**Golang Notes**

**Introduction**

* Go language is a programming language initially developed at Google in the year 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.
* It is general purpose and used in apis, cloud services, cli tools, etc.
* It is a statically-typed language having syntax similar to that of C.
* It is strongly typed (variable types cannot change after its been declared) and statically typed, provides inbuilt support for garbage collection, and supports concurrent programming.
* Programs are constructed using packages, for efficient management of dependencies.
* It also provides a rich standard library.
* The most important features of Go programming are listed below −
  + Support for environment adopting patterns similar to dynamic languages. For example, type inference (x := 0 is valid declaration of a variable x of type int)
  + Compilation time is fast and runtime is fast since it’s compiled
  + Inbuilt concurrency support: lightweight processes (via go routines), channels, select statement.
  + Go programs are simple, concise, and safe.
  + Support for Interfaces and Type embedding.
  + Production of statically linked native binaries without external dependencies.
* To keep the language simple and concise, the following features commonly available in other similar languages are omitted in Go −
  + Support for type inheritance
  + Support for method or operator overloading
  + Support for circular dependencies among packages
  + Support for pointer arithmetic
  + Support for assertions
  + Support for generic programming

**Hello world**

* A screenshot of a computer code

  Description automatically generated
* All go files will be part of a package which is a collection of files of code. The package can be called “main” or something else. If the package is “main”, this tells the go compiler that our code should be compiled into an executable program at the end. All the go files we create that a part of the program will specify this main package at the start of the file. If you are making some kind of shared library or utility code that could be used in other programs, then you will call your package something else depending on what you're making. Since we'll be making a program to run on our computer, we'll use package main.
* The next line import "fmt" is a preprocessor command which tells the Go compiler to include the files lying in the package fmt.
* Notice that our function declaration starts off with the “func” keyword. All functions we create will start with this keyword. The function name in our case is main. This main function is the entry point of our application. So when we run our program go will look through our files in the program and it will look for this main function and it will fire this first one automatically so this main function is the entry point of the application. There must be one and only one main function in the application. We don't have one in every file that's inside our package, we just have one main function in the entry file. In other files we can create other functions which can be invoked from inside this main function.
* The next line /\*...\*/ is ignored by the compiler and it is there to add comments in the program. Comments are also represented using // similar to Java or C++ comments.
* The next line fmt.Println(...) is another function available in Go which causes the message "Hello, World!" to be displayed on the screen. Here fmt package has exported Println method which is used to display the message on the screen. Notice the capital P of Println method. In Go language, a name is exported if it starts with capital letter. Exported means the function or variable/constant is accessible to the importer of the respective package.
* To run the file, run the below:
* A close up of words

  Description automatically generated

**File Naming**

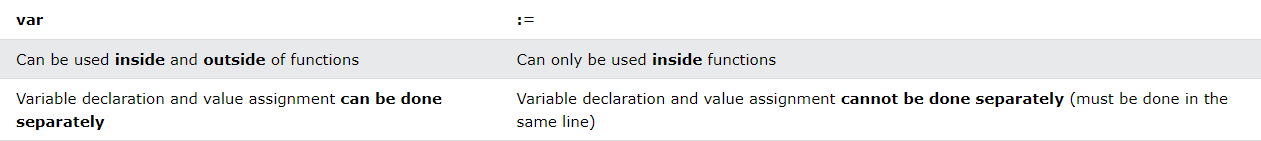
* Generally, file names are single lowercase words.
* Go follows a convention where source files are all lower case with underscore separating multiple words.
* Compound file names are separated with \_
* File names that begin with “.” or “\_” are ignored by the go tool
* Test files in Go come with suffix \_test.go . These are only compiled and run by the go test tool.
* Files with os and architecture specific suffixes automatically follow those same constraints, e.g. name\_linux.go will only build on linux, name\_amd64.go will only build on amd64.

**Comments**

* Single-line comments start with two forward slashes (//).
* Multi-line comments start with /\* and ends with \*/.

**Variables**

* A computer screen with text and numbers

  Description automatically generated
* Output: 
* In the above code, we initialize the 4 variables, str1, str2, str3, str4.
* Str1 is an explicit way of declaring a variable. This approach is more explicit and might be preferred when the variable needs a specific initial value or when declaring package-level variables.
* Str2 implicitly determines the type of the variable based on the value its assigned (we can think of this as a shorthand for they way we initialized str1).
* Str3 is an even more implicit way of declaring a string(we can think of this as a shorthand for they way we initialized str2). We can only use := the first time we are declaring the variable, so you can’t update the a variable with :=. This method is concise and is frequently used for declaring and initializing variables within a limited scope, like function bodies or short code segments.
* Str4 isn’t given an explicit value and we’re setting it up for future use. It defaults to a value of empty string “”.
* String values must be surrounded by double quotes
* There are some small differences between the “var” and “:=”:
* 
* You can declare variables outside a function as shown with str0, but you cannot use the := assignment to do so.
  + Ex:  
    A screen shot of a computer code

    Description automatically generated
  + Suppose I have the above code. Notice how I say b, err := foo() even though err was already defined. As long as we have one new value on the left side of the :=, we can use as many old variables as we want and reassign them with the := statement.
* Naming Rules
  + A variable name must start with a letter or an underscore character (\_)
  + A variable name cannot start with a digit
  + A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and \_ )
  + Variable names are case-sensitive (age, Age and AGE are three different variables)
  + There is no limit on the length of the variable name
  + A variable name cannot contain spaces
  + The variable name cannot be any Go keywords
  + Use camelCase and PascalCase depending on its scope:
    - Lowercase initial letter: Variables with lowercase initial letters are considered private to the package they are defined in.
    - Uppercase initial letter: Variables with uppercase initial letters are exported and can be accessed by code outside the package they are defined in.
* Declaring Multiple Variables
  + In Go, it is possible to declare multiple variables in the same line.
  + A screenshot of a computer program

    Description automatically generated
  + Output: 
  + Note: If you use the type keyword as shown above, it is only possible to declare one type of variable per line.
  + If the type keyword is not specified, you can declare different types of variables in the same line:
  + A screenshot of a computer code

    Description automatically generated
  + Output: A black background with white text

    Description automatically generated
  + Multiple variable declarations can also be grouped together into a block for greater readability:
  + A computer code with text

    Description automatically generated with medium confidence
  + Output: 

**Scope Rules**

* Variables can be block scope or package scope. (Also module/universe scope but we’ll mention that later)
* Block Scope
  + Block scope is anything inside curly braces {}
  + A computer screen shot of text

    Description automatically generated
* Package Scope
  + Suppose I have the following main.go and hello.go files:
  + A screen shot of a computer program

    Description automatically generatedA screenshot of a computer

    Description automatically generated
  + Output: A black screen with white text

    Description automatically generated
  + Note in the above example, the variables x and y are declared with package scope.
  + Note that it is impossible to create a variable that is only accessible from the file it is declared it and no other file. As long as there are other files from within the same package scope, they can access all variables that are package scope.
  + Note that it is convention to put all files that are under the same package inside a directory with that package name:
  + A screenshot of a computer

    Description automatically generated

**Constants**

* If a variable should have a fixed value that cannot be changed, you can use the const keyword.
* The const keyword declares the variable as "constant", which means that it is unchangeable and read-only.
* Note: The value of a constant must be assigned when you declare it.
* A white background with black text

  Description automatically generated
* **Constant Rules**
  + Constant names follow the same naming rules as variables
  + Constant names are usually written in uppercase letters (for easy identification and differentiation from variables)
  + Constants can be declared both inside and outside of a function
* There are two types of constants:
  + Typed constants
    - Typed constants are declared with a defined type:
    - 
  + Untyped constants
    - Untyped constants are declared without a type:
    - 
    - Note: In this case, the type of the constant is inferred from the value (means the compiler decides the type of the constant, based on the value).
* Multiple constants can be grouped together into a block for readability:
* A white background with black and red text

  Description automatically generated

**Strings**

* Strings, which are widely used in Go programming, are a readonly slice of bytes.
* In the Go programming language, strings are slices. The Go platform provides various libraries to manipulate strings.
* String Concatenation
  + Strings can be concatenated using the + operator or the Join() function from the strings package.
  + A screen shot of a computer program

    Description automatically generated
  + Output: 
* Length
  + A computer screen with text and numbers

    Description automatically generated
  + Output: 
* Substring
  + A computer screen shot of text

    Description automatically generated
  + A screen shot of a computer code

    Description automatically generated
* Splitting and Joining
  + A screen shot of a computer code

    Description automatically generated
  + Output: 
* Equality
  + A screen shot of a computer screen

