Packaging & Tooling :

* All Go programs are organized into groups of files called packages, so that code has the ability to be included into other projects as smaller reusable pieces. Let’s look at the packages that make up Go’s http functionality in the standard library.

*net/http/*

*cgi/*

*cookiejar/*

*testdata/*

*fcgi/*

*httptest/*

*httputil/*

*pprof/*

*testdata/*

These directories contain a series of related files with the .go extension, and provide clear separation of smaller units of code relating to the implementation of HTTP servers, clients, and utilities to test and profile them.

* Each package can be imported and used individually so that developers can import only the specific functionality that they need
* All .go files must declare the package that they belong to as the first line of the file excluding whitespace and comments.
* Packages are contained in a single directory. You may not have multiple packages in the same directory, nor may you split a package across multiple directories. This means that all .go files in a single directory must declare the same package name
* The convention for naming your package is to use the name of the directory containing it. Our example from the net/http package, all the files contained within the http directory are a part of the http package.
* When naming your packages and their directories, you should use short, concise, lowercase names, because they will be typed often while you’re developing.
* Keep in mind that a unique name is not required, because you import the package using its full path. Your package name is used as the default name when your package is imported, but it can be overridden.
* The package name main has special meaning in Go. It designates to the Go command that this package is intended to be compiled into a binary executable. All of the executable programs you build in Go must have a package called main. When the main package is encountered by the compiler, it must also find a function called main(); otherwise a binary executable won’t be created. The main() function is the entry point for the program, so without one, the program has no starting point. The name of the final binary will take the name of the directory the main package is declared in.
* Importing:
  + Packages in the standard library are found under where Go is installed on your computer. Packages that are created by you or other Go developers live inside the GOPATH, which is your own personal workspace for packages
  + Let’s take a look at an example. If Go was installed under */usr/local/go* and your GOPATH was set to */home/myproject;/home/mylibraries*, the compiler would look for the net/http package in the following order:

*/usr/local/go/src/pkg/net/http*

*/home/myproject/src/net/http*

*/home/mylibraries/src/net/http*

* Named imports :- multiple packages with the same name ?

*import (*

*"fmt"*

*myfmt "mylib/fmt"*

*)*

* The Go compiler will fail the build and output an error whenever you import a package that you don’t use
* BLANK IDENTIFIER :- The \_ (underscore character) is known as the blank identifier and has many uses within Go. It’s used when you want to throw away the assignment of a value, including the assignment of an import to its package name, or ignore return values from a function when you’re only interested in the others.
* Init functions :- Each package has the ability to provide as many init functions as necessary to be invoked at the beginning of execution time. All the init functions that are discovered by the compiler are scheduled to be executed prior to the main function being executed. The init functions are great for setting up packages, initializing variables, or performing any other bootstrapping you may need prior to the program running.

*package postgres*

*import (*

*"database/sql"*

*)*

*func init() {*

*sql.Register("postgres", new(PostgresDriver)) # registerdriver*

*}*

*package main*

*import (*

*"database/sql"*

*\_ "github.com/goinaction/code/chapter3/dbdriver/postgres"*

*)*

*func main() {*

*sql.Open("postgres", "mydb")*

*}*

* GO tools :- read
* GO Developer tools :
  + go vet : It won’t write code for you, but once you’ve written some code, the vet command will check your code for common errors :
    - Bad parameters in Printf-style function calls
    - Method signature errors for common method definitions
    - Bad struct tags
    - Unkeyed composite literals
  + go format : The fmt command is a favorite in the Go community. Instead of arguing about where curly braces should go, or whether to use tabs or spaces when you indent, the fmt tool makes these decisions moot by applying a predetermined layout to Go source code. To invoke this code formatter, type go fmt followed by a file or package specification. The fmt command will automatically format the source code files you specify and save them. Many Go developers configure their development environment to perform a go fmt on save or before committing to a code repository. Do yourself a favor and configure this right now.
  + go documentation : There’s another tool that will make your Go development process easier. Go has two ways to deliver documentation to developers. If you’re working at a command prompt, you can use the go doc command to print documentation directly to your terminal session. You can view a quick reference for a command or package without leaving your terminal
* Collaborating with other Go developers : Read in book
* Dependency management : Read in book

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Arrays, slices, and maps

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **Array internals & fundamentals :**
  + An array in Go is a fixed-length data type that contains a contiguous block of elements of the same type. This could be a built-in type such as integers and strings, or it can be a struct type.
  + An array is declared by specifying the type of data to be stored and the total number of elements required, also known as the array’s length. eg, *var array [5]int*
  + Once an array is declared, neither the type of data being stored nor its length can be changed. If you need more elements, you need to create a new array with the length needed and then copy the values from one array to the other.
  + When variables in Go are declared, they’re always initialized to their zero value for their respective type, and arrays are no different. When an array is initialized, each individual element that belongs to the array is initialized to its zero value
  + When variables in Go are declared, they’re always initialized to their zero value for their respective type, and arrays are no different. When an array is initialized, each individual element that belongs to the array is initialized to its zero value. eg

