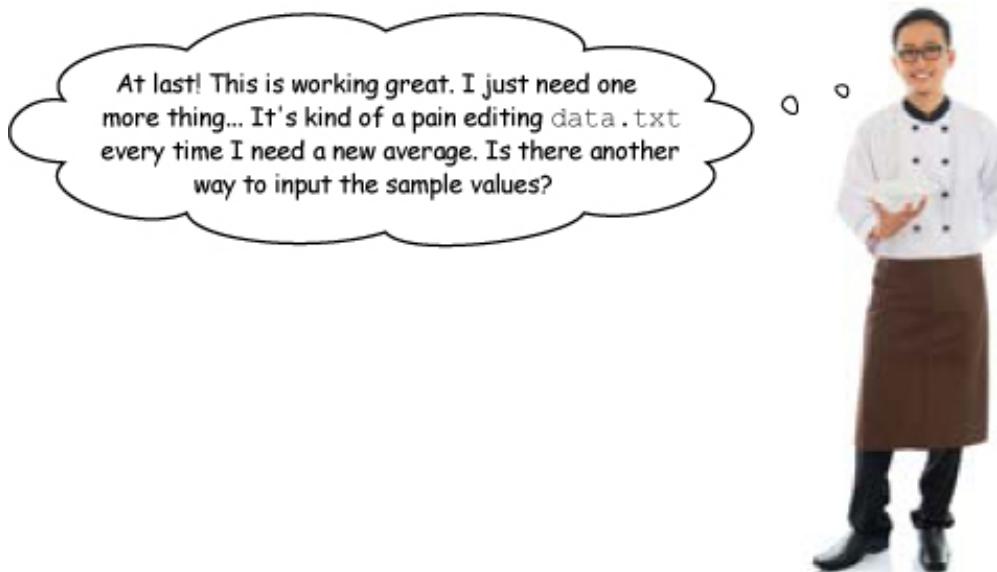
func main() {
    array := [5]string{"a", "b", "c", "d", "e"}
    slice := array[1:3]
    slice = append(slice, "x")
    slice = append(slice, "y", "z")
    for _, letter := range slice {
        fmt.Println(letter)
    }
}
```

Output:

We've provided
more blanks
than you
actually need.
How many
more? That's
up to you to
figure out!

COMMAND-LINE ARGUMENTS

Answers on page 28.



There is an alternative – users could pass the values to the program as command-line arguments.

Just as you can control the behavior of many Go functions by passing them arguments, you can pass arguments to many programs you run from the terminal or command prompt. This is known as a program's *command-line interface*.

You've already seen command-line arguments used in this very book. When we run the `cd` ("change directory") command, we pass it the name of the directory we want to change to as an argument. When we run the `go` command, we often pass it multiple arguments: the subcommand (`run`, `install`, etc.) we want to use, and the name of the file or package we want the subcommand to work on.

```

cd /Users/jay/go/bin
go install github.com/headfirstgo/average

```

GETTING COMMAND-LINE ARGUMENTS FROM THE OS.ARGS SLICE

Let's set up a new version of the `average` program, called `average2`, that takes the values to average as command line arguments.

The `os` package has a package variable, `os.Args`, that gets set to a slice of strings representing the command-line arguments the currently-running program was executed with. We'll start by simply printing the `os.Args` slice to see what it contains.

Create a new `average2` directory alongside the `average` directory in your workspace, and save a `command.go` file within it.



Then, save the following code in `command.go`. It simply imports the `fmt` and `os` packages, and passes the `os.Args` slice to `fmt.Println`.

```
// average2 calculates the average of several numbers.
package main
```

```
import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Args)
}
```

Let's try it out. From your terminal or command prompt, run this command to compile and install the program:

```
go install github.com/headfirstgo/average2
```

That will install an executable file named `average2` (or `average2.exe` on Windows) to your Go workspace's `bin` subdirectory. Use the `cd` command to change to `bin`, and run `average2`, but don't hit the Enter key just yet. Following the program name, type a space, and then one or more arguments, separated by spaces. *Then* hit Enter. The program will run, and print the value of `os.Args`.

Re-run `average2` with different arguments, and you should see different output.

The terminal window shows the following session:

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
[./average2 71.8 56.2 89.5]
$ ./average2 do re mi fa so
[./average2 do re mi fa so]
```

Annotations from the original image:

- Compile and install the executable.
- Change to the "bin" subdirectory.
- Run the executable with several arguments.
- It will print the value of `os.Args`.
- Run `average2` with different arguments to see different results.

THE SLICE OPERATOR CAN BE USED ON OTHER SLICES

This is working pretty well, but there's one problem: The name of the executable is being included as the first element of `os.Args`.

That should be easy to remove, though. Remember how we used the slice operator to get a slice that included everything but the first element of an array?

The terminal window shows the command:

```
$ ./average2 71.8 56.2 89.5
[./average2 71.8 56.2 89.5]
```

Annotation: The first element is the name of the program.

The code shown is:

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice5 := underlyingArray[1:]
fmt.Println(slice5)
```

Annotation: Index 1 - slice will start here.

Annotation: End of the array - slice will end here.

Annotation: Elements 1 through the end of underlyingArray.

The output is displayed in a box:

```
[b c d e]
```

The slice operator can be used on slices just like it can on arrays. If we use a slice operator of `[1:]` on `os.Args`, it will give us a new slice that omits the first element (whose index is 0), and includes the second element (index 1) through the end of the slice.

```
// average2 calculates the average of several numbers.
package main

import (
    "fmt"
    "os"
)
func main() {
    fmt.Println(os.Args[1:])
}
```

Get a new slice that includes the second element
(index 1) through the end of os.Args.

If we recompile and re-run `average2`, this time we'll see that the output includes only the actual command-line arguments.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ ./average2 71.8 56.2 89.5
Omits the executable name. → [71.8 56.2 89.5]
$ ./average2 do re mi fa so
Omits the executable name. → [do re mi fa so]
```

UPDATING OUR PROGRAM TO USE COMMAND-LINE ARGUMENTS

Now that we're able to get the command-line arguments as a slice of strings, let's update the `average2` program to convert the arguments to actual numbers, and calculate their average. We'll mostly be able to re-use the concepts we learned about in our original `average` program and the `datafile` package.

We use the slice operator on `os.Args` to omit the program name, and assign the resulting slice to an `arguments` variable. We set up a `sum` variable that will hold the total of all the numbers we're given. Then we use a `for ... range` loop to process the elements of the `arguments` slice (using the `_` blank identifier to ignore the element index). We use `strconv.ParseFloat` to convert the argument string to a `float64`. If we get an error, we log it and exit, but otherwise we add the current number to `sum`.

When we've looped through all the arguments, we use `len(arguments)` to determine how many data samples we're averaging. We then divide `sum` by this sample count to get the average.

```
// average2 calculates the average of several numbers.
package main

import (
    "fmt"
    "log" ← Import the "log" and
    "os" ← "strconv" packages.
    "strconv"
)

func main() {
    arguments := os.Args[1:] ← Get a slice of strings with all but
                                the first element of os.Args.

    var sum float64 = 0 ← Set up a variable to hold the sum of the numbers.
    for _, argument := range arguments { ← Process each command line argument.
        number, err := strconv.ParseFloat(argument, 64)
        If there was an error { if err != nil {
            converting the string, log log.Fatal(err)
            it and exit. } } ← Convert the string to a float64.

        sum += number ← Add the number to
                        the total.
    }
    sampleCount := float64(len(arguments)) ← The length of the arguments slice can
                                              be used as the number of samples.
    fmt.Printf("Average: %0.2f\n", sum/sampleCount) ← Calculate the average
}                                              and print it.
```

With these changes saved, we can recompile and re-run the program. It will take the numbers you provide as arguments, and average them. Give as few or as many arguments as you like; it will still work!

Run the executable with several arguments.

It will print the average.

You can use any number of arguments you like.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

VARIADIC FUNCTIONS

Now that we know about slices, we can cover a feature of Go that we haven't talked about so far. Have you noticed that some function calls can take as few, or as many, arguments as needed? Look at `fmt.Println` or `append`, for example:

```
fmt.Println(1) ← "Println" can take one argument...
                  ...Or five!
fmt.Println(1, 2, 3, 4, 5) ←
letters := []string{"a"} ← "append" can take two arguments...
letters = append(letters, "b") ←
letters = append(letters, "c", "d", "e", "f", "g") ← ...Or six!
```

Don't try doing this with just any function, though! With all the functions we've defined so far, there had to be an *exact* match between the number of parameters in the function definition and the number of arguments in the function call. Any difference

would result in a compile error.

```
func twoInts(first int, second int) { ← If two parameters are expected...
    fmt.Println(first, second)
}
```

```
func main() {
    twoInts(1) ← Then we can't pass just one...
    twoInts(1, 2, 3) ← ...And we can't pass three.
}
```

```
tmp/sandbox815038307/main.go:10:9: not enough arguments in call to twoInts
    have (number)
    want (int, int)
tmp/sandbox815038307/main.go:11:9: too many arguments in call to twoInts
    have (number, number, number)
    want (int, int)
```

So how do `Println` and `append` do it? They're declared as variadic functions. A **variadic function** is one that can be called with a *varying* number of arguments. To make a function variadic, use an ellipsis (...) before the type of the last (or only) function parameter in the function declaration.

```
func myFunc(param1 int, param2 ...string) {
    // function code here
}
```

The last parameter of a variadic function receives the variadic arguments as a slice, which the function can then process like any other slice.

Here's a variadic version of the `twoInts` function, that works just fine with any number of arguments:

```
func severalInts(numbers ...int) {
    fmt.Println(numbers)
}

func main() {
    severalInts(1)
    severalInts(1, 2, 3)      [1]           [1 2 3]
}
```

Here's a similar function that works with strings. Notice that if we provide no variadic arguments, it's not an error; the function just receives an empty slice.

```

The "strings" variable
will hold a slice with the
arguments.
↓
func severalStrings(strings ...string) {
    fmt.Println(strings)
}

func main() {
    severalStrings("a", "b")
    severalStrings("a", "b", "c", "d", "e")
    severalStrings() ←
}

```

If there are no arguments,
an empty slice is received.

[a b]
 [a b c d e]
 []

A function can take one or more non-variadic arguments as well. Although a function caller can omit variadic arguments (resulting in an empty slice), non-variadic arguments are always required; it's a compile error to omit those. Only the *last* parameter in a function definition can be variadic; you can't place it in front of required parameters.

An int argument will be required first.

A boolean argument will be required second.

Any remaining arguments must be strings and will be stored as a slice here.

```

func mix(num int, flag bool, strings ...string) {
    fmt.Println(num, flag, strings)
}

func main() {
    mix(1, true, "a", "b")
    mix(2, false, "a", "b", "c", "d")
}

```

1 true [a b]
 2 false [a b c d]

USING VARIADIC FUNCTIONS

Here's a `maximum` function that takes any number of `float64` arguments and returns the greatest value out of all of them. The arguments to `maximum` are stored in a slice in the `numbers` parameter. To start, we set the current maximum value to `-Inf`, a special value representing negative infinity, obtained by calling `math.Inf`. (We could start with a current maximum of `0`, but this way `maximum` will work with negative numbers.) Then we use `for ... range` to process each argument in the `numbers` slice, comparing it to the current maximum, and setting it as the new maximum if it's greater. Whatever maximum remains after processing all the arguments is the one we return.

```

package main

import (
    "fmt"
    "math"
)

func maximum(numbers ...float64) float64 {
    max := math.Inf(-1) ← Start with a very low value.
    Process { for _, number := range numbers {
        each variadic argument } if number > max {
            max = number } return max
    }
}

func main() {
    fmt.Println(maximum(71.8, 56.2, 89.5))
    fmt.Println(maximum(90.7, 89.7, 98.5, 92.3))
}

```

89.5
98.5

Here's an `inRange` function that takes a minimum value, a maximum value, and any number of additional `float64` arguments. It will discard any argument that is below the given minimum or above the given maximum, returning a slice containing only the arguments that were in the specified range.

```

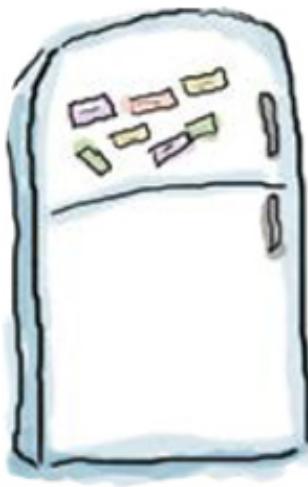
package main      The minimum value in the range.
import "fmt"      The maximum value in the range.
                  Any number of additional float64 arguments.

func inRange(min float64, max float64, numbers ...float64) []float64 {
    result := []float64{} ← This slice will hold arguments that were within range.
    Process { for _, number := range numbers {
        each variadic argument } if number >= min && number <= max { ← If this argument isn't below the minimum or above the maximum...
            result = append(result, number) ← Add it to the slice to be returned.
        }
    return result
}

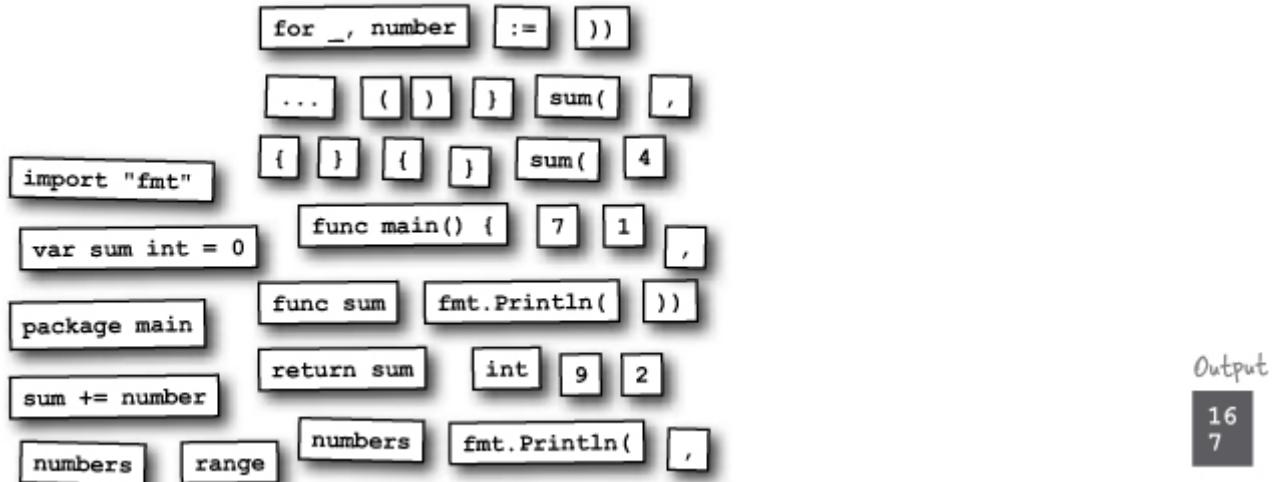
func main() {      Find arguments >= 1 and <= 100.
    fmt.Println(inRange(1, 100, -12.5, 3.2, 0, 50, 103.5)) [3.2 50]
    fmt.Println(inRange(-10, 10, 4.1, 12, -12, -5.2)) [4.1 -5.2]
}

```

CODE MAGNETS



A Go program that defines and uses a variadic function is scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output?



USING A VARIADIC FUNCTION TO CALCULATE AVERAGES

Answers on page 29.

Let's create a variadic `average` function that can take any number of `float64` arguments and return their average. It will look much like the logic from our `average2` program. We'll set up a `sum` variable to hold the total of the argument values. Then we'll loop through the range of arguments, adding each one to the value in `sum`. Finally, we'll divide `sum` by the number of arguments (converted to a `float64`) to get the average. The result is a function that can average as many (or as few) numbers as we want.

```

package main
import "fmt"

func average(numbers ...float64) float64 {
    var sum float64 = 0 ← Set up a variable to hold the sum of the arguments.
    Process each for _, number := range numbers {
        variadic { sum += number ← Add the argument value to the total.
        argument. } }
    return sum / float64(len(numbers)) ← Divide the total by the number
}                                         of arguments to get the average.

func main() {
    fmt.Println(average(100, 50))
    fmt.Println(average(90.7, 89.7, 98.5, 92.3)) 75
}                                         92.8

```

PASSING SLICES TO VARIADIC FUNCTIONS

Our new `average` variadic function works so well, we should try updating our `average2` program to make use of it. We can paste the `average` function into our `average2` code as-is.

In the `main` function, we're still going to need to convert each of the command-line arguments from a `string` to a `float64` value. We'll create a slice to hold the resulting values, and store it in a variable named `numbers`. After each command-line argument is converted, instead of using it to calculate the average directly, we'll just append it to the `numbers` slice.

We then *attempt* to pass the `numbers` slice to the `average` function. But when we go to compile the program, that results in an error...

```
// average2 calculates the average of several numbers.
package main

import (
    "fmt"
    "log"
    "os"
    "strconv"
)

func average(numbers ...float64) float64 {
    var sum float64 = 0
    for _, number := range numbers {
        sum += number
    }
    return sum / float64(len(numbers))
}

func main() {
    arguments := os.Args[1:]
    numbers := []float64{} ← This slice will hold the numbers we're averaging
    for _, argument := range arguments {
        number, err := strconv.ParseFloat(argument, 64)
        if err != nil {
            log.Fatal(err)
        }
        numbers = append(numbers, number) ← Append the converted number to the slice.
    }
    fmt.Printf("Average: %0.2f\n", average(numbers)) ← Attempt to pass the numbers
} ← to the variadic function...

```

Error → cannot use numbers (type []float64)
as type float64 in argument to average

The `average` function is expecting one or more `float64` arguments, not a *slice* of `float64` values...

So what now? Are we forced to choose between making our functions variadic and being able to pass slices to them?

Fortunately, Go provides special syntax for this situation. When calling a variadic function, simply add an ellipsis (...) following the slice you want to use in place of variadic arguments.

```
func severalInts(numbers ...int) {
    fmt.Println(numbers)
}

func mix(num int, flag bool, strings ...string) {
    fmt.Println(num, flag, strings)
}

func main() {
    intSlice := []int{1, 2, 3} ← Use an int slice
    in place of the
    severalInts(intSlice...) ← variadic arguments.
    stringSlice := []string{"a", "b", "c", "d"} ← Use a string slice
    mix(1, true, stringSlice...) ← in place of the
} ← variadic arguments.

[1 2 3]
1 true [a b c d]
```

So all we need to do is add an ellipsis following the `numbers` slice in our call to `average...`

```
func main() {
    arguments := os.Args[1:]
    numbers := []float64{}
    for _, argument := range arguments {
        number, err := strconv.ParseFloat(argument, 64)
        if err != nil {
            log.Fatal(err)
        }
        numbers = append(numbers, number)
    }
    fmt.Printf("Average: %0.2f\n", average(numbers...))
}
```

Pass the slice to the variadic function.

With that change made, we should be able to compile and run our program again. It will convert our command-line arguments to a slice of `float64` values, then pass that slice to the variadic `average` function.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

It works!

SLICES HAVE SAVED THE DAY!



```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

Working with lists of values is essential for any programming language. With arrays and slices, you can keep your data in collections of whatever size you need. And with features like `for ... range` loops, Go makes it easy to process the data in those collections, too!



YOUR GO TOOLBOX

That's it for Chapter 6! You've added slices to your toolbox.

Part Two

Arrays

An array is a list of values of a particular type.

Each item in an array is referred to as an array element.

An array holds a specific number of elements; no means are available to easily add more elements to an array.

Slices

A slice is also a list of elements of a particular type, but unlike arrays, tools are available to add or remove elements.

Slices don't hold any data themselves. A slice is merely a view into the elements of an underlying array.



BULLET POINTS

- A slice variable is declared just like an array variable, except the length is omitted:

```
var myslice []int
```

- For the most part, code for working with slices is identical to code that works with arrays. This includes: accessing elements, using zero values, passing slices to the `len` function, and `for ... range` loops.

- A slice literal looks just like an array literal, except the length is omitted: `[]int{1, 7, 10}`

- You can get a slice that contains elements `i` through `j - 1` of an array or slice using the slice operator: `s[i:j]`

- The `os.Args` package variable contains a slice of strings with the command-line arguments the current program was run with.

- A variadic function is one that can be called with a varying number of arguments.
 - To declare a variadic function, place an ellipsis (...) before the type of the last parameter in the function declaration. That parameter will then receive all the variadic arguments as a slice.
 - When calling a variadic function, you can use a slice in place of the variadic arguments by typing an ellipsis after the slice:
- ```
inRange(1, 10, myslice...)
```

## POOL PUZZLE SOLUTION

```
package main

import "fmt"

func main() {
 numbers := make([]float64, 3)
 numbers[0] = 19.7
 numbers[2] = 25.2
 for i, number := range numbers {
 fmt.Println(i, number)
 }
 var letters = []string {"a", "b", "c"}
 for i, letter := range letters {
 fmt.Println(i, letter)
 }
}
```



Below is a program that takes a slice of an array and then appends elements to the slice. Write down what the program output would be.

```

package main
import "fmt"

func main() {
 array := [5]string{"a", "b", "c", "d", "e"}
 slice := array[1:3]
 slice = append(slice, "x")
 slice = append(slice, "y", "z")
 for _, letter := range slice {
 fmt.Println(letter)
 }
}

```

Output:

b

c

x

y

z

.....

.....

.....

## CODE MAGNETS SOLUTION

package main

import "fmt"

func sum ( numbers ... int ) {

var sum int = 0

for \_, number := range numbers {

sum += number

}

return sum

}

func main() {

fmt.Println( sum( 7 , 9 ))

fmt.Println( sum( 1 , 2 , 4 ))

}

Output

16

7

## 7 maps

### *Labeling Data*



**Throwing things in piles is fine, until you need to find something again.** You've already seen how to create lists of values using *arrays* and *slices*. You've seen how to apply the same operation to *every value* in an array or slice. But what if you need to work with a *particular* value? To find it, you'll have to start at the beginning of the array or hash, and *look through Every. Single. Value.*

What if there were a kind of collection where every value had a label on it? You could quickly find just the value you needed! In this chapter, we'll look at **maps**, which do just that.

# COUNTING VOTES

A seat on the Sleepy Creek County School Board is up for grabs this year, and polls have been showing that the election is really close. Now that it's election night, the candidates are excitedly watching the votes roll in.

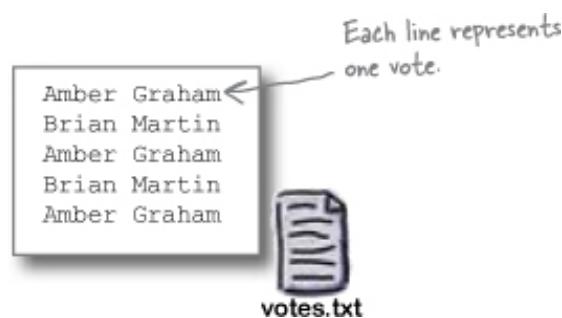
This is another example that debuted in Head First Ruby, in the hashes chapter. Ruby hashes are a lot like Go maps, so this example works great here, too!



There are two candidates on the ballot, Amber Graham and Brian Martin. Voters also have the option to "write in" a candidate's name (that is, type in a name that doesn't appear on the ballot). Those won't be as common as the main candidates, but we can expect a few such names to appear.

The electronic voting machines in use this year record the votes to text files, one vote per line. (Budgets are tight, so the city council chose the cheap voting machine vendor.)

Here's a file with all the votes for District A:



We need to process each line of the file and tally the total number of times each name

occurs. The name with the most votes will be our winner!

## READING NAMES FROM A FILE

Our first order of business is to read the contents of the `votes.txt` file. The `datafile` package from previous chapters already has a `GetFloats` function that reads each line of a file into a slice, but `GetFloats` can only read `float64` values. We're going to need a separate function that can return the file lines as a slice of `string` values.

So let's start by creating a `strings.go` file alongside the `floats.go` file in the `datafile` package directory. In that file, we'll add a `GetStrings` function. The code in `GetStrings` will look much like the code in `GetFloats` (we've grayed out the code that's identical below). But instead of converting each line to a `float64` value, `GetStrings` will just add the line directly to the slice we're returning, as a `string` value.

```
// Package datafile allows reading data samples from files.
package datafile

import (
 "bufio"
 "os"
)

// GetStrings reads a string from each line of a file.
func GetStrings(fileName string) ([]string, error) {
 Create a slice of strings. → strings := []string{}
 Identical except for the name of the slice variable. →
 Instead of converting the file line string to a float64, add it to the slice directly. →
 {
 file, err := os.Open(fileName)
 if err != nil {
 return strings, err
 }
 scanner := bufio.NewScanner(file)
 for scanner.Scan() {
 strings = append(strings, scanner.Text())
 }
 err = file.Close()
 if err != nil {
 return strings, err
 }
 if scanner.Err() != nil {
 return strings, scanner.Err()
 }
 return strings, nil
 }
}

Still part of the same package as GetFloats.
Don't import the "strconv" package; we don't need it in this file.
Return a slice of strings instead of a slice of float64 values.
```

Now let's create the program that will actually count the votes. We'll name it `count`.

Within your Go workspace, go into the `src/github.com/headfirstgo` directory and create a new directory named `count`. Then create a file named `command.go` within the `count` directory.

Before writing the full program, let's confirm that our `GetStrings` function is working. At

the top of the `main` function, we'll call `datafile.GetStrings`, passing it "votes.txt" as the name of the file to read from. We'll store the returned slice of strings in a new variable named `lines`, and any error in a variable named `err`. As usual, if `err` is not `nil`, we'll log the error and exit. Otherwise, we'll simply call `fmt.Println` to print out the contents of the `strings` slice.



```
// count tallies the number of times each line
// occurs within a file.
package main

import (
 "fmt"
 "github.com/headfirstgo/datafile"
 "log"
)

func main() {
 lines, err := datafile.GetStrings("votes.txt")
 If there was an error, { if err != nil {
 log.Fatal(err)
 }
 fmt.Println(lines) ← Print the slice of strings.
}
```

Annotations on the code:

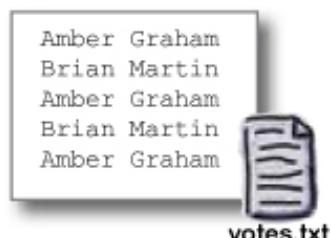
- "Import the "datafile" package, which now includes the `GetFloats` function." - points to the `import` statement for `datafile`.
- "Read the votes.txt file and return a slice of strings with every line from the file." - points to the `GetStrings` method call.
- "Print the slice of strings." - points to the `fmt.Println` call.
- "If there was an error, log it and exit" - points to the `if err != nil { log.Fatal(err) }` block.

As we've done with other programs, you can compile this program (plus any packages it depends on, `datafile` in this case) by running `go install` and providing it the package import path. If you used the directory structure shown above, that import path should be `github.com/headfirstgo/count`.

Compile the contents of the "count" directory, and install the resulting executable. → 

That will save an executable file named `count` (or `count.exe` on Windows) in the `bin` subdirectory of your Go workspace.

As with the `data.txt` file in previous chapters, we need to ensure a `votes.txt` file is saved in the current directory when we run our program. In the `bin` subdirectory of your Go workspace, save a file with the contents shown at right. In your terminal, use the `cd` command to change to that same subdirectory.



Now you should be able to run the executable by typing `./count` (or `count.exe` on Windows). It should read every line of `votes.txt` into a slice of strings, then print that slice out.

Change to the "bin" directory  
within your workspace. →  
Run the executable. →

```
Shell Edit View Window Help
$ cd /Users/jay/go/bin
$./count
[Amber Graham Brian Martin Amber Graham Brian Martin
 Brian Martin]
$
```

## COUNTING NAMES THE HARD WAY, WITH SLICES

Reading a slice of names from the file didn't require learning anything new. But now comes the challenge: how do we count the number of times each name occurs? We'll show you two ways, first with slices, and then with a new data structure, *maps*.

For our first solution, we'll create two slices, each with the same number of elements, in a specific order. The first slice would hold the names we found in the file, with each name occurring once. We could call that one `names`. The second slice, `counts`, would hold the number of times each name was found in the file. The element `counts[0]` would hold the count for `names[0]`, `counts[1]` would hold the count for `names[1]`, and so on.

| Index | names          | Index | counts |
|-------|----------------|-------|--------|
| 0     | "Amber Graham" | 0     | 3      |
| 1     | "Brian Martin" | 1     | 2      |
| 2     | "Carlos Diaz"  | 2     | 1      |
| 3     | ...            | 3     | ...    |

Let's update the `count` program to actually count the number of times each name occurs in the file. We'll try this plan of using a `names` slice to hold each unique candidate name, and a corresponding `counts` slice to track the number of times each name occurs.

```
// ...Preceding code omitted...
func main() {
 lines, err := datafile.GetStrings("votes.txt")
 if err != nil {
 log.Fatal(err)
 }
 names := []string{} ← Create a slice to hold candidate names.
 counts := []int{} ← Create a slice to hold the number of times each name occurs.
 for _, line := range lines {
 matched := false
 for i, name := range names {
 ← Loop over each value in the names slice.
 if name == line { ← If this line matches the current name...
 counts[i] += 1 ← ...Increment the corresponding count
 matched = true ← Mark that we found a match.
 }
 }
 if matched == false { ← If no match was found...
 names = append(names, line) ← ...Add it as a new name...
 counts = append(counts, 1) ←
 And add a new count (this line will
 be the first occurrence).
 }
 }
 All done; print the results. { for i, name := range names {
 fmt.Printf("%s: %d\n", name, counts[i])
 }
 Print each element from the names slice...
 ...And the corresponding element from the counts slice.
}
```

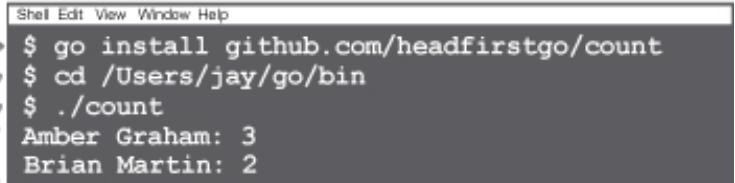
As always, we can re-compile the program with `go install`. If we run the resulting executable, it will read the `votes.txt` file and print each name it finds, along with the number of times that name occurs!

Compiles the updated "datafile" package, because "average" depends on it.

Ensure we're in the "bin" subdirectory.

Run the updated program.

Counts for each name will be printed.



```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$./count
Amber Graham: 3
Brian Martin: 2
```

Let's take a closer look at how this works...

Our `count` program uses a loop nested *inside* another loop to tally the name counts. The outer loop assigns lines of the file to the `line` variable, one at a time. `for _, line := range lines {`

Process each line of the file. { for \_, line := range lines {
 // ...
}

The *inner* loop searches each element of the `names` hash, looking for a name equal to the current line from the file.

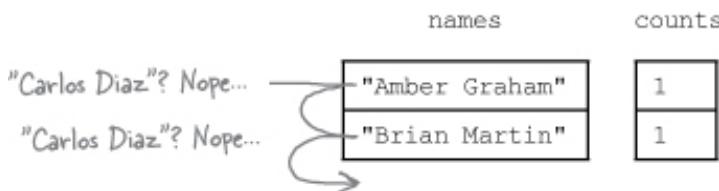
```

for i, name := range names {
 if name == line {
 counts[i] += 1
 matched = true
 }
}

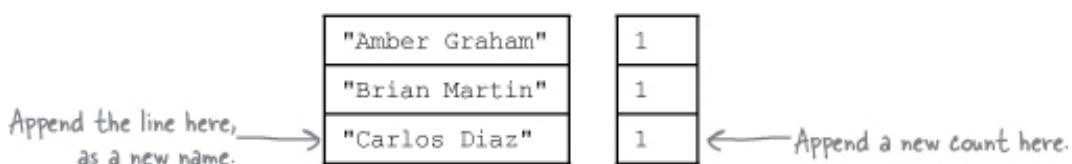
```

Search the "names" slice for one matching the current file line.

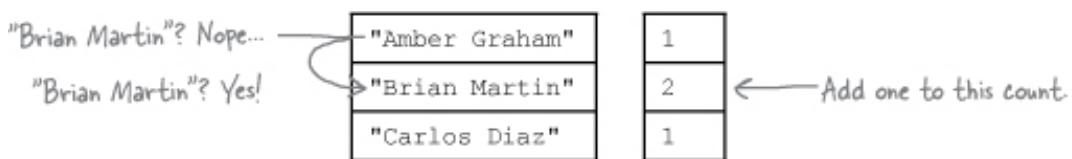
Say someone adds a write-in candidate to their ballot, causing a line from the text file to be loaded with the string "Carlos Diaz". The program will check the elements of `names`, one by one, to see if any of them equal "Carlos Diaz".



If none matches, the program will append the string "Carlos Diaz" to the `names` slice, and a corresponding count of 1 to the `counts` slice (because this line represents the first vote for "Carlos Diaz").



But suppose the next line is the string "Brian Martin"... Because that string already exists in the `names` slice, the program will find it and add 1 to the corresponding value in `counts`, instead.



## MAPS

But here's the problem with storing the names in slices: for each and every line of the file, you have to search through many (if not all) of the values in the `names` slice to compare them. That may work okay in a small district like Sleepy Creek County, but in a bigger district with lots of votes, this approach will be way too slow! "Mikey Moose"? Nope...

|                        | names          | counts |
|------------------------|----------------|--------|
| "Mikey Moose"? Nope... | "Amber Graham" | 1      |
| "Mikey Moose"? Nope... | "Brian Martin" | 1      |
| "Mikey Moose"? Nope... | >"Carlos Diaz" | 1      |

Putting data in a slice is like stacking it in a big pile; you can get particular items back out, but you'll have to search through *everything* to find them.



**Slice**

Go has another way of storing collections of data: *maps*. A **map** is a collection where a value is accessed via a *key*. Keys are an easy way to get data back out of your map. It's like having neatly-labeled file folders instead of a messy pile.



Map

Whereas arrays and slices can only use *integers* as indexes, a map can use *any* type for keys (as long as values of that type can be compared using `==`). That includes numbers, strings, and more. The values all have to all be of the same type, and the keys all have to be of the same type, but the keys don't have to be the same type as the values.

To declare a variable that holds a map, you type the `map` keyword, followed by square brackets (`[]`) containing the key type. Then, following the brackets, provide the value type.