    Description automatically generated
* String Iteration
  + Recall: ASCII (American Standard Code for Information Interchange) is a subset of UTF-8 (Unicode Transformation Format-8). ASCII was one of the earliest character encoding standards, using 7 bits to represent characters (128 possible characters, including control characters and symbols).
  + UTF-8, on the other hand, is a variable-width character encoding capable of representing every character in the Unicode character set. It's backward compatible with ASCII because the first 128 characters in Unicode are directly mapped to the ASCII character set.
  + When you iterate over a string using a for range loop, the loop variable represents Unicode code points (runes), not characters directly. If you specifically want to convert a Unicode code point (rune) to its corresponding character as a string, you can simply convert it by using string():
  + A computer screen shot of a code

    Description automatically generated
  + Output: A screenshot of a computer

    Description automatically generated

**Numeric Types:**

* int: Signed integers (can be int8, int16, int32, or int64 based on size).
* uint: Unsigned integers (can be uint8, uint16, uint32, or uint64 based on size).
* float32, float64: Floating-point numbers.
* complex64, complex128: Complex numbers with float32 and float64 real and imaginary parts.
* A computer screen shot of a number and numbers

  Description automatically generated
* Output: 
* Notice that int4 has a default value of 0. Int8 means that it is a signed integer (can be negative or positive) and takes up 8 bits. Thus int8 values range from -128 to 127 inclusive. Uint16 means is the set of all unsigned 16-bit integers, range: 0 through 65535.
* For more information about the types: <https://go.dev/src/builtin/builtin.go>

**Boolean Type:**

* A computer screen shot of a code

  Description automatically generated

**Printing and Formatting Strings**

* Go has three functions to output text:
  + Print()
  + Println()
  + Printf()
  + Sprintf()
* Print()
  + The Print() function prints its arguments with their default format.
  + A computer code with black text

    Description automatically generated
  + Output: 
  + If we want to print the arguments in new lines, we need to use \n.
  + Output: A black background with white text

    Description automatically generated
  + It is also possible to only use one Print() for printing multiple variables.
  + 
  + It is also possible to only use one Print() for printing multiple variables.
  + A computer code with black text

    Description automatically generated
  + Output: 
  + Print() inserts a space between the arguments if neither are strings:
  + A white background with black text

    Description automatically generated
  + Output: 
  + If we want to add a space between string arguments, we need to use " ":
  + A computer code with black text

    Description automatically generated
  + Output: 
* Println()
  + The Println() function is similar to Print() with the difference that a whitespace is added between the arguments, and a newline is added at the end:
  + A computer screen shot of a program code

    Description automatically generated
  + Output: A black screen with white text

    Description automatically generated
* Printf()
  + The Printf() function first formats its argument based on the given formatting verb and then prints them.
  + Here we will use two formatting verbs:
  + %v is used to print the value of the arguments
  + %q is used to put quotes around arguments (should only be used on strings)
  + %T is used to print the type of the arguments
  + %f is used to print floats
  + A screen shot of a computer

    Description automatically generated
  + Output: A screen shot of a computer

    Description automatically generated
  + Notice that %0.2f means that we print 225.555 up to 2 decimal places
* Sprintf()
  + Saves a formatted string
  + A computer screen shot of a code

    Description automatically generated
  + Output: 

**Arrays**

* In Go, arrays refer to static arrays
* Declaring Arrays
  + In Go, there are two ways to declare an array:
  + 1. With the var keyword:
  + A white background with black text

    Description automatically generated
  + 2. With the := sign:
  + A white background with black text

    Description automatically generated
* Access Elements of an Array
  + A computer code with black text

    Description automatically generated with medium confidence
  + Output: 
* Change Elements of an Array
  + You can also change the value of a specific array element by referring to the index number.
  + A white background with black text

    Description automatically generated
  + Output: 
* Array Initialization
  + If an array or one of its elements has not been initialized in the code, it is assigned the default value of its type.
  + Recall: The default value for int is 0, and the default value for string is "".
  + A screenshot of a computer code

    Description automatically generated
  + Output: A number on a black background

    Description automatically generated
* Initialize Only Specific Elements
  + A computer code with black text

    Description automatically generated
  + Output: 
  + The array above has 5 elements.
  + 1:10 means: assign 10 to array index 1 (second element).
  + 2:40 means: assign 40 to array index 2 (third element).
* Find the Length of an Array
  + A computer code with red and blue text

    Description automatically generated
  + 

**Slices**

* Slices are dynamic arrays
* In Go, there are several ways to create a slice:
  + Using the []datatype{values} format
  + Create a slice from an array
  + Using the make() function
* Create a Slice With []datatype{values}
  + 
  + The code above declares an empty slice of 0 length and 0 capacity.
  + To initialize the slice during declaration, use this:
  + 
  + The code above declares a slice of integers of length 3 and also the capacity of 3.
  + In Go, there are two functions that can be used to return the length and capacity of a slice:
    - Length: It refers to the number of elements in the slice. It's what you get when you use the len() function on a slice. The length of a slice can change when you append or remove elements from it.
    - Capacity: It represents the maximum number of elements that the slice can hold without needing to allocate more memory. You can get the capacity of a slice using the cap() function. The capacity of a slice is determined by the size of the underlying array. As you add elements to a slice, if the capacity is exceeded, a new larger array is allocated, and the elements are copied over to it.
  + A computer screen shot of a code

    Description automatically generated
  + Output: A black screen with white text

    Description automatically generated
  + When you create a slice directly using []datatype{values}, the time complexity is O(n), where n is the number of elements in the slice. This is because the slice needs to allocate memory for each element and copy the provided values into that allocated memory. The time taken will be proportional to the number of elements being copied.
* Create a Slice From an Array
  + A screenshot of a computer code

    Description automatically generated
  + Output: A black background with white text

    Description automatically generated
  + In the example above myslice is a slice with length 2. It is made from arr1 which is an array with length 6.
  + The slice starts from the third element of the array which has value 12. The slice can grow to the end of the array. This means that the capacity of the slice is 4.
  + If myslice started from the element at index 0, the slice capacity would be 6.
  + Creating a slice from an array doesn't involve copying elements; it merely creates a "view" or reference to the underlying array. This operation has a time complexity of O(1) because it doesn't involve any element copying. It just creates a new slice header that points to the existing array.
  + In Go, a slice header is a data structure that describes a slice. It contains metadata about the slice such as its length, capacity, and a reference to the underlying array. The header itself is a small structure composed of three fields:
    - Pointer to the underlying array: This points to the starting address of the underlying array from which the slice is formed.
    - Length: Indicates the number of elements in the slice.
    - Capacity: Represents the maximum number of elements that the slice can contain without requiring a new allocation. Capacity is determined by the number of elements between the start of the slice and the end of the underlying array.
  + When you create a slice in Go, like slice := array[2:4], it's important to understand that the slice variable holds this slice header. This header doesn’t contain the actual elements of the slice; rather, it's a reference to a subset of elements within the original array.
* Create a Slice With The make() Function
  + The make() function can also be used to create a slice.
  + A computer screen shot of a program

    Description automatically generated
  + Output: A black screen with white text

    Description automatically generated
  + Note: If the capacity parameter is not defined, it will be equal to length.
  + The make() function is used to create a slice with a specified length and capacity. The time complexity for this operation is O(1) for the allocation of the slice's internal structure. The time complexity is constant because make() only initializes the slice header and allocates a backing array (if required based on the provided capacity) without actually initializing or populating the elements.
* Access Elements of a Slice
  + A computer code with black text

    Description automatically generated
  + Output: 
* Change Elements of a Slice
  + A screenshot of a computer code

    Description automatically generated
  + Output: 
* Append Elements To a Slice
  + You can append elements to the end of a slice using the append()function:
  + 
  + A screenshot of a computer code

    Description automatically generated
  + Output: A black background with white numbers and symbols

    Description automatically generated
  + The append function in Go is used to add elements to a slice. Under the hood, append manages the growth of the slice by creating a new underlying array when necessary. Here's a simplified explanation of what happens:
    - Initial Check: When appending to a slice, Go checks if the underlying array has enough capacity to accommodate the new elements. If it does, the elements are added directly to the slice, and its length is updated.
    - Capacity Check and Allocation: If the underlying array doesn't have enough capacity to accommodate the new elements, a new array is allocated. The size of this new array might be larger than the original one to allow for future appends without needing frequent reallocation.
    - Copying Elements: The elements from the existing slice are copied over to the new array, along with the new elements being added. This is done to maintain the order of elements and ensure that the slice remains contiguous in memory.
    - Returning a New Slice: Finally, append returns a new slice that points to the newly allocated array. This slice might reference the same underlying array as the original slice if sufficient capacity was available, or it might reference a new, larger array if reallocation was necessary.
    - It's important to note that since Go uses a garbage collector, the old underlying array, if it's not referenced elsewhere, will be eventually cleaned up by the garbage collector.
  + Most of the time, the return value is the same array that was passed into the append function since the array doesn’t change (so the return value is useless), but we need the return value to account for the times when we do need a new array.
* Append One Slice To Another Slice
  + To append all the elements of one slice to another slice, use the append()function:
  + 
  + Note: The '...' after slice2 is necessary when appending the elements of one slice to another.
  + A screenshot of a computer code

