

MODULE-1

INTRODUCTION TO GO PROGRAMMING

Topics

- 1)Introduction
- 2)Types
- 3)Variables
- 4)The Terminal
- 5)The Editors
- 6) Your First Program
- 7) How to Read a Go Program:
- 8)Go Program Types: Numbers, Strings, Booleans
- 9)Variables: how to name a variable, scope, constants, defining multiple variables, An Example Program

Introduction

Getting Started

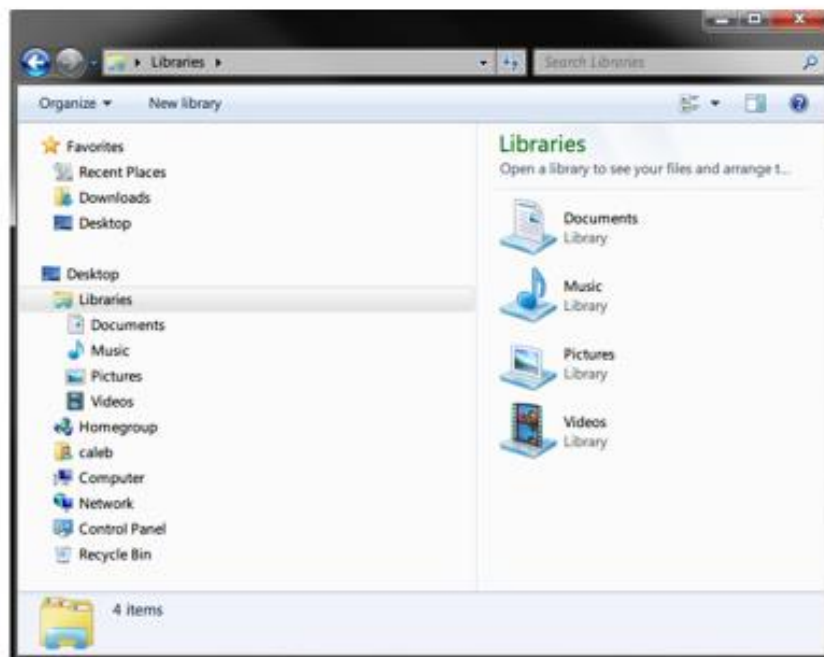
1. Go is a general-purpose programming language with advanced features and a clean syntax.
2. Because of its wide availability on a variety of platforms, its robust well-documented common library, and its focus on good software engineering

principles, Go is an ideal language to learn as your first programming language.

The process we use to write software using Go (and most programming languages) is fairly straightforward:

1. Gather requirements
 2. Find a solution
 3. Write source code to implement the solution
 4. Compile the source code into an executable
 5. Run and test the program to make sure it works
- This process is iterative (meaning its done many times) and the steps usually overlap. But before we write our first program in Go there are a few prerequisite concepts we need to understand.
 - **Files and Folders:** A file is a collection of data stored as a unit with a name. Modern operating systems (like Windows or Mac OSX) contain millions of files which store a large variety of different types of information – everything from text documents to executable programs to multimedia files.
 - All files are stored in the same way on a computer: they all have a name, a definite size (measured in bytes) and an associated type.
 - Typically, the file's type is signified by the file's extension – the part of the file name that comes after the last. For example, a file with the name hello.txt has the extension txt which is used to represent textual data.

- Folders (also called directories) are used to group files together. They can also contain other folders. On Win- 3 Getting Started downs file and folder paths (locations) are represented with the \ (backslash) character
- for example: C:\Users\john\example.txt.
- example.txt is the file name, it is contained in the folder john, which is itself contained in the folder Users which is stored on drive C (which represents the primary physical hard drive in Windows).
- On Windows files and folders can be browsed using Windows Explorer (accessible by double-clicking “My Computer” or typing win+e):



- The Terminal Most of the interactions we have with computers today are through sophisticated graphical user interfaces (GUIs). We use keyboards, mice and touchscreens to interact with visual buttons or other types of controls that are displayed on a screen. It wasn't always this way. Before the GUI we had the terminal – a simpler textual interface to the computer.