```
var mymap map[string]float64
```

Annotations:

- "map" keyword
- Key type
- Value type

Just as with slices, declaring a map variable doesn't automatically create a map; you need to call the `make` function (the same function you can use to create slices). Instead of a slice type, you can pass `make` the type of the map you want to create (which should be the same as the type of the variable you're going to assign it to).

```
var ranks map[string]int
ranks = make(map[string]int)
```

You may find it's easier to just use a short variable declaration, though:

```
ranks := make(map[string]int)
```

The syntax to assign values to a map and get them back out again looks a lot like the syntax to assign and get values for arrays or slices. But while arrays and slices only let you use integers as element indexes, maps let you use any value of the type you chose for keys.

```
ranks["gold"] = 1
ranks["silver"] = 2
ranks["bronze"] = 3
fmt.Println(ranks["bronze"]) 3
fmt.Println(ranks["gold"]) 1
```

Here's another map with strings as keys and strings as values:

```
elements := make(map[string]string)
elements["H"] = "Hydrogen"
elements["Li"] = "Lithium"
fmt.Println(elements["Li"])
fmt.Println(elements["H"])
```

Lithium  
Hydrogen

Here's a map with integers as keys and booleans as values:

```
isPrime := make(map[int]bool)
isPrime[4] = false
isPrime[7] = true
fmt.Println(isPrime[4])
fmt.Println(isPrime[7])
```

false  
true

## MAP LITERALS

Just as with arrays and slices, if you know keys and values that you want your map to have in advance, you can use a **map literal** to create it. A map literal starts with the map type (in the form `map[KeyType]ValueType`). This is followed by curly braces containing key/value pairs you want the map to start with. For each key/value pair, you include the key, a colon, and then the value. Multiple key/value pairs are separated by commas.

Map type                      Key                      Value                      Key                      Value  
mymap := map[string]float64{"a": 1.2, "b": 5.6}

Here are a couple of the preceding map examples, re-created using map literals:

```
ranks := map[string]int{"bronze": 3, "silver": 2, "gold": 1}
fmt.Println(ranks["gold"])
fmt.Println(ranks["bronze"])
elements := map[string]string{"H": "Hydrogen", "Li": "Lithium"}
fmt.Println(elements["H"])
fmt.Println(elements["Li"])
```

1  
3  
Hydrogen  
Lithium

As with slice literals, leaving the curly braces empty creates a map that starts empty.

You may find this preferable to using `make` to create maps.

```
emptyMap := map[string]float64{}
Create an empty map.
```



Fill in the blanks in the below program, so it will produce the output shown.

```
jewelry := _____(map[string]float64)
jewelry["necklace"] = 89.99
jewelry[_____] = 79.99
clothing := _____[string]float64{_____ : 59.99, "shirt": 39.99}
fmt.Println("Earrings:", jewelry["earrings"])
fmt.Println("Necklace:", jewelry[_____])
fmt.Println("Shirt:", clothing[_____])
fmt.Println("Pants:", clothing["pants"])
```

```
Earrings: 79.99
Necklace: 89.99
Shirt: 39.99
Pants: 59.99
```



**Answers on page 23.**

## ZERO VALUES IN MAPS

As with arrays and slices, if you access a map key that hasn't been assigned to, you'll get a zero value back.

```
Create a map with string keys and int values.
numbers := map[string]int{}
numbers["I've been assigned"] = 12
fmt.Println(numbers["I've been assigned"])
fmt.Println(numbers["I haven't been assigned"])
```

Print an assigned value.

Print an unassigned value.

```
12
0
```

Prints the zero value.

Depending on the value type, the zero value may not actually be `0`. For maps with a value type of `string`, for example, the zero value will be an empty string.

```
Print an assigned value.
strings := map[string]string{}
strings["I've been assigned"] = "hi"
fmt.Println(strings["I've been assigned"])
fmt.Println(strings["I haven't been assigned"])
```

Print an unassigned value.

```
hi
```

Prints the zero value (an empty string).

As with arrays and slices, zero values can make it safe to manipulate a map value even if you haven't explicitly assigned to it yet.

```
counters := map[string]int()
counters["a"]++
counters["a"]++
counters["c"]++
fmt.Println(counters["a"], counters["b"], counters["c"])

Still at its
zero value.
Has been
incremented twice.
Has been
incremented once.

2 0 1
```

## HOW TO TELL ZERO VALUES APART FROM ASSIGNED VALUES

Zero values, although useful, can sometimes make it difficult to tell whether a given key has been assigned the zero value, or if it has never been assigned.

Here's an example of a program where this could be an issue. This code erroneously reports that the student "Carl" is failing, when in reality he just hasn't had any grades logged:

```
func status(name string) {
 grades := map[string]float64{"Alma": 0, "Rohit": 86.5}
 grade := grades[name]
 if grade < 60 {
 fmt.Printf("%s is failing!\n", name)
 }
}

func main() {
 A map key with a value of 0 assigned.————→ status("Alma")
 A map key with no value assigned.————→ status("Carl")
}
```

Alma is failing!  
Carl is failing!

To address situations like this, accessing a map key optionally returns a second, boolean value. It will be `true` if the returned value has actually been assigned to the map, or `false` if the returned value just represents the default zero value. Most Go developers assign this boolean value to a variable named `ok` (because the name is nice and short).

```
counters := map[string]int{"a": 3, "b": 0}
var value int
var ok bool
value, ok = counters["a"] ← Access an assigned value.
 "ok" will be true.
fmt.Println(value, ok) ←
value, ok = counters["b"] ← Access an assigned value.
 "ok" will be true.
fmt.Println(value, ok) ←
value, ok = counters["c"] ← Access an unassigned value.
 "ok" will be false.
fmt.Println(value, ok)
```

3 true  
0 true  
0 false

If you only want to test whether a value is present, the value itself can be ignored by assigning it to the `_` blank identifier.

```

 counters := map[string]int{"a": 3, "b": 0}
 var ok bool
Test for the value's presence, but ignore it → _, ok = counters["b"]
 fmt.Println(ok)
Test for the value's presence, but ignore it → _, ok = counters["c"] true
 fmt.Println(ok) false

```

The second return value can be used to decide whether you should treat the value you got from the map as an assigned value that just happens to match the zero value for that type, or as an unassigned value.

Here's an update to our code that tests whether the requested key has actually had a value assigned before it reports a failing grade:

```

Get the value, plus a
boolean indicating whether func status(name string) {
this is an assigned value. grades := map[string]float64{"Alma": 0, "Rohit": 86.5}
If no value was assigned grade, ok := grades[name] ...Report that no grade has
to the specified key... if !ok { been logged for the student.
 fmt.Printf("No grade recorded for %s.\n", name)
Otherwise, follow the logic { } else if grade < 60 {
for reporting a failing grade. } fmt.Printf("%s is failing!\n", name)
}
}

func main() {
 status("Alma")
 status("Carl") Alma is failing!
 No grade recorded for Carl.
}

```



## Exercise

Write down what the output of the below program snippet would be.

```

data := []string{"a", "c", "e", "a", "e"}
counts := map[string]int{}
for _, item := range data {
 counts[item]++
}
letters := []string{"a", "b", "c", "d", "e"}
for _, letter := range letters {
 count, ok := counts[letter]
 if !ok {
 fmt.Printf("%s: not found\n", letter)
 } else {
 fmt.Printf("%s: %d\n", letter, count)
 }
}

```

Output:

.....  
.....  
.....  
.....  
.....  
.....

## REMOVING KEY/VALUE PAIRS WITH THE "DELETE" FUNCTION

At some point after assigning a value to a key, you may want to remove it from your map. Go provides the built-in `delete` function for this purpose. Just pass `delete` the map you want to delete a key from, and the key you want deleted. That key and its corresponding value will be removed from the map.

In the code below, we assign values to keys in two different hashes, then delete them again. After that, when we try accessing those keys, we get a zero value (which is `0` for the `ranks` map, `false` for the `isPrime` map). The secondary boolean value is also `false` in each case, which means that the key is not present.

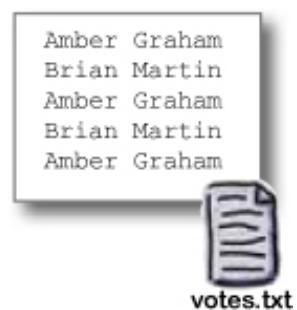
```
var ok bool
ranks := map[string]int{}
var rank int
ranks["bronze"] = 3 // Assign a value to the "bronze" key.
rank, ok = ranks["bronze"] // "ok" will be true because a value is present.
fmt.Printf("rank: %d, ok: %v\n", rank, ok)
delete(ranks, "bronze") // Delete the "bronze" key and its corresponding value.
rank, ok = ranks["bronze"] // "ok" will be false because the value's been deleted.
fmt.Printf("rank: %d, ok: %v\n", rank, ok)

isPrime := map[int]bool{}
var prime bool
isPrime[5] = true // Assign a value to the 5 key.
prime, ok = isPrime[5] // "ok" will be true because a value is present.
fmt.Printf("prime: %v, ok: %v\n", prime, ok)
delete(isPrime, 5) // Delete the 5 key and its corresponding value.
prime, ok = isPrime[5] // "ok" will be false because the value's been deleted.
fmt.Printf("prime: %v, ok: %v\n", prime, ok)
```

```
rank: 3, ok: true
rank: 0, ok: false
prime: true, ok: true
prime: false, ok: false
```

## UPDATING OUR VOTE COUNTING PROGRAM TO USE MAPS

Now that we understand maps a bit better, let's see if we can use what we've learned to simplify our vote counting program.



Previously, we used a pair of slices, one called `names` that held candidate names, and one called `counts` held vote counts for each name. For each name we read from the file, we

had to search through the slice of names, one by one, for a match. We then incremented the vote count for that name in the corresponding element of the `counts` slice.

```
// ...
names := []string{} ← Create a slice to hold candidate names.
counts := []int{} ← Create a slice to hold the number of times each name occurs.
for _, line := range lines {
 matched := false
 for i, name := range names { ← Loop over each value in the names slice.
 if name == line { ← If this line matches the current name...
 counts[i] += 1 ← ...Increment the corresponding count.
 }
}
// ...
```

Using a map will be much simpler. We can replace the two slices with a single map (which we'll also call `counts`). Our map will use candidate names as its keys, and integers (which will hold the vote counts for that name) as its values. Once that's set up, all we have to do is use each candidate name we read from the file as a map key, and increment the value that key holds.

Here's some simplified code that creates a map and increments the values for some candidate names directly:

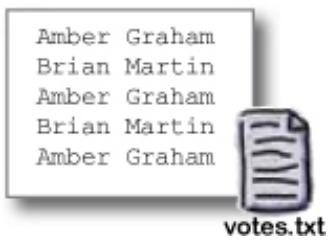
```
counts := map[string]int{}
counts["Amber Graham"]++
counts["Brian Martin"]++
counts["Amber Graham"]++
fmt.Println(counts)
map[Amber Graham:2 Brian Martin:1]
```

Our previous program needed separate logic to add new elements to both slices if the name wasn't found...

```
if matched == false { ← If no match was found...
 names = append(names, line) ← ...Add it as a new name...
 counts = append(counts, 1) ←
}
And add a new count (this line will
be the first occurrence).
```

But we don't need to do that with a map. If the key we're accessing doesn't already exist, we'll get the zero value back (literally `0` in this case, since our values are integers). We then increment that value, giving us `1`, which gets assigned to the map. When we encounter that name again, we'll get the assigned value, which we can then increment as normal.

Next, let's try incorporating our `counts` map into the actual program, so it can tally the votes from the actual file.



We'll be honest; after all that work to learn about maps, the final code looks a little anti-climactic! We replace the two slice declarations with a single map declaration. Next is the code in the loop that processes strings from the file. We replace the original eleven lines of code there with a single line, which increments the count in the map for the current candidate name. And we replace the loop at the end that prints the results with a single line that prints the whole `counts` map.

```

your workspace > src > github.com > headfirstgo > count > command.go

package main

import (
 "fmt"
 "github.com/headfirstgo/datafile"
 "log"
)

func main() {
 lines, err := datafile.GetStrings("votes.txt")
 if err != nil {
 log.Fatal(err)
 }
 counts := map[string]int{} ← Declare a map that will use candidate names
 for _, line := range lines { ← as keys, and vote counts as values.
 counts[line]++ ← Increment the vote count for the
 }
 fmt.Println(counts) ← Print the populated map.
}

```

Trust us, though, the code only *looks* anti-climactic. There are still complex operations going on here. But the map is handling them all for you, which means you don't have to write as much code!

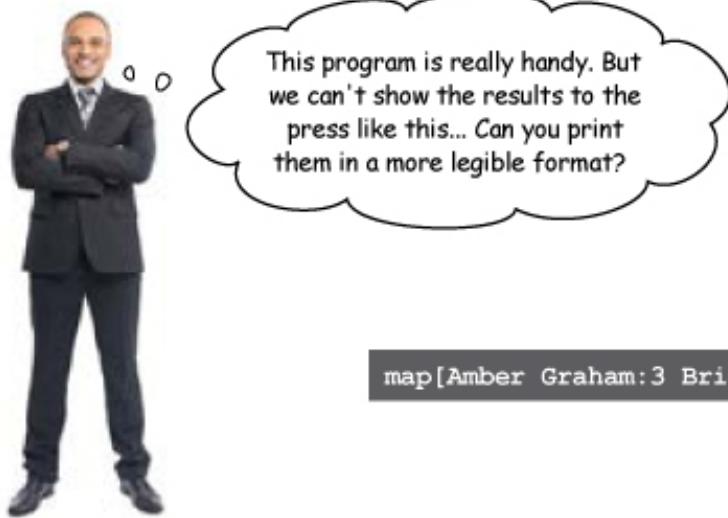
As before, you can re-compile the program using the `go install` command. When we re-run the executable, the `votes.txt` file will be loaded and processed. We'll see the `counts` map printed, with the number of times each name was encountered in the file.

```

Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$./count
map[Amber Graham:3 Brian Martin:2]

```

## USING FOR ... RANGE LOOPS WITH MAPS



The format we have...

```
map[Amber Graham:3 Brian Martin:2]
```

Name: Kevin Wagner  
Occupation: Election Volunteer

That's true. A format of one name and one vote count per line would probably be better:

...The format we want...

```
Amber Graham: 3
Brian Martin: 2
```

To format each key and value from the map as a separate line, we're going to need to loop through each entry in the map.

The same `for ... range` loop we've been using to process array and slice elements works on maps, too. Instead of assigning an integer index to the first variable you provide, however, the current map key will be assigned.

```
Variable that
will hold each
map key.
for [key], [value] := [range] myMap {
 Variable that
 will hold each
 corresponding value.
 "range" keyword.
 // Loop block here.
}
```

The map being processed.

The `for ... range` loop makes it easy to loop through a map's keys and values. Just provide a variable to hold each key, and another to hold the corresponding value, and it will automatically loop through each entry in the map.

```

package main

import "fmt"

func main() {
 grades := map[string]float64{"Alma": 74.2, "Rohit": 86.5, "Carl": 59.7}
Loop through each key/value pair. { for name, grade := range grades {
 fmt.Printf("%s has a grade of %0.1f%\n", name, grade)
}
}

```

Print each key and its corresponding value.

Carl has a grade of 59.7%
 Alma has a grade of 74.2%
 Rohit has a grade of 86.5%

If you only need to loop through the keys, you can omit the variable that holds the values:

Process only the keys: →

```

fmt.Println("Class roster:")
for name := range grades {
 fmt.Println(name)
}

```

**Class roster:**
 Alma
 Rohit
 Carl

And if you only need the values, you can assign the keys to the `_` blank identifier:

Process only the values: →

```

fmt.Println("Grades:")
for _, grade := range grades {
 fmt.Println(grade)
}

```

**Grades:**
 59.7
 74.2
 86.5

But there's one potential issue with this example... If you save the preceding example to a file and run it with `go run`, you'll find that the map keys and values are printed in a random order. If you run the program multiple times, you'll get a different order each time.

(Note: The same is not true of code run via the online Go Playground site. There, the order will still be random, but it will produce the same output each time it's run.)

The loop follows a different order each time! →

Shell Edit View Window Help
 

```
$ go run temp.go
Alma has a grade of 74.2%
Rohit has a grade of 86.5%
Carl has a grade of 59.7%
$ go run temp.go
Carl has a grade of 59.7%
Alma has a grade of 74.2%
Rohit has a grade of 86.5%
```

## THE FOR ... RANGE LOOP HANDLES MAPS IN RANDOM ORDER!

The `for ... range` loop processes map keys and values in a random order because a map is an *unordered* collection of keys and values. When you use `for ... range` loop with a map, you never know what order you'll get the map's contents in! Sometimes that's fine, but if you need more consistent ordering, you'll need to write the code for that yourself.

Here's an update to the previous program that always prints the names in alphabetical order. It does this using two separate `for` loops. The first loops over each key in the map, ignoring the values, and adds them to a slice of strings. Then, the slice is passed to the `sort` package's `Strings` method to sort it alphabetically, in-place.

The second `for` loop doesn't loop over the map, it loops over the sorted slice of names. (Which, thanks to the preceding code, now contains every key from the map in alphabetical order.) It prints the name, and then gets the value that matches that name from the map. It still processes every key and value in the map, but it gets the keys from the sorted slice, not the map itself.

```
package main

import (
 "fmt"
 "sort"
)

func main() {
 grades := map[string]float64{"Alma": 74.2, "Rohit": 86.5, "Carl": 59.7}
 names := []string{}
 for name := range grades {
 names = append(names, name)
 }
 sort.Strings(names)
 for _, name := range names {
 fmt.Printf("%s has a grade of %0.1f%%\n", name, grades[name])
 }
}
```

Annotations on the code:

- Build a slice with all the map keys.** A bracket groups the declaration of `names` and the first `for` loop.
- Sort the slice alphabetically.** An arrow points from the `sort.Strings(names)` call to the `for` loop below it.
- Process the names alphabetically.** An arrow points from the `for` loop below to the `fmt` call.
- Use the current student name to get the grade from the map.** An arrow points from the `name` variable in the `fmt` call to the `grades[name]` access in the same line.

If we save the above code and run it, this time the student names are printed in alphabetical order. This will be true no matter how many times we run the program.

If it doesn't matter what order your map data is processed in, using a `for ... range` loop directly on the map will probably work for you. But if order matters, you may want to consider setting up your own code to handle the processing order.

```
Shell Edit View Window Help
$ go run temp.go
Alma has a grade of 74.2%
Carl has a grade of 59.7%
Rohit has a grade of 86.5%
$ go run temp.go
Alma has a grade of 74.2%
Carl has a grade of 59.7%
Rohit has a grade of 86.5%
```

The names are processed in alphabetical order each time. →

## UPDATING OUR VOTE COUNTING PROGRAM WITH A FOR ... RANGE LOOP

There aren't a lot of candidates in Sleepy Creek County, so we don't see a need to sort the output by name. We'll just use a `for ... range` loop to process the keys and values

directly from the map.



It's a pretty simple change to make; we just replace the line that prints the entire map with a `for ... range` loop. We'll assign each key to a `name` variable, and each value to a `count` variable. Then we'll call `Printf` to print the current candidate name and vote count.

```
your workspace > src > github.com > headfirstgo > count > command.go

package main

import (
 "fmt"
 "github.com/headfirstgo/datafile"
 "log"
)

func main() {
 lines, err := datafile.GetStrings("votes.txt")
 if err != nil {
 log.Fatal(err)
 }
 counts := map[string]int{}
 for _, line := range lines {
 counts[line]++
 }
 Process each map key and value. { for name, count := range counts {
 fmt.Printf("Votes for %s: %d\n", name, count)
 }
}

Print the key (the candidate name). ↑
Print the value (the vote count). ↑
```

Another compilation via `go install`, another run of the executable, and we'll see our output in its new format. Each candidate name and their vote count is here, neatly formatted on its own line.

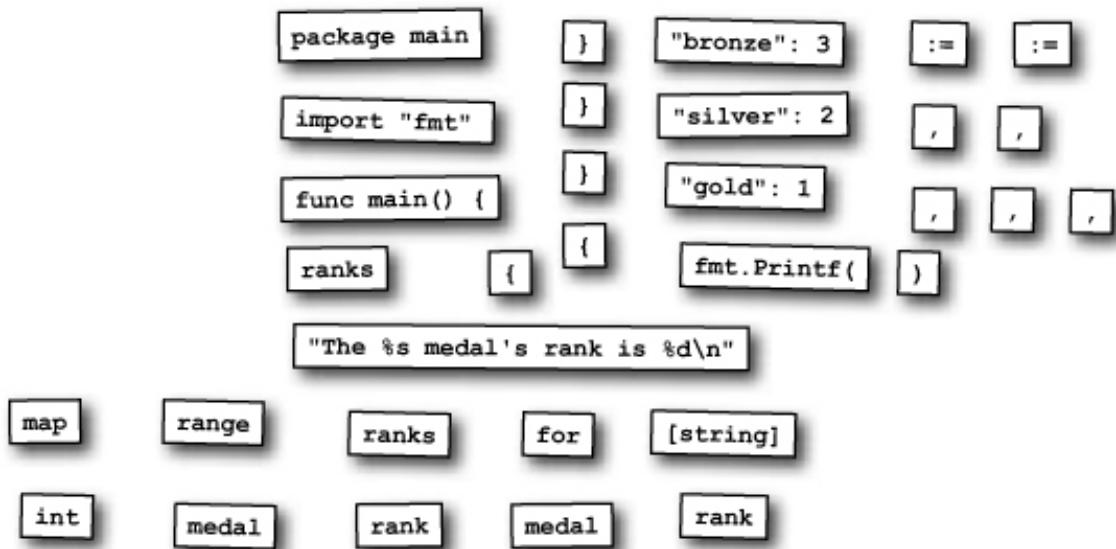
When the only data collections we had available were arrays and slices, we needed a lot of extra code and processing time to look values up. But maps have made the process easy! Anytime you need to be able to find a collection's values again, you should consider using a map!

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$./count
Amber Graham: 3
Brian Martin: 2
```

# CODE MAGNETS



A Go program that uses a `for ... range` loop to print out the contents of a map is scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output? (It's okay if the output order differs between runs of the program.)



## YOUR GO TOOLBOX

That's it for Chapter 7! You've added maps to your toolbox.

## Arrays

## Slices

## Maps

A map is a collection where each value is stored under a corresponding key.

Whereas arrays and slices can only use integers as indexes, a map can use (almost) any type for keys.

All of a map's keys must be the same type, and all the values must be the same type, but the keys don't have to be the same type as the values.



## BULLET POINTS

- When declaring a map variable, you must provide the types for its keys and its values:  
`var mymap map[string]int`
- To assign a value to a map, provide the key you want to assign it to in square brackets:  
`mymap["my key"] = 12`
- To get a value, you provide the key as well:  
`fmt.Println(mymap["my key"])`
- You can create a map and initialize it with data at the same time using a map literal:  
`map[string]int{"a": 2, "b": 3}`
- As with arrays and slices, if you access a map key that hasn't been assigned a value, you'll get a zero value back.
- Getting a value from a map can return a second, optional boolean value that indicates whether that value was assigned, or if it represents a default zero value:  
`value, ok := mymap["c"]`
- If you only want to test whether a key has had a value assigned, you can ignore the actual value using the \_ blank identifier:  
`_, ok := mymap["c"]`

- You can delete keys and their corresponding values from a map using the `delete` built-in function:

```
delete(mymap, "b")
```

- You can use `for ... range` loops with maps, much like you can with arrays or slices. You provide one variable that will be assigned each key in turn, and a second variable that will be assigned each value in turn.

```
for key, value := range mymap {
 fmt.Println(key, value)
}
```

- The `for ... range` loop processes map key/value pairs in random order. If you need a specific order, you'll need to handle that yourself.



Fill in the blanks in the below program, so it will produce the output shown.

```
jewelry := make(map[string]float64)
jewelry["necklace"] = 89.99
jewelry["earrings"] = 79.99
clothing := map[string]float64{"pants": 59.99, "shirt": 39.99}
fmt.Println("Earrings:", jewelry["earrings"])
fmt.Println("Necklace:", jewelry["necklace"])
fmt.Println("Shirt:", clothing["shirt"])
fmt.Println("Pants:", clothing["pants"])
```

Output  
↓  
**Earrings: 79.99  
Necklace: 89.99  
Shirt: 39.99  
Pants: 59.99**



Write down what the output of the below program snippet would be.

```

data := []string{"a", "c", "e", "a", "e"}
counts := map[string]int{}
for _, item := range data {
 counts[item]++
}
letters := []string{"a", "b", "c", "d", "e"}
for _, letter := range letters {
 count, ok := counts[letter]
 if !ok {
 fmt.Printf("%s: not found\n", letter)
 } else {
 fmt.Printf("%s: %d\n", letter, count)
 }
}

```

Output:

a: 2

b: not found

c: 1

d: not found

e: 2

## CODE MAGNETS SOLUTION

```

package main

import "fmt"

func main() {

 ranks := map[string] int { "bronze": 3, "silver": 2, "gold": 1 }

 for medal, rank := range ranks {
 fmt.Printf("The %s medal's rank is %d\n", medal, rank)
 }
}


```

↓ Output

The gold medal's rank is 1  
 The bronze medal's rank is 3  
 The silver medal's rank is 2

## 8 structs

### Building Storage



**Sometimes you need to store more than one type of data.** We learned about slices, which store a list of values. Then we learned about maps, which map a list of keys to a list of values. But both of these data structures can only hold values of *one* type. Sometimes, you need to group together values of *several* types. Think of mailing addresses, where you have to mix street names (strings) with postal codes (integers). Or student records, where you have to mix student names (strings) with grade point averages (floating-point numbers). You can't mix value types in slices or maps. But you *can* if you use another type called a **struct**. We'll learn all about structs in this chapter!

# SLICES AND MAPS HOLD VALUES OF ONE TYPE

Gopher Fancy is a new magazine devoted to lovable rodents. They're currently working on a system to keep track of their subscriber base.



To start, we need to store the subscriber's name, the monthly rate we're charging them, and whether their subscription is active. But the name is a string, the rate is a float64, and the active indicator is a bool. We can't make one slice hold all those types!

A slice can only be set up to hold one type of values.

```
subscriber := []string{}
subscriber = append(subscriber, "Aman Singh")
subscriber = append(subscriber, 4.99) ← We can't add this float64!
subscriber = append(subscriber, true) ← We can't add this boolean!
```

```
cannot use 4.99 (type float64) as type string in append
cannot use true (type bool) as type string in append
```



Then we tried maps. We wish that would have worked, because we could have used the keys to label what each value represented. But just like slices, maps can only hold one type of value!

A map can only hold one type of values.

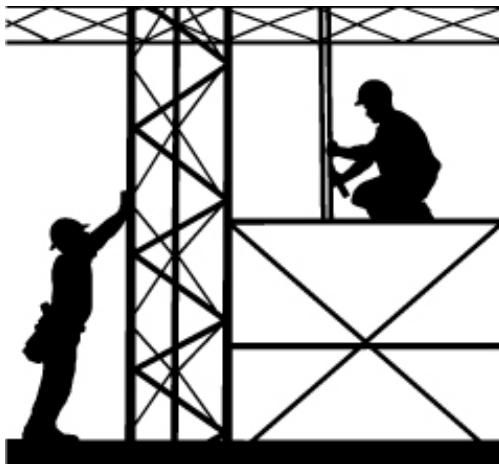
```
subscriber := map[string]float64{}
subscriber["name"] = "Aman Singh" ← We can't store this string!
subscriber["rate"] = 4.99
subscriber["active"] = true ← We can't store this boolean!
```

```
cannot use "Aman Singh" (type string)
as type float64 in assignment
cannot use true (type bool)
as type float64 in assignment
```

**It's true: arrays, slices, and maps are no help if you need to mix values of different types. They can only be set up to hold values of a single type. But Go does have a way to solve this problem...**

## STRUCTS ARE BUILT OUT OF VALUES OF MANY TYPES

A **struct** (short for "structure") is a value that is constructed out of other values of many different types. Whereas a slice might only be able to hold `string` values or a map might only be able to hold `int` values, you can create a struct that holds `string` values, `int` values, `float64` values, `bool` values, and more — all in one convenient grouping.



You declare a struct type using the `struct` keyword, followed by curly braces. Within the braces, you can define one or more **fields**: values that the struct groups together. Each field definition appears on a separate line, and consists of a field name, followed by the type of value that field will hold.

```
"struct" keyword.

struct {
Field name.-----field1 string-----Field type.
Field name.-----field2 int-----Field type.
}
```

You can use a struct type as the type of a variable you're declaring. This code declares a variable named `myStruct` that holds structs that have a `float64` field named `number`, a `string` field named `word`, and a `bool` field named `toggle`:

(It's more common to use a defined type to declare struct variables, but we won't cover type definitions for a few more pages, so we'll write it this way for now.)

```
Declare a variable
named "myStruct".

var myStruct struct {
 number float64
 word string
 toggle bool
}
fmt.Printf("%#v\n", myStruct) ← Print out the struct value as it would appear in Go code.
The "myStruct" variable can hold structs that
have a float64 "number" field, a string "word"
field, and a bool "toggle" field.
The struct fields, each set
to that type's zero value.→ struct { number float64; word string; toggle bool }
{number:0, word:"", toggle:false}
```

When we call `Printf` with the `%#v` verb above, it prints the value in `myStruct` as a struct literal. We'll be covering struct literals later in the chapter, but for now you can see that the struct's `number` field has been set to `0`, the `word` field to an empty string, and the `toggle` field to `false`. Each field has been set to the zero value for its type.

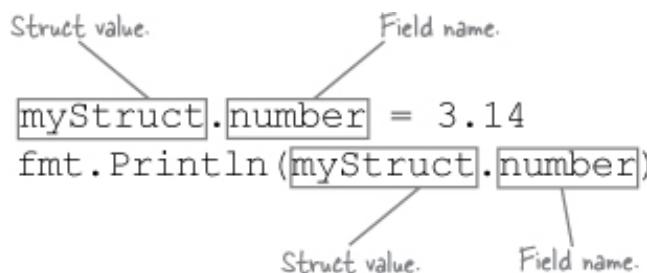
## ACCESS STRUCT FIELDS USING THE DOT OPERATOR

Now we can define a struct, but to actually use it, we need a way to store new values in the struct's fields and retrieve them again.

All along, we've been using the dot operator to indicate functions that "belong to" another package, or methods that "belong to" a value:

```
fmt.Println("hi") var myTime time.Time
 myTime.Year()
Call a function belonging
to the "fmt" package. Call a method belonging
to a "Time" value.
```

Similarly, we can use a dot operator to indicate fields that "belong to" a struct. This works both for assigning values, and retrieving them.



We can use dot operators to assign values to all the fields of `myStruct`, and then print them back out:

```
var myStruct struct {
 number float64
 word string
 toggle bool
}
Assign values to struct fields. {myStruct.number = 3.14
 myStruct.word = "pie"
 myStruct.toggle = true
Retrieve values from struct fields. {fmt.Println(myStruct.number)
 fmt.Println(myStruct.word)
 fmt.Println(myStruct.toggle)}
```

```
3.14
pie
true
```



**Don't worry about the number of spaces between struct field names and their types.**

When you write your struct fields, just insert a single space between the field name and its type. When you run the `go fmt` command on your files (which you should always do) it will insert extra spaces so that all the types align vertically. The alignment just makes the code easier to read; it doesn't change its meaning at all!

```
var aStruct struct {
 shortName int
 longerName float64
 longestName string
} Extra spaces align
 ↑
 the field types.
```

## STORING SUBSCRIBER DATA IN A STRUCT

Now that we know how to declare a variable that holds a struct and assign values to its fields, we can create a struct to hold magazine subscriber data.