    Description automatically generated
  + A black background with white numbers and symbols

    Description automatically generated
* Change The Length of a Slice
  + A screenshot of a computer program

    Description automatically generated
  + Output: A screen shot of a computer

    Description automatically generated
* Copy()
  + If the array is large and you need only a few elements, it is better to copy those elements using the copy() function.
  + The copy() function creates a new underlying array with only the required elements for the slice. This will reduce the memory used for the program.
  + 
  + The copy() function takes in two slices dest and src, and copies data from src to dest. It returns the number of elements copied.
  + A computer screen shot of code

    Description automatically generated
  + Output: A screenshot of a computer

    Description automatically generated

**Maps**

* Maps are used to store data values in key:value pairs.
* Each element in a map is a key:value pair.
* A map is an unordered and changeable collection that does not allow duplicates.
* The length of a map is the number of its elements. You can find it using the len() function.
* The default value of a map is nil.
* Maps hold references to an underlying hash table.
* Go has multiple ways for creating maps.
* Create Maps Using var and :=
  + 
  + A text on a white background

    Description automatically generated
  + Output: 
  + Note: The order of the map elements defined in the code is different from the way that they are stored. The data are stored in a way to have efficient data retrieval from the map.
* Create Maps Using make()Function:
  + 
  + A screenshot of a computer code

    Description automatically generated
  + 
* Create an Empty Map
  + There are two ways to create an empty map. One is by using the make()function and the other is by using the following syntax.
  + 
  + Note: The make()function is the right way to create an empty map. If you make an empty map in a different way and write to it, it will cause a runtime panic.
  + This example shows the difference between declaring an empty map using with the make()function and without it.
  + A computer code with black text

    Description automatically generated
  + Output: 
  + In Go, when you use make with map, it initializes the map, assigning an empty map value to the variable. On the other hand, declaring a map without initializing it assigns a nil value to the variable.
  + In the code, a is initialized using make, so it's not nil. However, b is declared but not initialized, so its value is nil.
* Allowed Key Types
  + The map key can be of any data type for which the equality operator (==) is defined. These include:
    - Booleans
    - Numbers
    - Strings
    - Arrays
    - Pointers
    - Structs
    - Interfaces (as long as the dynamic type supports equality)
  + Invalid key types are:
    - Slices
    - Maps
    - Functions
    - These types are invalid because the equality operator (==) is not defined for them.
* Allowed Value Types
  + The map values can be any type.
* Access Map Elements
  + You can access map elements by:
  + A computer screen shot of a program

    Description automatically generated
  + Output: 
  + In Go, accessing elements in a map has an average time complexity of O(1).
* Update and Add Map Elements
  + A screenshot of a computer program

    Description automatically generated
* Remove Element from Map
  + Removing elements is done using the delete() function.
  + A screenshot of a computer program

    Description automatically generated
* Check For Specific Elements in a Map
  + You can check if a certain key exists in a map using:
  + 
  + If you only want to check the existence of a certain key, you can use the blank identifier (\_) in place of val.
  + In Go, when you check if a key exists in a map using the two-value assignment, the second value (boolean) indicates whether the key exists. If the key doesn't exist, the second value will be false, and the first value will be the zero value of the map's value type.
  + A screenshot of a computer code

    Description automatically generated
  + Output:
  + A black background with white text

    Description automatically generated
  + In this example, we checked for existence of different keys in the map.
  + The key "color" does not exist in the map. So the value is an empty string ('').
  + The ok2 variable is used to find out if the key exist or not. Because we would have got the same value if the value of the "color" key was empty. This is the case for val3.
  + Checking for the existence of an element in a map in Go has an average time complexity of O(1).
* Iterate Over Maps
  + You can use range to iterate over maps.
  + A computer code with red and black text

    Description automatically generated
  + Output: 
  + Notice the order is random

**Bytes**

* the byte type represents an 8-bit unsigned integer. It's an alias for uint8 type and is commonly used to handle byte-sized values or when dealing with raw binary data.
* A screenshot of a computer program

  Description automatically generated

**Rune**

* the rune type represents a Unicode code point. It is an alias for the int32 type and is used to store single Unicode characters. Since Go uses UTF-8 encoding for strings, which can have variable byte lengths for different characters (especially non-ASCII ones), the rune type provides a way to work with individual Unicode characters irrespective of their byte representation.
* The rune type is often used when dealing with characters in strings that may go beyond the ASCII range. It allows you to iterate over a string and access each Unicode code point individually.
* A screen shot of a computer

  Description automatically generated
* Output: A screen shot of a computer screen

  Description automatically generated
* String to rune
* A computer screen with text

  Description automatically generated

**Operators**

* A screenshot of a computer

  Description automatically generated
* A screenshot of a math table

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* A screenshot of a phone

  Description automatically generated
* A screenshot of a computer program

  Description automatically generated

**Conditions**

* A computer code with text

  Description automatically generated
* Note that we can’t do the following:
  + A screen shot of a computer error

    Description automatically generated
  + We need curly braces around the if block
  + A computer code with text

    Description automatically generated
* In Go, the if statement allows you to do more than just check a condition. You can also perform an action before the condition check. Here's a step-by-step breakdown of the snippet:
* A screen shot of a computer error

  Description automatically generated
* Initialization within the if statement: Go allows you to perform an assignment or initialization right before the condition check in an if statement. In this case, err := thisCouldFail() is doing two things:
  + It calls the function thisCouldFail().
  + It assigns the result of this function call to the variable err.
* Condition check: After the semicolon (;), the condition err != nil is evaluated. err != nil checks if the value of err is not nil. In Go, nil represents the absence of a value for pointers, interfaces, channels, maps, and functions.
* Execution inside the if block: If the condition err != nil is true (meaning there is an error), the code inside the curly braces {} following the if statement will execute. In this case, log.Fatal(err) logs the error message using Go's log package and terminates the program.
* So, the semicolon after the initialization statement (err := thisCouldFail()) separates the initialization from the condition check within the if statement in Go. It's a concise way to handle potential errors immediately after calling a function.
* The scope of the err variable declared within the if statement is limited to the block of code encompassed by that if statement.

**Switch**

* Use the switch statement to select one of many code blocks to be executed.
* The switch statement in Go is similar to the ones in C, C++, Java, JavaScript, and PHP. The difference is that it only runs the matched case so it does not need a break statement.
* A white background with text

  Description automatically generated
* This is how it works:
  + The expression is evaluated once
  + The value of the switch expression is compared with the values of each case
  + If there is a match, the associated block of code is executed
  + The default keyword is optional. It specifies some code to run if there is no case match
* A screenshot of a computer program

  Description automatically generated
* All the case values should have the same type as the switch expression. Otherwise, the compiler will raise an error:
* A screenshot of a computer program

  Description automatically generated

**Multi-case switch Statement**

* It is possible to have multiple values for each case in the switch statement:
* A white background with green text

  Description automatically generated
* A screenshot of a computer program

  Description automatically generated
* Output: 

**Loops**

* The for loop loops through a block of code a specified number of times.
* The for loop is the only loop available in Go.
* The for loop can take up to three statements:
* A close-up of a text

  Description automatically generated
* statement1 Initializes the loop counter value.
* statement2 Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends.
* statement3 Increases the loop counter value.
* Note: These statements don't need to be present as loops arguments. However, they need to be present in the code in some form.
* Ex:
  + A computer code with red and black text

    Description automatically generated
  + Output: 
* The continue statement is used to skip one or more iterations in the loop. It then continues with the next iteration in the loop.
* A computer code with numbers and symbols

  Description automatically generated
* The break statement is used to break/terminate the loop execution.
* A computer code with numbers and symbols

  Description automatically generated
* Infinite Loops
  + A black background with white text

    Description automatically generated
* Conditional Looping
  + A black background with white text

    Description automatically generated
  + Ex:
  + A computer screen with text

    Description automatically generated
* Range
* The range keyword is used to more easily iterate over an array, slice or map. It returns both the index and the value.
* The range keyword is used like this:
* A white background with black text

  Description automatically generated
* This example uses range to iterate over an array and print both the indexes and the values at each (idx stores the index, val stores the value):
* A computer code with red and black text

  Description automatically generated
* Tip: To only show the value or the index, you can omit the other output using an underscore (\_).
* A computer code with red and black text

  Description automatically generated
* Note that when we use a range loop, the loop variables (index and value) are copies of the original elements in the collection. If you attempt to modify these loop variables directly, it won't change the original collection.
* A screenshot of a computer program

  Description automatically generated
* If you want to modify elements in the original collection, you would need to access them directly using their indices.
* A screen shot of a computer

  Description automatically generated
* This applies to slices, arrays, and similar data structures. If you're iterating over maps, modifications can be made directly to the map elements within the loop since you're accessing the original references.