- where rather than manipulating buttons on a screen we issued commands and received replies. We had a conversation with the computer.
- And although it might appear that most of the computing world has left behind the terminal as a relic of the past, the truth is that the terminal is still the fundamental user interface used by most programming languages on most computers.
- The Go programming language is no different, and so before we write a program in Go, we need to have a rudimentary understanding of how a terminal works.
- Windows in Windows the terminal (also known as the command line) can be brought up by typing the windows key + r (hold down the windows key then press r), typing cmd.exe and hitting enter. You should see a black window appear that looks like this:



- By default, the command line starts in your home directory. (In my case this is C:\Users\caleb) You issue commands by typing them in and hitting enter. Try entering the command dir., which lists the contents of a directory. You should see something like this:

```
C:\Users\caleb>dir
Volume in drive C has no label.
Volume Serial Number is B2F5-F125
```

- **Text Editors** The primary tool programmers use to write software is a text editor. Text editors are similar to word processing programs (Microsoft Word, Open Office, ...) but unlike such programs they don't do any formatting, (No bold, italic, ...) instead they operate only on plain text. Both OSX and Windows come with text editors but they are highly limited and I recommend installing a better one. To make the installation of this software easier an installer is available at the book's website: <http://www.golang-book.com/>. This installer will install the Go tool suite, setup environmental variables and install a text editor.
- **Windows** For windows the installer will install the Scite text editor. You can open it by going to Start All Programs → → → Go Scite. You should see something like this: The text editor contains a large white text area where text can be entered. To the left of this text area, you can see the line numbers. At the bottom of the window is a status bar which displays information about the file and your current location in it (right now it says that we are on line 1, column 1, text is being inserted normally, and we are using windows-style newlines).
- You can open files by going to
 - File Open and brows → - ing to your desired file.
 - Files can be saved by going to File Save or File Save As.
 - → → As you work in a text editor it is useful to learn keyboard shortcuts.
 - The menus list the shortcuts to their right. Here are a few of the most common:
 - Ctrl + S – save the current file

- Ctrl + X – cut the currently selected text (remove it and put it in your clipboard so it can be pasted later)
- Ctrl + C – copy the currently selected text
- Ctrl + V – paste the text currently in the clipboard
- Use the arrow keys to navigate, Home to go to the beginning of the line and End to go to the end of the line
- Hold down shift while using the arrow keys (or Home and End) to select text without using the mouse
- Ctrl + F – brings up a find in file dialog that you can

Go Tools

Go is a compiled programming language, which means source code (the code you write) is translated into a language that your computer can understand. Therefore, before we can write a Go program, we need the Go compiler. The installer will setup Go for you automatically. We will be using version 1 of the language. (More information can be found at <http://www.golang.org>) Let's make sure everything is working. Open up a terminal and type the following

```
go version
```

You should see the following:

```
go version go1.0.2
```

- Your version number may be slightly different. If you get an error about the command not being recognized try restarting your computer. The Go tool suite is made up of several

different commands and sub-commands. A list of those commands is available by typing:

```
go help
```

Your First Program

- Traditionally the first program you write in any programming language is called a “Hello World” program – a program that simply outputs Hello World to your terminal. Let's write one using Go.
- First create a new folder where we can store our program.
- The installer you used in chapter 1 created a folder in your home directory named Go. Create a folder named `~/Go/src/golang-book/chapter2`.
- (Where `~` means your home directory) From the terminal you can do this by entering the following commands:

```
mkdir Go/src/golang-book  
mkdir Go/src/golang-book/chapter2
```

Using your text editor type in the following:

```
package main

import "fmt"

// this is a comment

func main() {
    fmt.Println("Hello World")
}
```

- Make sure your file is identical to what is shown here and save it as main. Go in the folder we just created. Open up a new terminal and type in the following:

```
cd Go/src/golang-book/chapter2
go run main.go
```

- You should see Hello World displayed in your terminal.
- The go run command takes the subsequent files (separated by spaces), compiles them into an executable saved in a temporary directory and then runs the program.
- If you didn't see Hello World displayed you may have made a mistake when typing in the program.
- The Go compiler will give you hints about where the mistake lies.
- Like most compilers, the Go compiler is extremely pedantic and has no tolerance for mistakes.