First, we'll define a variable named `subscriber`. We'll give `subscriber` a struct type with `name` (`string`), `rate` (`float64`), and `active` (`bool`) fields.

With the variable and its type declared, we can then use dot operators to access the struct's fields. We assign values of the appropriate type to each field, and then print the values back out again.

Declare a "subscriber" variable... ↘ ...That holds structs... ↘

```
var subscriber struct {
 name string
 rate float64
 active bool
}
```

The struct will have a "name" field that holds a string... ↗ ...A "rate" field that holds a float64... ↗ ...And an "active" field that holds a bool.

Assign values to struct fields. { subscriber.name = "Aman Singh"  
subscriber.rate = 4.99  
subscriber.active = true }

Retrieve values from struct fields. { fmt.Println("Name:", subscriber.name)  
fmt.Println("Monthly rate:", subscriber.rate)  
fmt.Println("Active?", subscriber.active) }

**Name: Aman Singh**  
**Monthly rate: 4.99**  
**Active? true**

Even though the data we have for a subscriber is stored using a variety of types, structs let us keep it all in one convenient package!



At the right is a program that creates a struct variable which holds a pet's name (a string) and age (an int). Fill in the blanks so that the code will produce the output shown.

```
package main

import "fmt"

func main() {
 var pet _____ {
 name _____
 _____ int
 }
 pet._____ = "Max"
 pet.age = 5
 fmt.Println("Name:", _____.name)
 fmt.Println("Age:", pet._____)
}
```

**Name: Max**  
**Age: 5**

## DEFINED TYPES AND STRUCTS

Structs seem promising... but declaring struct variables is really tedious for us. We have to repeat the entire struct type declaration for each new variable!

```
var subscriber1 struct {
 name string
 rate float64
 active bool
}

subscriber1.name = "Aman Singh"
fmt.Println("Name:", subscriber1.name)

var subscriber2 struct {
 name string
 rate float64
 active bool
}

subscriber2.name = "Beth Ryan"
fmt.Println("Name:", subscriber2.name)
```

Define the struct type for the "subscriber1" variable.

Define an identical type all over again for the "subscriber2" variable!

Name: Aman Singh  
Name: Beth Ryan

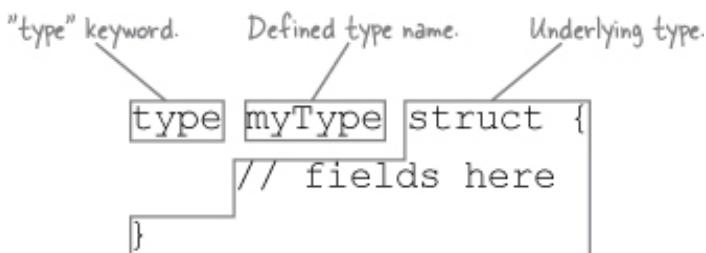


Throughout this book, you've used a variety of types, like `int`, `string`, `bool`, slices, maps, and now structs. But you haven't been able to create completely *new* types.

**Type definitions** allow you to create types of your own. They let you create a new **defined type** that's based on an **underlying type**.

Although you can use any type as an underlying type, such as `float64`, `string`, or even slices or maps, in this chapter we're going to focus on using struct types as underlying types. We'll try using other underlying types when we take a deeper look at defined types in the next chapter.

To write a type definition, use the `type` keyword, followed by the name for your new defined type, and then the underlying type you want to base it on. If you're using a struct type as your underlying type, you'll use the `struct` keyword followed by a list of field definitions in curly braces, just as you did when declaring struct variables.



Just like variables, type definitions *can* be written within a function. But that will limit its scope to that function's block, meaning you won't be able to use it outside that function. So types are usually defined outside of any functions, at the package level.

As a quick demonstration, the below code defines two types: `part`, and `car`. Each defined type uses a struct as its underlying type.

Then, within the `main` function, we declare a `porsche` variable of the `car` type, and a `bolts` variable of the `part` type. There's no need to re-write the lengthy struct definitions when declaring the variables; we just use the names of the defined types.

```
package main

import "fmt"

Define a type named "part".
type part struct {
 description string
 count int
}

Define a type named "car".
type car struct {
 name string
 topSpeed float64
}

func main() {
 var porsche car
 porsche.name = "Porsche 911 R"
 porsche.topSpeed = 323
 fmt.Println("Name:", porsche.name)
 fmt.Println("Top speed:", porsche.topSpeed)

 Access the struct fields. Declare a variable of type "part".
 var bolts part
 bolts.description = "Hex bolts"
 bolts.count = 24
 fmt.Println("Description:", bolts.description)
 fmt.Println("Count:", bolts.count)
}
```

The underlying type for "part" will be a struct with these fields.

The underlying type for "car" will be a struct with these fields.

Access the struct fields.

Access the struct fields.

Declare a variable of type "car".

Declare a variable of type "part".

Name: Porsche 911 R  
Top speed: 323  
Description: Hex bolts  
Count: 24

With the variables declared, we can set the values of their struct fields and get the values back out, just as we did in previous programs.

## USING A DEFINED TYPE FOR MAGAZINE SUBSCRIBERS

Previously, to create more than one variable that stored magazine subscriber data in a struct, we had to write out the full struct type (including all its fields) for each variable.

```

var subscriber1 struct {
 name string
 rate float64
 active bool
}
// ...
var subscriber2 struct {
 name string
 rate float64
 active bool
}
// ...

```

Define a struct type.

Define an identical type.

But now, we can simply define a `subscriber` type at the package level. We write the struct type just once, as the underlying type for the defined type. When we're ready to declare variables, we don't have to write the struct type again; we simply use `subscriber` as their type. No more need to repeat the entire struct definition!

```

package main

import "fmt"
Define a type named "subscriber".
type subscriber struct {
 name string
 rate float64
 active bool
}

func main() {
 var subscriber1 subscriber
 subscriber1.name = "Aman Singh"
 fmt.Println("Name:", subscriber1.name)
 var subscriber2 subscriber
 subscriber2.name = "Beth Ryan" ← Use the "subscriber" type for
 fmt.Println("Name:", subscriber2.name)
}

```

Use the struct type that was on the variables as the underlying type for the type definition.

Declare a variable of type "subscriber".

Use the "subscriber" type for the second variable, too.

Name: Aman Singh  
 Name: Beth Ryan

## USING DEFINED TYPES WITH FUNCTIONS

Defined types can be used for more than just variable types. They also work for function parameters and return values.

Here's our `part` type again, together with a new `printInfo` function that prints a part's fields. The function takes a single parameter, with `part` as its type. Within `printInfo`, we access the fields via the parameter variable just like any other struct variable's.

```

package main

import "fmt"

type part struct {
 description string
 count int
}

func showInfo(p part) {
 Access the parameter's fields.
 fmt.Println("Description:", p.description)
 fmt.Println("Count:", p.count)
}

func main() {
 var bolts part
 bolts.description = "Hex bolts"
 bolts.count = 24
 showInfo(bolts)
}

```

Create a "part" value.

Pass the "part" to the function.

Description: Hex bolts  
 Count: 24

And here's a `minimumOrder` function that creates a `part` with a specified description and a predefined value for the `count` field. We declare `minimumOrder`'s return type to be `part` so it can return the new struct.

```

// Package, imports, type definition omitted

func minimumOrder(description string) part {
 var p part
 p.description = description
 p.count = 100
 return p
}

func main() {
 p := minimumOrder("Hex bolts")
 fmt.Println(p.description, p.count)
}

```

Create a new "part" value.

Return the "part".

Call minimumOrder. Use a short variable declaration to store the returned "part".

Hex bolts 100

Here are a couple functions that work with the magazine's `subscriber` type...

The `printInfo` function takes a `subscriber` as a parameter, and prints the values of its fields.

We also have a `defaultSubscriber` function that sets up a new `subscriber` struct with some default values. It takes a string parameter called `name`, and uses that to set a new `subscriber` value's `name` field. Then it sets the `rate` and `active` fields to default values. Finally, it returns the completed `subscriber` struct to its caller.

```

package main

import "fmt"

type subscriber struct {
 name string
 rate float64
 active bool
}

Declare one parameter... ↴ ...With a type of "subscriber".
func printInfo(s subscriber) {
 fmt.Println("Name:", s.name)
 fmt.Println("Monthly rate:", s.rate)
 fmt.Println("Active?", s.active)
}

func defaultSubscriber(name string) subscriber {
 var s subscriber ← Create a new "subscriber".
 Set the struct's fields. { s.name = name
 s.rate = 5.99
 s.active = true
 return s ← Return the "subscriber".
}

func main() {
 subscriber1 := defaultSubscriber("Aman Singh") ← Set up a subscriber
 subscriber1.rate = 4.99 ← with this name.
 printInfo(subscriber1) ← Print the field values.

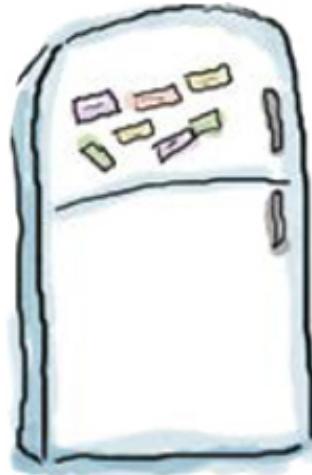
 subscriber2 := defaultSubscriber("Beth Ryan") ← Set up a subscriber
 printInfo(subscriber2) ← with this name.
}

```

Name: Aman Singh  
 Monthly rate: 4.99  
 Active? true  
 Name: Beth Ryan  
 Monthly rate: 5.99  
 Active? true

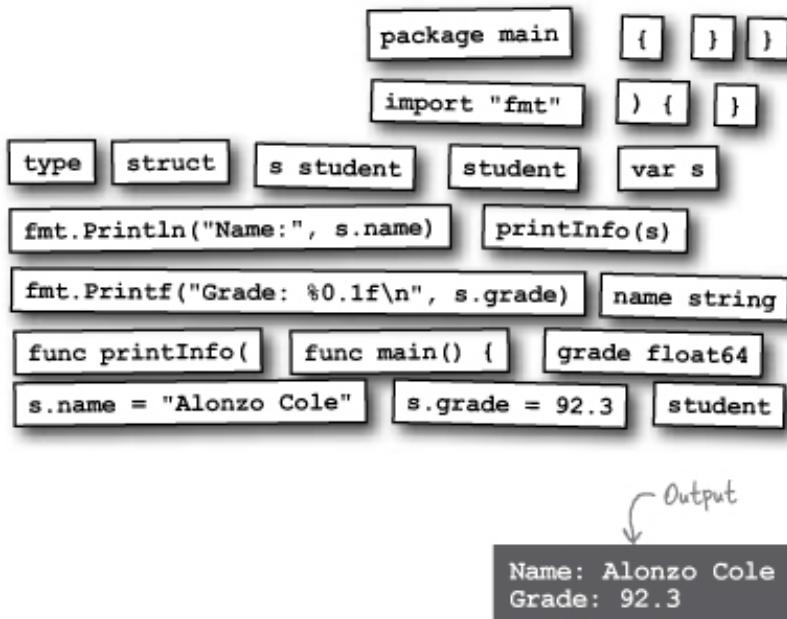
In our `main` function, we can pass subscriber names to `defaultSubscriber` to get a new `subscriber` structs. One subscriber gets a discounted `rate`, so we re-set that struct field directly. We can pass filled-out `subscriber` structs to `printInfo` to print out their contents.

## CODE MAGNETS



A Go program is scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that will produce the given output? The finished program will

have a defined struct type named `student`, and a `printInfo` function that accepts a `student` value as a parameter.



**Don't use an existing type name as a variable name!**

If you've defined a type named `car` in the current package, and you declare a variable that's also named `car`, the variable name will **shadow** (take precedence over) the type name, making it inaccessible.

Refers to the type.  
↓  
`var car car`  
`var car2 car`

Refers to the variable,  
↑  
resulting in an error!

This isn't a common problem in practice, because defined types are often exported from their packages (and their names are therefore capitalized), and variables often are not (and their names are therefore lower-case). `Car` (an exported type name) can't conflict with `car` (an unexported variable name). We'll see more about exporting defined types later in the chapter. Still, shadowing is a confusing problem when it occurs, so it's good to be aware that it can happen.

# MODIFYING A STRUCT USING A FUNCTION



We'll be offering this \$4.99 discounted rate to a lot of subscribers, so I tried to create an `applyDiscount` function to set the `rate` field for us. But it's not working!

```
func applyDiscount(s subscriber) {
 s.rate = 4.99 ← Set the "rate" field.
}

func main() {
 var s subscriber
 applyDiscount(s) ← Attempt to set a
 fmt.Println(s.rate) "subscriber" struct's
 "rate" field to 4.99.
}

```

0 ← But it's still set to 0!

Our friends at Gopher Fancy are trying to write a function that takes a struct as a parameter, and updates one of the fields in that struct.

Remember way back in [Chapter 3](#), when we were trying to write a `double` function that took a number and doubled it? After `double` returned, the number was back to its original value!

```
func main() {
 amount := 6
 double(amount) ← Pass an argument to the function.
 fmt.Println(amount) ← Prints the original value!
}
func double(number int) {
 number *= 2
}

```

Parameter is set to a copy of the argument  
Alters the copied value, not the original!

6 ← Prints the unchanged amount!

That's when we learned that Go is a "pass-by-value" language, meaning that function parameters receive a *copy* of the arguments the function was called with. If a function changes a parameter value, it's changing the *copy*, not the *original*.

The same thing is true for structs. When we pass a `subscriber` struct to `applyDiscount`, the function receives a *copy* of the struct. So when we set the `rate` field on the struct, we're modifying the *copied* struct, not the *original*.

```
func applyDiscount(s subscriber) {
 s.rate = 4.99 ← Modifies the copy, not the original!
}
```

↳ Receives a copy of the struct!

Back in [Chapter 3](#), our solution was to update the function parameter to accept a *pointer* to a value, instead of accepting a value directly. When calling the function, we used the address-of operator (`&`) to pass a pointer to the value we wanted to update. Then, within the function, we used the `*` operator to update the value at that pointer.

As a result, the updated value was still visible after the function returned.

A hand-drawn diagram of a C code snippet. The code defines a `main` function that initializes `amount` to 6, then calls `double` with a pointer to `amount`. The `double` function takes a pointer to an `int` and multiplies its value by 2. The output is 12. Annotations explain: 'Pass a pointer instead of the variable value.' points to the `double(&amount)` call; 'Accept a pointer instead of an int value.' points to the `double(number *int)` declaration; 'Update the value at the pointer.' points to the `*number *= 2` assignment; and 'Prints the doubled amount' points to the number 12.

```
func main() {
 amount := 6
 double(&amount) ← Pass a pointer instead of the variable value.
 fmt.Println(amount)
}

func double(number *int) {
 *number *= 2
} ← Accept a pointer instead of an int value.
 ← Update the value at the pointer.
 12 ← Prints the doubled amount
```

We can use pointers to allow a function to update a struct, as well.

Here's an updated version of the `applyDiscount` function that should work correctly. We update the `s` parameter to accept a pointer to a `subscriber` struct, rather than the struct itself. Then we update the value in the struct's `rate` field.

In `main`, we call `applyDiscount` with a pointer to a `subscriber` struct. When we print the value in the struct's `rate` field, we can see that it's been updated successfully!

```

package main

import "fmt"

type subscriber struct {
 name string
 rate float64
 active bool
}

func applyDiscount(s *subscriber) {
 s.rate = 4.99 // Update the struct field.
}

func main() {
 var s subscriber
 applyDiscount(&s)
 fmt.Println(s.rate)
}

```

*Take a pointer to a struct, not the struct itself.*

*Pass a pointer, not a struct.*

4.99



Wait, how does that work? In the double function, we had to use the \* operator to get the value at the pointer. Don't you need \* when you set the rate field in applyDiscount?

**Actually, no! The dot notation to access fields works on struct pointers as well as the structs themselves.**

## ACCESSING STRUCT FIELDS THROUGH A POINTER

If you try to print a pointer variable, what you'll see is the memory address it points to. This is generally not what you want.

```

func main() {
 var value int = 2 // Create a value.
 var pointer *int = &value // Get a pointer to the value.
 fmt.Println(pointer) // Oops! This prints the
} // pointer, not the value!

```

0xc420014100

Instead, you need to use the \* operator (what we like to call the "value-at operator") to get the value at the pointer.

```

func main() {
 var value int = 2
 var pointer *int = &value
 fmt.Println(*pointer) // Print the value at
}

```

3

So you might think you'd need to use the \* operator with pointers to structs as well. But

just putting a \* before the struct pointer won't work:

```
type myStruct struct {
 myField int
}

func main() {
 var value myStruct ← Create a struct value.
 value.myField = 3
 var pointer *myStruct = &value ← Get a pointer to the struct value.
 fmt.Println(*pointer.myField)
}
 ↗ Attempt to get the struct
 value at the pointer. ↗ Error!
 invalid indirect of
 pointer.myField (type int)
```

If you write `*pointer.myField`, Go thinks that `myField` must contain a pointer. It doesn't, though, so an error results. To get this to work, you need to wrap `*pointer` in parentheses. That will cause the `myStruct` value to be retrieved, after which you can access the struct field.

```
func main() {
 var value myStruct
 value.myField = 3
 var pointer *myStruct = &value
 fmt.Println((*pointer).myField) ← 3
}
```

Get the struct value at  
the pointer, then access  
the struct field.

Having to write `(*pointer).myField` all the time would get tedious quickly, though. For this reason, the dot operator lets you access fields via pointers to structs, just as you can access fields directly from struct values. You can leave off the parentheses and the `*` operator.

```
func main() {
 var value myStruct
 value.myField = 3
 var pointer *myStruct = &value
 fmt.Println(pointer.myField) ← 3
}
```

Access the struct field  
through the pointer.

This works for assigning to struct fields through a pointer as well:

```
func main() {
 var value myStruct
 var pointer *myStruct = &value
 pointer.myField = 9 ← Assign to a struct field through the pointer.
 fmt.Println(pointer.myField)
} 9
```

And that's how the `applyDiscount` function is able to update the struct field without using the `*` operator. It assigns to the `rate` field *through* the struct pointer.

```
func applyDiscount(s *subscriber) {
 s.rate = 4.99 ←
}
func main() {
 var s subscriber
 applyDiscount(&s)
 fmt.Println(s.rate) 4.99
```

Assign to the struct field through the pointer.

## THERE ARE NO DUMB QUESTIONS

**Q:** You showed a `defaultSubscriber` function before that set a struct's fields, but it didn't need to use any pointers! Why not?

**A:** The `defaultSubscriber` function *returned* a struct value. If a caller stores the returned value, then the values in its fields will be preserved. Only functions that *modify existing* structs without returning them have to use pointers for those changes to be preserved.

But `defaultSubscriber` *could* have returned a pointer to a struct, if we had wanted it to. In fact, we make just that change in the next section!

## PASS LARGE STRUCTS USING POINTERS



So function parameters receive a copy of the arguments from the function call, even for structs...  
If you pass a big struct with a lot of fields, won't that take up a lot of the computer's memory?

**Yes, it will. It has to make room for the original struct and the copy.**

```
// Code above here omitted
type subscriber struct {
 name string
 rate float64
 active bool
}

func printInfo(s *subscriber) {
 fmt.Println("Name:", s.name)
 fmt.Println("Monthly rate:", s.rate)
 fmt.Println("Active?", s.active)
}

func defaultSubscriber(name string) *subscriber {
 var s subscriber
 s.name = name
 s.rate = 5.99
 s.active = true
 return &s ← Return a pointer to a struct
 instead of the struct itself.
}

func applyDiscount(s *subscriber) {
 s.rate = 4.99
}

func main() {
 subscriber1 := defaultSubscriber("Aman Singh")
 applyDiscount(subscriber1)
 printInfo(subscriber1) ↑ Since this is already a struct,
 remove the address-of operator.
 subscriber2 := defaultSubscriber("Beth Ryan")
 printInfo(subscriber2)
}
```

```
Name: Aman Singh
Monthly rate: 4.99
Active? true
Name: Beth Ryan
Monthly rate: 5.99
Active? true
```

Functions receive a copy of the arguments they're called with, even if they're a big value like a struct.

That's why, unless your struct has only a couple small fields, it's often a good idea to pass functions a *pointer* to a struct, rather than the struct itself. (This is true even if the function doesn't need to modify the struct.) When you pass a struct pointer, only one copy of the original struct exists in memory. The function just receives the memory address of that single struct, and can read the struct, modify it, or whatever else it needs to do, all without making an extra copy.

Here's our `defaultSubscriber` function, updated to return a pointer, and our `printInfo` function, updated to receive a pointer. Neither of these functions needs to change an existing struct like `applyDiscount` does. But using pointers ensures that only one copy of each struct needs to be kept in memory, while still allowing the program to work as normal.



The two programs below aren't working quite right. The `nitroBoost` function in the left-hand program is supposed to add 50 kilometers/hour to a `car`'s top speed, but it's not. And the `doublePack` function in the right-hand program is supposed to double a `part` value's `count` field, but it's not, either.

See if you can fix the programs. Only minimal changes will be necessary; we've left a little extra space in the code so you can make the necessary updates.

```
package main package main
import "fmt" import "fmt"
type car struct { type part struct {
 name string description string
 topSpeed float64 count int
}
func nitroBoost(c car) { func doublePack(p part) {
 c.topSpeed += 50 p.count *= 2
}
func main() { func main() {
 var mustang car var fuses part
 mustang.name = "Mustang Cobra" fuses.description = "Fuses"
 mustang.topSpeed = 225 fuses.count = 5
 nitroBoost(mustang) doublePack(fuses)
 fmt.Println(mustang.name) fmt.Println(fuses.description)
 fmt.Println(mustang.topSpeed) fmt.Println(fuses.count)
}
}
This is supposed to → Mustang Cobra
be 50 km/h higher! 225
This is supposed to → Fuses
be doubled! 5
```

## MOVING OUR STRUCT TYPE TO A DIFFERENT PACKAGE



We're definitely starting to appreciate the convenience of this subscriber struct type. But the code in our main package is getting a little long. Can we move subscriber out to another package?

That should be easy to do. Find the `headfirstgo` directory within your Go workspace, and create a new directory in there to hold a package named `magazine`. Within `magazine`, create a file named `magazine.go`.

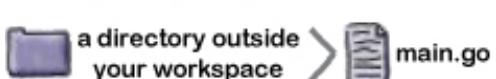


Be sure to add a package `magazine` declaration at the top of `magazine.go`. Then, copy the `subscriber` struct definition from your existing code, and paste it into `magazine.go`.

```
package magazine

We'll try pasting the { type subscriber struct {
 name string
 rate float64
 active bool
}
without any changes. }
```

Next, let's create a program to try out the new package. Since we're just experimenting for now, let's not create a separate package folder for this code; we'll just run it using the `go run` command. Create a file named `main.go`. You can save it in any directory you want, but make sure you save it *outside* your Go workspace, so it doesn't interfere with any other packages.



(You can move this code into your Go workspace later, if you want, as long as you create a separate package directory for it.)

Within `main.go`, save this code, which simply creates a new `subscriber` struct and accesses one of its fields.

There are two differences from the previous examples. First, we need to import the `magazine` package at the top of the file. Second, we need to use `magazine.subscriber` as the type name, since it belongs to another package now.

```
package main
import (
 "fmt" Import packages we need...
 "magazine" ...including our new
 "github.com/headfirstgo/magazine" package.
)
Type name needs to be prefixed
func main() { with the package name now.
 var s magazine.subscriber
 s.rate = 4.99
 fmt.Println(s.rate)
}
```

## A DEFINED TYPE'S NAME MUST BE CAPITALIZED TO BE EXPORTED

Let's see if our experimental code can still access the `subscriber` struct type in its new package. In your terminal, change into the directory where you saved `main.go`, then enter `go run main.go`.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
./main.go:9:18: cannot refer to unexported name magazine.subscriber
./main.go:9:18: undefined: magazine.subscriber
```

We get a couple errors, but here's the important one: `cannot refer to unexported name magazine.subscriber`.

Go type names follow the same rule as variable and function names: if the name of variable, function, or type begins with a capital letter, it is considered *exported* and can be accessed from outside the package it's declared in. But our `subscriber` type name begins with a lower-case letter. That means it can only be used within the `magazine` package.

Well, that seems like an easy fix. We'll just open our `magazine.go` file and capitalize the name of the defined type. Then, we'll open `main.go` and capitalize the names of any references to that type. (There's just one right now.)

**For a type to be accessed outside the package it's defined in, it must be exported: its name must begin with a capital letter.**



```
package magazine
type Subscriber struct {
 name string
 rate float64
 active bool
}
```



```
package main
import (
 "fmt"
 "github.com/headfirstgo/magazine"
)
func main() {
 var s magazine.Subscriber
 s.rate = 4.99
 fmt.Println(s.rate)
}
```

If we try running the updated code with `go run main.go`, we no longer get the error saying that the `magazine.Subscriber` type is unexported. So that seems to be fixed. But we get a couple new errors in its place...

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

## STRUCT FIELD NAMES MUST BE CAPITALIZED TO BE EXPORTED

With the `Subscriber` type name capitalized, we seem to be able to access it from the `main` package. But now we're getting an error saying that we can't refer to the `rate` field, because *that* is unexported.

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

Even if a struct type is exported from a package, its fields will be *unexported* if their names don't begin with a capital letter. Let's try capitalizing `Rate` (in both `magazine.go` and `main.go`)...

**Struct field names must also be capitalized if you want to export them from their package.**



magazine.go

```
package magazine

type Subscriber struct {
 name string
 Capitalize. → Rate float64
 active bool
}
```



main.go

```
package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)

func main() {
 var s magazine.Subscriber
 s.Rate = 4.99 ← Capitalize.
 fmt.Println(s.Rate) ← Capitalize.
}
```

```
Shell Edit View Window Help
$ go run main.go
4.99
```

Run `main.go` again, and you'll see that everything works this time. Now that they're exported, we can access the `Subscriber` type *and* its `Rate` field from the `main` package.

Notice that the code worked even though the `name` and `active` fields were still unexported. You can have a mixture of exported and unexported fields within a single struct type, if you want.

That's probably not advisable in the case of the `Subscriber` type, though. It wouldn't make sense to be able to access the subscription rate from other packages, but not the name or address. So let's go back into `magazine.go` and export the other fields as well. Simply capitalize their names: `Name` and `Active`.



magazine.go

```
package magazine

type Subscriber struct {
 Capitalize. → Name string
 Rate float64
 Capitalize. → Active bool
}
```

## STRUCT LITERALS

The code to define a struct and then assign values to its fields one by one can get a bit tedious:

```
var subscriber magazine.Subscriber
subscriber.Name = "Aman Singh"
subscriber.Rate = 4.99
subscriber.Active = true
```

So, just as with slices and maps, Go offers **struct literals** to let you create a struct and set its fields at the same time.

The syntax looks similar to a map literal. The type is listed first, followed by curly braces. Within the braces, you can specify values for some or all of the struct fields, using the field name, a colon, and then the value. If you specify multiple fields, separate them with commas.

```
myCar := car{Name: "Corvette", topSpeed: 337}
```

Above, we showed some code that creates a `Subscriber` struct and sets its fields, one by one. This code does the same thing in a single line, using a struct literal:

```
subscriber := magazine.Subscriber{Name: "Aman Singh", Rate: 4.99, Active: true}
fmt.Println("Name:", subscriber.Name)
fmt.Println("Rate:", subscriber.Rate)
fmt.Println("Active:", subscriber.Active)
```

Name: Aman Singh  
Rate: 4.99  
Active: true

You may have noticed that for most of the chapter, we've had to use long-form declarations for struct variables (unless the struct was being returned from a function). Struct literals allow us to use short variable declarations for a struct we've just created.

You can omit some or even all of the fields from the curly braces. Omitted fields will be set to the zero value for their type.

```
subscriber := magazine.Subscriber{Rate: 4.99}
fmt.Println("Name:", subscriber.Name)
fmt.Println("Rate:", subscriber.Rate)
fmt.Println("Active:", subscriber.Active)
```

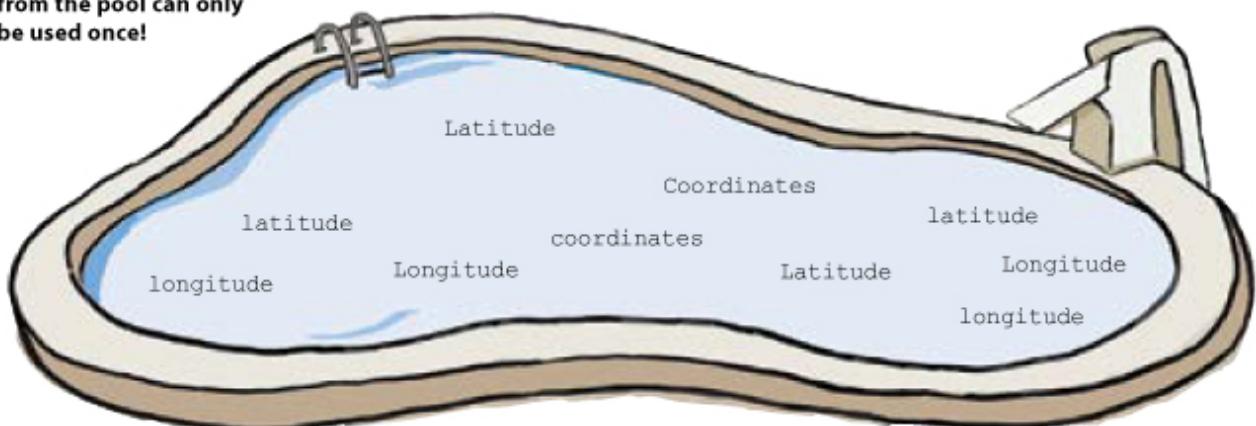
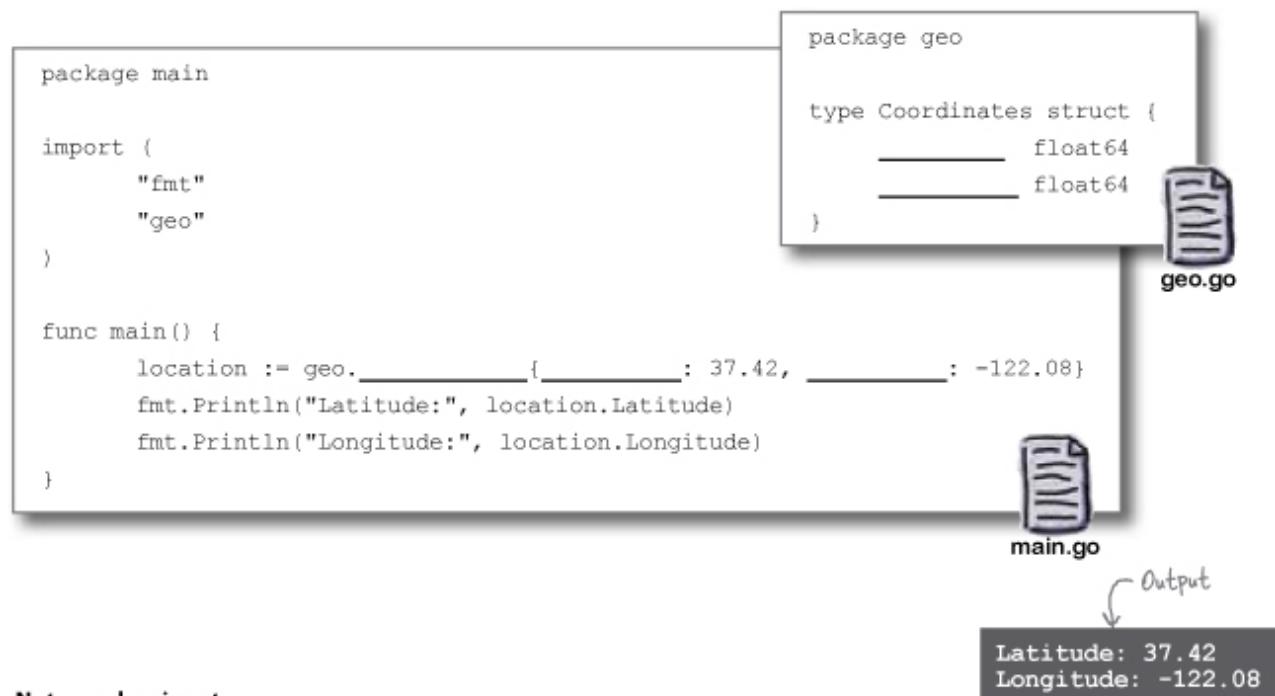
Name:  
Rate: 4.99  
Active: false



## POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in this code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a program that will run and produce the output

shown.



## CREATING AN EMPLOYEE STRUCT TYPE

Answers on page 22.



This new magazine package is working out great! Just a couple more things before we can publish our first issue... We need an employee struct type to track the names and salaries of our employees. And we need to store mailing addresses for both employees and subscribers.

Adding an `Employee` struct type should be pretty easy. We'll just add it to the `magazine` package, alongside the `subscriber` type. In `magazine.go`, define a new `Employee` type, with a

struct underlying type. Give the struct type a `Name` field with a type of `string`, and a `Salary` field with a type of `float64`. Be sure to capitalize the type name *and* all the fields, so that they're exported from the `magazine` package.

We can update the `main` function in `main.go` to try the new type out. First, declare a variable with the type `magazine.Employee`. Then assign values of the appropriate type to each of the fields. Finally, print those values out.