*array := [5]int{10, 20, 30, 40, 50},*

*array := [...]int{10, 20, 30, 40, 50},*

*array := [5]int{1: 10, 2: 20}*,

* + To access an individual element, use the [ ] operator.
  + You can have an array of pointers,

*array := [5]\*int {0: new(int), 1: new(int)}*

*\*array[0] = 10*

*\*array[1] = 20*

* + An array is a value in Go. This means you can use it in an assignment operation. The variable name denotes the entire array and, therefore, an array can be assigned to other arrays of the same type. Only arrays of the same type can be assigned.

*var array1 [5]string*

*array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}*

*// Copy the values from array2 into array1.*

*array1 = array2*

* + The type of an array variable includes both the length and the type of data that can be stored in each element. Only arrays of the same type can be assigned

*var array1 [4] string*

*array2 := [5] string {"Red", "Blue", "Green", "Yellow", "Pink"}*

*array1 = array2*

*// Compiler Error : cannot use array2 (type [5]string) as type [4]string in assignment*

* + Copying an array of pointers copies the pointer values and not the values that the pointers are pointing to.

*var array1 [3]\*string*

*array2 := [3]\*string{new(string), new(string), new(string)}*

*\*array2[0] = "Red"*

*\*array2[1] = "Blue"*

*\*array2[2] = "Green"*

*array1 = array2*

* + Arrays are always one-dimensional, but they can be composed to create multidimensional arrays. Multidimensional arrays come in handy when you need to manage data that may have parent/child relationships or is associated with a coordinate system.

*var array [4][2]int*

*array := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}*

*array := [4][2]int{1: {20, 21}, 3: {40, 41}}*

*array := [4][2]int{1: {0: 20}, 3: {1: 41}}*

* + To access an individual element, use the [ ] operator again and a bit of composition

*var array [2][2]int*

*array[0][0] = 10*

*array[0][1] = 20*

*array[1][0] = 30*

*array[1][1] = 40*

* + You can copy multidimensional arrays into each other as long as they have the same type. The type of a multidimensional array is based on the length of each dimension and the type of data that can be stored in each element

*var array1 [2][2]int*

*var array2 [2][2]int*

*array2[0][0] = 10*

*array2[0][1] = 20*

*array2[1][0] = 30*

*array2[1][1] = 40*

*array1 = array2*

* + Because an array is a value, you can copy individual dimensions,

*var array3 [2]int = array1[1]*

*// provided that array1 is multidimensional*

* + **Passing arrays between functions**
  + Passing an array between functions can be an expensive operation in terms of memory and performance. When you pass variables between functions, they’re always passed by value. When your variable is an array, this means the entire array, regardless of its size, is copied and passed to the function.
  + there’s a better and more efficient way of doing this. You can pass a pointer to the array and only copy eight bytes, instead of eight megabytes of memory on the stack.

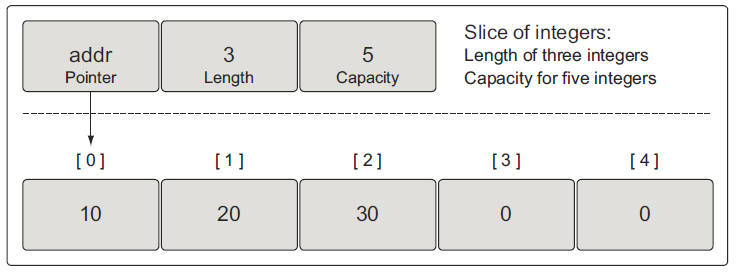
*var array [1e6]int*

*foo(&array)*

*func foo(array \*[1e6]int) { ...*

*}*

* **Slice internals and fundamentals :**
  + A slice is a data structure that provides a way for you to work with and manage collections of data. Slices are built around the concept of dynamic arrays that can grow and shrink as you see fit. They’re flexible in terms of growth because they have their own built-in function called append, which can grow a slice quickly with efficiency. You can also reduce the size of a slice by slicing out a part of the underlying memory. Slices give you all the benefits of indexing, iteration, and garbage collection optimizations because the underlying memory is allocated in contiguous blocks.
  + Slices are tiny objects that abstract and manipulate an underlying array. They’re three field data structures that contain the metadata Go needs to manipulate the underlying arrays (see figure). The three fields are a pointer to the underlying array, the length or the number of elements the slice has access to, and the capacity or the number of elements the slice has available for growth



* + One way to create a slice is to use the built-in function make. When you use make, one option you have is to specify the length of the slice

*// Create a slice of strings.*

*slice := make([]string, 5) // length and capacity of 5 elements.*

*slice := make([]int, 3, 5) // length of 3 and capacity of 5 elements.*

*slice := make([]int, 5, 3) // Compiler Error: len larger than cap in make([]int)*

* + When you specify the length and capacity separately, you can create a slice with available capacity in the underlying array that you don’t have access to initially. eg, *slice := make([]int, 3, 5)*. The two elements not associated with the length of the slice can be incorporated so the slice can use those elements as well.
  + New slices can also be created to share this same underlying array and use any existing capacity.
  + An idiomatic way of creating a slice is to use a slice literal, It’s similar to creating an array, except you don’t specify a value inside of the [ ] operator. The initial length and capacity will be based on the number of elements you initialize.