**Functions**

* Naming Rules for Go Functions
  + A function name must start with a letter
  + A function name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
  + Function names are case-sensitive
  + A function name cannot contain spaces
* A screen shot of a computer code

  Description automatically generated
* If we don’t return anything, we can do something like the following:
* A black text on a white background

  Description automatically generated
* Named Return Values
  + In Go, you can name the return values of a function.
  + A screenshot of a computer program

    Description automatically generated
  + Output: 
  + Here, we name the return value as result (of type int), and return the value with a naked return (means that we use the return statement without specifying the variable name)
  + The example above can also be written like this. Here, the return statement specifies the variable name:
  + A screenshot of a computer program

    Description automatically generated
* Multiple Return Values
  + Go functions can also return multiple values.
  + A screen shot of a computer code

    Description automatically generated
  + Output: 
  + In the above, myFunction() returns one integer (result) and one string (txt1)
  + We can also store the two return values into two variables (a and b):
  + A white screen with black text

    Description automatically generated
  + Output: 
  + If we (for some reason) do not want to use some of the returned values, we can add an underscore (\_), to omit this value.
  + A screenshot of a computer code

    Description automatically generated
  + Output: 
  + Here, we want to omit the second returned value
* Callback functions
  + Callback functions in GoLang are functions that are passed as an argument to other functions.
  + A computer screen shot of text

    Description automatically generated
  + Output: 
  + If a callback function needs to return a value, you can specify the return type in the callback function's signature.
  + A computer screen shot of text

    Description automatically generated
  + A black screen with white text

    Description automatically generated
* Pass by Value
  + Go makes a copy of the values when passed into functions
  + A blue background with white text

    Description automatically generated
  + Group A types are copied when passed into functions, so modifying them in the function doesn’t actually change the value. Modifying Group B types in the function does change the value
    - Strings, Ints, Floats, Booleans: These are all basic types in Go, and they are immutable. When you pass them around, they are passed by value, meaning a copy of the value is made. Modifying a variable with these types creates a new instance in memory.Structs: Structs are collections of fields. When passed to a function, the entire struct is copied (passed by value). However, modifying fields within the struct won't affect the original struct if it's passed as an argument.
    - Arrays: Arrays have a fixed size and are passed by value. If you pass an array to a function, it creates a copy of the entire array. Any modifications made within the function scope won't affect the original array.
    - Slices: Slices are like dynamic arrays in Go. They contain a pointer to an array, a length, and a capacity. Slices are mutable, and when passed to a function, the pointer to the underlying array is copied, not the entire data. Modifying elements in a slice within a function will affect the original slice as it refers to the same underlying array.
    - Maps: Maps are reference types, passed by reference. When you pass a map to a function, you're passing a reference to the original map. Changes made to the map within the function will reflect in the original map.
  + Ex:
    - A screen shot of a computer program

      Description automatically generated
    - Output: A black screen with white text

      Description automatically generated
  + Ex:
    - A screen shot of a computer program

      Description automatically generated
    - Output: A black screen with white text

      Description automatically generated
* Variadic Functions
  + Variadic functions can be called with any number of trailing arguments. For example, fmt.Println is a common variadic function.
  + Ex:
    - A screenshot of a computer code

      Description automatically generated
    - Output: A white background with black text

      Description automatically generated
    - Explanation: Here’s a function that will take an arbitrary number of ints as arguments. Within the function, the type of nums is equivalent to []int. We can call len(nums), iterate over it with range, etc.
  + If you already have multiple args in a slice, apply them to a variadic function using func(slice...) like this.
* Anonymous Functions
  + In Go, anonymous functions are functions defined without a name. They're often used for short, immediate tasks or as arguments to other functions. Here's an example:
  + A computer screen shot of a black screen

    Description automatically generated
  + In Go, you can define an anonymous function and use it as a callback function. Here's an example of using an anonymous function as a callback:
  + A computer screen shot of a black screen

    Description automatically generated
* Closures
  + Closures in GoLang are powerful constructs that allow functions to capture and use variables defined outside of their body. When you define a function within another function, the inner function can access the variables of the enclosing function. This is particularly useful when you want to create functions that carry some state or behavior with them.
  + Ex:
    - A screen shot of a computer code

      Description automatically generated
    - Output: A white background with black text

      Description automatically generated
    - This function intSeq returns another function, which we define anonymously in the body of intSeq. The returned function closes over the variable i to form a closure.
    - We call intSeq, assigning the result (a function) to nextInt. This function value captures its own i value, which will be updated each time we call nextInt.
    - See the effect of the closure by calling nextInt a few times.
    - To confirm that the state is unique to that particular function, create and test a new one.

**Pointers**

* Pointers in Go are easy and fun to learn. Some Go programming tasks are performed more easily with pointers, and other tasks, such as call by reference, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect Go programmer.
* As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined –
* A white background with black text

  Description automatically generated
* Output: 
* What Are Pointers?
  + A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is –
  + 
  + Here, var-type is the pointer's base type; it must be a valid data type and var-name is the name of the pointer variable. The asterisk \* you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:
  + A close-up of a sign

    Description automatically generated
  + The actual data type of the value of all pointers, whether integer, float, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.
  + There are a few important operations, which we frequently perform with pointers: (a) we define pointer variables, (b) assign the address of a variable to a pointer, and (c) access the value at the address stored in the pointer variable.
* How to Use Pointers?
  + All these operations are carried out using the unary operator \* that returns the value of the variable located at the address specified by its operand. The following example demonstrates how to perform these operations
  + A screenshot of a computer program

    Description automatically generated
  + When the above code is compiled and executed, it produces the following result
  + 
* Nil Pointers in Go
  + Go compiler assign a Nil value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a nil pointer.
  + The nil pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –
  + A white background with black text

    Description automatically generated
  + 
  + On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the nil (zero) value, it is assumed to point to nothing.
  + To check for a nil pointer you can use an if statement as follows:
  + A close-up of words

    Description automatically generated

**Struct**

* A struct (short for structure) is used to create a collection of members of different data types, into a single variable.
* While arrays are used to store multiple values of the same data type into a single variable, structs are used to store multiple values of different data types into a single variable.
* A struct can be useful for grouping data together to create records.
* Declare a Struct
  + To declare a structure in Go, use the type and struct keywords:
  + A white background with black text

    Description automatically generated
  + Tip: Notice that the struct members above have different data types. name and job is of type string, while age and salary is of type int.
* Access Struct Members
  + To access any member of a structure, use the dot operator (.) between the structure variable name and the structure member:
  + A screen shot of a computer program

    Description automatically generated
  + Output: A screenshot of a computer

    Description automatically generated
* Pass Struct as Function Arguments
  + A screen shot of a computer program

    Description automatically generated
  + Output: A screenshot of a computer

    Description automatically generated
* Receiver Functions
* Receiver functions in Go, also known as methods, allow you to associate a function with a particular type. They enable you to define behavior specific to a type by attaching functions to it.
* Receiver functions allow you to work with the data encapsulated in the type and perform operations or computations specific to that type.
* A screen shot of a computer program

  Description automatically generated
* Output: 
* Notice that the Area function is only available to Rectangle structs.
* Receiver functions in Go can be defined with either values or pointers as receivers. Using pointers as receivers can be beneficial when you want to modify the original value of the receiver within the method.
* When you define a method with a pointer receiver for a type, you can call that method using either a value of that type or a pointer to that type.
* A screen shot of a computer program

  Description automatically generated
* Output: 
* In this example, rect is an instance of the Rectangle type, not a pointer to Rectangle. However, you can still call the Scale() method, which has a pointer receiver, on rect. Go allows this and automatically converts the value receiver to a pointer receiver when necessary.
* Under the hood, Go performs an implicit address-of operation (&rect) to call the method with a pointer receiver. This behavior simplifies the syntax and allows you to call methods uniformly whether you have a value or a pointer to a type with a pointer receiver method.
* Below is an example with a pointer to a rectangle
* A screenshot of a computer program

  Description automatically generated

**Type Casting**

* In Go, type casting is known as type conversion. It allows you to convert variables from one type to another, provided they are compatible. Here's a brief overview of how type conversion works in Go:
* 
* A screenshot of a computer program