```
package magazine

type Subscriber struct {
 Name string
 Rate float64
 Active bool
}
 Capitalize the name, so
 it's exported.
type Employee struct {
 Name string
 Salary float64
}
 Export field names, too.
```



```
package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)
 Try creating an Employee value.
func main() {
 var employee magazine.Employee
 employee.Name = "Joy Carr"
 employee.Salary = 60000
 fmt.Println(employee.Name)
 fmt.Println(employee.Salary)
}
```

If you execute `go run main.go` from your terminal, it should run, create a new `magazine.Employee` struct, set its field values, and then print those values out.

```
Joy Carr
60000
```

## CREATING AN ADDRESS STRUCT TYPE

Next, we need to track mailing addresses for both the `Subscriber` and `Employee` types. We're going to need fields for the street address, city, state, and postal code (zip code).

We could add separate fields to both the `Subscriber` and `Employee` types, like this:

```
type Subscriber struct {
 Name string
 Rate float64
 Active bool
}
 If we added fields here...
 Street string
 City string
 State string
 PostalCode string
}

type Employee struct {
 Name string
 Salary float64
}
 We'd have to repeat them here...
 Street string
 City string
 State string
 PostalCode string
}
```

...But mailing addresses are going to have the same format, no matter what type they belong to. It's a pain to have to repeat all those fields between multiple types.

Struct fields can hold values of any type, *including other structs*. So instead, let's try building an `Address` struct type, and then adding an `Address` field on the `Subscriber` and `Employee` types. That will save us some effort now, and ensure consistency between the types later if we have to change the address format.

We'll create just the `Address` type first, so we can ensure it's working correctly. Place it in the `magazine` package, alongside the `Subscriber` and `Employee` types. Then, replace the code in `main.go` with a few lines to create an `Address` and ensure its fields are accessible.



```
magazine.go
package magazine

// Subscriber and Employee
// code omitted...
type Address struct {
 Street string
 City string
 State string
 PostalCode string
}

main.go
package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)
func main() {
 var address magazine.Address
 address.Street = "123 Oak St"
 address.City = "Omaha"
 address.State = "NE"
 address.PostalCode = "68111"
 fmt.Println(address)
}
```

Type `go run main.go` in your terminal, and it should create an `Address` struct, populate its fields, and then print the whole struct out.

```
{123 Oak St Omaha NE 68111}
```

## ADDING A STRUCT AS A FIELD ON ANOTHER TYPE

Now that we're sure the `Address` struct type works by itself, let's add `HomeAddress` fields to the `Subscriber` and `Employee` types.

Adding a struct field that is itself a struct type is no different than adding a field of any other type. You provide a name for the field, followed by the field's type (which in this case will be a struct type).

Add a field named `HomeAddress` to the `Subscriber` struct. Make sure to capitalize the field name, so that it's accessible from outside the `magazine` package. Then specify the field type, which is `Address`.

Add a `HomeAddress` field to the `Employee` type as well.



```
package magazine

type Subscriber struct {
 Name string
 Rate float64
 Active bool
 Capitalized
 field name. → HomeAddress Address
}
 ↑
 The field type.

type Employee struct {
 Name string
 Salary float64
 Capitalized
 field name. → HomeAddress Address
}
 ↑
 The field type.

type Address struct {
 // Fields omitted
}
```

## SETTING UP A STRUCT WITHIN ANOTHER STRUCT

Now let's see if we can populate the fields of the `Address` struct *within* the `Subscriber` struct. There are a couple ways to go about this.

The first approach is to create an entirely separate `Address` struct, and then use it to set the entire `Address` field of the `Subscriber` struct. Here's an update to `main.go` that follows this approach.

```
main.go package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)

func main() {
 Create the
 Subscriber struct
 that the Address
 will belong to.
 address := magazine.Address{Street: "123 Oak St",
 City: "Omaha", State: "NE", PostalCode: "68111"}
 subscriber := magazine.Subscriber{Name: "Aman Singh"}
 subscriber.HomeAddress = address ← Set the HomeAddress field.
 fmt.Println(subscriber.HomeAddress)
}

 ↑ Create an Address value
 and populate its fields.

 ↑ Print the HomeAddress field.
```

```
{123 Oak St Omaha NE 68111}
```

Type `go run main.go` in your terminal, and you'll see the subscriber's `HomeAddress` field has been set to the struct you built.

Another approach is to set the fields of the inner struct *through* the outer struct.

When a `Subscriber` struct is created, its `HomeAddress` field is already set: it's an `Address` struct with all its fields set to their zero values. If we print `HomeAddress` using the "%#v" verb for

`fmt.Println`, it will print the struct as it would appear in Go code, that is, as a struct literal. We'll see that each of the `Address` fields is set to an empty string, which is the zero value for the `string` type.

```
subscriber := magazine.Subscriber{}
fmt.Printf("%#v\n", subscriber.HomeAddress)
```

The field is already set  
as a new Address struct.

Each of the `Address` struct's fields is  
set to an empty string (which is the  
zero value for strings).

```
magazine.Address{Street:"", City:"", State:"", PostalCode:""}
```

If `subscriber` is a variable that contains a `Subscriber` struct, then when you type `subscriber.HomeAddress`, you'll get an `Address` struct, even if you haven't explicitly set `HomeAddress`.

You can use this fact to "chain" dot operators together so you can access the fields of the `Address` struct. Simply type `subscriber.HomeAddress` to access the `Address` struct, followed by *another* dot operator and the name of the field you want to access on that `Address` struct.

```
subscriber.HomeAddress.City
```

This part gives you an Address struct.  
This part accesses the City field on that Address struct.

This works both for assigning values to the inner struct's fields...

```
subscriber.HomeAddress.PostalCode = "68111"
```

...And for retrieving those values again later.

```
fmt.Println("Postal Code:", subscriber.HomeAddress.PostalCode)
```

Here's an update to `main.go` that uses dot operator chaining. First we store a `Subscriber` struct in the `subscriber` variable. That will automatically create an `Address` struct in `subscriber`'s `HomeAddress` field. We set values for `subscriber.HomeAddress.Street`, `subscriber.HomeAddress.City`, etc., and then print those values out again.

Then we store an `Employee` struct in the `employee` variable, and do the same for its `HomeAddress` struct.



main.go

```
package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)

func main() {
 subscriber := magazine.Subscriber{Name: "Aman Singh"}
 Set the fields of subscriber.HomeAddress. {
 subscriber.HomeAddress.Street = "123 Oak St"
 subscriber.HomeAddress.City = "Omaha"
 subscriber.HomeAddress.State = "NE"
 subscriber.HomeAddress.PostalCode = "68111"
 fmt.Println("Subscriber Name:", subscriber.Name)
 fmt.Println("Street:", subscriber.HomeAddress.Street)
 Retrieve field values from subscriber.HomeAddress. {
 fmt.Println("City:", subscriber.HomeAddress.City)
 fmt.Println("State:", subscriber.HomeAddress.State)
 fmt.Println("Postal Code:", subscriber.HomeAddress.PostalCode)

 Set the fields of employee.HomeAddress. {
 employee := magazine.Employee{Name: "Joy Carr"}
 employee.HomeAddress.Street = "456 Elm St"
 employee.HomeAddress.City = "Portland"
 employee.HomeAddress.State = "OR"
 employee.HomeAddress.PostalCode = "97222"
 fmt.Println("Employee Name:", employee.Name)
 Retrieve field values from employee.HomeAddress. {
 fmt.Println("Street:", employee.HomeAddress.Street)
 fmt.Println("City:", employee.HomeAddress.City)
 fmt.Println("State:", employee.HomeAddress.State)
 fmt.Println("Postal Code:", employee.HomeAddress.PostalCode)
 }
}
```

```
Subscriber Name: Aman Singh
Street: 123 Oak St
City: Omaha
State: NE
Postal Code: 68111
Employee Name: Joy Carr
Street: 456 Elm St
City: Portland
State: OR
Postal Code: 97222
```

Type `go run main.go` in your terminal, and the program will print out the completed fields of both `subscriber.HomeAddress` and `employee.HomeAddress`.

## ANONYMOUS STRUCT FIELDS

The code to access the fields of an inner struct through its outer struct can be a bit tedious, though. You have to write the field name of the inner struct (`HomeAddress`) each time you want to access any of the fields it contains.

```
subscriber := magazine.Subscriber{Name: "Aman Singh"}
subscriber.HomeAddress.Street = "123 Oak St"
subscriber.HomeAddress.City = "Omaha"
subscriber.HomeAddress.State = "NE"
subscriber.HomeAddress.PostalCode = "68111"

Have to write the field name of the inner struct...
...Only then can you access its fields.
```

Go allows you to define **anonymous fields**: struct fields that have no name of their own, just a type. We can use an anonymous field to make our inner struct easier to access.

Here's an update to the `Subscriber` and `Employee` types to convert their `HomeAddress` fields to an anonymous field. To do this, we simply remove the field name, leaving only the type.



`magazine.go` package magazine

```
type Subscriber struct {
 Name string
 Rate float64
 Active bool
}

Delete the field name ("HomeAddress"), leaving only the type. → Address
}

type Employee struct {
 Name string
 Salary float64
}

Delete the field name ("HomeAddress"), leaving only the type. → Address
}

type Address struct {
 // Fields omitted
}
```

When you declare an anonymous field, you can use the field's type name as if it were the name of the field. So `subscriber.Address` and `employee.Address` in the code below still access the `Address` structs:

```
subscriber := magazine.Subscriber{Name: "Aman Singh"}
subscriber.Address.Street = "123 Oak St" ← Access the inner struct field through
subscriber.Address.City = "Omaha" its new "name", which is "Address".
fmt.Println("Street:", subscriber.Address.Street)
fmt.Println("City:", subscriber.Address.City)
employee := magazine.Employee{Name: "Joy Carr"}
employee.Address.State = "OR"
employee.Address.PostalCode = "97222"
fmt.Println("State:", employee.Address.State)
fmt.Println("Postal Code:", employee.Address.PostalCode)
```

Street: 123 Oak St  
City: Omaha  
State: OR  
Postal Code: 97222

## EMBEDDING STRUCTS

But anonymous fields offer much more than just the ability to skip providing a name for a field in a struct definition.

An inner struct that is stored within an outer struct using an anonymous field is said to be **embedded** within the outer struct. You can access fields for an embedded struct as if they belonged to the outer struct.

So now that the `Address` struct type is embedded within the `Subscriber` and `Employee` struct types, you don't have to write out `subscriber.Address.City` to get at the `city` field; you can just write `subscriber.City`. You don't need to write `employee.Address.State`; you can just write `employee.State`.

Here's one last version of `main.go`, updated to treat `Address` as an embedded type. You can

write the code as if there was no `Address` type at all; it's like the `Address` fields belong to the struct type they're embedded within.



**main.go**

```
package main

import (
 "fmt"
 "github.com/headfirstgo/magazine"
)

func main() {
 Set the fields of
 the Address as if
 they were defined
 on Subscriber. subscriber := magazine.Subscriber{Name: "Aman Singh"}
 subscriber.Street = "123 Oak St"
 subscriber.City = "Omaha"
 subscriber.State = "NE"
 subscriber.PostalCode = "68111"
 Retrieve Address
 fields through the
 Subscriber. fmt.Println("Street:", subscriber.Street)
 fmt.Println("City:", subscriber.City)
 fmt.Println("State:", subscriber.State)
 fmt.Println("Postal Code:", subscriber.PostalCode)

 Set the fields of
 the Address as if
 they were defined
 on Employee. employee := magazine.Employee{Name: "Joy Carr"}
 employee.Street = "456 Elm St"
 employee.City = "Portland"
 employee.State = "OR"
 employee.PostalCode = "97222"
 Retrieve Address
 fields through the
 Employee. fmt.Println("Street:", employee.Street)
 fmt.Println("City:", employee.City)
 fmt.Println("State:", employee.State)
 fmt.Println("Postal Code:", employee.PostalCode)
}
```

```
Street: 123 Oak St
City: Omaha
State: NE
Postal Code: 68111
Street: 456 Elm St
City: Portland
State: OR
Postal Code: 97222
```

**You can access fields for an embedded struct as if they belong to the outer struct.**

**OUR DEFINED TYPES ARE COMPLETE!**



These struct types you've made for us are great! No more passing around bunches of variables just to represent one subscriber. Everything we need is stored in one convenient bundle. Thanks to you, we're ready to fire up the presses and mail out our first issue!

Nice work! You've defined `Subscriber` and `Employee` struct types, and embedded an `Address` struct in each of them. You've found a way to represent all the data the magazine needed!

You're still missing an important aspect to defined types, though. In previous chapters, you've used types like `time.Time` and `strings.Replacer` that have *methods*: functions that you can call *on* their values. But you haven't learned how to define methods for your own types yet. Don't worry; we'll learn all about it in the next chapter!



Here's a source file from the `geo` package, which we saw in a previous exercise. Your goal is to make the code in `main.go` work correctly. But here's the catch: you need to do it by adding just *two* fields to the `Landmark` struct type within `geo.go`.

```
package geo

type Coordinates struct {
 Latitude float64
 Longitude float64
}

type Landmark struct {
 
 
}
```

Add two fields here!

```
package main

import (
 "fmt"
 "geo"
)

func main() {
 location := geo.Landmark{}
 location.Name = "The Googleplex"
 location.Latitude = 37.42
 location.Longitude = -122.08
 fmt.Println(location)
}
```



Output → {The Googleplex {37.42 -122.08}}



## YOUR GO TOOLBOX

That's it for Chapter 7! You've added maps to your toolbox.

Arrays

Slices

Maps

Structs

A struct is a value that's constructed by joining together other values of different types.

The separate values that form a struct are known as fields.

Each field has a name and a type.

## Defined Types

Type definitions allow you to create new types of your own.

Each defined type is based on an underlying type that determines how values are stored.

Defined types can use any type as an underlying type, although structs are most commonly used.



## BULLET POINTS

- You can declare a variable with a struct type. To specify a struct type, use the `struct` keyword, followed by a list of field names and types within curly braces.

```
var myStruct struct {
 field1 string
 field2 int
}
```

- Writing struct types repeatedly can get tedious, so it's usually best to define a type with an underlying struct type. Then the defined type can be used for variables, function parameters or return values, etc.

```
type myType struct {
```

```
 field1 string
}

var myVar myType
```

- Struct fields are accessed via the dot operator.

```
myVar.field1 = "value"
fmt.Println(myVar.field1)
```

- If a function needs to modify a struct or if a struct is large, it should be passed to the function as a pointer.
- Types will only be exported from the package they're defined in if their name begins with a capital letter.
- Likewise, struct fields will not be accessible outside their package unless their name is capitalized.
- Struct literals let you create a struct and set its fields at the same time.

```
myVar := myType{field1: "value"}
```

- Adding a struct field with no name, only a type, defines an anonymous field.
- An inner struct that is added as part of an outer struct using an anonymous field is said to be embedded within the outer struct.
- You can access the fields of an embedded struct as if they belong to the outer struct.



At the right is a program that creates a struct variable which holds a pet's name (a `string`) and age (an `int`). Fill in the blanks so that the code will produce the output shown.

```

package main

import "fmt"

func main() {
 var pet struct {
 name string
 age int
 }
 pet.name = "Max"
 pet.age = 5
 fmt.Println("Name:", pet.name)
 fmt.Println("Age:", pet.age)
}

```

Name: Max  
Age: 5

## CODE MAGNETS SOLUTION

```

package main

import "fmt"

type student struct {
 name string
 grade float64
}

func printInfo(s student) {
 fmt.Println("Name:", s.name)
 fmt.Printf("Grade: %0.1f\n", s.grade)
}

func main() {
 var s student
 s.name = "Alonzo Cole"
 s.grade = 92.3
 printInfo(s)
}

```

Output  
Name: Alonzo Cole  
Grade: 92.3



The two programs below weren't working quite right. The `nitroBoost` function in the left-hand program is supposed to add 50 kilometers/hour to a `car`'s top speed, but it wasn't. And the `doublePack` function in the right-hand program is supposed to double a `part` value's `count` field, but it wasn't, either.

Fixing both programs was simply a matter of updating the functions to accept pointers, and updating the function calls to pass pointers. The code within the functions that updates the struct fields doesn't need to be changed; the code to access a field through a pointer to a struct is the same as the code to access a field on the struct directly.

```
package main

import "fmt"

type car struct {
 name string
 topSpeed float64
}

func nitroBoost(c *car) {
 c.topSpeed += 50
}

func main() {
 var mustang car
 mustang.name = "Mustang Cobra"
 mustang.topSpeed = 225
 nitroBoost(&mustang)
 fmt.Println(mustang.name)
 fmt.Println(mustang.topSpeed)
}

Fixed; it's 50 km/h higher. → Mustang Cobra 275
```

```
package main

import "fmt"

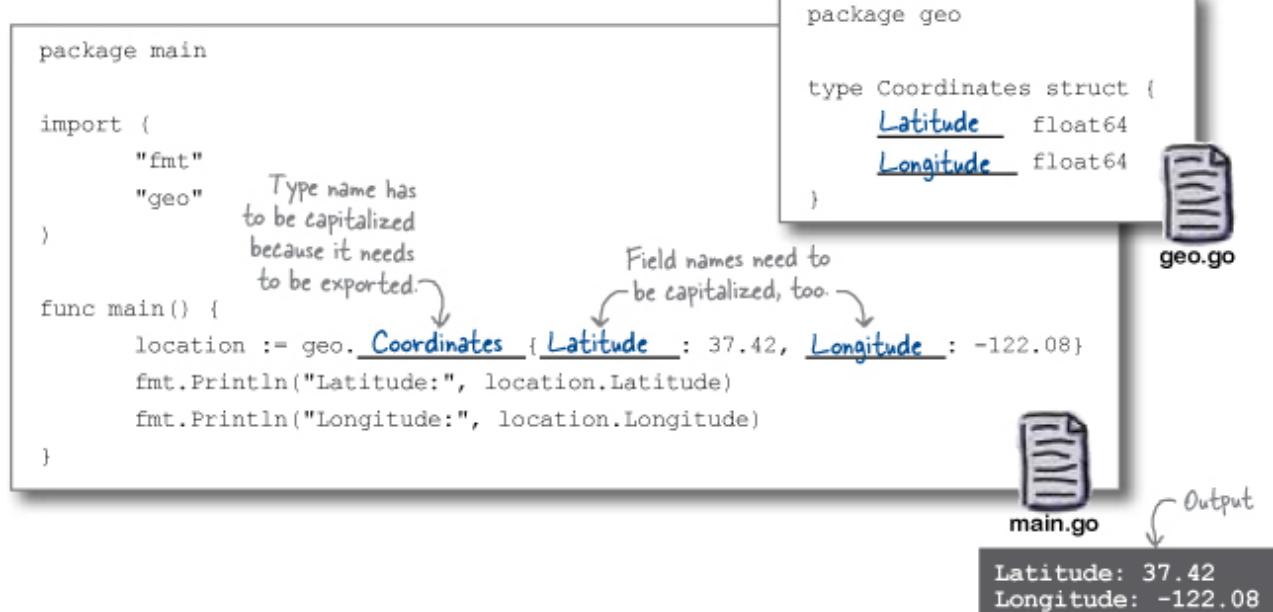
type part struct {
 description string
 count int
}

func doublePack(p *part) {
 p.count *= 2
}

func main() {
 var fuses part
 fuses.description = "Fuses"
 fuses.count = 5
 doublePack(&fuses)
 fmt.Println(fuses.description)
 fmt.Println(fuses.count)
}

Fixed; it's double the original value. → Fuses 10
```

## POOL PUZZLE SOLUTION





The `geo.go` source file is from the `geo` package, which we saw in a previous exercise. Your goal was to make the code in `main.go` work correctly, by adding just *two* fields to the `Landmark` struct type within `geo.go`.

```
package geo

type Coordinates struct {
 Latitude float64
 Longitude float64
}

type Landmark struct {
 Name string
 Coordinates
}
```



```
package main

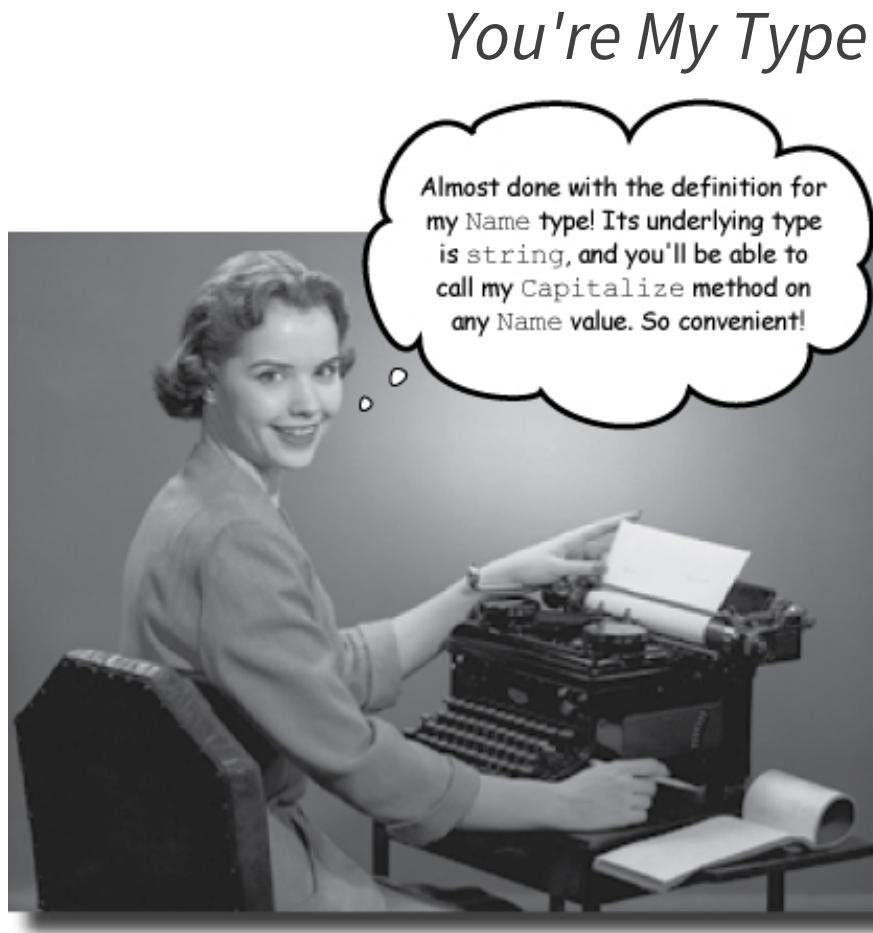
import (
 "fmt"
 "geo"
)

func main() {
 location := geo.Landmark()
 location.Name = "The Googleplex"
 location.Latitude = 37.42
 location.Longitude = -122.08
 fmt.Println(location)
}
```



Output → {The Googleplex {37.42 -122.08}}

## 9 defined types



**There's more to learn about defined types.** In the previous chapter, we showed you how to define a type with a struct underlying type. What we *didn't* show you was that you can use *any* type as an underlying type.

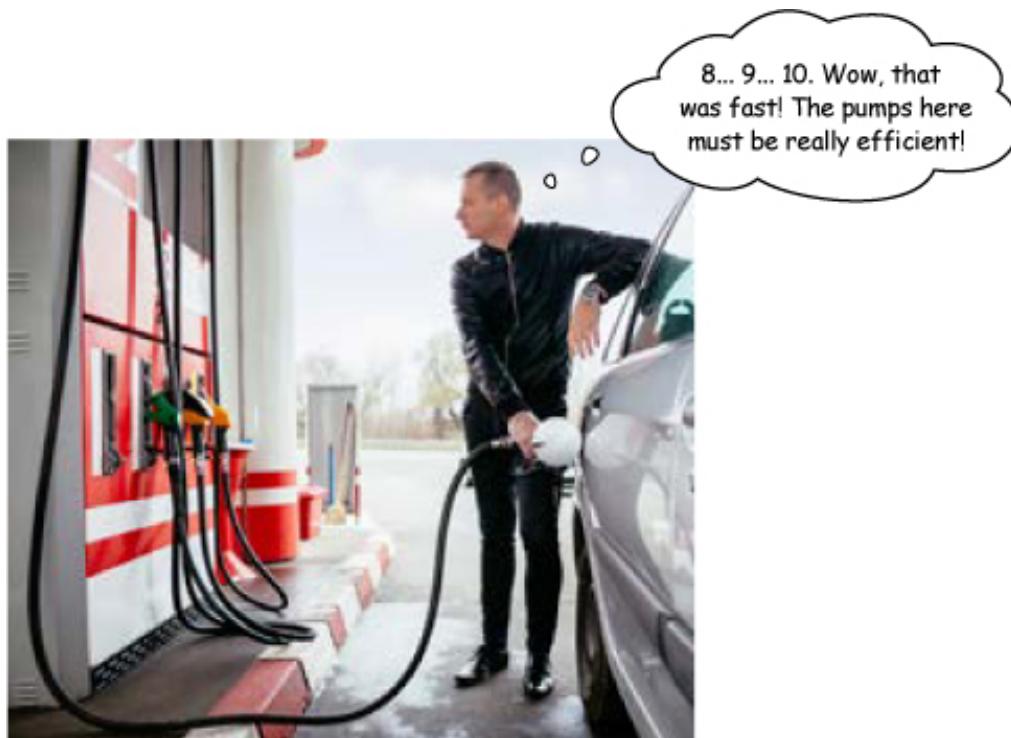
And do you remember methods — the special kind of function that's associated with values of a particular type? We've been calling methods on various values throughout the book, but we haven't shown you how to define your *own* methods.

In this chapter, we're going to fix all of that. Let's get started!

## TYPE ERRORS IN REAL LIFE

If you live in the U.S.A., you are probably used to the quirky system of measurement used here. At gas stations, for example, fuel is sold by the "gallon", a volume nearly four times the size of the "liter" used in much of the rest of the world.

Steve is an American, renting a car in another country. He pulls into a gas station to refuel. He intends to purchase 10 gallons, figuring that will be enough to reach his hotel in another city.



He gets back on the road, but only gets one-fourth of the way to his destination before running out of fuel.

If Steve had looked at the labels on the gas pump more closely, he would have realized that it was measuring the fuel in liters, not gallons, and that he needed to purchase 37.85 liters to get the equivalent of 10 gallons.



When you have a number, it's best to be certain what that number is measuring. You want to know if it's liters or gallons, kilograms or pounds, dollars or yen.

## DEFINED TYPES WITH UNDERLYING BASIC TYPES

If you have the following variable:

```
var fuel float64 = 10
```

...Does that represent 10 gallons or 10 liters? The person who wrote that declaration knows, but no one else does, not for sure.

You can use Go's defined types to make it clear what a value is to be used for. Although defined types most commonly use structs as their underlying types, they *can* be based on `int`, `float64`, `string`, `bool`, or any other type.

**Go defined types most often use structs as their underlying types, but they can also be based on ints, strings, booleans, or any other type.**

Here's a program that defines two new types, `Liters` and `Gallons`, both with an underlying type of `float64`. These are defined at package level, so that they're available within any function in the current package.

Within the `main` function, we declare a variable with a type of `Gallons`, and another with a type of `Liters`. We assign values to each variable, and then print them out.

```
package main

import "fmt"

Define two new types, each with a type of float64
an underlying type of float64. { type Liters float64
 type Gallons float64

func main() {
 var carFuel Gallons
 Define a variable with a
 type of Gallons.
 var busFuel Liters
 Define a variable with a
 type of Liters.
 carFuel = Gallons(10.0)
 Convert a float64 to Gallons.
 busFuel = Liters(240.0)
 Convert a float64 to Liters.
 fmt.Println(carFuel, busFuel)
}
 10 240
```

Once you've defined a type, you can do a conversion to that type from any value of the underlying type. As with any other conversion, you write the type you want to convert to, followed by the value you want to convert in parentheses.

If we had wanted, we could have written short variable declarations in the code above

using type conversions:

```
carFuel := Gallons(10.0)
busFuel := Liters(240.0)
```

You can assign a value of the underlying type to a variable using the defined type; the conversion will be performed automatically.

```
var carFuel Gallons
var busFuel Liters
carFuel = 10.0 ← Automatically converts the float64 to Gallons.
busFuel = 240.0 ← Automatically converts the float64 to Liters.
```

If you have a variable that uses a defined type, you *cannot* assign a value of a different defined type to it, even if the other type has the same underlying type. This helps protect developers from confusing the two types.

```
carFuel = Liters(240.0)
busFuel = Gallons(10.0)
```

Errors → cannot use Liters(240) (type Liters) as type Gallons in assignment  
cannot use Gallons(10) (type Gallons) as type Liters in assignment

You can convert between types that have the same underlying type. So `Liters` can be converted to `Gallons` and vice-versa, because both have an underlying type of `float64`. But Go only considers the value of the underlying type when doing a conversion; there is no difference between `Gallons(Liters(240.0))` and `Gallons(240.0)`. Simply converting raw values from one type to another defeats the protection against conversion errors that types are supposed to provide.

```
carFuel = Gallons(Liters(40.0)) ← 40 liters do NOT equal 40 gallons!
busFuel = Liters(Gallons(63.0)) ← 63 gallons do NOT equal 63 liters!
fmt.Printf("Gallons: %0.1f Liters: %0.1f\n", carFuel, busFuel)
```

Legal, but incorrect! → Gallons: 40.0 Liters: 63.0

Instead, you'll want to perform whatever operations are necessary to convert the underlying type value to a value appropriate for the type you're converting to.

A quick web search shows that one liter equals roughly 0.264 gallons, and that one gallon equals roughly 3.785 liters. We can multiply by these conversion rates to convert from `Gallons` to `Liters`, and vice versa.

```

carFuel = Gallons(Liters(40.0) * 0.264) ← Convert from Liters to Gallons.
busFuel = Liters(Gallons(63.0) * 3.785) ← Convert from Gallons to Liters.
fmt.Printf("Gallons: %0.1f Liters: %0.1f\n", carFuel, busFuel)

```

Properly-converted values. → **Gallons: 10.6 Liters: 238.5**

## DEFINED TYPES AND OPERATORS

A defined type supports all the same operations as its underlying type. Types based on `float64`, for example, support arithmetic operators like `+`, `-`, `*`, and `/`, as well as comparison operators like `==`, `>`, and `<`.

|                                                        |            |
|--------------------------------------------------------|------------|
| <code>fmt.Println(Liters(1.2) + Liters(3.4))</code>    | <b>4.6</b> |
| <code>fmt.Println(Gallons(5.5) - Gallons(2.2))</code>  | 3.3        |
| <code>fmt.Println(Liters(2.2) / Liters(1.1))</code>    | 2          |
| <code>fmt.Println(Gallons(1.2) == Gallons(1.2))</code> | true       |
| <code>fmt.Println(Liters(1.2) &lt; Liters(3.4))</code> | true       |
| <code>fmt.Println(Liters(1.2) &gt; Liters(3.4))</code> | false      |

A type based on an underlying type of `string`, however, would support `+`, `==`, `>`, and `<`, but not `-`, because `-` is not a valid operator for strings.

```

// package and import statements omitted
type Title string ← Define a type with an underlying type of "string".

func main() {
 fmt.Println>Title("Alien") == Title("Alien"))
 These work... { fmt.Println>Title("Alien") < Title("Zodiac"))
 { fmt.Println>Title("Alien") > Title("Zodiac"))
 { fmt.Println>Title("Alien") + "s")
 This doesn't! → fmt.Println>Title("Jaws 2") - " 2")
}

```

↓ Error

**invalid operation:**  
**Title("Jaws 2") - " 2"**  
**(operator - not defined on string)**

A defined type can be used in operations together with values of its underlying type:

|                                                |            |
|------------------------------------------------|------------|
| <code>fmt.Println(Liters(1.2) + 3.4)</code>    | <b>4.6</b> |
| <code>fmt.Println(Gallons(5.5) - 2.2)</code>   | 3.3        |
| <code>fmt.Println(Gallons(1.2) == 1.2)</code>  | true       |
| <code>fmt.Println(Liters(1.2) &lt; 3.4)</code> | true       |

But defined types *cannot* be used in operations together with values of a different type, even if the other type has the same underlying type. Again, this is to protect developers from accidentally mixing the two types.