How to Read a Go Program Let's look at this program in more detail. Go programs are read top to bottom, left to right. (like a book) The first line says this.

```
package main
```

- This is known as a “package declaration”. Every Go program must start with a package declaration. Packages are Go's way of organizing and reusing code.
- There are two types of Go programs: executables and libraries.
- Executable applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)
- Libraries are collections of code that we package together so that we can use them in other programs.
- We will explore libraries in more detail later, for now just make sure to include this line in any program you write.
- The next line is a blank line.
- Computers represent newlines with a special character (or several characters).
- Newlines, spaces and tabs are known as whitespace (because you can't see them). Go mostly doesn't care about whitespace, we use it to make programs easier to read. (You could remove this line and the program would behave in exactly the same way).

Then we see this:

```
import "fmt"
```

- The import keyword is how we include code from other packages to use with our program.
- The fmt package (shorthand for format) implements formatting for input and output.
- Given what we just learned about packages what do you think the fmt package's files would contain at the top of them?
- The import keyword is how we include code from other packages to use with our program. The fmt package (shorthand for format) implements formatting for input and output.
- Notice that fmt above is surrounded by double quotes. The use of double quotes like this is known as a “string literal” which is a type of “expression”.
- In Go strings represent a sequence of characters (letters, numbers, symbols, ...) of a definite length.
- Strings are described in more detail in the next chapter, but for now the important thing to keep in mind is that an opening " character must eventually be followed by another " character and anything in between the two is included in the string.
- (The " character itself is not part of the string) The line that starts with // is known as a comment.
- Comments are ignored by the Go compiler and are there for your own sake (or whoever picks up the source code for your program).
- Go supports two different comments: // comments in which all the text between the // and the end of the line is part of the comment and /* */ comments

- where everything between the *s is part of the comment. (And may include multiple lines) After this you see a function declaration:

```
func main() {  
    fmt.Println("Hello World")  
}
```

- Functions are the building blocks of a Go program.
- They have inputs, outputs and a series of steps called statements which are executed in order.
- All functions start with the keyword `func` followed by the name of the function (`main` in this case), a list of zero or more “parameters” surrounded by parentheses, an optional return type and a “body” which is surrounded by curly braces.
- This function has no parameters, doesn't return anything and has only one statement.
- The name `main` is special because it's the function that gets called when you execute the program.
- The final piece of our program is this line:

```
fmt.Println("Hello World")
```

This statement is made of three components.

- First, we access another function inside of the `fmt` package called `Println` (that's the `fmt. Println` piece, `Println` means Print Line).
- Then we create a new string that contains `Hello World` and invoke (also known as call or execute) that function with the string as the first and only argument. At this point we've already seen a lot of new terminology and you may be a bit overwhelmed.

- Sometimes it's helpful to deliberately read your program out loud. One reading of the program we just wrote might go like this: Create a new executable program, which references the `fmt` library and contains one function called `main`.
- That function takes no arguments, doesn't return anything and does the following: Access the `Println` function contained inside of the `fmt` package and invoke it using one argument – the string `Hello World`. The `Println` function does the real work in this program. You can find out more about it by typing the following in your terminal:

```
godoc fmt Println
```

Among other things you should see this

```
Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.
```

- Go is a very well documented programming language but this documentation can be difficult to understand unless you are already familiar with programming languages.
- Nevertheless, the `godoc` command is extremely useful and a good place to start whenever you have a question.
- Back to the function at hand, this documentation is telling you that the `Println` function will send whatever you give to it to standard output – a name for the output of the terminal you are working in.