  Description automatically generated
* Type conversion doesn't change the underlying value; it creates a new value of the desired type.
* Go requires explicit type conversion; it won't automatically convert types even if they seem related.
* Conversion is only possible between compatible types, like int to float, string to []byte, etc.
* If the conversion is not possible, it will result in a compilation error.
* In Go, you can convert a string to an integer and vice versa using functions from the strconv package.
* String to Integer (Parsing):
  + To convert a string to an integer, you can use the Atoi() function from the strconv package:
  + A computer screen shot of a program code

    Description automatically generated
* Integer to String (Formatting):
  + To convert an integer to a string, you can use the Itoa() function
  + A computer screen with text and images

    Description automatically generated

**Interface**

* Interfaces in GoLang are types that declare a set of methods. An interface defines the behavior of an object without specifying the implementation. It allows you to define a contract that a concrete type can choose to fulfill by implementing those methods.
* A screen shot of a computer program

  Description automatically generated
* Shape is an interface that declares the Area() method.
* Rectangle and Circle are concrete types that implement the Shape interface by defining the Area() method for their specific shapes.
* The main() function creates instances of Rectangle and Circle, stores them in a slice of Shape, and then iterates through the shapes to calculate and print their areas.
* Interfaces are powerful in GoLang because they enable polymorphism and allow different types to be treated in a uniform way if they fulfill the interface contract by implementing the specified methods.

**Errors**

* Errors are handled using the error interface. The error interface is a built-in type with the following definition:
* A black background with white and blue text

  Description automatically generated
* In Go, functions often return an error as the last return value to indicate whether the function execution was successful or encountered an issue. It's a common pattern to see functions returning both the result and an error:
* A screen shot of a computer program

  Description automatically generated
* Here's an example of how you might use this:
* A screen shot of a computer program

  Description automatically generated
* The errors.New() function is commonly used to create a new error. However, Go allows you to create custom error types by implementing the Error() method on a custom struct.
* A screen shot of a computer program

  Description automatically generated
* Then you handle it similarly:
* A screen shot of a computer program

  Description automatically generated
* Defer
  + the defer statement is used to schedule a function call to be executed just before the surrounding function returns. It's a way to ensure that certain cleanup or finalization tasks are performed regardless of how the function exits—whether it's due to a return statement, a panic, or reaching the end of the function.
  + When you use defer, the function call and its arguments are evaluated immediately, but the actual execution of that function call is delayed until the surrounding function completes. This creates a Last In, First Out (LIFO) stack of deferred function calls within the function's scope
  + A screen shot of a computer code

    Description automatically generated
  + Here, "Hello" will be printed first, and then "Deferred 2" and "Deferred 1" will be printed in reverse order because of the LIFO stack of deferred functions.
  + defer is commonly used for tasks like closing files, unlocking mutexes, or releasing resources, ensuring that these actions are performed before exiting the function. It's particularly helpful for keeping code clean and maintaining readability by associating cleanup actions with their corresponding setup actions.
* Panic
  + panic is a built-in function that's used to signal a situation where the program cannot continue executing as expected. When a function encounters an error or an exceptional condition that it cannot handle, it can trigger a panic.
  + When a panic occurs, the normal flow of the program is disrupted, similar to throwing an exception in other languages. It immediately stops the execution of the current function, unwinding the stack of function calls (deferred functions are executed during this unwind process), and then continues until it reaches the top of the goroutine, where the program will terminate, displaying information about the panic, including the stack trace.
  + A screen shot of a computer code

    Description automatically generated
  + However, panics are not typically used for ordinary error handling. In Go, idiomatic error handling is done using the error type, returning errors from functions, and handling them explicitly using conditional statements like if err != nil.
  + Panics are more suitable for unrecoverable errors such as invariant violations or situations where continuing the program's execution would lead to an unstable or corrupt state. For instance, if a critical invariant of your program is broken, like a nil pointer dereference that cannot be recovered gracefully, triggering a panic might be appropriate. Panics in Go are meant to represent extreme, unexpected, and typically unrecoverable errors.
* Recover
  + recover is a built-in function used in conjunction with defer to manage and potentially regain control after a panic has occurred.
  + When a panic happens in a function, it immediately stops the normal execution flow, unwinds the function call stack, and begins executing any deferred functions (those functions scheduled using defer) in the reverse order they were deferred. This includes deferred calls up until the point of the panic.
  + During this unwinding process, if a deferred function containing a recover call is encountered, and that function is directly called by the panicking function (i.e., it's on the call stack of the panicking function), recover can be used to capture the panic value and restore some degree of control.
  + A screen shot of a computer code

    Description automatically generated
  + Output: 
  + In this example, recoverFromPanic is a deferred function that contains a call to recover. If a panic occurs in the main function, the deferred recoverFromPanic function is called during the unwinding process. The recover function checks if there was a panic and retrieves the value passed to panic.
  + A common use of recover is to restore some basic functionality or perform cleanup tasks after capturing the panic value. However, it's crucial to note a few important points about recover:
    - Context is critical: recover only works if it's called in a deferred function directly invoked by the panicking function.
    - Control flow recovery: It allows limited recovery of control after a panic, but it's generally used for logging, cleanup, and potentially resuming a limited amount of functionality.
    - Not a general error handling mechanism: Using recover as a general error-handling mechanism is discouraged in Go. It's primarily intended for specific scenarios where controlled recovery from a panic is necessary, not as a replacement for regular error handling.

**Modules**

* In version 1.13, the authors of Go added a new way of managing the libraries a Go project depends on, called Go modules.
* A module has a number of Go code files implementing the functionality of a package, but it also has two additional and important files in the root: the go.mod file and the go.sum file.
* These files contain information the go tool uses to keep track of your module’s configuration, and are commonly maintained by the tool so you don’t need to.
* The first thing to do is decide the directory the module will live in. Next, you’ll create the module directory itself. Usually, the module’s top-level directory name is the same as the module name, which makes things easier to keep track of.
* 
* The next step is to create a go.mod file within the mymodule directory to define the Go module itself. To do this, you’ll use the go tool’s mod init command and provide it with the module’s name, which in this case is mymodule. Now create the module by running go mod init from the mymodule directory and provide it with the module’s name, mymodule:
* 
* With the module created, your directory structure will now look like this:
* A close-up of a blue background

  Description automatically generated
* go.mod file
  + It’s the file that contains the name of the module and versions of other modules your own module depends on.
  + projects/mymodule/go.mod will look like:
  + A screen shot of a computer

    Description automatically generated
  + The first line, the module directive, tells Go the name of your module so that when it’s looking at import paths in a package, it knows not to look elsewhere for mymodule. The mymodule value comes from the parameter you passed to go mod init
  + The only other line in the file at this point, the go directive, tells Go which version of the language the module is targeting. In this case, since the module was created using Go 1.16, the go directive says 1.16
* Also create the following main.go file:
* A screen shot of a computer code

  Description automatically generated
* Once you have created the main.go file, the module’s directory structure will look similar to this:
* A screenshot of a computer

  Description automatically generated
* Adding a Package to your module:
  + Create a new mypackage package directory
  + A screenshot of a computer

    Description automatically generated
  + Inside this package, have the following mypackage.go file:
  + A screen shot of a computer

    Description automatically generated
  + Identifiers starting with a lowercase letter are considered unexported and can only be accessed within the package where they are defined. On the other hand, identifiers starting with an uppercase letter are exported and can be accessed from outside the package.
  + Since you want the PrintHello function to be available from another package, the capital P in the function name is important. The capital letter means the function is exported and available to any outside program.
  + Now that you’ve created the mypackage package with an exported function, you will need to import it from the mymodule package to use it. This is similar to how you would import other packages, such as the fmt package previously, except this time you’ll include your module’s name at the beginning of the import path. Open your main.go file from the mymodule directory and add a call to PrintHello by adding the highlighted lines below:
  + A screen shot of a computer program

    Description automatically generated
  + If you take a closer look at the import statement, you’ll see the new import begins with mymodule, which is the same module name you set in the go.mod file. This is followed by the path separator and the package you want to import, mypackage in this case:
  + 
  + In the future, if you add packages inside mypackage, you would also add them to the end of the import path in a similar way. For example, If you had another package called extrapackage inside mypackage, your import path for that package would be mymodule/mypackage/extrapackage.
  + Run your updated module with go run and main.go from the mymodule directory as before:
  + 
  + Output: A blue background with white text