```

fmt.Println(Liters(1.2) + Gallons(3.4))
fmt.Println(Gallons(1.2) == Liters(1.2))

```

If you want to add a value in `Liters` to a value in `Gallons`, you'll need to convert one type to match the other first.

Errors

```
invalid operation: Liters(1.2) + Gallons(3.4)
(mismatched types Liters and Gallons)
invalid operation: Gallons(1.2) == Liters(1.2)
(mismatched types Gallons and Liters)
```



## POOL PUZZLE

Your **job** is to take code snippets from the pool and place them into the blank lines in this code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a program that will run and produce the output shown.

```
package main

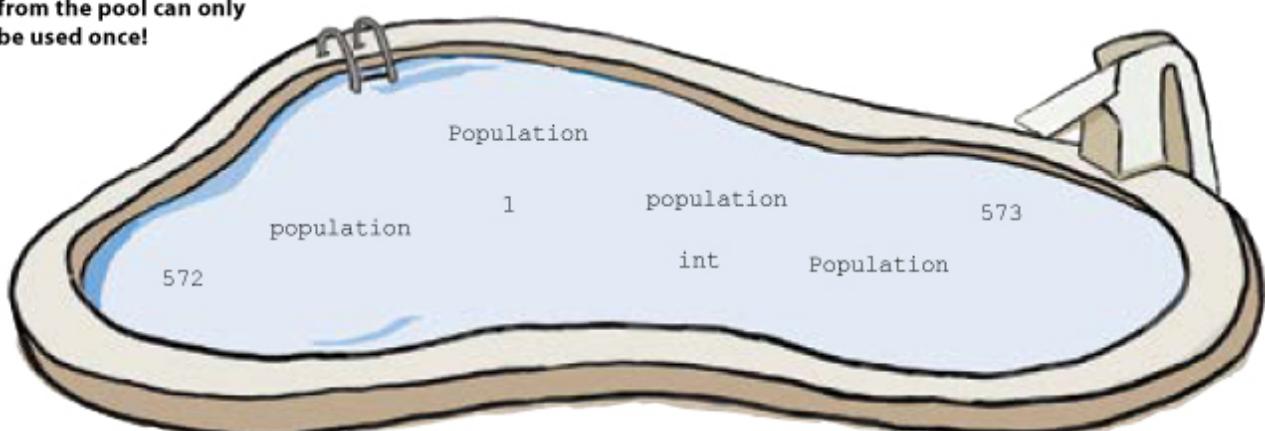
import "fmt"

type _____ int

func main() {
 var _____ Population
 population = _____(_____)
 fmt.Println("Sleepy Creek County population:", population)
 fmt.Println("Congratulations, Kevin and Anna! It's a girl!")
 population += _____
 fmt.Println("Sleepy Creek County population:", population)
}
```

Output → Sleepy Creek County population: 572  
Congratulations, Kevin and Anna! It's a girl!  
Sleepy Creek County population: 573

Note: each snippet from the pool can only be used once!



## CONVERTING BETWEEN TYPES USING FUNCTIONS

Suppose we wanted to take a car whose fuel level is measured in `Gallons` and refill it at a gas pump that measures in `Liters`. Or take a bus whose fuel is measured in `Liters` and refill it at a gas pump that measures in `Gallons`. To protect us from inaccurate measurements, Go will give us a compile error if we try to combine values of different types:

```
package main

import "fmt"

type Liters float64
type Gallons float64

func main() {
 carFuel := Gallons(1.2)
 busFuel := Liters(2.5)
 carFuel += Liters(8.0) ← Can't add a
 Liters value to
 a Gallons value!
 busFuel += Gallons(30.0) ← Can't add a
 Gallons value to a
 Liters value!
}

Errors → invalid operation: carFuel += Liters(8)
 (mismatched types Gallons and Liters)
invalid operation: busFuel += Gallons(20)
 (mismatched types Liters and Gallons)
```

In order to do operations with values of different types, we need to convert the types to match first. Previously, we demonstrated multiplying a `Liters` value by 0.264 and converted the result to `Gallons`. We also multiplied a `Gallons` value by 3.785 and converted the result to `Liters`.

```
carFuel = Gallons(Liters(40.0) * 0.264) ← Convert from Liters to Gallons.
busFuel = Liters(Gallons(63.0) * 3.785) ← Convert from Gallons to Liters.

// Imports, type declarations omitted is just over 1/4 the
// number of liters.
func ToGallons(l Liters) Gallons {
 return Gallons(l * 0.264) ← The number of gallons
 is just over 1/4 the
 number of liters.
}

func ToLiters(g Gallons) Liters {
 return Liters(g * 3.785) ← The number of liters is
 just under 4 times the
 number of gallons.
}

func main() {
 carFuel := Gallons(1.2) ← Convert Liters to
 Gallons before adding.
 busFuel := Liters(4.5) ← Convert Gallons to
 Liters before adding.
 carFuel += ToGallons(Liters(40.0)) ←
 busFuel += ToLiters(Gallons(30.0)) ←
 fmt.Printf("Car fuel: %0.1f gallons\n", carFuel)
 fmt.Printf("Bus fuel: %0.1f liters\n", busFuel)
}
```

Car fuel: 11.8 gallons  
 Bus fuel: 118.1 liters

We can create `ToGallons` and `ToLiters` functions that do the same thing, then call them to perform the conversion for us:

Gasoline isn't the only liquid we need to measure the volume of. There's cooking oil, bottles of soda, juice, etc. And so there are many more measures of volume than just liters and gallons. In the U.S.A. there are teaspoons, cups, quarts, and more. The metric system has other units of measure as well, but the milliliter (1/1000th of a liter) is the most commonly used.

Let's add a new type, `Milliliters`. Like the others, it will use `float64` as an underlying type.

```
type Liters float64
type Milliliters float64
type Gallons float64
```

Add a  
new type.

We're also going to want a way to convert from `Milliliters` to the other types. But if we start adding a function to convert from `Milliliters` to `Gallons`, we run into a problem: we can't have two `ToGallons` functions in the same package!

```
func ToGallons(l Liters) Gallons {
 return Gallons(l * 0.264)
}
func ToGallons(m Milliliters) Gallons { ←
 return Gallons(m * 0.000264)
}
```

Error → 12:31: `ToGallons` redeclared in this block  
previous declaration at `prog.go:9:26`

We can't add another function to convert from  
Milliliters to Gallons if it has the same name!

We could rename the two `ToGallons` functions to include the type they're converting from: `LitersToGallons` and `MillilitersToGallons`, respectively. But those names would be a pain to write out all the time, and as we start adding functions to convert between the other types, it becomes clear this isn't sustainable.

```
func LitersToGallons(l Liters) Gallons { ←
 return Gallons(l * 0.264)
} ← Eliminates the conflict, but the name is really long!
func MillilitersToGallons(m Milliliters) Gallons { ←
 return Gallons(m * 0.000264)
} ← Eliminates the conflict, but the name is really long!
func GallonsToLiters(g Gallons) Liters { ←
 return Liters(g * 3.785)
} ← Avoids conflict, but the name is really long!
func GallonsToMilliliters(g Gallons) Milliliters { ←
 return Milliliters(g * 3785.41)
}
```

## THERE ARE NO DUMB QUESTIONS

**Q:** I've seen other languages that support function *overloading*: they allow you to have multiple functions with the same name, as long as their

## parameter types are different. Doesn't Go support that?

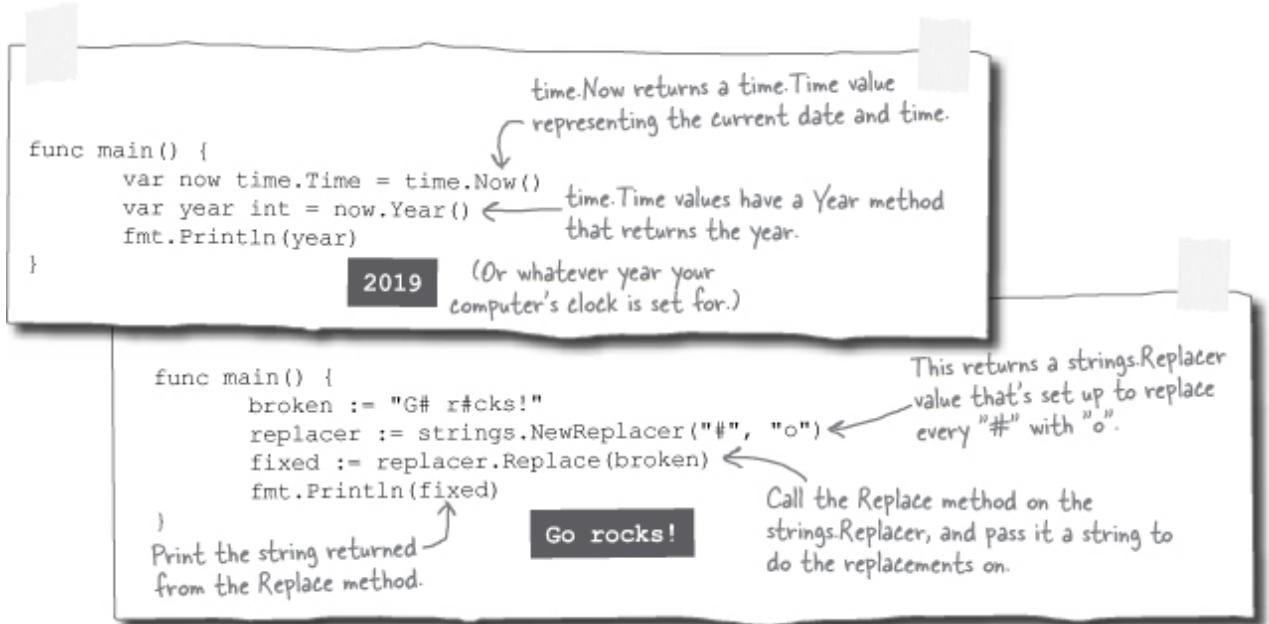
**A:** The Go maintainers get this question frequently too, and they answer it at <https://golang.org/doc/faq#overloading>: "Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice." The Go language is simplified by *not* supporting overloading, and so it doesn't support it. As you'll see later in the book, the Go team made similar decisions in other areas of the language, too; when they have to choose between simplicity and adding more features, they generally choose simplicity. But that's okay! As we'll see shortly, there are other ways to get the same benefits...



## FIXING OUR FUNCTION NAME CONFLICT USING METHODS

Remember way back in [Chapter 2](#), we introduced you to *methods*, which are functions associated with values of a given type? Among other things, we created a `time.Time` value

and called its `Year` method, and we created a `strings.Replacer` value and called its `Replace` method.



We can define methods of our own to help with our type conversion problem.

We're not allowed to have multiple functions named `ToGallons`, so we had to write long, cumbersome function names that incorporated the type we were converting:

```
LitersToGallons(Liters(2))
MillilitersToGallons(Milliliters(500))
```

But we *can* have multiple *methods* named `ToGallons`, as long as they're defined on separate types. Not having to worry about name conflicts will let us make our method names much shorter.

```
Liters(2).ToGallons()
Milliliters(500).ToGallons()
```

But let's not get ahead of ourselves. Before we can do anything else, we need to know how to define a method...

## DEFINING METHODS

A method definition is very similar to a function definition. In fact, there's really only one difference: you add one extra parameter, a **receiver parameter**, in parentheses *before* the function name.

As with any function parameter, you need to provide a name for the receiver parameter, followed by a type.

```
Receiver
parameter name Receiver
parameter type
func (m MyType) sayHi () {
 fmt.Println("Hi from", m)
}
```

To call a method you've defined, you write the value you're calling the method on, a dot, and the name of the method you're calling followed by parentheses. The value you're calling the method on is known as the method **receiver**.

The similarity between method calls and method definitions can help you remember the syntax: the receiver is listed first when *calling* a method, and the receiver parameter is listed first when *defining* a method.

```
value := MyType ("a MyType value")
value.sayHi()
Method receiver Method name
```

The name of the receiver parameter in the method definition isn't important, but its type is: the method you're defining becomes associated with all values of that type.

Below, we define a type named `MyType`, with an underlying type of `string`. Then, we define a method named `sayHi`. Because `sayHi` has a receiver parameter with a type of `MyType`, we'll be able to call the `sayHi` method on any `MyType` value. (Most developers would say that `sayHi` is defined "on" `MyType`.)

```
package main

import "fmt"

type MyType string
Define a new type.
Define a receiver parameter.
func (m MyType) sayHi () {
 fmt.Println("Hi")
}
The method will be defined on MyType.

func main () {
 value := MyType ("a MyType value")
 Create a MyType value.
 value.sayHi () ← Call sayHi on that value.
 anotherValue := MyType ("another value") ← Create another
 anotherValue.sayHi () ← Call sayHi on
 the new value.
}
```

Hi  
Hi

Once a method is defined on a type, it can be called on any value of that type.

Here, we create two different `MyType` values, and call `sayHi` on each of them.

## THE RECEIVER PARAMETER IS (PRETTY MUCH) JUST ANOTHER PARAMETER

The type of the receiver parameter is the type that the method becomes associated with. But aside from that, the receiver parameter doesn't get special treatment from Go. You can access its contents within the method block just like you would any other function parameter.

The code sample below is almost identical to the previous one, except that we've updated it to print the value of the receiver parameter. You can see the receivers in the resulting output.

```
package main

import "fmt"

type MyType string

func (m MyType) sayHi() {
 fmt.Println("Hi from", m)
}

func main() {
 value := MyType("a MyType value")
 value.sayHi()
 anotherValue := MyType("another value")
 anotherValue.sayHi()
}

Receivers passed to receiver parameter.

Value to call method on.

Value to call method on.

See receiver values in the output.


```

Go lets you name a receiver parameter whatever you want, but it's more readable if all the methods you define for a type have receiver parameters with the same name.

By convention, Go developers usually use a name consisting of a single letter — the first letter of the receiver's type name, in lower case. (This is why we used `m` as the name for our `MyType` receiver parameter.)

**The receiver parameter is Go's equivalent to "self" or "this" values in other languages.**

*THERE ARE NO DUMB QUESTIONS*

## **Q: Can I define new methods on *any* type?**

**A:** Only types that are defined in the same package where you define the method. That means no defining methods for types from someone else's `security` package from your `hacking` package, and no defining new methods on universal types like `int` or `string`.

## **Q: But I need to be able to use methods of my own with someone else's type!**

**A:** First you should consider whether a function would work well enough; a function can take any type you want as a parameter. But if you *really* need a value that has some methods of your own, plus some methods from a type in another package, you can make a struct type that embeds the other package's type as an anonymous field. We'll look at how that works in the next chapter.

## **Q: I've seen other languages where a method receiver was available in a method block in a special variable named `self` or `this`. Does Go do that?**

**A:** The receiver parameter is Go's equivalent to `self`/`this`. You can use receiver parameters in the same way, and there's no need for Go to reserve `self` or `this` as keywords! (You could even name your receiver parameter `this` if you wanted, but don't do that; the convention is to use the first letter of the receiver's type name instead.)

# A METHOD IS (PRETTY MUCH) JUST LIKE A FUNCTION

Aside from the fact that they're called on a receiver, methods are otherwise pretty similar to any other function.

As with any other function, you can define addition parameters within parentheses following the method name. These parameter variables can be accessed in the method block, along with the receiver parameter. When you call the method, you'll need to provide an argument for each parameter.

```
func (m MyType) MethodWithParameters(number int, flag bool) {
 fmt.Println(m)
 fmt.Println(number)
 fmt.Println(flag)
}

func main() {
 value := MyType("MyType value")
 value.MethodWithParameters(4, true)
}
```

The diagram illustrates the execution of the provided Go code. The `value` variable is annotated as the `Receiver`. The `number` and `flag` parameters are annotated as `Parameter.`. The output of the `fmt.Println(flag)` statement is annotated as `Parameter.`. The terminal window shows the final output: `MyType value`, `4`, and `true`.

As with any other function, you can declare one or more return values for a method, which will be returned when the method is called:

```
func (m MyType) WithReturn() int {
 return len(m)
}
func main() {
 value := MyType("MyType value")
 fmt.Println(value.WithReturn()) ← Print the method's
} return value.
12
```

Return the length of the receiver's underlying string value.

As with any other function, a method is considered exported from the current package if its name begins with a capital letter, and it's considered unexported if its name begins with a lower-case letter. If you want to use your method outside the current package, be sure its name begins with a capital letter.

```
func (m MyType) ExportedMethod() {
}
func (m MyType) unexportedMethod() {
}
```

Exported; name begins with a capital letter.

Unexported; name begins with a lower-case letter.

## POINTER RECEIVER PARAMETERS

Here's an issue that may look familiar by now. We've defined a new `Number` type with an underlying type of `int`. We've given `Number` a `Double` method that is supposed to multiply the underlying value of its receiver by two, and then update the receiver. But we can see from the output that the method receiver isn't actually getting updated.

```
package main

import "fmt"
type Number int
func (n Number) Double() {
 n *= 2
}
func main() {
 number := Number(4)
 fmt.Println("Original value of number:", number)
 number.Double()
 fmt.Println("number after calling Double:", number)
}
Original value of number: 4
number after calling Double: 4
```

Define a type with an underlying type of "int".

Define a method on the Number type.

Multiply the receiver by 2, and attempt to update the receiver.

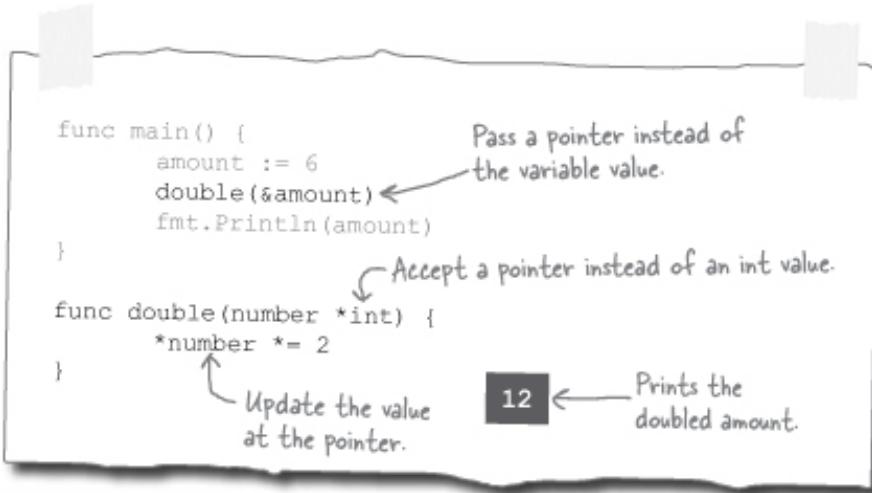
Create a Number value.

Attempt to double the Number.

Number is unchanged!

Back in Chapter 3, we had a `double` function with a similar problem. Back then, we learned that function parameters receive a copy of the values the function is called with,

not the original values, and that any updates to the copy would be lost when the function exited. To make the `double` function work, we had to pass a *pointer* to the value we wanted to update, and then update the value at that pointer within the function.



We've said that receiver parameters are treated no differently than ordinary parameters. And like any other parameter, a receiver parameter receives a *copy* of the receiver value. If you make changes to the receiver within a method, you're changing the copy, not the original.

As with the `double` function in Chapter 3, the solution is to update our `Double` method to use a pointer for its receiver parameter. This is done in the same way as any other parameter: we place a `*` in front of the receiver type to indicate it's a pointer type. We'll also need to modify the method block so that it updates the value at the pointer. Once that's done, when we call `Double` on a `Number` value, the `Number` should be updated.

```
// Package, imports, type omitted
func (n *Number) Double() {
 *n *= 2
}

func main() {
 number := Number(4)
 fmt.Println("Original value of number:", number)
 number.Double() ← We DON'T have to update the method call!
 fmt.Println("number after calling Double:", number)
}

Original value of number: 4
number after calling Double: 8 ← Value at pointer was updated.
```

Notice that we *didn't* have to change the method call at all. When you call a method that requires a pointer receiver on a variable with a non-pointer type, Go will automatically convert the receiver to a pointer for you. The same is true for variables with pointer types; if you call a method requiring a value receiver, Go will automatically get the value at the pointer for you and pass that to the method.

```
// Package, imports omitted
type MyType string

func (m MyType) method() {
 fmt.Println("Method with value receiver")
}
func (m *MyType) pointerMethod() {
 fmt.Println("Method with pointer receiver")
}

func main() {
 value := MyType("a value")
 pointer := &value
 value.method() Value automatically
 value.pointerMethod() converted to pointer.
 pointer.method() Value at pointer
 pointer.pointerMethod() ← automatically retrieved.
 pointer.pointerMethod()
}
```

```
Method with value receiver
Method with pointer receiver
Method with value receiver
Method with pointer receiver
```

You can see this at work in the code at right. The method named `method` takes a value receiver, but we can call it using both direct values and pointers, because Go auto-converts if needed. And the method named `pointerMethod` takes a pointer receiver, but we can call it on both direct values and pointers, because Go will auto-convert if needed.

By the way, the code at right breaks a convention: for consistency, all of your type's methods can take value receivers, or they can all take pointer receivers, but you should avoid mixing the two. We're only mixing the two kinds here for demonstration purposes.



**To call a method that requires a pointer receiver, you have to be able to get a pointer to the value!**

*You can only get pointers to values that are stored in variables. If you try to get the address of a value that's not stored in a variable, you'll get an error:*

```
&MyType("a value")
```

Error → cannot take the address  
of MyType("a value")

*The same limitation applies when calling methods with pointer receivers. Go can automatically convert values to pointers for you, but only if the receiver*

*value is stored in a variable. If you try to call a method on the value itself, Go won't be able to get a pointer, and you'll get a similar error:*

```
MyType("a value").pointerMethod()
Errors → cannot call pointer method
on MyType("a value")
cannot take the address
of MyType("a value")
```

*Instead, you'll need to store the value in a variable, which will then allow Go to get a pointer to it:*

```
value := MyType("a value")
value.pointerMethod()
↑
Go converts this to a pointer.
```



## BREAKING STUFF IS EDUCATIONAL!

Here is our `Number` type again, with definitions for a couple methods. Make one of the changes below, and try to compile the code. Then undo your change, and try the next one. See what happens!

```
package main
```

```
import "fmt"
```

```
type Number int
```

```
func (n *Number) Display() {
```

```
 fmt.Println(*n)
```

```
}
```

```
func (n *Number) Double() {
```

```
 *n *= 2
```

```
}
```

```
func main() {
```

```
 number := Number(4)
```

```
 number.Double()
```

```
 number.Display()
```

```
}
```

| If you do this...                                                                                                                                              | ...the code will break because...                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Change a receiver parameter to a type not defined in this package:</p> <pre>func (n *Number)int)<br/>Double() {<br/>    *n *= 2<br/>}</pre>                 | <p>You can only define new methods on types that were declared in the current package. Defining a method on a globally-defined type like <code>int</code> will result in a compile error.</p>                                |
| <p>Change the receiver parameter for <code>Double</code> to a non-pointer type:</p> <pre>func (n *Number)<br/>Double() {<br/>    *n *= 2<br/>}</pre>           | <p>Receiver parameters receive a copy of the value the method was called on. If the <code>Double</code> function only modifies the copy, the original value will be unchanged when <code>Double</code> exits.</p>            |
| <p>Call a method that requires a pointer receiver on a value that's not in a variable:</p> <pre>Number(4).Double()</pre>                                       | <p>When calling a method that takes a pointer receiver, Go can automatically convert a value to a pointer to a receiver if it's stored in a variable. If it's not, you'll get an error.</p>                                  |
| <p>Change the receiver parameter for <code>Display</code> to a non-pointer type:</p> <pre>func (n *Number)<br/>Display() {<br/>    fmt.Println(*n)<br/>}</pre> | <p>The code will actually still <i>work</i> after making this change, but it breaks convention! Receiver parameters in the methods for a type can be all pointers, or all values, but it's best to avoid mixing the two.</p> |

# CONVERTING LITERS AND MILLILITERS TO GALLONS USING METHODS

When we added a `Milliliters` type to our defined types for measuring volume, we discovered we couldn't have `ToGallons` functions for both `Liters` and `Milliliters`. To work around this, we had to create functions with lengthy names:

```
func LitersToGallons(l Liters) Gallons {
 return Gallons(l * 0.264)
}
func MillilitersToGallons(m Milliliters) Gallons {
 return Gallons(m * 0.000264)
}
```

But unlike functions, method names don't have to be unique, as long as they're defined on different types.

Let's try implementing a `ToGallons` method on the `Liters` type. The code will be almost identical to the `LitersToGallons` function, but we'll make the `Liters` value a receiver parameter rather than an ordinary parameter. Then we'll do the same for the `Milliliters` type, converting the `MillilitersToGallons` function to a `ToGallons` method.

Notice that we're not using pointer types for the receiver parameters. We're not modifying the receivers, and the values don't consume much memory, so it's fine for the parameter to receive a copy of the value.

```

package main

import "fmt"

type Liters float64
type Milliliters float64
type Gallons float64

Method for Liters. ↴ Names can be identical, if they're on separate types.
func (l Liters) ToGallons() Gallons { ↴ Method block unchanged from function block.
 return Gallons(l * 0.264) ←
}

Method for Milliliters. ↴ Names can be identical, if they're on separate types.
func (m Milliliters) ToGallons() Gallons {
 return Gallons(m * 0.000264) ← Method block unchanged from function block.
}

func main() { ↴ Create Liters value.
 soda := Liters(2)
 fmt.Printf("%0.3f liters equals %0.3f gallons\n", soda, soda.ToGallons())
 water := Milliliters(500) ← Create Milliliters value.
 fmt.Printf("%0.3f milliliters equals %0.3f gallons\n", water, water.ToGallons())
}

2.000 liters equals 0.528 gallons
500.000 milliliters equals 0.132 gallons

```

Convert Liters to Gallons. ↴  
Convert Milliliters to Gallons. ↴

In our `main` function, we create a `Liters` value, then call `ToGallons` on it. Because the receiver has the type `Liters`, the `ToGallons` method for the `Liters` type is called. Likewise, calling `ToGallons` on a `Milliliters` value causes the `ToGallons` method for the `Milliliters` type to be called.

## CONVERTING GALLONS TO LITERS AND MILLILITERS USING METHODS

The process is similar when converting the `GallonsToLiters` and `GallonsToMilliliters` functions to methods. We just move the `Gallons` parameter to a receiver parameter in each.

```

func (g Gallons) ToLiters() Liters { ← Define a ToLiters method on the Gallons type.
 return Liters(g * 3.785)
}
func (g Gallons) ToMilliliters() Milliliters { ← Define a ToMilliliters method
 return Milliliters(g * 3785.41) on the Gallons type.
}

func main() { ↴ Create a Gallons value.
 milk := Gallons(2)
 fmt.Printf("%0.3f gallons equals %0.3f liters\n", milk, milk.ToLiters())
 fmt.Printf("%0.3f gallons equals %0.3f milliliters\n", milk, milk.ToMilliliters())
}

2.000 gallons equals 7.570 liters
2.000 gallons equals 7570.820 milliliters

```

Convert it to Liters. ↴      Convert it to Milliliters. ↴



The below code should add a `ToMilliliters` method on the `Liters` type, and a `ToLiters` method on the `Milliliters` type. The code in the `main` function should produce the output shown. Fill in the blanks to complete the code.

```
type Liters float64
type Milliliters float64
type Gallons float64

func (l Liters) ToMilliliters() Milliliters {
 return Milliliters(l * 1000)
}
func (m Milliliters) ToLiters() Liters {
 return Liters(m / 1000)
}

func main() {
 l := Liters(3)
 fmt.Printf("%0.1f liters is %0.1f milliliters\n", l, l.ToMilliliters())
 ml := Milliliters(500)
 fmt.Printf("%0.1f milliliters is %0.1f liters\n", ml, ml.ToLiters())
}
```

3.0 liters is 3000.0 milliliters  
500.0 milliliters is 0.5 liters



## YOUR GO TOOLBOX

That's it for Chapter 9! You've added method definitions to your toolbox.

## Defined Types

Type definitions allow you to create new types of your own.

Each defined type is based on an underlying type that determines how values are stored.

Defined types can use any type as an underlying type, although structs are most commonly used.

## Method Definitions

A method definition is just like a function definition, except that it includes a receiver parameter.

The method becomes associated with the type of the receiver parameter. From then on, that method can be called on any value of that type.



### BULLET POINTS

- Once you've defined a type, you can do a conversion to that type from any value of the underlying type.  
`Gallons(10.0)`
- Once a variable's type is defined, values of other types cannot be assigned to that variable, even if they have the same underlying type.
- A defined type supports all the same operators as its underlying type. A type based on `int`, for example, would support `+`, `-`, `*`, `/`, `==`, `>`, and `<` operators.
- A defined type can be used in operations together with values of its underlying type:  
`Gallons(10.) + 2.3`
- To define a method, provide a receiver parameter in parentheses before the method name:

```
func (m MyType) MyMethod() {
}
```

- The receiver parameter can be used within the method block like any other parameter:

```
func (m MyType) MyMethod() {
 fmt.Println("called on", m)
}
```

- You can define additional parameters or return values on a method, just as you would with any other function.
- Defining multiple functions with the same name in the same package is not allowed, even if they have parameters of different types. But you *can* define multiple *methods* with the same name, as long as each is defined on a different type.
- You can only define methods on types that were defined in the same package.
- As with any other parameter, receiver parameters receive a copy of the original value. If your method needs to modify the receiver, you should use a pointer type for the receiver parameter, and modify the value at that pointer.

## POOL PUZZLE SOLUTION

```
package main

import "fmt"

type Population int

func main() {
 var population Population
 population = Population(572)
 fmt.Println("Sleepy Creek County population:", population)
 fmt.Println("Congratulations, Kevin and Anna! It's a girl!")
 population += 1
 fmt.Println("Sleepy Creek County population:", population)
}

Output → Sleepy Creek County population: 572
 Congratulations, Kevin and Anna! It's a girl!
 Sleepy Creek County population: 573
```



The below code should add a `ToMilliliters` method on the `Liters` type, and a `ToLiters` method on the `Milliliters` type. The code in the `main` function should produce the output shown. Fill in the blanks to complete the code.

```
type Liters float64
type Milliliters float64
type Gallons float64

func (l Liters) ToMilliliters() Milliliters {
 return Milliliters(l * 1000)
}
func (m Milliliters) ToLiters() Liters {
 return Liters(m / 1000)
}

func main() {
 l := Liters(3)
 fmt.Printf("%0.1f liters is %0.1f milliliters\n", l, l.ToMilliliters())
 ml := Milliliters(500)
 fmt.Printf("%0.1f milliliters is %0.1f liters\n", ml, ml.ToLiters())
}
```

3.0 liters is 3000.0 milliliters  
500.0 milliliters is 0.5 liters

# 10 encapsulation and embedding

## *Keep it to Yourself*

I heard that Paragraph type of hers stores its data in a simple string field! And that fancy Replace method? It's just promoted from an embedded strings.Replacer! You'd never know it from using Paragraph, though!



**Mistakes happen.** Your struct type's methods work great, *if* the struct's fields are set correctly. But suppose you're using data from a file to set those struct fields. Are you *sure* all that data is valid? If it's not, users could get strange errors when your type's methods are called! In this chapter, you'll learn about **encapsulation**: a way to protect those fields from invalid data.

And what if your struct type needs methods that already exist on another type? You don't have to copy and paste the method code. If you embed the other type within your struct type, its methods will be promoted to your struct type. You can use the methods just as if they were defined on your own type! This chapter will show you how that works, too.

# CREATING A DATE STRUCT TYPE

A local startup called Remind Me is developing a calendar application to help users remember birthdays, anniversaries, and more.



The year, month, and day sound like they all need to be grouped together; none of those values would be useful by itself. A struct type would probably be useful for keeping those separate values together in a single bundle.

As we've seen, defined types can use any other type as their underlying type, including structs. In fact, struct types served as our introduction to defined types, back in chapter 8.

Let's create a `Date` struct type to hold our year, month, and day values. We'll add `Year`, `Month`, and `Day` fields to the struct, each with a type of `int`. In our `main` function, we'll run a quick test of the new type, using a struct literal to create a `Date` value with all its fields populated. We'll just use `Println` to print the `Date` out for now.

```
package main

import "fmt"

type Date struct {
 Year int
 Month int
 Day int
}

func main() {
 date := Date{Year: 2019, Month: 5, Day: 27}
 fmt.Println(date)
}
```

Annotations on the code:

- An arrow points from the word `struct` in `Date struct` to the text "Define a new struct type."
- An arrow points from the opening brace of the struct definition to the text "Define struct fields."
- An arrow points from the opening brace of the struct literal in the `main` function to the text "Use a struct literal to create a Date value."

If we run the finished program, we'll see the `Year`, `Month`, and `Day` fields of our `Date` struct. It looks like everything's working!

## PEOPLE ARE SETTING THE DATE STRUCT FIELD TO INVALID VALUES!



Ah, we can see how that might happen. Only year numbers 1 or greater are valid, but we don't have anything preventing users from accidentally setting the `Year` field to 0 or -999. Only month numbers from 1 through 12 are valid, but nothing prevents users from setting the `Month` field to 0 or 13. Only the numbers 1 through 31 are valid for the `Day` field, but users can enter days like -2 or 50.