- This function is what causes Hello World to be displayed.
- In the next chapter we will explore how Go stores and represents things like Hello World by learning about types. Your First Program

Problems

- 1. What is whitespace?**
- 2. What is a comment? What are the two ways of writing a comment?**
- 3. Our program began with package main. What would the files in the fmt package begin with?**
- 4. We used the Println function defined in the fmt package. If we wanted to use the Exit function from the os package what would we need to do?**
- 5. Modify the program we wrote so that instead of printing Hello World it prints Hello, my name is followed by your name.**

Types

In the last chapter we used the data type string to store Hello World. Data types categorize a set of related values, describe the operations that can be done on them and define the way they are stored. Since types can be a difficult concept to grasp we will look at them from a couple different perspectives before we see how they are implemented in Go.

Philosophers sometimes make a distinction between types and tokens. For example suppose you have a dog named Max. Max is the token (a particular instance or member) and dog is the type (the general concept). “Dog” or “dogness” describes a set of properties that all dogs have in common. Although

oversimplistic we might reason like this: All dogs have 4 legs, Max is a dog, therefore Max has 4 legs. Types in programming languages work in a similar way: All strings have a length, x is a string, therefore x has a length

In mathematics we often talk about sets. For example: \mathbb{R} (the set of all real numbers) or \mathbb{N} (the set of all natural numbers). Each member of these sets shares properties with all the other members of the set. For example all natural numbers are associative: “for all natural numbers a , b , and c , $a + (b + c) = (a + b) + c$ and $a \times (b \times c) = (a \times b) \times c$.” In this way sets are similar to types in programming languages since all the values of a particular type share certain properties.

Go is a statically typed programming language. This means that variables always have a specific type and that type cannot change. Static typing may seem cumbersome at first. You'll spend a large amount of your time just trying to fix your program so that it finally compiles. But types help us reason about what our program is doing and catch a wide variety of common mistakes.

Go comes with several built-in data types which we will now look at in more detail.

Numbers

Go has several different types to represent numbers. Generally, we split numbers into two different kinds: integers and floating-point numbers.

Integers

Integers – like their mathematical counterpart – are 25 Types numbers without a decimal component. (... , -3, -2, -1, 0, 1, ...) Unlike the base-10 decimal system we use to represent numbers, computers use a base-2 binary system.

Our system is made up of 10 different digits. Once we've exhausted our available digits we represent larger numbers by using 2 (then 3, 4, 5, ...) digits put next to each other. For example the number after 9 is 10, the number after 99 is 100 and so on. Computers do the same, but they only have 2 digits instead of 10. So counting looks like this: 0, 1, 10, 11, 100, 101, 110, 111 and so on.

The other difference between the number system we use and the one computers use is that all of the integer types have a definite size. They only have room for a certain number of digits. So a 4 bit integer might look like this: 0000, 0001, 0010, 0011, 0100. Eventually we run out of space and most computers just wrap around to the beginning. (Which can result in some very strange behavior)

Go's integer types are: uint8, uint16, uint32, uint64, int8, int16, int32 and int64.

8, 16, 32 and 64 tell us how many bits each of the types use. uint means “unsigned integer” while int means “signed integer”. Unsigned integers only contain positive numbers (or zero). In addition there two alias types: byte which is the same as uint8 and rune which is the same as

int32. Bytes are an extremely common unit of measurement used on computers (1 byte = 8 bits, 1024 bytes = 1 kilobyte, 1024 kilobytes = 1 megabyte, ...) and therefore Go's byte data type is often used in the definition of other types. There are also 3 machine dependent integer types: uint, int and uintptr. They are machine dependent because their size depends on the type of architecture you are using.

Generally if you are working with integers you should just use the int type.