    Description automatically generated
* Adding a Remote Module as a Dependency
  + Go modules are distributed from version control repositories, commonly Git repositories. When you want to add a new module as a dependency to your own, you use the repository’s path as a way to reference the module you’d like to use. When Go sees the import path for these modules, it can infer where to find it remotely based on this repository path.
  + For this example, you’ll add a dependency on the github.com/spf13/cobra library to your module. Cobra is a popular library for creating console applications, but we won’t address that in this tutorial.
  + Similar to when you created the mymodule module, you’ll again use the go tool. However, this time, you’ll run the go get command from the mymodule directory. Run go get and provide the module you’d like to add. In this case, you’ll get github.com/spf13/cobra:
  + 
  + When you run this command, the go tool will look up the Cobra repository from the path you specified and determine which version of Cobra is the latest by looking at the repository’s branches and tags. It will then download that version and keep track of the one it chose by adding the module name and the version to the go.mod file for future reference.
  + Now, open the go.mod file in the mymodule directory to see how the go tool updated the go.mod file when you added the new dependency. The example below could change depending on the current version of Cobra that’s been released or the version of the Go tooling you’re using, but the overall structure of the changes should be similar:
  + A computer screen shot of a computer code

    Description automatically generated
  + A new section using the require directive has been added. This directive tells Go which module you want, such as github.com/spf13/cobra, and the version of the module you added. Sometimes require directives will also include an // indirect comment. This comment says that, at the time the require directive was added, the module is not referenced directly in any of the module’s source files. A few additional require lines were also added to the file. These lines are other modules Cobra depends on that the Go tool determined should be referenced as well.
  + You may have also noticed a new file, go.sum, was created in the mymodule directory after running the go run command. This is another important file for Go modules and contains information used by Go to record specific hashes and versions of dependencies. This ensures consistency of the dependencies, even if they are installed on a different machine.
  + Once you have the dependency downloaded you’ll want to update your main.go file with some minimal Cobra code to use the new dependency. Update your main.go file in the mymodule directory with the Cobra code below to use the new dependency:
  + A computer screen shot of a program code

    Description automatically generated
  + This code creates a cobra.Command structure with a Run function containing your existing “Hello” statements, which will then be executed with a call to cmd.Execute(). Now, run the updated code:
  + 
  + Output: A screen shot of a computer

    Description automatically generated
  + Using go get to add the latest version of a remote dependency, such as github.com/sp13/cobra here, makes it easier to keep your dependencies updated with the latest bug fixes. However, sometimes there may be times where you’d rather use a specific version of a module, a repository tag, or a repository branch
* Using a Specific Version of a Module
  + Since Go modules are distributed from a version control repository, they can use version control features such as tags, branches, and even commits.
  + The go tool knows that if a specific version isn’t provided using @, it should use the special version latest. The latest version isn’t actually in the repository, like my-tag or my-branch may be. It’s built into the go tool as a helper so you don’t need to search for the latest version yourself.
  + For example, when you added your dependency initially, you could have also used the following command for the same result:
  + 
  + Now, imagine there’s a module you use that’s currently in development. For this example, call it your\_domain/sammy/awesome. There’s a new feature being added to this awesome module and work is being done in a branch called new-feature. To add this branch as a dependency of your own module you would provide go get with the module path, followed by the @ symbol, followed by the name of the branch:
  + 
  + Running this command would cause go to connect to the your\_domain/sammy/awesome repository, download the new-feature branch at the current latest commit for the branch, and add that information to the go.mod file.
  + Using your module’s Cobra dependency as an example, suppose you need to reference commit 07445ea of github.com/spf13/cobra because it has some changes you need and you can’t use another version for some reason. In this case, you can provide the commit hash after the @ symbol the same as you would for a branch or a tag. Run the go get command in your mymodule directory with the module and version to download the new version:
  + 
  + If you open your module’s go.mod file again you’ll see that go get has updated the require line for github.com/spf13/cobra to reference the commit you specified:
  + A screenshot of a computer program

    Description automatically generated
  + suppose you want to use Cobra version 1.1.1. You could look at the Cobra repository and see it has a tag named v1.1.1, among others. To use this tagged version, you would use the @ symbol in a go get command, just as you would use a non-version tag or branch. Now, update your module to use Cobra 1.1.1 by running the go get command with v1.1.1 as the version:
  + 
  + Now if you open your module’s go.mod file, you’ll see go get has updated the require line for github.com/spf13/cobra to reference the version you provided:
  + A computer screen shot of a computer code

    Description automatically generated

File IO

* File I/O in Go (or Golang) involves performing input/output operations on files. The os and io/ioutil packages provide functionalities to work with files in Go.
  + os.Create() : The os.Create() method is used to creates a file with the desired name. If a file with the same name already exists, then the create function truncates the file.
  + ioutil.ReadFile() : The ioutil.ReadFile() method takes the path to the file to be read as it’s the only parameter. This method returns either the data of the file or an error.
  + ioutil.WriteFile() : The ioutil.WriteFile() is used to write data to a file. The WriteFile() method takes in 3 different parameters, the first is the location of the file we wish to write to, the second is the data object, and the third is the FileMode, which represents the file’s mode and permission bits.
  + log.Fatalf : Fatalf will cause the program to terminate after printing the log message. It is equivalent to Printf() followed by a call to os.Exit(1).
  + log.Panicf : Panic is just like an exception that may arise at runtime. Panicln is equivalent to Println() followed by a call to panic(). The argument passed to panic() will be printed when the program terminates.
  + bufio.NewReader(os.Stdin) : This method returns a new Reader whose buffer has the default size(4096 bytes).
  + inputReader.ReadString(‘\n’) : This method is used to read user input from stdin and reads until the first occurrence of delimiter in the input, returning a string containing the data up to and including the delimiter. If an error is encountered before finding a delimiter, it returns the data read before the error and the error itself.
* Creating and Writing to a File:
  + A screen shot of a computer program

    Description automatically generated
  + Output: A black screen with white text

    Description automatically generated
* User Input
  + A computer screen shot of a program code

    Description automatically generated
  + Output: 

**Type Aliases**

* In Go, type aliases are declarations that create an alternative name for an existing type. They don't create a new type; instead, they introduce a new name to refer to an existing type, making the code more readable and maintainable.
* A screen shot of a computer program

  Description automatically generated
* In this example, MyInt is a type alias for the built-in int type. You can use MyInt as if it were an int. Type aliases can be beneficial when you want to provide more descriptive names for types or when you want to distinguish between different uses of the same type.

**Generics**

* Generics in Go allow you to write functions or data structures that can be used with different types.
* Suppose we have the following code:  
  A screen shot of a computer program

  Description automatically generated
* What if we want to make a similar function for float64? We can do the following:
* A computer screen shot of a program code

  Description automatically generated
* We can also let golang infer the type being passed to the add function as shown below:
* A computer screen shot of text

  Description automatically generated
* What if we want to add more numeric types? We can make a type alias to make the code clean:  
  A screen shot of a computer program

  Description automatically generated
* What if we have a type alias that has an underlying type of int? We can adjust the code to account for the by adding the following to Numeric type:
* A screen shot of a computer program

  Description automatically generated

**Goroutines**

* A goroutine is a lightweight thread of execution in Go (Golang). It's a fundamental part of the Go language's concurrency model. Goroutines allow functions or methods to be executed concurrently, independently of one another.
* Unlike traditional threads in many other programming languages, goroutines are managed by the Go runtime and are multiplexed onto multiple OS threads. They're designed to be extremely lightweight, allowing thousands or even millions of goroutines to exist within a single Go program without consuming a lot of memory.
* Concurrency vs Parallelism:
  + In Go, when you use the go keyword to launch a function as a goroutine, it creates a concurrent execution. This means the code inside the goroutine can potentially run concurrently with other parts of your program.
  + The distinction lies in how the Go runtime manages these goroutines. Under the hood, the Go runtime scheduler decides how these goroutines are executed on available CPU cores. If you have multiple CPU cores, these goroutines can indeed execute in parallel, utilizing the available cores for simultaneous execution.
  + However, if your machine has a single CPU core or if the workload doesn't allow for simultaneous execution, the Go scheduler will employ a technique called "multiplexing" to manage these goroutines concurrently, allowing them to take turns executing in a way that gives the appearance of parallelism, even on a single core.
  + So, while using the go keyword creates concurrent execution, whether it results in actual parallelism depends on the resources available and the workload. The Go runtime handles this underlying complexity, allowing you to write concurrent code without explicitly managing the low-level details of parallelism.
* Consider the following code
* A screen shot of a computer code

  Description automatically generated
* Output: 
* This code is synchronous since fmt.Println(“hi”) has to wait for f(1) to finish before it gets executed.
* With goroutines, we can make f fork off of our main function asynchronously, meaning main doesn’t have to wait for f(1) to finish before it continues to fmt.Println(). To do so, we simply put the go keyword in front of f(1)
* A screen shot of a computer program