```
date := Date{Year: 2019, Month: 14, Day: 50)
fmt.Println(date)
date = Date{Year: 0, Month: 0, Day: -2}
fmt.Println(date)
date = Date{Year: -999, Month: -1, Day: 0}
fmt.Println(date)
```

(2019 14 50)  
(0 0 -2)  
(-999 -1 0)

What we need is a way for our programs to ensure the user data is valid before accepting it. In computer science, this is known as "data validation". We need to test that the `Year` is being set to a value of 1 or greater, the `Month` is being set between 1 and 12, and the `Day` is being set between 1 and 31.

(Yes, some months have fewer than 31 days, but to keep our code samples a reasonable length, we'll just check that it's between 1 and 31.)

# SETTER METHODS

A struct type is just another defined type, and that means you can define methods on it just like any other. We should be able to create `SetYear`, `SetMonth`, and `SetDay` methods on the `Date` type that take a value, check whether it's valid, and if so, set the appropriate struct field.

This kind of method is often called a **setter method**. By convention, Go setter methods are usually named in the form `SetX`, where `X` is the thing that you're setting.

Here's a prototype of the `SetYear` method. The receiver parameter is a pointer to the `Date` struct you're calling the method on. `SetYear` accepts the year you want to set as a parameter, and sets the `Year` field on the receiver `Date` struct. It doesn't validate the value at all currently, but we'll add validation in a little bit.

In our `main` method, we create a `Date`, and call `SetYear` on it. Then we print the struct's `Year` field.

**Setter methods are methods used to set fields or other values within a defined type's underlying value.**

```
package main

import "fmt"

type Date struct {
 Year int
 Month int
 Day int
}

func (d Date) SetYear(year int) {
 d.Year = year
}

func main() {
 date := Date{} // Create a Date.
 date.SetYear(2019) // Set its Year field via the method.
 fmt.Println(date.Year) // Print the Year field.
}

```

Accepts the value the field should be set to.  
Set the struct field.

0 ← Year is still set to its zero value!

When we run the program, though, we'll see that it didn't work quite right. Even though we create a `Date` and call `SetYear` with a new value, the `Year` field is still set to its zero value!

## SETTER METHODS NEED POINTER RECEIVERS

Remember the `Double` method on the `Number` type we showed you earlier? Originally, we

wrote it with a plain value receiver type, `Number`. But we learned that, like any other parameter, receiver parameters receive a *copy* of the original value. The `Double` method was updating the copy, which was lost when the function exited.

```
func (n *Number) Double() {
 *n *= 2
}
```

Change the receiver parameter to a pointer type.  
Update the value at the pointer.

We needed to update `Double` to take a pointer receiver type, `*Number`. When we updated the value at the pointer, the changes were preserved after `Double` exited.

The same holds true for `SetYear`. The `Date` receiver gets a *copy* of the original struct. Any updates to the fields of the copy are lost when `SetYear` exits!

```
func (d Date) SetYear(year int) {
 d.Year = year
}
```

Receives a copy of the Date struct.  
Updates the copy, not the original!

We can fix `SetYear` by updating it to take a pointer receiver: `(d *Date)`. That's the only change that's necessary. We don't have to update the `SetYear` method block, because `d.Year` automatically gets the value at the pointer for us (as if we'd typed `(*d).Year`). The call to `date.SetYear` in `main` doesn't need to be changed either, because the `Date` value is automatically converted to a `*Date` when it's passed to the method.

```

package main

import "fmt"

type Date struct {
 Year int
 Month int
 Day int
}
func (d *Date) SetYear(year int) {
 d.Year = year
}
func main() {
 date := Date{}
 date.SetYear(2019)
 fmt.Println(date.Year)
}

Automatically converted to a pointer. → Needs to be a pointer receiver, so original value can be updated.
Automatically gets value at pointer. ← Now updates original value, not a copy.
2019 ← Year field has been updated.

```

Now that `SetYear` takes a pointer receiver, if we re-run the code, we'll see that the `Year` field has been updated.

## ADDING THE REMAINING SETTER METHODS

Now it should be easy to follow the same pattern to define `SetMonth` and `SetDay` methods on the `Date` type. We just need to be sure to use a pointer receiver in the method definition. Go will convert the receiver to a pointer when we call each method, and convert the pointer back to a struct value when updating its fields.

```

package main

import "fmt"

type Date struct {
 Year int
 Month int
 Day int
}

func (d *Date) SetYear(year int) {
 d.Year = year
}
func (d *Date) SetMonth(month int) {
 d.Month = month
}
func (d *Date) SetDay(day int) {
 d.Day = day
}

func main() {
 date := Date{}
 date.SetYear(2019)
 date.SetMonth(5) ← Set the month.
 date.SetDay(27) ← Set the day of the month.
 fmt.Println(date) ← Print all fields.
}

{2019 5 27}

```

In `main`, we can create a `Date` struct value, set its `Year`, `Month`, and `Day` fields via our new methods, and print the whole struct out to see the results.

Now we have setter methods for each of our `Date` type's fields. But even if they use the methods, users can still accidentally set the fields to invalid values. We'll look at preventing that next.

```
date := Date{}
date.SetYear(0) ← |invalid|
date.SetMonth(14) ← |invalid|
date.SetDay(50) ← |invalid|
fmt.Println(date)
```

{0 14 50}



In the [Chapter 8](#) exercises, you saw code for a `Coordinates` struct type. We've moved that type definition to a `coordinates.go` file within the `geo` package directory.

We need to add setter methods to the `Coordinates` type for each of its fields. Fill in the blanks in the `coordinates.go` file below, so that the code in `main.go` will run and produce the output shown.

```
package geo

type Coordinates struct {
 Latitude float64
 Longitude float64
}

func (c _____) SetLatitude(_____ float64) {
 _____ = latitude
}

func (c _____) SetLongitude(_____ float64) {
 _____ = longitude
}
```



coordinates.go

```
package main

import (
 "fmt"
 "geo"
)

func main() {
 coordinates := geo.Coordinates{}
 coordinates.SetLatitude(37.42)
 coordinates.SetLongitude(-122.08)
 fmt.Println(coordinates)
}
```



main.go



Output  
↓  
{37.42 -122.08}

## ADDING VALIDATION TO THE SETTER METHODS

Adding validation to our setter methods will take a bit of work, but we learned everything we need to do it in [Chapter 3](#).

In each setter method, we'll test whether the value is in a valid range. If it's invalid, we'll return an `error` value. If it's valid, we'll set the `Date` struct field as normal, and return `nil` for the error value.

Let's add validation to the `SetYear` method first. We add a declaration that the method will return a value, of type `error`. At the start of the method block, we test whether the `year` parameter provided by the caller is any number less than `1`. If it is, we return an `error` with a message of `"invalid year"`. If not, we set the struct's `Year` field, and return `nil`, indicating there was no error.

In `main`, we call `SetYear` and store its return value in a variable named `err`. If `err` is not `nil`, it means the assigned value was invalid, so we log the error and exit. Otherwise, we proceed to print the `Date` struct's `Year` field.

Passing an invalid value to `SetYear` causes the program to report the error and exit. But if we pass a valid value, the program will proceed to print it out. Looks like our `SetYear` method is working!

```
package main

import (
 "errors" ← Lets us create error values.
 "fmt"
 "log" ← Lets us log an error and exit.
)

type Date struct {
 Year int
 Month int
 Day int
} ← Add an error
 return value.↓

func (d *Date) SetYear(year int) error {
 If the given year is invalid... → if year < 1 {
 ...Return an error. → return errors.New("invalid year")
 }

 Otherwise, set the field... → d.Year = year
 ...And return an error of "nil". → return nil
}
// SetMonth, SetDay omitted

func main() {
 date := Date{}
 Capture any error. → err := date.SetYear(0)
 If the value was invalid... → if err != nil {
 Log the error and exit. → log.Fatal(err)
 }
 fmt.Println(date.Year)
}

Error gets logged. → 2018/03/17 19:58:02 invalid year
exit status 1

date := Date{}
err := date.SetYear(2019) ← Valid value.
if err != nil {
 log.Fatal(err)
}
fmt.Println(date.Year) 2019 ← Field gets printed.
```

Validation code in the `SetMonth` and `SetDay` methods will be similar to the code in `SetYear`.

In `SetMonth`, we test whether the provided month number is less than `1` or greater than `12`, and return an error if so. Otherwise, we set the field and return `nil`.

And in `SetDay`, we test whether the provided day of the month is less than `1` or greater than `31`. Invalid values result in a returned error, but valid values cause the field to be set and `nil` to be returned.

```
// Package, imports, type declaration omitted
func (d *Date) SetYear(year int) error {
 if year < 1 {
 return errors.New("invalid year")
 }
 d.Year = year
 return nil
}
func (d *Date) SetMonth(month int) error {
 if month < 1 || month > 12 {
 return errors.New("invalid month")
 }
 d.Month = month
 return nil
}
func (d *Date) SetDay(day int) error {
 if day < 1 || day > 31 {
 return errors.New("invalid day")
 }
 d.Day = day
 return nil
}

func main() {
 // Try the below code snippets here
}
```

You can test the setter methods by inserting the code snippets below into the block for `main...`

Passing 14 to `SetMonth` results in an error:

```
date := Date{}
err := date.SetMonth(14)
if err != nil {
 log.Fatal(err)
}
fmt.Println(date.Month)
```

2018/03/17 20:17:42  
invalid month  
exit status 1

Passing 50 to `SetDay` results in an error:

```
date := Date{}
err := date.SetDay(50)
if err != nil {
 log.Fatal(err)
}
fmt.Println(date.Day)
```

2018/03/17 20:30:54  
invalid day  
exit status 1

But passing 5 to `SetMonth` works:

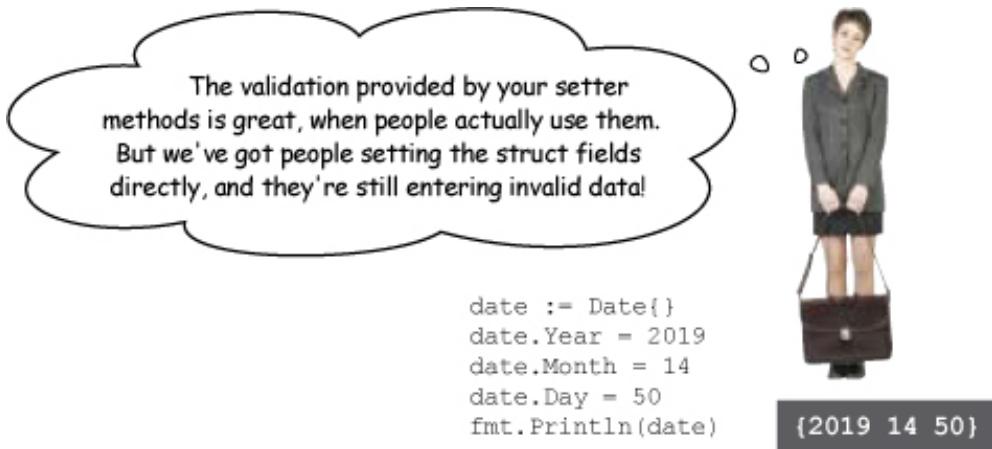
```
date := Date{}
err := date.SetMonth(5)
if err != nil {
 log.Fatal(err)
}
fmt.Println(date.Month)
```

But passing 27 to SetDay works:

```
date := Date{}
err := date.SetDay(27)
if err != nil {
 log.Fatal(err)
}
fmt.Println(date.Day)
```

27

## THE FIELDS CAN STILL BE SET TO INVALID VALUES!



It's true; there's nothing preventing anyone from setting the `Date` struct fields directly. And if they do so, it bypasses the validation code in the setter methods. They can set any value they want!

We need a way to protect these fields, so that users of our `Date` type can only update the fields using the setter methods.

Go provides a way of doing this: we can move the `Date` type to another package, and make its date fields unexported.

So far, unexported variables, functions, etc. have mostly gotten in our way. The most recent example of this was in [Chapter 8](#), when we discovered that even though our `Subscriber` struct type was exported from the `magazine` package, its fields were *unexported*, making them inaccessible outside the `magazine` package.

With the `Subscriber` type name capitalized, we seem to be able to access it from the `main` package. But now we're getting an error saying that we can't refer to the `rate` field, because that is unexported.

Even if a struct type is exported from a package, its fields will be *unexported* if their names don't begin with a capital letter. Let's try capitalizing `Rate` (in both `magazine.go` and `main.go`)...

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

But in this case, we don't *want* the fields to be accessible. Unexported struct fields are exactly what we need!

Let's try moving our `Date` type to another package and making its fields unexported, and see if that fixes our problem.

## MOVING THE DATE TYPE TO ANOTHER PACKAGE

In the `headfirstgo` directory within your Go workspace, create a new directory to hold a package named `calendar`. Within `calendar`, create a file named `date.go`. (Remember, you can name the files within a package directory anything you want; they'll all become part of the same package.)



Within `date.go`, add a `package calendar` declaration, and import the "`errors`" package. (That's the only package that the code in this file will be using.) Then, copy all your old code for the `Date` type and paste it into this file.

```

package calendar ← This file is part of the
 "calendar" package.

import "errors" ←

type Date struct { This file only uses functions
 Year int from the "error" package.
 Month int
 Day int } Copy and paste all the code for the
 Date type into the new file.

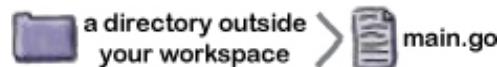
func (d *Date) SetYear(year int) error {
 if year < 1 {
 return errors.New("invalid year")
 }
 d.Year = year
 return nil
}

func (d *Date) SetMonth(month int) error {
 if month < 1 || month > 12 {
 return errors.New("invalid month")
 }
 d.Month = month
 return nil
}

func (d *Date) SetDay(day int) error {
 if day < 1 || day > 31 {
 return errors.New("invalid day")
 }
 d.Day = day
 return nil
}

```

Next, create a file named `main.go`. You can save it in any directory you want, but save it *outside* your Go workspace, so it doesn't interfere with any other packages.



At this point, code we add in `main.go` will still be able to create an invalid `Date`, either by setting its fields directly, or by using a struct literal.

```

package main ← Use the "main" package, since we'll
 be running this code as a program.

import (
 "fmt"
 "github.com/headfirstgo/calendar" ← Import our new package.
)

func main() {
 date := calendar.Date{} ← Create a new Date value.

Set the Date's fields directly. { date.Year = 2019
 date.Month = 14
 date.Day = 50
 fmt.Println(date)
 Specify package. } Set another Date's fields
 using a struct literal.
 date = calendar.Date{Year: 0, Month: 0, Day: -2}
 fmt.Println(date)
}

```

If we run `main.go` from the terminal, we'll see that both ways of setting the fields worked, and two invalid dates are printed.

```

Shell Edit View Window Help
$ cd temp
$ go run main.go
{2019 14 50}
{0 0 -2}

```

## MAKING DATE FIELDS UNEXPORTED

Now let's try updating the `Date` struct so that its fields are unexported. That simply consists of changing the field names to begin with lower-case letters, in the type definition and everywhere else they occur.

The `Date` type itself needs to remain exported, as do all of the setter methods, as we *will* need to access these from outside the `calendar` package.



```

date.go
package calendar

import "errors"
type Date struct {
 year int
 month int
 day int
}

func (d *Date) SetYear(year int) error {
 if year < 1 {
 return errors.New("invalid year")
 }
 d.year = year
 return nil
}

func (d *Date) SetMonth(month int) error {
 if month < 1 || month > 12 {
 return errors.New("invalid month")
 }
 d.month = month
 return nil
}

func (d *Date) SetDay(day int) error {
 if day < 1 || day > 31 {
 return errors.New("invalid day")
 }
 d.day = day
 return nil
}

```

*Annotations:*

- Date type needs to remain exported!* (points to the `Date` type definition)
- Change field names so they are unexported.* (points to the `year`, `month`, and `day` fields)
- No changes to method names.* (points to the `SetYear`, `SetMonth`, and `SetDay` method signatures)
- No changes to method parameters.* (points to the `year`, `month`, and `day` parameters in the method signatures)
- Update field name to match declaration above.* (points to the `d.year = year` assignment in `SetYear`)
- Update field name to match declaration above.* (points to the `d.month = month` assignment in `SetMonth`)
- Update field name to match declaration above.* (points to the `d.day = day` assignment in `SetDay`)

To test our changes, update the field names in `main.go` to match the field names in `date.go`.



main.go

```
// Package, import statements omitted
func main() {
 date := calendar.Date{}
 date.year = 2019
 date.month = 14
 date.day = 50
 fmt.Println(date)

 date = calendar.Date{year: 0, month: 0, day: -2}
 fmt.Println(date)
}
```

Change field names to match.

Change field names to match.

## ACCESSING UNEXPORTED FIELDS THROUGH EXPORTED METHODS

As you might expect, now that we've converted the fields of `Date` to unexported, trying to access them from the `main` package results in compile errors. This is true both when trying to set the field values directly, and when using them in a struct literal.

Can't access fields directly.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
./main.go:10:6: date.year undefined (cannot refer to unexported field or method year)
./main.go:11:6: date.month undefined (cannot refer to unexported field or method month)
./main.go:12:6: date.day undefined (cannot refer to unexported field or method day)
./main.go:15:27: unknown field 'year' in struct literal of type calendar.Date
./main.go:15:37: unknown field 'month' in struct literal of type calendar.Date
./main.go:15:45: unknown field 'day' in struct literal of type calendar.Date
```

But, we can still access the fields indirectly. *Unexported* variables, struct fields, functions, methods, etc. can still be accessed by *exported* functions and methods in the same package. So when code in the `main` package calls the exported `SetYear` method on a `Date` value, `SetYear` can update the `Date`'s `year` struct field, even though it's unexported. The exported `SetMonth` method can update the unexported `month` field. And so on.

If we modify `main.go` to use the setter methods, we'll be able to update a `Date` value's fields:



main.go

```
package main

import (
 "fmt"
 "github.com/headfirstgo/calendar"
 "log"
)

func main() {
 date := calendar.Date()
 err := date.SetYear(2019) ← Use the
 if err != nil { setter
 log.Fatal(err) method.
 }
 err = date.SetMonth(5) ← Use the
 if err != nil { setter
 log.Fatal(err) method.
 }
 err = date.SetDay(27) ← Use the
 if err != nil { setter
 log.Fatal(err) method.
 }
 fmt.Println(date)
}
```

You can update fields via the setter  
methods!

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
{2019 5 27}
```

**Unexported variables, struct fields, functions, and methods can still be accessed by exported functions and methods in the same package.**

If we update `main.go` to call `SetYear` with an invalid value, we'll get an error when we run it:



main.go

```
func main() {
 date := calendar.Date()
 err := date.SetYear(0) ← Call the setter
 if err != nil { method with an
 log.Fatal(err) invalid value.
 }
 fmt.Println(date)
}
```

Invalid values get reported!

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
2018/03/23 19:20:17 invalid year
exit status 1
```

Now that a `Date` value's fields can only be updated via its setter methods, programs are protected against accidentally entering invalid data.

That should cut down on the invalid dates we've been seeing. But there's a new problem. We can set the field values, but how do we get those values back out?



Ah, that's right. We provided setter methods that let us set `Date` fields, even though those fields are unexported from the `calendar` package. But we haven't provided any methods to *get* the field values.

We can print an entire `Date` struct. But if we try to update `main.go` to print an individual `Date` field, we won't be able to access it!

The screenshot shows a code editor with a file named `main.go`. The code defines a `main()` function that creates a `calendar.Date` object, sets its year to 2019, and then tries to print its `year` field using `fmt.Println(date.year)`. Handwritten annotations explain that setting the year is a "Setter method" and printing it is a "Getter method". A callout points to the `year` field in the code with the text "Try to print the year field." Another callout points to the terminal window below with the text "Get an error, because the field is unexported!".

```
main.go func main() {
 date := calendar.Date{}
 err := date.SetYear(2019) ← Set to a valid year.
 if err != nil {
 log.Fatal(err)
 }
 fmt.Println(date.year) ← Try to print the year field.
}
```

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
command-line-arguments
./main.go:16:18: date.year undefined
(cannot refer to unexported field or method year)
```

## GETTER METHODS

As we've seen, methods whose main purpose is to *set* the value of a struct field or variable are called "setter methods". And as you might expect, methods whose main purpose is to *get* the value of a struct field or variable are called **getter methods**.

Compared to the setter methods, adding getter methods to the `Date` type will be easy. They don't need to do anything except return the field value when they're called.

By convention, a getter method's name should be the same as the name of the field or variable it accesses. (Of course, if you want the method to be exported, its name will need to start with a capital letter.) So `Date` will need a `Year` method to access the `year` field,

a `Month` method for the `month` field, and a `Day` method for the `day` field.

Getter methods don't need to modify the receiver at all, so we *could* use a direct `Date` value as a receiver. But if any method on a type takes a pointer receiver, convention says that they all should, for consistency's sake. Since we have to use a pointer receiver for our setter methods, we use a pointer for the getter methods as well.

With the changes to `date.go` complete, we can update `main.go` to set all the `Date` fields, then use the getter methods to print them all out.

 **date.go**

```
package calendar

import "errors"

type Date struct {
 year int
 month int
 day int
}

func (d *Date) Year() int {
 return d.year
}

func (d *Date) Month() int {
 return d.month
}

func (d *Date) Day() int {
 return d.day
}

// Setter methods omitted
```

Annotations on the code:

- A bracket above the `year`, `month`, and `day` fields is labeled "Use a pointer receiver type for consistency with the setter methods."
- A bracket below the `Year()`, `Month()`, and `Day()` methods is labeled "Same name as the field (but capitalized so it's exported)."
- An arrow points from the `return d.year` line to the text "Return the field value."

 **main.go**

```
// Package, import statements omitted
func main() {
 date := calendar.Date{}
 err := date.SetYear(2019)
 if err != nil {
 log.Fatal(err)
 }
 err = date.SetMonth(5)
 if err != nil {
 log.Fatal(err)
 }
 err = date.SetDay(27)
 if err != nil {
 log.Fatal(err)
 }
 fmt.Println(date.Year())
 fmt.Println(date.Month())
 fmt.Println(date.Day())
}
```

Values returned from getter methods.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
2019
5
27
```

## ENCAPSULATION

The practice of hiding data in one part of a program from code in another part is known

as **encapsulation**, and it's not unique to Go. Encapsulation is valuable because it can be used to protect against invalid data (as we've seen). Also, you can change an encapsulated portion of a program without worrying about breaking other code that accesses it, because direct access isn't allowed.

Many other programming languages encapsulate data within classes. (Classes are a concept similar, but not identical, to a type). In Go, data is encapsulated within packages, using unexported variables, struct fields, functions, or methods.

Encapsulation is used far more frequently in other languages than it is in Go. In some languages it is conventional to define getters and setters for every field, even when accessing those fields directly would work just as well. Go developers generally only rely on encapsulation when it's necessary, such as when field data needs to be validated by setter methods. In Go, if you don't see a need to encapsulate a field, it's generally okay to export it and allow direct access to it.

## *THERE ARE NO DUMB QUESTIONS*

**Q:** Many other languages don't allow access to encapsulated values outside of the class where they're defined. Is it safe for Go to allow other code in the same package to access unexported fields?

**A:** Generally, all the code in a package is the work of a single developer (or group of developers). Generally all the code in a package has a similar purpose, as well. The authors of code within the same package are most likely to need access to unexported data, and they're also likely to only use that data in valid ways. So, yes, sharing unexported data with the rest of the package is generally safe.

Code *outside* the package is likely to be written by *other* developers, but that's okay because the unexported fields are hidden from them, so they can't accidentally change their values to something invalid.

**Q:** I've seen other languages where the name of every getter method started with "Get", as in `GetName`, `GetCity`, etc. Can I do that in Go?

**A:** The Go language will allow you to do that, but you shouldn't. The Go community has decided on a convention of leaving the `Get` prefix off of getter method names. Including it would only lead to confusion for your fellow developers!

Go still uses a `Set` prefix for setter methods, just like many other languages, because it's needed to distinguish setter method names from getter method names for the same field.



Bear with us; we'll need two pages to fit all the code for this exercise...  
Fill in the blanks to make the following changes to the `Coordinates` type:

- Update its fields so they're unexported.
- Add getter methods for each field. (Be sure to follow the convention: a getter method's name should be the same as the name of the field it accesses, with capitalization if the method needs to be exported.)
- Add validation to the setter methods. `SetLatitude` should return an error if the passed-in value is less than `-90` or greater than `90`. `SetLongitude` should return an error if the new value is less than `-180` or greater than `180`.



coordinates.go

```
package geo

import "errors"

type Coordinates struct {
 _____ float64
 _____ float64
}

func (c *Coordinates) _____() _____ {
 return c.latitude
}

func (c *Coordinates) _____() _____ {
 return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) _____ {
 if latitude < -90 || latitude > 90 {
 return _____("invalid latitude")
 }
 c.latitude = latitude
 return _____
}
func (c *Coordinates) SetLongitude(longitude float64) _____ {
 if longitude < -180 || longitude > 180 {
 return _____("invalid longitude")
 }
 c.longitude = longitude
 return _____
}
```

Next, update the `main` package code to make use of the revised `Coordinates` type.

- For each call to a setter method, store the `error` return value.
- If the `error` is not `nil`, use the `log.Fatal` function to log the error message and exit.
- If there were no errors setting the fields, call both getter methods to print the field values.

The completed code should produce the output shown when it runs. (The call to `SetLatitude` should be successful, but we're passing an invalid value to `SetLongitude`, so it should log an error and exit at that point.)



```
package main

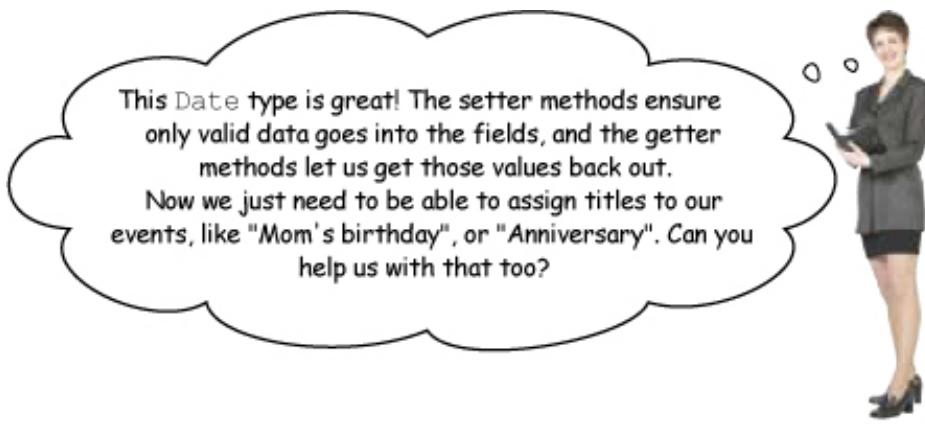
import (
 "fmt"
 "geo"
 "log"
)

func main() {
 coordinates := geo.Coordinates{}
 __ := coordinates.SetLatitude(37.42)
 if err != __ {
 log.Fatal(err)
 }
 err = coordinates.SetLongitude(-1122.08) ← (An invalid value!)
 if err != __ {
 log.Fatal(err)
 }
 fmt.Println(coordinates._____())
 fmt.Println(coordinates._____())
}
```

Output

```
2018/03/23 20:12:49 invalid longitude
exit status 1
```

## EMBEDDING THE DATE TYPE IN AN EVENT TYPE



That shouldn't take much work. Remember how we embedded an `Address` struct type within two other struct types back in Chapter 8?

The `Address` type was considered "embedded" because we used an anonymous field (a field with no name, just a type) in the outer struct to store it. This caused the fields of `Address` to be promoted to the outer struct, allowing us to access fields of the inner struct as if they belonged to the outer struct.

```

package magazine

type Subscriber struct {
 Name string
 Rate float64
 Active bool
 Address
}

type Employee struct {
 Name string
 Salary float64
 Address
}

type Address struct {
 // Fields omitted
}

```

**Set the fields of subscriber as if they were defined on Subscriber.**

`subscriber.Street = "123 Oak St"`  
`subscriber.City = "Omaha"`  
`subscriber.State = "NE"`  
`subscriber.PostalCode = "68111"`

your workspace > src > github.com > headfirstgo > calendar > event.go

Since that strategy worked so well before, let's define an `Event` type that embeds a `Date` with an anonymous field.

Create another file within the `calendar` package folder, named `event.go`. (We could put it within the existing `date.go` field, but this organizes things a bit more neatly.) Within that file, define an `Event` type with two fields: a `Title` field with a type of `string`, and an anonymous `Date` field.

```
package calendar

Embed a type Event struct {
 Title string
 Date Date
}
Date using an anonymous field.
```

## UNEXPORTED FIELDS DON'T GET PROMOTED

Embedding a `Date` in the `Event` type will *not* cause the `Date` fields to be promoted to the `Event`, though. The `Date` fields are unexported, and Go doesn't promote unexported fields to the enclosing type. That makes sense; we made sure the fields were encapsulated so they can only be accessed through setter and getter methods, and we don't want that encapsulation to be circumvented through field promotion.

```
event.go
package calendar

type Event struct {
 Title string
 Date Date
}
Embedded using an anonymous field.
```

In our `main` package, if we try to set the `month` field of a `Date` through its enclosing `Event`, we'll get an error:

```
main.go
package main

import "github.com/headfirstgo/calendar"

func main() {
 event := calendar.Event{}
 event.month = 5
}
Unexported Date fields aren't promoted to the Event!
event.month undefined (type calendar.Event has no field or method month)
```

And of course, using dot operator chaining to retrieve the `Date` field and then access fields on it directly won't work, either. You can't access a `Date` value's unexported fields when it's by itself, and you can't access its unexported fields when it's part of an `Event`, either.

```
main.go
package main

func main() {
 event := calendar.Event{}
 event.Date.year = 2019
}
Can't access Date fields directly on the Date!
event.Date.year undefined (cannot refer to unexported field or method year)
```

So does that mean we won't be able to access the fields of the `Date` type, if it's embedded

within the `Event` type? Don't worry; there's another way!

## EXPORTED METHODS GET PROMOTED JUST LIKE FIELDS

If you embed a struct type within another struct type, the embedded type's exported fields get promoted to the outer type. In the same way, if you embed a type with exported methods within a struct type, its *methods* will be promoted to the outer type, meaning you can call the methods as if they were defined on the outer type. (Only defined types with an underlying struct type can have fields, but *any* defined type can have methods, so the embedded type doesn't have to have an underlying struct type.)

Here's a package that defines two types. `MyType` is a struct type. `MyType` embeds a second type, `EmbeddedType`, as an anonymous field.

```
package mypackage ← These types are in their own package.
import "fmt" ↗ Declare MyType as a struct type.
type MyType struct {
 EmbeddedType ← EmbeddedType is embedded in MyType.
} ↗ Declare a type to embed (doesn't matter whether it's a struct).
type EmbeddedType string ↗ This method will be promoted to MyType.
func (e EmbeddedType) ExportedMethod() {
 fmt.Println("Hi from ExportedMethod on EmbeddedType")
} ↗ This method will not be promoted.
func (e EmbeddedType) unexportedMethod() {
}
```

Because `EmbeddedType` defines an exported method (named `ExportedMethod`), that method is promoted to `MyType`, and can be called on `MyType` values.

```
package main

import "mypackage"

func main() {
 value := mypackage.MyType()
 value.ExportedMethod() ↗ Hi from ExportedMethod on EmbeddedType
}
 ↗ Error
```

As with unexported fields, unexported methods are *not* promoted. You'll get an error if you try to call one.

```
value.unexportedMethod() ← Attempt to call unexported method.
 ↗ Error
 ↗ value.unexportedMethod undefined (type mypackage.MyType
 has no field or method unexportedMethod)
```

Our `Date` fields weren't promoted to the `Event` type, because they're unexported. But the getter and setter methods on `Date` *are* exported, and they *do* get promoted to the `Event` type!