Floating Point Numbers

Floating point numbers are numbers that contain a decimal component (real numbers). (1.234, 123.4, 0.00001234, 12340000) Their actual representation on a computer is fairly complicated and not really necessary in order to know how to use them. So for now we need only keep the following in mind: 1. Floating point numbers are inexact. Occasionally it is not possible to represent a number. For example computing $1.01 - 0.99$ results in 0.0200000000000000018 – A number extremely close to what we would expect, but not exactly the same.

2. Like integers floating point numbers have a certain size (32 bit or 64 bit). Using a larger sized floating point number increases its precision. (how many digits it can represent)

3. In addition to numbers there are several other values which can be represented: “not a number” (NaN, for things like $0/0$) and positive and negative infinity. ($+\infty$ and $-\infty$)

Go has two floating point types: float32 and float64 (also often referred to as single precision and double precision respectively) as well as two additional types for representing complex numbers (numbers with imaginary parts): complex64 and complex128. Generally we should stick with float64 when working with floating point numbers.

Example

Let's write an example program using numbers. First create a folder called chapter3 and make a main.go file containing the following:

```
package main

import "fmt"

func main() {
    fmt.Println("1 + 1 =", 1 + 1)
}
```

If you run the program and you should see this:

```
$ go run main.go
1 + 1 = 2
```

Notice that this program is very similar to the program we wrote in chapter 2. It contains the same package line, the same import line, the same function declaration and uses the same Println function. This time instead of printing the string Hello World we print the string 1 + 1 = followed by the result of the expression 1 + 1. This expression is made up of three parts: the numeric literal 1 (which is of type int), the + operator (which represents addition) and another numeric literal 1. Let's try the same thing using floating point numbers:

```
fmt.Println("1 + 1 =", 1.0 + 1.0)
```

Notice that we use the .0 to tell Go that this is a float-ing point number instead of an integer. Running this program will give you the same result as before. In addition to addition Go has several other operators:

+	addition
-	subtraction
*	multiplication
/	division
%	remainder

Strings

As we saw in chapter 2 a string is a sequence of characters with a definite length used to represent text. Go strings are made up of individual bytes, usually one for each character. (Characters from other languages like Chinese are represented by more than one byte)

String literals can be created using double quotes "Hello World" or back ticks `Hello World`. The difference between these is that double quoted strings cannot contain newlines and they allow special escape sequences. For example `\n` gets replaced with a newline and `\t` gets replaced with a tab character.

Several common operations on strings include finding the length of a string: `len("Hello World")`, accessing an individual character in the string: `"Hello World"[1]`, and concatenating two strings together: `"Hello " + "World"`. Let's modify the program we created earlier to test these out:

```
package main

import "fmt"

func main() {
    fmt.Println(len("Hello World"))
    fmt.Println("Hello World"[1])
    fmt.Println("Hello " + "World")
}
```

1. A space is also considered a character, so the string's length is 11 not 10 and the 3rd line has "Hello " instead of "Hello".

2. Strings are “indexed” starting at 0 not 1. [1] gives you the 2nd element not the 1st. Also notice that you see 101 instead of e when you run this program. This is because the character is represented by a byte (remember a byte is an integer).

One way to think about indexing would be to show it like this instead: "Hello World"1. You'd read that as “The string Hello World sub 1,” “The string Hello World at 1” or “The second character of the string Hello World”.

Concatenation uses the same symbol as addition. The Go compiler figures out what to do based on the types of the arguments. Since both sides of the + are strings the compiler assumes you mean concatenation and not addition. (Addition is meaningless for strings)

Booleans

- A boolean value (named after George Boole) is a special 1 bit integer type used to represent true and false (or on and off). Three logical operators are used with boolean values:

&&	and
	or
!	not

- Here is an example program showing how they can be

Used:

```
func main() {
    fmt.Println(true && true)
    fmt.Println(true && false)
    fmt.Println(true || true)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

- Running this program should give you:

```
$ go run main.go
true
false
true
true
false
```

- We usually use truth tables to define how these operators work:

Expression	Value
true && true	true
true && false	false
false && true	false
false && false	false

Expression	Value
true true	true
true false	true
false true	true
false false	false

- These are the simplest types included with Go and form the foundation from which all later types are built.

Problems:

1. How are integers stored on a computer?
2. We know that (in base 10) the largest 1 digit number is 9 and the largest 2 digit number is 99. Given that in binary the largest 2 digit number is 11 (3), the largest 3 digit number is 111 (7) and the largest 4 digit number is 1111 (15) what's the largest 8 digit number? (hint: $10^1 - 1 = 9$ and $10^2 - 1 = 99$)
3. Although overpowered for the task you can use Go as a calculator. Write a program that computes 32132×42452 and prints it to the terminal. (Use the * operator for multiplication)
4. What is a string? How do you find its length?
5. What's the value of the expression `(true && false) || (false && true) || !(false && false)?`

Variables

- Up until now we have only seen programs that use literal values (numbers, strings, etc.) but such programs aren't particularly useful. To make truly useful programs we need to learn two new concepts: variables and control flow statements. This chapter will explore variables in more detail.
- A variable is a storage location, with a specific type and an associated name. Let's change the program we wrote in chapter 2 so that it uses a variable:

- Notice that the string literal from the original program still appears in this program, but rather than send it directly to the `Println` function we assign it to a variable instead.
- Variables in Go are created by first using the `var` keyword, then specifying the variable name (`x`), the type (`string`) and finally assigning a value to the variable (`Hello World`). The last step is optional so an alternative way of writing the program would be like this:

```
package main

import "fmt"

func main() {
    var x string
    x = "Hello World"
    fmt.Println(x)
}
```

- Variables in Go are similar to variables in algebra but there are some subtle differences:

First when we see the `=` symbol we have a tendency to read that as “`x` equals the string `Hello World`”. There's nothing wrong with reading our program that way, but it's better to read it as “`x` takes the string `Hello World`” or “`x` is assigned the string `Hello World`”. This distinction is important because (as their name would suggest) variables can change their value throughout the lifetime of a program. Try running the following:

```
package main

import "fmt"

func main() {
    var x string
    x = "first"
    fmt.Println(x)
    x = "second"
    fmt.Println(x)
}
```

In fact, you can even do this:

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

This program would be nonsense if you read it like an algebraic theorem. But it makes sense if you are careful to read the program as a list of commands. When we see `x = x + "second"` we should read it as “assign the concatenation of the value of the variable `x` and the string literal `second` to the variable `x`.” The right side of the `=` is done first and the result is then assigned to the left side of the `=`.

The `x = x + y` form is so common in programming that

Go has a special assignment statement: `+=`. We could have written `x = x + "second"` as `x += "second"` and it would have done the same thing. (Other operators can be used the same way)

```
var x string = "hello"
var y string = "world"
fmt.Println(x == y)
```

This program should print false because hello is not the same as world. On the other hand:

```
var x string = "hello"  
var y string = "hello"  
fmt.Println(x == y)
```

This will print true because the two strings are the same.

Since creating a new variable with a starting value is so common Go also supports a shorter statement:

```
x := "Hello World"
```

Notice the: before the = and that no type was specified. The type is not necessary because the Go compiler is able to infer the type based on the literal value you assign the variable. (Since you are assigning a string literal, x is given the type string) The compiler can also do inference with the var statement:

```
var x = "Hello World"
```

The same thing works for other types:

```
x := 5  
fmt.Println(x)
```

Generally, you should use this shorter form whenever possible.

How to Name a Variable

Naming a variable properly is an important part of software development.

Names must start with a letter and may contain letters, numbers or the _

(underscore) symbol. The Go compiler doesn't care what you name a variable so the name is meant for your (and others) benefit.

Pick names which clearly describe the variable's purpose. Suppose we had the following:

```
x := "Max"
fmt.Println("My dog's name is", x)
```

In this case x is not a very good name for a variable. A better name would be:

```
name := "Max"
fmt.Println("My dog's name is", name)
```

or even:

```
dogsName := "Max"
fmt.Println("My dog's name is", dogsName)
```

In this last case we use a special way to represent multiple words in a variable name known as lower camel case (also known as mixed case, bumpy caps, camel back or hump back). The first letter of the first word is lowercase, the first letter of the subsequent words is uppercase and all the other letters are lowercase.

Scope

Going back to the program we saw at the beginning of

the chapter:

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

Another way of writing this program would be like this:

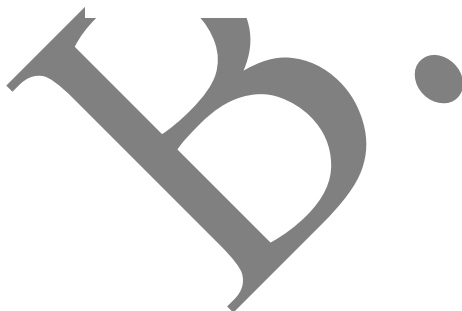
```
package main

import "fmt"

var x string = "Hello World"

func main() {
    fmt.Println(x)
}
```

Notice that we moved the variable outside of the main function. This means that other functions can access this variable:



```

var x string = "Hello World"

func main() {
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}

```

The `f` function now has access to the `x` variable. Now suppose that we wrote this instead:

```

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}

```

If you run this program you should see an error:

```

.\main.go:11: undefined: x

```

The compiler is telling you that the `x` variable inside of the `f` function doesn't exist.

It only exists inside of the `main` function. The range of places where you are allowed to use `x` is called the scope of the variable.

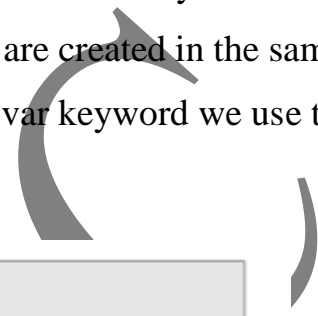
According to the language specification “Go is lexically scoped using blocks”.

Basically, this means that the variable exists within the nearest curly braces `{ }` (a block) including any nested curly braces (blocks), but not outside of them.

Scope can be a little confusing at first; as we see more Go examples it should become clearer.

Constants

- Go also has support for constants. Constants are basically variables whose values cannot be changed later. They are created in the same way you create variables but instead of using the var keyword we use the const keyword:




```
package main

import "fmt"

func main() {
    const x string = "Hello World"
    fmt.Println(x)
}
```

This:



```
const x string = "Hello World"
x = "Some other string"
```

Results in a compile-time error:

```
.\main.go:7: cannot assign to x
```

- Constants are a good way to reuse common values in a program without writing them out each time.
- For example, Pi in the math package is defined as a constant.

Defining Multiple Variables

- Go also has another shorthand when you need to define multiple variables:

```
var (  
    a = 5  
    b = 10  
    c = 15  
)
```

- Use the keyword var (or const) followed by parentheses with each variable on its own line.

An Example Program

- Here's an example program which takes in a number entered by the user and doubles it:

```

package main

import "fmt"

func main() {
    fmt.Print("Enter a number: ")
    var input float64
    fmt.Scanf("%f", &input)

    output := input * 2

    fmt.Println(output)
}

```

- We use another function from the fmt package to read the user input (Scanf). &input will be explained in a later chapter, for now all we need to know is that Scanf fills input with the number we enter.

Problems

1. What are two ways to create a new variable?
2. What is the value of x after running: $x := 5$; $x += 1$?
3. What is scope and how do you determine the scope of a variable in Go?
4. What is the difference between var and const?
5. Using the example program as a starting point, write a program that converts from Fahrenheit into Celsius. ($C = (F - 32) * 5/9$)
6. Write another program that converts from feet into meters. ($1 \text{ ft} = 0.3048 \text{ m}$)