  Description automatically generated
* Output: 
* Notice that 1 never gets printed. In this code, we’re using goroutines with go f(1) in the main function. Goroutines are independently executing functions, and when you execute go f(1), it starts a new goroutine to execute the function f(1) concurrently with the rest of the code in main.
* However, the main function itself doesn't wait for the goroutine to finish executing before it exits. This means that the program might terminate before the goroutine has a chance to execute fully. In your case, the main goroutine immediately prints "hi" and then exits, and because the program exits, the goroutine may not have enough time to run and print 1 as it's intended to do in function f.
* In order to have a temporary solution to fix this, we’ll adjust the code to the following (we also added more goroutines for example’s sake):
* A screen shot of a computer program

  Description automatically generated
* Output: 
* In this code, you're launching three goroutines (go f(1), go f(2), and go f(3)) to execute the f function with different integer arguments concurrently with the fmt.Println("hi") statement in the main function.
* However, similar to the previous example, the main goroutine doesn't wait for the other goroutines to complete their execution. It immediately prints "hi" and then waits for 2 seconds using time.Sleep(time.Second \* 2). It just happens that during this sleep time, the other goroutines (f(1), f(2), f(3)) have enough time to execute and print their respective numbers.
* The output can be non-deterministic due to the concurrent nature of goroutines and the fact that the main goroutine doesn't wait for them to finish before exiting. Sometimes you might see the numbers 1, 2, 3 printed, but other times you might only see "hi" printed.

**Concurrency Structure**

* Go follows a model of concurrency called the fork-join model as shown below:
* A screen shot of a computer

  Description automatically generated
* Child processes (goroutines) are forked off the main function. So go someFunc(“1”) is representative of a goroutine being forked off the main function.
* With this fork-join model, we see that child process is forked off the main function. But we also see that this child process rejoins the main function. This point is called the join point. And the main function is responsible for implementing the rejoining of the child process to the main process.
* In the example without time.sleep, the results of the goroutines never got printed since we didn’t rejoin the child process. In fact, the time.sleep doesn’t actually sync our main process with the goroutines as explained previously. For example, if we only slept for 1ms, the results of the goroutines might never have been printed.
* So as of now, we technically don’t have a join point.

**WaitGroup**

* The sync.WaitGroup is a useful synchronization primitive used to wait for a collection of goroutines to finish their execution before proceeding further in the main program. It helps in coordinating the execution of concurrent tasks.
* A computer screen shot of a program code

  Description automatically generated
* Output: A screen shot of a computer

  Description automatically generated
* worker is a function that simulates some work by printing a start message, sleeping for one second (simulating work duration), and then printing a completion message.
* main initializes a sync.WaitGroup named wg to coordinate the goroutines. numWorkers is set to 5, indicating the number of worker goroutines to be launched.
* A loop is used to launch numWorkers goroutines.
* Inside the loop, wg.Add(1) increments the WaitGroup counter before starting each goroutine. This counter keeps track of the number of active goroutines.
* A goroutine is launched using an anonymous function. This anonymous function takes an argument x (which captures the loop variable i).
* Inside the anonymous function, defer wg.Done() is used to defer the execution of wg.Done(), which decrements the WaitGroup counter when the goroutine finishes its execution. The worker function is called with the argument x, which represents the current loop iteration value.
* wg.Wait() blocks the execution of the main goroutine until the WaitGroup counter goes back to 0. It waits for all the launched goroutines to call wg.Done() and signal their completion.
* Once all the worker goroutines have finished their tasks, the program proceeds to the next line and prints "All workers have finished their tasks".
* In summary, this code showcases how to use sync.WaitGroup to coordinate multiple goroutines performing concurrent tasks and ensure that the main program waits for all of them to finish before proceeding further. It also demonstrates how to capture loop variables safely by passing them as arguments to goroutines to avoid shared references to mutable variables.

**Channels**

* Go provides channels that you can use for bidirectional communication between goroutines.
* Bidirectional communication means that one goroutine will send a message and the other will read it. Sends and receives are blocking. Code execution will be stopped until the write and read are done successfully.
* Recall that in our previous example, the goroutines were running independently of each other. So in order for them to communicate, we can have them reference the same place in memory which is where a channel would live.
* Suppose we have the following example:  
  A blackboard with different channels

  Description automatically generated
* Suppose goroutine2 depends on information from goroutine3. In other words, goroutine3 needs to send data to goroutine2.
* We can think of channels as FIFO queues.
* Now suppose goroutine3 sends three units of data (represented by hearts) to the channel:
* A blackboard with different channels

  Description automatically generated
* Then, goroutine2 can just read off those units of data in a FIFO order.
* Recall that that our main function is also a goroutine as well. This means that our main function could communicate with its child goroutines via channels as well as shown below:
* A diagram of different channels

  Description automatically generated
* Buffered and unbuffered channels differ in how they handle the sending and receiving of values.
* Unbuffered Channels
  + An unbuffered channel doesn't have any capacity to store data. When a value is sent on an unbuffered channel, the sender will wait until there's a receiver ready to receive the value. Similarly, when a receiver tries to receive from an unbuffered channel, it will wait until there's a sender ready to send.
  + Below is an example of channels in code:
  + A screen shot of a computer program

    Description automatically generated
  + Output: 
  + Explanation
    - myChannel := make(chan string): Creates a channel named myChannel of type string. Channels are used for communication and synchronization between goroutines.
    - go func() { myChannel <- "my message" }(): This line starts a new goroutine (concurrent execution) using the go keyword. Inside this goroutine, an anonymous function is defined that sends the string "my message" into the myChannel using the channel operator <-.
    - message := <-myChannel: This line retrieves data from the myChannel and assigns it to the variable message. This line of code is blocking. This means that the main function will wait for either the channel to close (more on this later) or for a message to be received from this channel. This is the join point as mentioned in the concurrency structure section.
* Buffered Channels
  + Buffered channels, on the other hand, have a capacity defined when they are created. They can store a specific number of elements without a corresponding receiver. Sending to a buffered channel will only block when the buffer is full, and receiving will block when the buffer is empty.
  + A screen shot of a computer program

    Description automatically generated
  + Output: A black background with white text

    Description automatically generated
  + ch := make(chan int, 2): Creates a buffered channel capable of holding two integers.
  + ch <- 1: Sends the value 1 into the channel. Since the channel has a buffer size of 2, this operation succeeds immediately.
  + fmt.Println("Sent 1 to the channel"): Prints "Sent 1 to the channel" to the console.
  + ch <- 2: Sends the value 2 into the channel. The buffer still has space, so this operation succeeds immediately.
  + fmt.Println("Sent 2 to the channel"): Prints "Sent 2 to the channel" to the console.
  + go func() { ... }(): Starts a new goroutine that attempts to send 3 into the channel. Since the channel's buffer is full (already contains 1 and 2), this goroutine will be blocked until there's space in the buffer to send the value.
  + fmt.Println("Received:", <-ch): Receives a value from the channel. It receives 1 from the channel because it was the first value sent into the channel and was subsequently received. The buffer now has space for another value, so the goroutine sending 3 (ch <- 3) continues executing. But before it is able to continue the next two lines jus
  + fmt.Println("Received:", <-ch): Receives the next value from the channel. It receives 2 because it's the next available value in the channel after 1.
  + fmt.Println("Received:", <-ch): Receives the next value from the channel. It receives 3 because it's the next available value in the channel after 2.
  + When the main function reaches its end, it doesn't wait for any remaining goroutines to finish their execution. Therefore, in some cases, the goroutine attempting to send 3 might not have enough time to execute and print "Sent 3 to the channel" before the program exits. This is because the program doesn't wait for the goroutines to finish their execution before terminating.
  + If you want to ensure that the goroutine has enough time to execute and print its message, you might consider using synchronization mechanisms like sync.WaitGroup to coordinate the main function's exit until all goroutines finish their execution. This way, the main function will wait for the goroutines to complete before terminating, ensuring that "Sent 3 to the channel" gets a chance to be printed.
  + Here's an example using sync.WaitGroup:
  + A computer screen shot of text

    Description automatically generated
  + Output: A screen shot of a computer

    Description automatically generated
  + Notice that received 3 is printed before sent 3 to the channel is printed. This is misleading because sent 3 to the channel (ch <-3) happens first, its just that the fmt.Println("Sent 3 to the channel") takes some time.
* Channel Iteration and Closing channels
  + Channel iteration is useful when you're unsure about the exact number of values that will be sent through the channel. This construct allows you to continuously receive values from the channel until it's closed by the sender, without needing to know the specific count in advance.
  + Closing a channel is a way to signal that no more values will be sent on it. It's a built-in mechanism to communicate completion or that the sender is finished sending data. Once a channel is closed, any further attempts to send data into that channel will cause a panic, but receiving from a closed channel can still yield values until the channel is drained of all its remaining values.
  + Suppose we have the following code:
  + A screen shot of a computer program