That means we can create an `Event` value, and then call the getter and setter methods for the `Date` directly on the `Event`. That's just what we do in the updated `main.go` code below. As always, the exported methods are able to access the unexported `Date` fields for us.



```
main.go package main

import (
 "fmt"
 "github.com/headfirstgo/calendar"
 "log"
)

func main() {
 event := calendar.Event{}
 err := event.SetYear(2019) ← This setter method
 if err != nil {
 log.Fatal(err)
 }
 err = event.SetMonth(5) ← This setter method
 if err != nil {
 log.Fatal(err)
 }
 err = event.SetDay(27) ← This setter method
 if err != nil {
 log.Fatal(err)
 }
}

These getter methods for Date have been promoted to Event.
{
 fmt.Println(event.Year()) 2019
 fmt.Println(event.Month()) 5
 fmt.Println(event.Day()) 27
}
```

And if you prefer to use dot operator chaining to call methods on the `Date` value directly, you can do that too:

Get the Event's Date field, then call getter methods on it.

```
{fmt.Println(event.Date.Year())
 fmt.Println(event.Date.Month())
 fmt.Println(event.Date.Day())}
```

|      |
|------|
| 2019 |
| 5    |
| 27   |

## ENCAPSULATING THE EVENT TITLE FIELD

Because the `Event` struct's `Title` field is exported, we can still access it directly:



```
event.go package calendar

type Event struct {
 Exported field → Title string
 Date
}
```



main.go

```
// Package, imports omitted
func main() {
 event := calendar.Event{}
 event.Title = "Mom's birthday"
 fmt.Println(event.Title)
}

```

Mom's birthday

This exposes us to the same sort of issues that we had with the `Date` fields, though. For example, there's no limit on the length of the `Title` string:



main.go

```
func main() {
 event := calendar.Event{}
 event.Title = "An extremely long title that is impractical to print"
 fmt.Println(event.Title)
}

```

An extremely long title that is impractical to print

It seems like a good idea to encapsulate the `Title` field as well, so we can validate new values. Here's an update to the `Event` type that does so. We change the field's name to `title` so it's unexported, then add getter and setter methods. The `RuneCountInString` function from the `unicode/utf8` package is used to ensure there aren't too many runes (characters) in the string.



event.go

```
package calendar
import (
 "errors" Add this package for creating error values.
 "unicode/utf8" Add this package so we can count
 the number of runes in a string.
)
type Event struct {
 title string
 Date
}
Change to → title string
unexported. → Date
→ Getter method. → func (e *Event) Title() string {
 return e.title
 } → Must use pointer.
→ Setter method. → func (e *Event) SetTitle(title string) error {
 if utf8.RuneCountInString(title) > 30 { ← If the title has more than 30
 return errors.New("invalid title") characters, return an error.
 }
 e.title = title
 return nil
}
```

## PROMOTED METHODS LIVE ALONGSIDE THE OUTER TYPE'S METHODS

Now that we've added setter and getter methods for the `title` field, our programs can report an error if a title longer than 30 characters is used. An attempt to set a 39-character title causes an error to be returned:



main.go

```
// Package, imports omitted
func main() {
 event := calendar.Event{}
 err := eventSetTitle("An extremely long and impractical title")
 if err != nil {
 log.Fatal(err)
 }
}
2018/03/23 20:44:17 invalid title
exit status 1
```

The `Event` type's `Title` and `SetTitle` methods live alongside the methods promoted from the embedded `Date` type. Importers of the `calendar` package can treat all the methods as if they belong to the `Event` type, without worrying about which type they're actually defined on.



main.go

```
// Package, imports omitted
func main() {
 event := calendar.Event{}
 err := eventSetTitle("Mom's birthday") ← Defined on Event itself.
 if err != nil {
 log.Fatal(err)
 }
 err = event.SetYear(2019) ← Promoted from Date.
 if err != nil {
 log.Fatal(err)
 }
 err = event.SetMonth(5) ← Promoted from Date.
 if err != nil {
 log.Fatal(err)
 }
 err = event.SetDay(27) ← Promoted from Date.
 if err != nil {
 log.Fatal(err)
 }
 fmt.Println(event.Title()) ← Defined on Event itself.
 fmt.Println(event.Year()) ← Promoted from Date.
 fmt.Println(event.Month()) ← Promoted from Date.
 fmt.Println(event.Day()) ← Promoted from Date.
}
```

Mom's birthday  
2019  
5  
27

## OUR CALENDAR PACKAGE IS COMPLETE!



Now we can call the `Title` and `SetTitle` methods directly on an `Event`, and call the methods to set a year, month, and day as if they belonged to the `Event`. They're actually defined on `Date`, but we don't have to worry about that. Our work here is done!

Method promotion allows you to easily use one type's methods as if they belonged to another. You can use this to compose types that combine the methods of several other

types. This can help you keep your code clean, without sacrificing convenience!



We completed the code for the `Coordinates` type in a previous exercise. You won't need to make any updates to it this time; it's just here for reference. On the next page, we're going to embed it in the `Landmark` type (which we also saw back in [Chapter 8](#)), so that its methods are promoted to `Landmark`.

```
package geo

import "errors"

type Coordinates struct {
 latitude float64
 longitude float64
}

func (c *Coordinates) Latitude() float64 {
 return c.latitude
}
func (c *Coordinates) Longitude() float64 {
 return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) error {
 if latitude < -90 || latitude > 90 {
 return errors.New("invalid latitude")
 }
 c.latitude = latitude
 return nil
}

func (c *Coordinates) SetLongitude(longitude float64) error {
 if longitude < -180 || longitude > 180 {
 return errors.New("invalid longitude")
 }
 c.longitude = longitude
 return nil
}
```



coordinates.go

Here's an update to the `Landmark` type. We want its `Name` field to be encapsulated, accessible only by a `Name` getter method and a `SetName` setter method. `SetName` should return an error if its argument is an empty string, or set the `Name` field and return a `nil` error otherwise. `Landmark` should also have an anonymous `Coordinates` field, so that the methods of `Coordinates` are promoted to `Landmark`.

Fill in the blanks to complete the code for the `Landmark` type.

```
package geo

import "errors"

type Landmark struct {
 name string
}

func (l *Landmark) Name() string {
 return l.name
}

func (l *Landmark) SetName(name string) error {
 if name == "" {
 return errors.New("invalid name")
 }
 l.name = name
 return nil
}
```



If the blanks in the code for `Landmark` are completed correctly, the code in the `main` package should run, and produce the output shown.

```
package main
// Imports omitted
func main() {
 location := geo.Landmark{}
 err := location.SetName("The Googleplex")
 if err != nil {
 log.Fatal(err)
 }
 err = location.SetLatitude(37.42)
 if err != nil {
 log.Fatal(err)
 }
 err = location.SetLongitude(-122.08)
 if err != nil {
 log.Fatal(err)
 }
 fmt.Println(location.Name())
 fmt.Println(location.Latitude())
 fmt.Println(location.Longitude())
}
```



Output  
The Googleplex  
37.42  
-122.08



## YOUR GO TOOLBOX

That's it for Chapter 10! You've added encapsulation and embedding to your toolbox.

## Encapsulation

Encapsulation is the practice of hiding data in one part of a program from code in another part.

Encapsulation can be used to protect against invalid data.

Encapsulated data is also easier to change. You can be sure you won't break other code that accesses the data, because no code is allowed to.

## Embedding

A type that is stored within a struct type using an anonymous field is said to be embedded within the struct.

Methods of an embedded type get promoted to the outer type. They can be called as if they were defined on the outer type.



### BULLET POINTS

- In Go, data is encapsulated within packages, using unexported package variables or struct fields.
- Unexported variables, struct fields, functions, methods, etc. can still be accessed by exported functions and methods in the same package.
- The practice of ensuring that data is valid before accepting it is known as "data validation".
- A method that is primarily used to set the value of an encapsulated field is known as a "setter method". Setter methods often include validation logic, to ensure the new value being provided is valid.

- Since setter methods need to modify their receiver, their receiver parameter should have a pointer type.
- A method that is primarily used to get the value of an encapsulated field is known as a "getter method".
- Methods defined on an outer struct type live alongside methods promoted from an embedded type.
- An embedded type's unexported methods don't get promoted to the outer type.



We need to add setter methods to the `Coordinates` type for each of its fields. Fill in the blanks in the `coordinates.go` file below, so that the code in `main.go` will run and produce the output shown.

```

package geo

type Coordinates struct {
 Latitude float64
 Longitude float64
}

func (c *Coordinates) SetLatitude(latitude float64) {
 c.Latitude = latitude
}

func (c *Coordinates) SetLongitude(longitude float64) {
 c.Longitude = longitude
}

func main() {
 coordinates := geo.Coordinates{}
 coordinates.SetLatitude(37.42)
 coordinates.SetLongitude(-122.08)
 fmt.Println(coordinates)
}

```

*Must use a pointer type so we can modify the receiver.*

*Must use a pointer type so we can modify the receiver.*

**coordinates.go**

**Output**

{37.42 -122.08}

**main.go**



Your goal with updating this code was to encapsulate the fields of the `Coordinates` type, and add validation to its setter methods.

- Update the fields of `Coordinates` so they're unexported.
- Add getter methods for each field.
- Add validation to the setter methods. `SetLatitude` should return an error if the passed-in value is less than `-90` or greater than `90`. `SetLongitude` should return an error if the new value is less than `-180` or greater than `180`.



coordinates.go

```
package geo

import "errors"

type Coordinates struct {
 latitude float64 } Fields should be
 longitude float64 } unexported.

} Getter method name should be
 same as field, but capitalized. } Same type as the field.

func (c *Coordinates) Latitude() float64 {
 return c.latitude
}

} Getter method name should be
 same as field, but capitalized. } Same type as the field.

func (c *Coordinates) Longitude() float64 {
 return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) error {
 if latitude < -90 || latitude > 90 {
 return errors.New("invalid latitude")
 }
 c.latitude = latitude } Return a new error value.
 return nil } Return nil if no error. } Need to return
} error type.

func (c *Coordinates) SetLongitude(longitude float64) error {
 if longitude < -180 || longitude > 180 {
 return errors.New("invalid longitude")
 }
 c.longitude = longitude } Return a new error value.
 return nil } Return nil if no error.
}
```

Your next task was to update the `main` package code to make use of the revised `Coordinates` type.

- For each call to a setter method, store the `error` return value.
- If the `error` is not `nil`, use the `log.Fatal` function to log the error message and exit.
- If there were no errors setting the fields, call both getter methods to print the field values.

The call to `SetLatitude` below is successful, but we're passing an invalid value to `SetLongitude`, so it logs an error and exits at that point.

```

package main

import (
 "fmt"
 "geo"
 "log"
)

func main() {
 coordinates := geo.Coordinates{}
 err := coordinates.SetLatitude(37.42)
 if err != nil { ← If there was an error,
 log.Fatal(err) log it and exit.
 }
 err = coordinates.SetLongitude(-1122.08) ← (An invalid value!)
 if err != nil { ← If there was an error,
 log.Fatal(err) log it and exit.
 }
 fmt.Println(coordinates.Latitude())
 fmt.Println(coordinates.Longitude())
}

```

Store the returned error value.

If there was an error, log it and exit.

(An invalid value!)

If there was an error, log it and exit.

Call the getter methods.

Output

2018/03/23 20:12:49 invalid longitude  
exit status 1



Here's an update to the `Landmark` type (which we also saw in [Chapter 8](#)). We want its name field to be encapsulated, accessible only by getter and setter methods. The `SetName` method should return an error if its argument is an empty string, or set the name field and return a `nil` error otherwise. `Landmark` should also have an anonymous `Coordinates` field, so that the methods of `Coordinates` are promoted to `Landmark`.

```
package geo
```

```
import "errors"
```

```
type Landmark struct {
```

```
 name string
 Coordinates
```

```
}
```

Ensure "name" field is unexported so it's encapsulated.

Embed using anonymous field.

```
func (l *Landmark) Name() string {
```

```
 return l.name
```

```
} Same name as field,
```

but with "Set" prefix.

```
func (l *Landmark) SetName(name string) error {
```

```
 if name == "" {
```

```
 return errors.New("invalid name")
```

```
}
```

```
 l.name = name
```

```
 return nil
```

```
}
```

```
package main
// Imports omitted
func main() {
```

```
 location := geo.Landmark{}
```

```
 err := location.SetName("The Googleplex")
```

```
 if err != nil {
```

log.Fatal(err)

```
 }
```

err = location.SetLatitude(37.42)

```
 if err != nil {
```

log.Fatal(err)

```
 }
```

err = location.SetLongitude(-122.08)

```
 if err != nil {
```

log.Fatal(err)

```
 }
```

```
 fmt.Println(location.Name())
```

```
 fmt.Println(location.Latitude())
```

```
 fmt.Println(location.Longitude())
```

```
}
```

Create Landmark value.

Defined on Landmark itself.

Promoted from Coordinates.

Promoted from Coordinates.

Defined on Landmark.

Output

The Googleplex

37.42

-122.08



The Googleplex

37.42

-122.08

## 11 interfaces

### *What Can You Do?*



**Sometimes you don't care about the particular type of a value.** You don't care about what it *is*. You just need to know that it will be able to *do* certain things. That you'll be able to call *certain methods* on it. You don't care whether you have a `Pen` or a `Pencil`, you just need something with a `Draw` method. You don't care whether you have a `Car` or a `Boat`, you just need something with a `Steer` method. That's what Go **interfaces** accomplish. They let you define variables and function parameters that will hold *any* type, as long as that type defines certain methods.

TWO DIFFERENT TYPES THAT HAVE THE SAME

# METHODS

Remember audio tape recorders? (We suppose some of you will be too young.) They were great, though. They let you easily record all your favorite songs together on a single tape, even if they were by different artists. Of course, the recorders were usually too bulky to carry around with you. If you wanted to take your tapes on the go, you needed a separate, battery-powered tape player. Those usually didn't have recording capabilities. Ah, but it was so great making custom mixtapes and sharing them with your friends!



We're so overwhelmed with nostalgia that we've created a `gadget` package to help us reminisce. It includes a type that simulates a tape recorder, and another type that simulates a tape player.



The `TapePlayer` type has a `Play` method to simulate playing a song, and a `Stop` method to stop the virtual playback.

The `TapeRecorder` type also has `Play` and `Stop` methods, and a `Record` method as well.

```

 package gadget

 import "fmt"

 type TapePlayer struct {
 Batteries string
 }
 func (t TapePlayer) Play(song string) {
 fmt.Println("Playing", song)
 }
 func (t TapePlayer) Stop() {
 fmt.Println("Stopped!")
 }

 type TapeRecorder struct {
 Microphones int
 }
 Has a Play method just like TapePlayer's. {func (t TapeRecorder) Play(song string) {
 fmt.Println("Playing", song)
 }
 func (t TapeRecorder) Record() {
 fmt.Println("Recording")
 }
}
Has a Stop method just like TapePlayer's. {func (t TapeRecorder) Stop() {
 fmt.Println("Stopped!")
}

```

## A METHOD PARAMETER THAT CAN ONLY ACCEPT ONE TYPE

Here's a sample program that uses the `gadget` package. We define a `playList` function that takes a `TapePlayer` value, and a slice of song titles to play on it. The function loops over each title in the slice, and passes it to the `TapePlayer`'s `Play` method. When it's done playing the list, it calls `stop` on the `TapePlayer`.

Then, in the `main` method, all we have to do is create the `TapePlayer` and the slice of song titles, and pass them to `playList`.

```

package main
import "github.com/headfirstgo/gadget"

func playList(device gadget.TapePlayer, songs []string) {
 for _, song := range songs { ← Loop over each song.
 device.Play(song) ← Play the current song.
 }
 device.Stop() ← Stop the player once we're done.
}

func main() {
 player := gadget.TapePlayer{} ← Create a TapePlayer.
 mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"} ← Create a slice of
 playList(player, mixtape) ← song titles.
}

Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!

```

The `playList` function works great with a `TapePlayer` value. You might hope that it would work with a `TapeRecorder` as well. (After all, a tape recorder is basically just a tape player with an extra record function.) But `playList`'s first parameter has a type of `TapePlayer`. Try to pass it an argument of any other type, and you'll get a compile error:

```
func main() {
 player := gadget.TapeRecorder{}
 mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
 playList(player, mixtape)
}

Pass the TapeRecorder
to playList
```

Create a TapeRecorder  
instead of a TapePlayer..

Error

cannot use player (type gadget.TapeRecorder)
 as type gadget.TapePlayer in argument to playList



That's too bad... All the `playList` function  
really needs is a value whose type defines `Play`  
and `Stop` methods. Both `TapePlayer` and  
`TapeRecorder` have those!

```
func playList(device gadget.TapePlayer, songs []string) {
 for _, song := range songs {
 device.Play(song) ← Needs the value to have a Play
 method with a string parameter.
 }
 device.Stop() ← Needs the value to have a Stop
 method with no parameters.
}

type TapePlayer struct {
 Batteries string
}
func (t TapePlayer) Play(song string) { ← TapePlayer has a Play method
 with a string parameter...
}
func (t TapePlayer) Stop() { ← TapePlayer has a Stop method
 with no parameters...
}

type TapeRecorder struct {
 Microphones int
}
func (t TapeRecorder) Play(song string) { ← TapePlayer also has a Play method
 with a string parameter.
}
func (t TapeRecorder) Record() {
 fmt.Println("Recording")
}
func (t TapeRecorder) Stop() { ← TapePlayer also has a Stop method
 with no parameters...
}
```

In this case, it does seem like the Go language's type safety is getting in our way, rather than helping us. The `TapeRecorder` type defines all the methods that the `playList` function needs, but we're being blocked from using it because `playList` only accepts `TapePlayer` values.

So what can we do? Write a second, nearly-identical `playListWithRecorder` function that

takes a `TapeRecorder` instead?

Actually, Go offers another way...

## INTERFACES

When you install a program on your computer, you usually expect the program to provide you with a way to interact with it. You expect a word processor to give you a place to type text. You expect a backup program to give you a way to select which files to save. You expect a spreadsheet to give you a way to insert columns and rows for data. The set of controls a program provides you so you can interact with it are often called its "interface".

**An interface is a set of methods that certain values are expected to have.**

Whether you've actually thought about it or not, you probably expect Go values to provide you with a way to interact with them, too. What's the most common way to interact with a Go value? Through its methods.

In Go, an **interface** is defined as a set of methods that certain values are expected to have. You can think of an interface as a set of actions you need a type to be able to perform.

You define an interface type using the `interface` keyword, followed by curly braces containing a list of method names, along with any parameters or return values the methods are expected to have.

```
type myInterface interface {
 methodWithoutParameters()
 methodWithParameter(float64)
 methodWithReturnValue() string
}
```

The diagram shows a Go code snippet defining an interface named `myInterface`. The code uses the `interface` keyword. Three methods are listed: `methodWithoutParameters()`, `methodWithParameter(float64)`, and `methodWithReturnValue() string`. Annotations with arrows point to specific parts of the code:

- A blue arrow points from the text "Method name" to the first two method definitions (`methodWithoutParameters()` and `methodWithParameter(float64)`).
- A green arrow points from the text "Type of parameter" to the type argument `float64` in the second method definition.
- A red arrow points from the text "Type of return value" to the type argument `string` in the third method definition.

Any type that has all the methods listed in an interface definition is said to **satisfy** that interface. A type that satisfies an interface can be used anywhere that interface is called for.

I once bought a coffee maker that had no "brew" button! Not what I expected. I wasn't very satisfied with that purchase.



The method names, parameter types (or lack thereof), and return value types (or lack thereof) all need to match those defined in the interface. A type can have methods *in addition* to those listed in the interface, but it mustn't be *missing* any, or it doesn't satisfy that interface.

A type can satisfy multiple interfaces, and an interface can (and usually should) have multiple types that satisfy it.

## DEFINING A TYPE THAT SATISFIES AN INTERFACE

The code below sets up a quick experimental package, named `mypkg`. It defines an interface type named `MyInterface`, with three methods. Then it defines a type named `MyType` that satisfies `MyInterface`.

There are three methods required to satisfy `MyInterface`: a `MethodWithoutParameters` method, a `MethodWithParameter` method that takes a `float64` parameter, and a `MethodWithReturnValue` method that returns a `string`.

Then we declare another type, `MyType`. The underlying type of `MyType` doesn't matter in this example; we just used `int`. We define all the methods on `MyType` that it needs to satisfy `MyInterface`, plus one extra method that isn't part of the interface.



```

package mypkg

import "fmt" Declare an interface type.

type MyInterface interface { A type satisfies this interface
 MethodWithoutParameters() if it has this method...
 MethodWithParameter(float64) ...And this method (with
 MethodWithReturnValue() string a float64 parameter)...
}
} ...And this method (with
 a string return value).

type MyType int Declare a type. We'll make
 it satisfy myInterface.

func (m MyType) MethodWithoutParameters() { First required method.
 fmt.Println("MethodWithoutParameters called")
}
func (m MyType) MethodWithParameter(f float64) { Second required method
 fmt.Println("MethodWithParameter called with", f)
}
func (m MyType) MethodWithReturnValue() string { Third required method
 return "Hi from MethodWithReturnValue"
}
func (my MyType) MethodNotInInterface() { A type can still satisfy an interface
 fmt.Println("MethodNotInInterface called") even if it has methods that aren't
 part of the interface.
}
}

```

Many other languages would require us to explicitly say that `MyType` satisfies `MyInterface`. But in Go, this happens *automatically*. If a type has all the methods declared in an interface, then it can be used anywhere that interface is required, with no further declarations needed.

Here's a quick program that will let us try `mypkg` out.

A variable declared with an interface type can hold any value whose type satisfies that interface. This code declares a `value` variable with `MyInterface` as its type, then creates a `MyType` value and assigns it to `value`. (Which is allowed, because `MyType` satisfies `MyInterface`.) Then we call all the methods on that value that are part of the interface.

```

package main

import (
 "fmt"
 "mypkg" Declare a variable using
 the interface type.
)
func main() {
 var value mypkg.MyInterface
 We can call value = mypkg.MyType(5) Values of myType satisfy myInterface,
 so we can assign this value to a variable
 with a type of myInterface.
 any method that's part of { value.MethodWithoutParameters()
 value.MethodWithParameter(127.3)
 myInterface. } fmt.Println(value.MethodWithReturnValue())
}

```

MethodWithoutParameters called  
 MethodWithParameter called with 127.3  
 Hi from MethodWithReturnValue

# CONCRETE TYPES, ABSTRACT TYPES

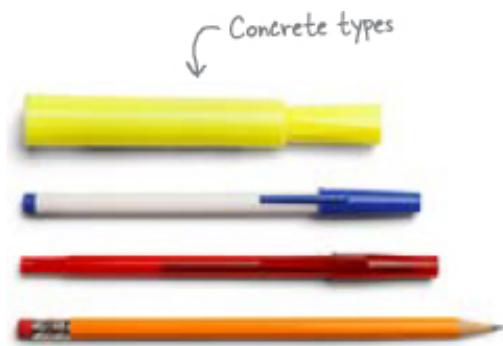
All the types we've defined in previous chapters have been concrete types. A **concrete type** specifies not only what its values can *do* (what methods you can call on them), but also what they *are*: they specify the underlying type that holds the value's data.

↓ Abstract type

**"I need something I  
can write with."**

An interface type is referred to as an **abstract type**. Abstract types don't describe what a value *is*: they don't say what its underlying type is, or how its data is stored. They only describe what a value can *do*: what methods it has.

Suppose you need to write down a quick note. In your desk drawer, you have values of several concrete types: `Pen`, `Pencil`, and `Marker`. Each of these concrete types defines a `write` method, so you don't really care which type you grab. You just want a `WritingInstrument`: an abstract type (interface type) that is satisfied by any concrete type with a `write` method.



## ASSIGN ANY TYPE THAT SATISFIES THE INTERFACE

When you have a variable with an interface type, it can hold values of any type that satisfies the interface.

Suppose we have `Whistle` and `Horn` types, each of which has a `makeSound` method. We can create a `NoiseMaker` interface that represents any type with a `makeSound` method. If we declare a `toy` variable with a type of `NoiseMaker`, we'll be able to assign either `Whistle` or `Horn` values to it. (Or any other type that we later declare, as long as it has a `makeSound` method.)

We can then call the `makeSound` method on any value assigned to the `toy` variable. Although we don't know exactly what concrete type the value in `toy` *is*, we know what it

can *do*: make sounds. If its type didn't have a `MakeSound` method, then it wouldn't satisfy the `NoiseMaker` interface, and we wouldn't have been able to assign it to the variable.

```
package main

import "fmt"

type Whistle string
func (w Whistle) MakeSound() {
 fmt.Println("Tweet!")
}

type Horn string
func (h Horn) MakeSound() {
 fmt.Println("Honk!")
}

type NoiseMaker interface {
 MakeSound()
}
Declare a NoiseMaker
variable.

func main() {
 var toy NoiseMaker
 toy = Whistle("Toyco Canary")
 toy.MakeSound()
 toy = Horn("Toyco Blaster")
 toy.MakeSound()
}

Tweet!
Honk!
```

Has a MakeSound method.

Also has a MakeSound method.

Represents any type with a MakeSound method.

Assign a value of a type that satisfies NoiseMaker to the variable.

Assign a value of another type that satisfies NoiseMaker to the variable.

You can declare function parameters with interface types as well. (After all, function parameters are really just variables too.) If we declare a `play` function that takes a `NoiseMaker`, for example, then we can pass any value from a type with a `MakeSound` method to `play`:

```
func play(n NoiseMaker) {
 n.MakeSound()
}

func main() {
 play(Whistle("Toyco Canary"))
 play(Horn("Toyco Blaster"))
}
```

Tweet!  
Honk!

## YOU CAN ONLY CALL METHODS DEFINED AS PART OF THE INTERFACE

Once you assign a value to a variable (or method parameter) with an interface type, you can *only* call methods that are specified by the interface on it.

Suppose we created a `Robot` class, which in addition to a `MakeSound` method, also has a `Walk` method. We add a call to `Walk` in the `play` function, and pass a new `Robot` value to `play`.

But the code doesn't compile, saying that `NoiseMaker` values don't have a `Walk` method.

Why is that? `Robot` values *do* have a `Walk` method; the definition is right there!

But it's *not* a `Robot` value that we're passing to the `play` function; it's a `NoiseMaker`. What if we had passed a `Whistle` or `Horn` to `play` instead? Those don't have `Walk` methods!

When we have a variable of an interface type, the only methods we can be sure it has are the methods that are defined in the interface. And so, those are the only methods Go allows you to call. (There is a way to get at the value's concrete type, so that you can call more specialized methods. We'll look at that shortly.)

```
package main

import "fmt"

type Whistle string

func (w Whistle) MakeSound() {
 fmt.Println("Tweet!")
}

type Horn string

func (h Horn) MakeSound() {
 fmt.Println("Honk!")
}

type Robot string
 ↴ Declare a new Robot type.
 ↴ Robot satisfies
 ↴ the NoiseMaker
 ↴ interface.

func (r Robot) MakeSound() {
 fmt.Println("Beep Boop")
}
 ↴ An additional method.

func (r Robot) Walk() {
 fmt.Println("Powering legs")
}

type NoiseMaker interface {
 MakeSound()
}

func play(n NoiseMaker) {
 ↪ OK. Part of NoiseMaker interface.
 ↪ Not OK! Not part of NoiseMaker!
 n.MakeSound()
}

func main() {
 play(Robot("Botco Ambler"))
}
 ↴ Error

 n.Walk undefined
 (type NoiseMaker has no
 field or method Walk)
```

Note that it *is* just fine to assign a type that *has* other methods to a variable with an interface type. As long as you don't actually call those other methods, everything will work.

```
func play(n NoiseMaker) {
 n.MakeSound() ← Call only methods that are
} part of the interface.

func main() {
 play(Robot("Botco Ambler")) Beep Boop
}
```



## BREAKING STUFF IS EDUCATIONAL!

Here are a couple concrete types, `Fan` and `CoffeePot`. We also have an `Appliance` interface with a `TurnOn` method. `Fan` and `CoffeePot` both have `TurnOn` methods, so they both satisfy the `Appliance` interface.

That's why, in the `main` function, we're able to define an `Appliance` variable, and assign both `Fan` and `CoffeePot` variables to it.

Make one of the changes below, and try to compile the code. Then undo your change, and try the next one. See what happens!

```
type Fan string
func (f Fan) TurnOn() {
 fmt.Println("Spinning")
}
```

```
type CoffeePot string
func (c CoffeePot) TurnOn() {
 fmt.Println("Powering up")
}
func (c CoffeePot) Brew() {
 fmt.Println("Heating Up")
}
```

```
type Appliance interface {
 TurnOn()
}

func main() {
 var device Appliance
 device = Fan("Windco Breeze")
```

```

device.TurnOn()
device = CoffeePot("LuxBrew")
device.TurnOn()
}

```

|                                                                                                                                                                              |                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| If you do this...                                                                                                                                                            | ...the code will break because...                                                                                                                                                                                      |
| Call a method from the concrete type that isn't defined in the interface:<br><code>device.Brew()</code>                                                                      | When you have a value in a variable with an interface type, you can only call methods defined as part of that interface, regardless of what methods the concrete type had.                                             |
| Remove the method that satisfies the interface from a type:<br><br><del>func (c CoffeePot) TurnOn() {     fmt.Println("Powering     up") }</del>                             | If a type doesn't satisfy an interface, you can't assign values of that type to variables that use that interface as their type.                                                                                       |
| Add a new return value or parameter on the method that satisfies the interface:<br><br><del>func (f Fan) TurnOn() error {     fmt.Println("Spinning")     return nil }</del> | If the number and types of all parameters and return values don't match between a concrete type's method definition and the method definition in the interface, then the concrete type does not satisfy the interface. |

## FIXING OUR PLAYLIST FUNCTION USING AN INTERFACE

Let's see if we can use an interface to allow our `playList` function to work with the `Play` and `Stop` methods on both of our concrete types: `TapePlayer` and `TapeRecorder`.

```
// TapePlayer type definition here
func (t TapePlayer) Play(song string) {
 fmt.Println("Playing", song)
}

func (t TapePlayer) Stop() {
 fmt.Println("Stopped!")
}

// TapeRecorder type definition here
func (t TapeRecorder) Play(song string) {
 fmt.Println("Playing", song)
}

func (t TapeRecorder) Record() {
 fmt.Println("Recording")
}

func (t TapeRecorder) Stop() {
 fmt.Println("Stopped!")
}
```

In our `main` package, we declare a `Player` interface. (We could define it in the `gadget` package instead, but defining the interface in the same package where we use it gives us more flexibility.) We specify that the interface requires both a `Play` method with a `string` parameter, and a `Stop` method with no parameters. This means that both the `TapePlayer` and `TapeRecorder` types will satisfy the `Player` interface.

We update the `playList` function to take any value that satisfies `Player` instead of `TapePlayer` specifically. We also change the type of the `player` variable from `TapePlayer` to `Player`. This allows us to assign either a `TapePlayer` or a `TapeRecorder` to `player`. We then pass values of both types to `playList`!

```

package main

import "github.com/headfirstgo/gadget"

type Player interface { ← Define an interface type...
 Play(string) ← Require a Play method with a string parameter.
 Stop() ← Also require a Stop method.
}

func playList(device Player, songs []string) {
 for _, song := range songs {
 device.Play(song)
 }
 device.Stop()
}

func main() {
 mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
 var player Player = gadget.TapePlayer{} ← Update the
 playList(player, mixtape) ← Pass a TapePlayer variable to hold
 player = gadget.TapeRecorder{} to playList. any Player.
 playList(player, mixtape) ←
} ← Pass a TapeRecorder
 to playList.

```

```

Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!

```



**Watch it!**

If a type declares methods with pointer receivers, then you'll only be able to use pointers to that type when assigning to interface variables.

The `toggle` method on the `Switch` type below has to use a pointer receiver so it can modify the receiver.

```
package main
```

```
import "fmt"
```

```

type Switch string
func (s *Switch) toggle() {
 if *s == "on" {
 *s = "off"
 } else {
 *s = "on"
 }
 fmt.Println(*s)
}

```

```

type Toggleable interface {
 toggle()
}

```

```
}
```

```
func main() {
 s := Switch("off")
 var t Toggleable = s
 t.toggle()
 t.toggle()
}
```

*But that results in an error when we assign a `Switch` value to a variable with the interface type `Toggleable`:*

`Switch does not implement Toggleable  
(toggle method has pointer receiver)`

*When Go decides whether a value satisfies an interface, pointer methods aren't included for direct values. But they are included for pointers. So the solution is to assign a pointer to a `Switch` to the `Toggleable` variable, instead of a direct `Switch` value:*

`var t Toggleable = &s` ← Assign a pointer instead.

*Make that change, and the code should work correctly.*

## THERE ARE NO DUMB QUESTIONS

**Q: Should interface type names begin with a capital letter, or a lower-case letter?**

**A:** The rules for interface type names are the same as the rules for any other type. If the name begins with a lower-case letter, then the interface type will be *unexported* and will not be accessible outside the current package. Sometimes you won't need to use the interface you're declaring from other packages, so making it unexported is fine. But if you *do* want to use it in other packages, you'll need to start the interface type's name with a capital letter, so that it's *exported*.