    Description automatically generated
  + In this specific case, we know how many values will be sent/received through the channel, 3.
  + But what if main doesn’t know how many values will be received, we can do the following, we also changed 3 to 2 for example sake:
  + A screen shot of a computer program

    Description automatically generated
  + In this code our main function fires f as a goroutine. Then it goes through an infinite for loop. It tries and get the data in the channel, but there isn’t any data yet, so it waits.
  + When f was fired off, f adds 0 to the channel. Then, f tries to add 1 to the channel which is unsuccessful just yet since it needs to wait for the value of 0 in the channel to first be received.
  + Back to the main function, we get the value from the channel, and also an isOpen variable which tells us if the channel is open or closed. Since it is open, we print 0, then try to get the next value from the channel, but there isn’t one just yet so we wait. (this section is explained earlier so not too much detail will be done)
  + Since the value of 0 has been received, f is able to add 1 to the channel. This is received by the for loop in the main function and 1 is printed out. Then f exists its for loop and closes the channel. Then the for loop in the main function tries to receive data from the channel, but it realizes the channel is closed, then it breaks out the loop.
  + Note that if we didn’t close the channel, we’d get a deadlock since all the goroutines would be asleep. But the main function’s for loop would still be waiting for data to be put in the channel.
  + Go has added syntactic sugar to the above to have the following equivalent code:
  + A screen shot of a computer program

    Description automatically generated

**Select**

* The select statement allows a goroutine to wait on multiple channel operations simultaneously. It's similar to a switch statement but works with channel operations.
* A screen shot of a computer program

  Description automatically generated
* Output: 
* A select statement will block until one of its cases can run. In this select statement waits until it receives a message from myChannel1 or a message from myChannel2. Once it receives a message from either channel, it’s going to execute the code inside that block (in our case, print out the received message). If the select receives multiple messages from multiple channels at the same time, it’s going to choose one at random. This is why only “my message 2” is printed instead of “my message1”, its random.

**Mutexes**

* a mutex (short for mutual exclusion) is a synchronization primitive used to control access to shared resources, allowing only one goroutine at a time to access the shared data.
* Mutexes are part of Go's sync package and are used to prevent race conditions where multiple goroutines access the same resource concurrently, potentially causing conflicts or unexpected behavior due to simultaneous read and write operations.
* Here's a basic example of how you might use a mutex in Go:
* A computer screen shot of a program code

  Description automatically generated
* Output: A screenshot of a computer program

  Description automatically generated
* sync.Mutex is used to create a mutex variable named mutex.
* The incrementCounter function increments a shared counter variable while locking the mutex using mutex.Lock() before accessing it and unlocking it using mutex.Unlock() after the operation.
* Multiple goroutines are started to concurrently execute incrementCounter().
* The program waits for user input (fmt.Scanln()) to prevent it from ending prematurely, allowing the goroutines to finish their execution.
* By using mutexes, you ensure that only one goroutine can access the critical section of code (in this case, modifying the counter variable) at any given time. This prevents race conditions and ensures data consistency when dealing with shared resources.
* RWMutex
  + The fundamental difference between a Mutex and a RWMutex (read-write mutex) in Go lies in their behavior regarding concurrent access:
  + Mutex:
    - Exclusive Lock: Mutex provides exclusive access, allowing only one goroutine to hold the lock at a time. It's used when you have critical sections of code where you want to prevent multiple goroutines from reading or writing simultaneously.
  + RWMutex:
    - Shared Read Lock, Exclusive Write Lock: RWMutex allows for multiple readers to hold a lock simultaneously while ensuring that only one writer can hold the lock exclusively. It's suitable when you have scenarios where reads are much more frequent than writes and concurrent reads can happen without interfering with each other.
  + A screen shot of a computer program

    Description automatically generated
  + Output: A screen shot of a computer

    Description automatically generated
  + This example demonstrates a scenario where multiple goroutines are reading data concurrently using readData, while another goroutine writes data using writeData. The sync.RWMutex ensures that multiple readers can access the data simultaneously, but when a writer needs to modify the data, it exclusively locks the resource, preventing concurrent reads or writes.
  + The RLock() and RUnlock() methods are used for reading operations, allowing multiple goroutines to read the data simultaneously. The Lock() and Unlock() methods are used for writing operations, allowing only one goroutine to write data at a time, blocking all other readers and writers until the write operation is complete.
  + This mechanism helps in scenarios where the data is frequently read but less frequently written, improving concurrency and performance by allowing multiple readers unless a writer needs exclusive access.
  + Note that you might notice Data written: key=key3, value=value3 is followed by reader 1 – Read data: key=key2, value=value2. You might expect Data written: key=key3, value=value3 is followed by reader 1 – Read data: key=key3, value=value3. This is misleading because value := readData(key) happens, and then fmt.Printf("Reader %d - Read data: key=%s, value=%s\n", id, key, value). After value := readData(key) happens but before fmt.Printf("Reader %d - Read data: key=%s, value=%s\n", id, key, value) is able to run, the data write happens which is why we see Data written: key=key3, value=value3, then reader 1 – Read data: key=key2, value=value2.

**Sending HTTP GET Requests**

* A screenshot of a computer program

  Description automatically generated
* Output: A screen shot of a computer

  Description automatically generated
* The http.Get function in Go is blocking, meaning that it will halt the execution of further code until it either receives a response or encounters an error. This function will wait until it gets a response from the server or until it times out due to network issues.
* If you need non-blocking behavior, you can use the http.NewRequest method along
* Errors that http.Get might raise:
  + Network Errors: These could include connection timeouts, DNS resolution errors, refused connections, etc. If there are issues with the network or the URL provided, http.Get might return an error.
* Errors that ioutil.ReadAll might raise:
  + Read Errors: Occur when there's an issue while reading the response body. This could happen due to issues like unexpected EOF (End of File), bad data, or the connection being closed prematurely.
  + Memory Errors: If the response body is excessively large and exceeds the available memory for reading, ioutil.ReadAll might raise an error.
* The resp.Body.Close() statement is crucial. In Go, when you make an HTTP request using http.Get() or similar methods, it returns a response object (resp). This response object has a Body property, which is a stream of data containing the response from the server. It's essential to close this body to free up resources once you're done processing the response. The defer statement ensures that resp.Body.Close() is executed when the surrounding function (main() in this case) finishes, regardless of where in the function the return statements occur.
* Leaving the response body open can cause the following issues:
  + Resource Exhaustion: If your code repeatedly makes HTTP requests without closing the response body, it consumes system resources (file descriptors, memory) each time. Over time, this can lead to resource exhaustion and potentially cause your program to crash.
  + Potential Connection Limits: In some scenarios, not closing the response body might keep connections open unnecessarily, which could hit connection limits, especially in scenarios where there's a maximum number of concurrent connections allowed to a server.
  + Memory Leaks: If the response body is not closed, it might prevent the underlying resources (like memory buffers) associated with the response body from being released. This can lead to memory leaks, degrading your program's performance over time.
* The body variable in this code is of type []byte (a byte slice). When you read the response body using ioutil.ReadAll(resp.Body), it reads the entire body of the HTTP response and stores it as a sequence of bytes in the body variable. The body variable is then converted to a string using string(body), resulting in a string type variable called bodyString.
* Remember that body is a byte slice because HTTP responses can contain various types of content (HTML, JSON, images, etc.), which are represented in bytes. Converting it to a string assumes the content is text-based and readable as a string.
* Notice how the raw string has \n in it which is why when we do Println(bodyString), it looks a dict in python.
* We had to convert the body to a map in case we need to use the data later.

**Sending HTTP POST Requests**

* A computer screen shot of many colorful lines

  Description automatically generated
* Output: 
* the json.Marshal() function is used to transform data into a JSON-encoded byte slice. It serializes the data structure (in this case, the map payload) into a JSON string. This process is often referred to as "marshaling" or "stringifying"
* bytes.NewBuffer() creates and initializes a new buffer using the provided byte slice. In the context of the code provided, bytes.NewBuffer(jsonPayload) is used to create a buffer containing the JSON payload (jsonPayload), which is then used as the request body in the HTTP POST request.
* The http.Post() function expects the request body to be of type io.Reader, and bytes.NewBuffer() returns a value that satisfies this interface (\*bytes.Buffer implements the io.Reader interface). Therefore, the JSON payload, converted into a byte slice and then wrapped in a buffer via bytes.NewBuffer(), is suitable for sending as the body of the POST request.