The code at the right defines `car` and `truck` types, each of which have `Accelerate`, `Brake`,

and `Steer` methods. Fill in the blanks to add a `Vehicle` interface that includes those three methods, so that the code in the `main` function will compile and produce the output shown.

```
package main

import "fmt"

type Car string
func (c Car) Accelerate() {
 fmt.Println("Speeding up")
}
func (c Car) Brake() {
 fmt.Println("Stopping")
}
func (c Car) Steer(direction string) {
 fmt.Println("Turning", direction)
}

type Truck string
func (t Truck) Accelerate() {
 fmt.Println("Speeding up")
}
func (t Truck) Brake() {
 fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
 fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
 fmt.Println("Loading", cargo)
}
```

Your code here! → \_\_\_\_\_

```
func main() {
 var vehicle Vehicle = Car("Toyoda Yarvic")
 vehicle.Accelerate()
 vehicle.Steer("left")

 vehicle = Truck("Fnord F180")
 vehicle.Brake()
 vehicle.Steer("right")
}
```

Speeding up  
Turning left  
Stopping  
Turning right

## TYPE ASSERTIONS

We've defined a new `Tryout` function that will let us test the various methods of our `TapePlayer` and `TapeRecorder` types. `Tryout` has a single parameter with the `Player` interface as its type, so that we can pass in either a `TapePlayer` OR `TapeRecorder`.

Within `TryOut`, we call the `Play` and `Stop` methods, which are both part of the `Player` interface. We also call the `Record` method, which is *not* part of the `Player` interface, but is defined on the `TapeRecorder` type. We're only passing a `TapeRecorder` value to `TryOut` for now, so we should be fine, right?

Unfortunately, no. We saw earlier that if a value of a concrete type is assigned to a variable with an interface type (including function parameters), then you can only call methods on it that are part of that interface, regardless of what other methods the concrete type has. Within the `TryOut` function, we don't have a `TapeRecorder` value (the concrete type), we have a `Player` value (the interface type). And the `Player` interface doesn't have a `Record` method!

```
type Player interface {
 Play(string)
 Stop()
}

func TryOut(player Player) {
 player.Play("Test Track")
 player.Stop()
 Not part of Player! → player.Record()
}

func main() {
 TryOut(gadget.TapeRecorder{})
}
```

These are fine; they're part of the Player interface.

Pass a TapeRecorder (which satisfies Player) to the function.

Error

player.Record undefined (type Player has no field or method Record)



We need a way to get the concrete type value (which *does* have a `Record` method) back.

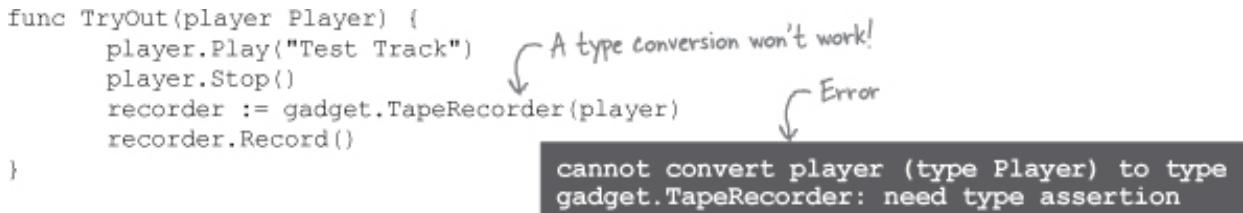
Your first instinct might be to try a type conversion to convert the `Player` value to a `TapeRecorder` value. But type conversions aren't meant for use with interface types, so that generates an error. The error message suggests trying something else:

```
func TryOut(player Player) {
 player.Play("Test Track")
 player.Stop()
 recorder := gadget.TapeRecorder(player)
 recorder.Record()
}
```

A type conversion won't work!

Error

cannot convert player (type Player) to type gadget.TapeRecorder: need type assertion



A "type assertion"? What's that?

When you have a value of a concrete type assigned to a variable with an interface type, a **type assertion** lets you get the concrete type back. It's kind of like a type conversion. Its syntax even looks like a cross between a method call and a type conversion. After an interface value, you type a dot, and a pair of parentheses with the original type. (Or rather, what you're *asserting* the value's original type is.)

```

var noiseMaker NoiseMaker = Robot("Botco Ambler")
var robot Robot = noiseMaker.(Robot)
 |
 +-- Interface value
 +-- Asserted type

```

In plain language, the type assertion above says something like "I know this variable uses the interface type `NoiseMaker`, but I'm pretty sure *this* `NoiseMaker` is actually a `Robot`".

Once you've used a type assertion to get a value of a concrete type back, you can call methods on it that are defined on that type, but aren't part of the interface.

This code assigns a `Robot` to a `NoiseMaker` interface value. We're able to call `MakeSound` on the `NoiseMaker`, because it's part of the interface. But to call the `Walk` method, we need to use a type assertion to get a `Robot` value. Once we have a `Robot` (rather than a `NoiseMaker`), we can call `Walk` on it.

```

type Robot string
func (r Robot) MakeSound() {
 fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
 fmt.Println("Powering legs")
}

type NoiseMaker interface {
 MakeSound()
 Define a variable with
 an interface type...
}
func main() {
 var noiseMaker NoiseMaker = Robot("Botco Ambler")
 noiseMaker.MakeSound() ← Call a method that's part of the interface.
 var robot Robot = noiseMaker.(Robot) ← Convert back to the original
 robot.Walk() ← Call a method that's
 defined on the original type
 type using a type assertion.
}

Beep Boop
Powering legs

```

...And assign a value of a type  
that satisfies the interface.

Call a method that's part of the interface.

Convert back to the original type using a type assertion.

Call a method that's defined on the original type (not the interface).

## TYPE ASSERTION FAILURES

Previously, our `TryOut` function wasn't able to call the `Record` method on a `Player` value, because it's not part of the `Player` interface. Let's see if we can get this working using a type assertion.

Just like before, we pass a `TapeRecorder` to `TryOut`, where it gets assigned to a parameter that uses the `Player` interface as its type. We're able to call the `Play` and `Stop` methods on the `Player` value, because those are both part of the `Player` interface.

Then, we use a type assertion to convert the `Player` back to a `TapeRecorder`. And we call

Record on the `TapeRecorder` value instead.

```
type Player interface {
 Play(string)
 Stop()
}

func TryOut(player Player) {
 player.Play("Test Track")
 player.Stop()
 recorder := player.(gadget.TapeRecorder) ← Use a type assertion to
 recorder.Record() ← get a TapeRecorder value.

 Store the TapeRecorder value. →
}

func main() {
 TryOut(gadget.TapeRecorder{})
}
```

Call the method that's only defined on the concrete type.

Playing Test Track  
Stopped!  
Recording

Everything seems to be working great... with a `TapeRecorder`. But what happens if we try to pass a `TapePlayer` to `Tryout`? How well will that work, considering we have a type assertion that says the parameter to `Tryout` is actually a `TapeRecorder`?

```
func main() {
 TryOut(gadget.TapeRecorder{})
 TryOut(gadget.TapePlayer{}) ← Pass a TapePlayer as well...
}
```

Everything compiles successfully, but when we try to run it, we get a runtime panic! As you might expect, trying to assert that a `TapePlayer` is actually a `TapeRecorder` did not go well. (It's simply not true, after all.)

Playing Test Track  
Stopped!  
Recording  
Playing Test Track  
Stopped!

Panic! → panic: interface conversion: main.Player  
is gadget.TapePlayer, not gadget.TapeRecorder

## AVOIDING PANICS WHEN TYPE ASSERTIONS FAIL

If a type assertion is used in a context that expects only one return value, then if the original type doesn't match the type in the assertion, the program will panic at run time (*not* when compiling):

```
var player Player = gadget.TapePlayer{}
recorder := player.(gadget.TapeRecorder) ← Assert that the original type is a TapeRecorder,
 when it's actually a TapePlayer....
```

Panic! → panic: interface conversion: main.Player  
is gadget.TapePlayer, not gadget.TapeRecorder

If a type assertions are used in a context where multiple return values are expected, they have a second, optional return value that indicates whether the assertion was successful or not. (And the assertion won't panic if it's unsuccessful.) The second value is a `bool`, and it will be `true` if the value's original type was the asserted type, or `false` if not. You can do whatever you want with this second return value, but by convention it's usually assigned to a variable named `ok`. (This is similar to accessing a `map`, which also has an optional return value that lets you tell zero values apart from explicitly assigned values.)

Here's an update to the above code that assigns the results of the type assertion to a variable for the concrete type's value, and a second `ok` variable. It uses the `ok` value in an `if` statement to determine whether it can safely call `Record` on the concrete value (because the `Player` value had an original type of `TapeRecorder`), or if it should skip doing so (because the `Player` had some other concrete value).

```
var player Player = gadget.TapePlayer()
recorder, ok := player.(gadget.TapeRecorder)
if ok { ↑ Assign the second return value to a variable.
 recorder.Record() ← If the original type was TapeRecorder, call Record on the value.
} else {
 fmt.Println("Player was not a TapeRecorder") ← Otherwise, report that
} the assertion failed.
```

Player was not a TapeRecorder

In this case, the concrete type was `TapePlayer`, not `TapeRecorder`, so the assertion is unsuccessful, and `ok` is `false`. The `if` statement's `else` clause runs, printing "Player was not a TapeRecorder". A runtime panic is averted.

When using type assertions, if you're not absolutely sure which original type is behind the interface value, then you should use the optional `ok` value to handle cases where it's a different type than you expected, and avoid a runtime panic.

## TESTING TAPEPLAYERS AND TAPERECORDERS USING TYPE ASSERTIONS

Let's see if we can use what we've learned to fix our `TryOut` function for `TapePlayer` and `TapeRecorder` values. Instead of ignoring the second return value from our type assertion, we'll assign it to an `ok` variable. The `ok` variable will be `true` if the type assertion is successful (indicating the `recorder` variable holds a `TapeRecorder` value, ready for us to call `Record` on it), or `false` otherwise (indicating it's *not* safe to call `Record`). We wrap the call to the `Record` method in an `if` statement to ensure it's only called when the type assertion is successful.

```

type Player interface {
 Play(string)
 Stop()
}

func TryOut(player Player) {
 player.Play("Test Track")
 player.Stop()
 recorder, ok := player.(gadget.TapeRecorder)
 Call the Record method only if the original value was a TapeRecorder.
 if ok { ↗ Assign the second return value to a variable.
 recorder.Record()
 }
}

func main() {
 TryOut(gadget.TapeRecorder{})
 TryOut(gadget.TapePlayer{})
}

```

TapeRecorder passed in... → Playing Test Track  
Type assertion succeeds, Record called. → Stopped!  
TapePlayer passed in... → Recording  
→ Playing Test Track  
→ Stopped!

Type assertion does not succeed, Record not called.

As before, in our `main` function, we first call `Tryout` with a `TapeRecorder` value. `Tryout` takes the `Player` interface value it receives, and calls the `Play` and `Stop` methods on it. The assertion that the `Player` value's concrete type is `TapeRecorder` succeeds, and the `Record` method is called on the resulting `TapeRecorder` value.

Then, we call `Tryout` again with a `TapePlayer`. (This is the call that halted the program previously because the type assertion panicked.) `Play` and `Stop` are called, as before. The type assertion fails, because the `Player` value holds a `TapePlayer` and not a `TapeRecorder`. But because we're capturing the second return value in the `ok` value, the type assertion doesn't panic this time. It just sets `ok` to `false`, which causes the code in our `if` statement not to run, which causes `Record` not to be called. (Which is good, because `TapePlayer` values don't have a `Record` method.)

Thanks to type assertions, we've got our `Tryout` function working with both `TapeRecorder` and `TapePlayer` values!



## POOL PUZZLE

Updated code from our previous exercise is at the right. We're creating a `TryVehicle` method that calls all the methods from the `Vehicle` interface. Then, it should attempt a type assertion to get a concrete `Truck` value. If successful, it should call `LoadCargo` on the

Truck value.

Your **job** is to take code snippets from the pool and place them into the blank lines in this code. **Don't** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a program that will run and produce the output shown.

```
type Truck string
func (t Truck) Accelerate() {
 fmt.Println("Speeding up")
}
func (t Truck) Brake() {
 fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
 fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
 fmt.Println("Loading", cargo)
}

type Vehicle interface {
 Accelerate()
 Brake()
 Steer(string)
}

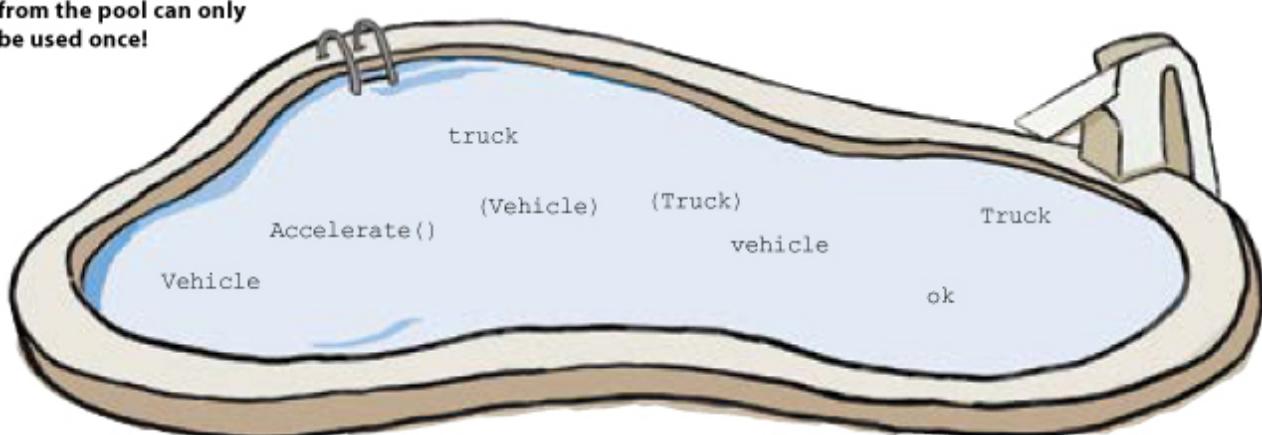
func TryVehicle(vehicle _____) {
 vehicle._____
 vehicle.Steer("left")
 vehicle.Steer("right")
 vehicle.Brake()
 truck, ___ := vehicle._____
 if ok {
 _____.LoadCargo("test cargo")
 }
}

func main() {
 TryVehicle(Truck("Fnord F180"))
}
```

Output →

Speeding up  
Turning left  
Turning right  
Stopping  
Loading test cargo

Note: each snippet  
from the pool can only  
be used once!

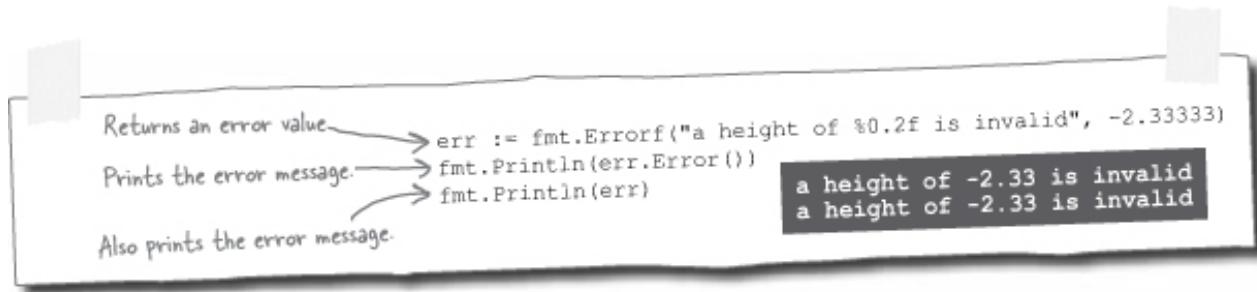


## THE "ERROR" INTERFACE

We'd like to wrap up the chapter by looking at a few interfaces that are built into Go. We haven't covered these interfaces explicitly, but you've actually been using them all

along.

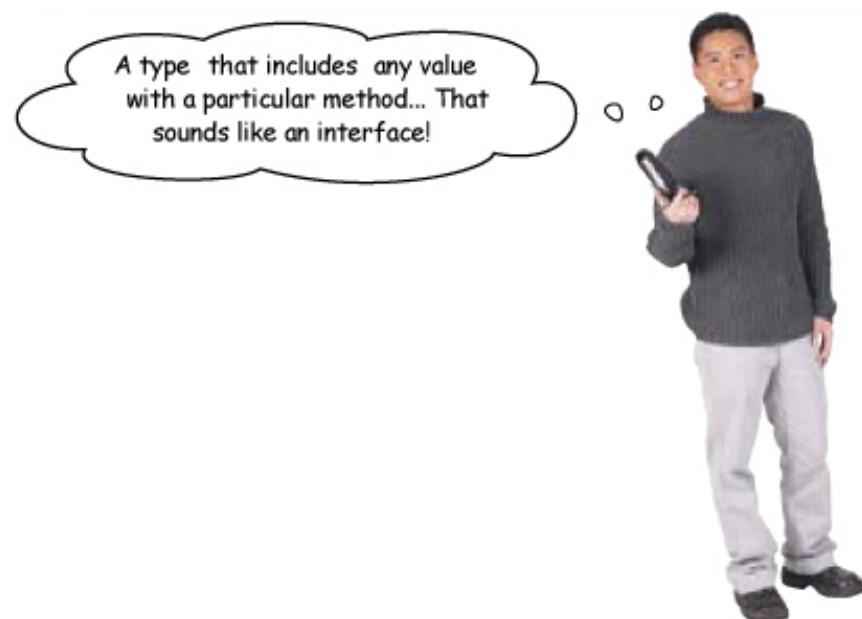
In Chapter 3, we learned how to create our own `error` values. We said, "An error value is any value with a method named `Error` that returns a string."



That's right. The `error` type is just an interface! It looks something like this:

```
type error interface {
 Error() string
}
```

Declaring the `error` type as an interface means that if it has an `Error` method that returns a `string`, it satisfies the `error` interface, and it's an `error` value. That means you can define your own types and use them anywhere an `error` value is required!



For example, here's a simple defined type, `ComedyError`. Because it has an `Error` method that returns a `string`, it satisfies the `error` interface, and we can assign it to a variable with the type `error`.

```

Define a type with an
underlying type of "string".
type ComedyError string
func (c ComedyError) Error() string { ← Satisfy the error interface.
 return string(c) ← The Error method needs to return a string, so do a type conversion.
}

Set up a variable with
a type of "error"
func main() {
 var err error
 err = ComedyError("What's a programmer's favorite beer? Logger!")
 fmt.Println(err)
}

What's a programmer's favorite beer? Logger!

```

If you need an `error` value, but also need to track more information about the error than just an error message string, you can create your own type that satisfies the `error` interface *and* stores the information you want.

Suppose you're writing a program that monitors some equipment to ensure it doesn't overheat. Here's an `OverheatError` type that might be useful. It has an `Error` method, so it satisfies `error`. But more interestingly, it uses `float64` as its underlying type, allowing us to track the degrees over capacity.

```

Define a type with an
underlying type of float64.
type OverheatError float64
func (o OverheatError) Error() string { ← Satisfy the error
 return fmt.Sprintf("Overheating by %0.2f degrees!", o) ← interface.
}

Use the temperature
in the error message.

```

Here's a `checkTemperature` function that uses `OverheatError`. It takes the system's actual temperature and the temperature that's considered safe as parameters. It specifies that it returns a value of type `error`, not an `OverheatError` specifically, but that's okay because `OverheatError` satisfies the `error` interface. If the `actual` temperature is over the `safe` temperature, `checkTemperature` returns a new `OverheatError` that records the excess.

```

func checkTemperature(actual float64, safe float64) error {
 excess := actual - safe If the actual temperature is in
 if excess > 0 { excess of the safe temperature...
 return OverheatError(excess)
 }
 return nil
}

func main() {
 var err error = checkTemperature(121.379, 100.0)
 if err != nil {
 log.Fatal(err)
 }
}

```

2018/04/02 19:27:44 Overheating by 21.38 degrees!

## *THERE ARE NO DUMB QUESTIONS*

**Q:** How is it we've been using the `error` interface type in all these different packages, without importing it? Its name begins with a lower-case letter. Doesn't that mean it's unexported, from whatever package it's declared in?

What package is

`error` declared in, anyway?

**A:** The `error` type is a "predeclared identifier", like `int` or `string`. And so, like other predeclared identifiers, it's not part of any package. It's part of the "universe block", meaning it's available everywhere, regardless of what package you're in.

Remember how there are `if` and `for` blocks, which are encompassed by function blocks, which are encompassed by package blocks? Well, the universe block encompasses all package blocks. That means you can use anything defined in the universe block from any package, without importing it. And that includes `error`, and all other predeclared identifiers.

## THE STRINGER INTERFACE

Remember our `Gallons`, `Liters`, and `Milliliters` types, which we created back in Chapter 9 to distinguish between various units for measuring volume? We're discovering that it's not so easy to distinguish between them after all. Twelve gallons is a very different amount than twelve liters or twelve milliliters, but they all look the same when printed. If there are too many decimal places of precision on a value, that looks awkward when printed, too.

```

type Gallons float64
type Liters float64
type Milliliters float64

func main() {
 fmt.Println(Gallons(12.09248342)) ← Create and print a
 fmt.Println(Liters(12.09248342)) ← Create and print a Liters value.
 fmt.Println(Milliliters(12.09248342)) ← Create and print a Milliliters value.
}

All three values look identical! { 12.09248342
 12.09248342
 12.09248342

```

You can use `Printf` to round the number off and add an abbreviation indicating the unit of measure, but doing that every place you need to use these types would quickly get tedious.

Format the numbers and add abbreviations. { `fmt.Printf("%0.2f gal\n", Gallons(12.09248342))` 12.09 gal  
`fmt.Printf("%0.2f L\n", Liters(12.09248342))` 12.09 L  
`fmt.Printf("%0.2f mL\n", Milliliters(12.09248342))` 12.09 mL }

That's why the `fmt` package defines the `fmt.Stringer` interface: to allow any type to decide how it will be displayed when printed. It's easy to set up any type to satisfy `Stringer`; just define a `String()` method that returns a `string`. The definition looks like this:

```

type Stringer interface { Any type is a fmt.Stringer
 String() string ← if it has a String method
} that returns a string.

```

For example, here we've set up this `CoffeePot` type to satisfy `Stringer`:

```

type CoffeePot string
func (c CoffeePot) String() string { ← Satisfy the Stringer interface.
 return "I'm a little teapot!" ← Method needs to return a string.
}
func main() {
 coffeePot := CoffeePot("LuxBrew")
 fmt.Println(coffeePot.String())
}

```

I'm a little teapot!

Many functions in the `fmt` package check whether the values passed to them satisfy the `Stringer` interface, and call their `String` methods if so. This includes the `Print`, `Println`, and `Printf` functions and more. Now that `CoffeePot` satisfies `Stringer`, we can pass `CoffeePot` values directly to these functions, and the return value of the `CoffeePot`'s `String` method will be used in the output:

Pass the CoffeePot to various fmt functions. { `coffeePot := CoffeePot("LuxBrew")` Create a CoffeePot value.  
`fmt.Print(coffeePot, "\n")`  
`fmt.Println(coffeePot)`  
`fmt.Printf("%s", coffeePot)`

I'm a little teapot!  
I'm a little teapot!  
I'm a little teapot!

The return value of String is used in the output.

Now for a more serious use of this interface type. Let's make our `Gallons`, `Liters`, and `Milliliters` types satisfy `Stringer`. We'll move our code to format their values to `String` methods associated with each type. We'll call the `Sprintf` function instead of `Printf`, and return the resulting value.

```
type Gallons float64
func (g Gallons) String() string { ← Make Gallons satisfy Stringer.
 return fmt.Sprintf("%0.2f gal", g)
}

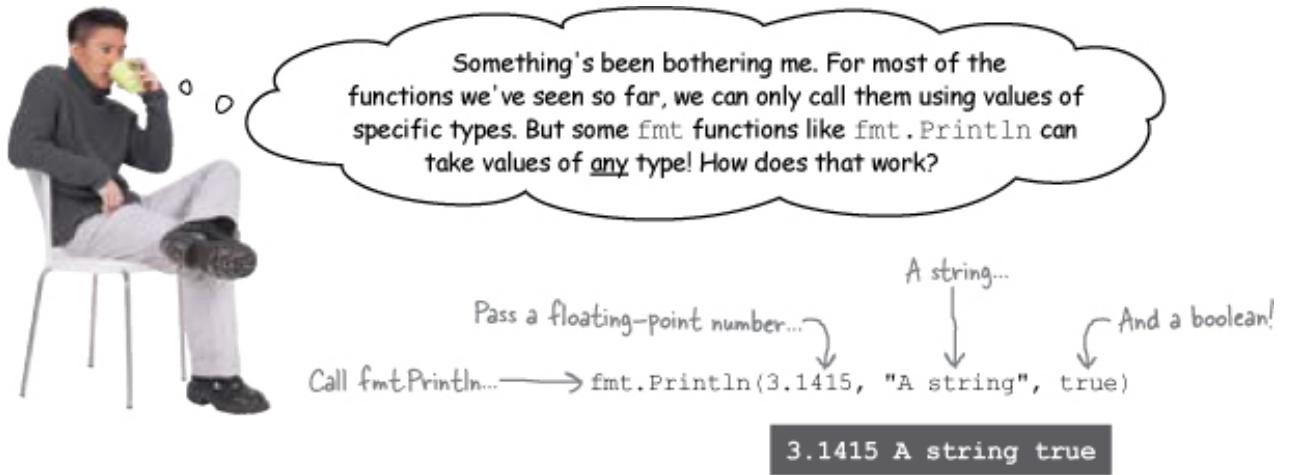
type Liters float64
func (l Liters) String() string { ← Make Liters satisfy Stringer.
 return fmt.Sprintf("%0.2f L", l)
}

type Milliliters float64
func (m Milliliters) String() string { ← Make Milliliters satisfy Stringer.
 return fmt.Sprintf("%0.2f mL", m)
}

func main() {
 Pass values of each type to Println. {fmt.Println(Gallons(12.09248342))
 fmt.Println(Liters(12.09248342))
 fmt.Println(Milliliters(12.09248342))} } The return values of each type's String method are used in the output.
}
```

Now, any time we pass `Gallons`, `Liters`, and `Milliliters` values to `Println` (or most other `fmt` functions), their `String` methods will be called, and the return values used in the output. We've set up a useful default format for printing each of these types!

## THE EMPTY INTERFACE



Good question! Let's run `go doc` to bring up the documentation for `fmt.Println` and see what type its parameters are declared as...

View documentation for  
the "fmt" package's  
"Println" function.

The "... shows it's a variadic  
function. But what's this  
"interface[]" type?

```
File Edit Window Help
$ go doc fmt.Println
func Println(a ...interface{}) (n int, err error)
 Println formats using the default formats for its operands and writes to
 standard output. Spaces are always added between operands and a newline...
```

As we saw in Chapter 6, the ... means that it's a variadic function, meaning it can take any number of parameters. But what's this interface{} type?

Remember, an interface declaration specifies the methods that a type is required to have in order to satisfy that interface. For example, our NoiseMaker interface is satisfied by any type that has a MakeSound method.

```
type NoiseMaker interface {
 MakeSound()
}
```

But what would happen if we declared an interface type that didn't require any methods at all? It would be satisfied by *any* type! It would be satisfied by *all* types!

```
type Anything interface { }
```

The type interface{} is known as **the empty interface**, and it's used to accept values of *any* type. The empty interface doesn't have any methods that are required to satisfy it, and so *every* type satisfies it.

If you declare a function that accepts a parameter with the empty interface as its type, then you can pass it values of any type as an argument:

func AcceptAnything(thing interface{}) { }

func main() { }

These are all valid types to pass to our function!

AcceptAnything(3.1415)

AcceptAnything("A string")

AcceptAnything(true)

AcceptAnything(Whistle("Toyco Canary"))

Accepts a parameter with the empty interface as its type.

**The empty interface doesn't require any methods to satisfy it, and so it's satisfied by all types.**

But don't rush out and start using the empty interface for all your function parameters! If you have a value with the empty interface as its type, there's not much you can *do* with it.

Most of the functions in `fmt` accept empty-interface values, so you can pass it on to those:

```
func AcceptAnything(thing interface{}) {
 fmt.Println(thing)
}

func main() {
 AcceptAnything(3.1415)
 AcceptAnything(Whistle("Toyco Canary"))
}
```

3.1415Toyco Canary

But don't try calling any methods on an empty-interface value! Remember, if you have a value with an interface type, you can only call methods on it that are part of the interface. And the empty interface doesn't *have* any methods. That means there are *no* methods you can call on a value with the empty interface type!

```
func AcceptAnything(thing interface{}) {
 fmt.Println(thing)
 thing.MakeSound() ← Try to call a method on the empty-interface value...
}

thing.MakeSound undefined (type interface {} is interface with no methods)
```

Error

To call methods on a value with the empty interface type, you'd need to use a type assertion to get a value of the concrete type back.

```
func AcceptAnything(thing interface{}) {
 fmt.Println(thing)
 whistle, ok := thing.(Whistle) ← Use a type assertion to
 get a Whistle.
 if ok {
 whistle.MakeSound() ← Call the method on the Whistle.
 }
}

func main() {
 AcceptAnything(3.1415)
 AcceptAnything(Whistle("Toyco Canary"))
}
```

3.1415Toyco CanaryTweet!

And by that point, you're probably better off writing a function that accepts only that specific concrete type.

```
func AcceptWhistle(whistle Whistle) { ← Accept a Whistle.
 fmt.Println(whistle)
 whistle.MakeSound() ← Call the method. No
 type conversion needed.
}
```

So there are limits to the usefulness of the empty interface when defining your own

functions. But you'll use the empty interface all the time with the functions in the `fmt` package, and in other places too. The next time you see an `interface{}` parameter in a function's documentation, you'll know exactly what it means!

When you're defining variables or function parameters, often you'll know exactly what the value you'll be working with *is*. You'll be able to use a concrete type like `Pen`, `Car`, or `Whistle`. Other times, though, you only care about what the value can *do*. In that case, you're going to want to define an interface type, like `WritingInstrument`, `Vehicle`, or `NoiseMaker`.

You'll define the methods you need to be able to call as part of the interface type. And you'll be able to assign to your variables or call your functions without worrying about the concrete type of your values. If it has the right methods, you'll be able to use it!



## YOUR GO TOOLBOX

**That's it for Chapter 11! You've added interfaces to your toolbox.**

### Interfaces

An interface is a set of methods certain values are expected to have.

Any type that has all the methods listed in an interface definition is said to satisfy that interface.

A type that satisfies an interface can be assigned to any variable or function parameter that uses that interface as its type.



### BULLET POINTS

- A concrete type specifies not only what its values can *do* (what methods you can call on them), but also what they *are*: they specify the underlying type that holds the value's data.
- An interface type is an abstract type. Interfaces don't describe what a value *is*: they don't say what its underlying type is, or how its data is stored. They only describe what a value can *do*: what methods it has.
- An interface definition needs to contain a list of method names, along with any parameters or return values those methods are expected to have.
- To satisfy an interface, a type must have all the methods the interface specifies. Method names, parameter types (or lack thereof), and return value types (or lack thereof) all need to match those defined in the interface.
- A type can have methods in addition to those listed in the interface, but it mustn't be missing any, or it doesn't satisfy that interface.
- A type can satisfy multiple interfaces, and an interface can have multiple types that satisfy it.
- Interface satisfaction is automatic. There is no need to explicitly declare that a concrete type satisfies an interface in Go.
- When you have a variable of an interface type, the only methods you can call on it are those defined in the interface.
- If you've assigned a value of a concrete type to a variable with an interface type, you can use a type assertion to get the concrete type value back. Only then can you call methods that are defined on the concrete type (but not the interface.)
- Type assertions return a second `bool` value that indicates whether the assertion was successful.

```
car, ok := vehicle.(Car)
```



```

type Car string
func (c Car) Accelerate() {
 fmt.Println("Speeding up")
}
func (c Car) Brake() {
 fmt.Println("Stopping")
}
func (c Car) Steer(direction string) {
 fmt.Println("Turning", direction)
}

type Truck string
func (t Truck) Accelerate() {
 fmt.Println("Speeding up")
}
func (t Truck) Brake() {
 fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
 fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
 fmt.Println("Loading", cargo)
}

type Vehicle interface {
 Accelerate()
 Brake()
 Steer(string)
}

func main() {
 var vehicle Vehicle = Car("Toyoda Yarvic")
 vehicle.Accelerate()
 vehicle.Steer("left")

 vehicle = Truck("Fnord F180")
 vehicle.Brake()
 vehicle.Steer("right") Speeding up
 Turning left
 Stopping
 Turning right
}

```

## POOL PUZZLE SOLUTION

```

type Truck string
func (t Truck) Accelerate() {
 fmt.Println("Speeding up")
}
func (t Truck) Brake() {
 fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
 fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
 fmt.Println("Loading", cargo)
}

type Vehicle interface {
 Accelerate()
 Brake()
 Steer(string)
}

func TryVehicle(vehicle Vehicle) {
 vehicle.Accelerate()
 vehicle.Steer("left")
 vehicle.Steer("right")
 vehicle.Brake()
 truck, ok := vehicle.(Truck)
 if ok {
 truck.LoadCargo("test cargo")
 }
}

func main() {
 TryVehicle(Truck("Fnord F180"))
}

```

Speeding up  
 Turning left  
 Turning right  
 Stopping  
 Loading test cargo