*// Contains a length and capacity of 5 elements.*

*slice := []string{"Red", "Blue", "Green", "Yellow", "Pink"}*

*// Contains a length and capacity of 3 elements.*

*slice := []int{10, 20, 30}*

* + When using a slice literal, you can set the initial length and capacity. All you need to do is initialize the index that represents the length and capacity you need. The following syntax will create a slice with a length and capacity of 100 elements.

*// Initialize the 100th element with an empty string.*

*slice := []string{99: ""}*

* + Remember, if you specify a value inside the [ ] operator, you’re creating an array. If you don’t specify a value, you’re creating a slice
  + Sometimes in your programs you may need to declare a nil slice. A nil slice is created by declaring a slice without any initialization. *var slice []int*, A nil slice is the most common way you create slices in Go. They can be used with many of the standard library and built-in functions that work with slices. They’re useful when you want to represent a slice that doesn’t exist, such as when an exception occurs in a function that returns a slice.
  + You can also create an empty slice by declaring a slice with initialization

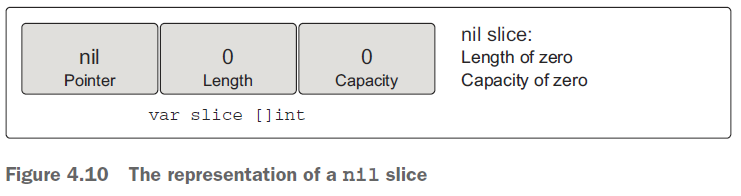
*// Use make to create an empty slice of integers.*

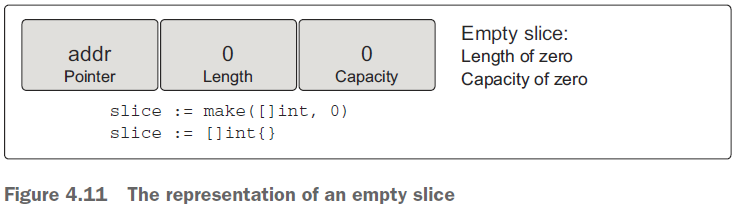
*slice := make([]int, 0)*

*// Use a slice literal to create an empty slice of integers.*

*slice := []int{}*

* + An empty slice contains a zero-element underlying array that allocates no storage. Empty slices are useful when you want to represent an empty collection, such as when a database query returns zero results.
  + Regardless of whether you’re using a nil slice or an empty slice, the built-in functions
  + append, len, and cap work the same





* + Assigning a value to any specific index within a slice is identical to how you do this with arrays. To change the value of an individual element, use the [ ] operator

*slice := []int{10, 20, 30, 40, 50}*

*// Change the value of index 1.*

*slice[1] = 25*

* + Slices are called such because you can slice a portion of the underlying array to create a new slice

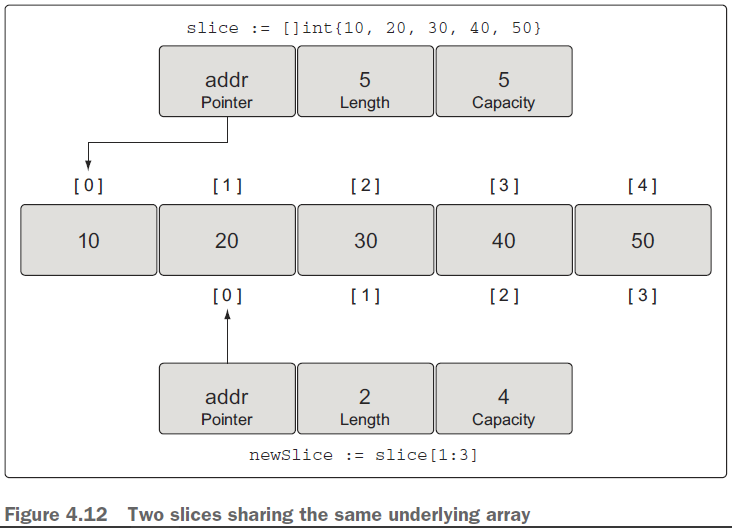
*slice := []int{10, 20, 30, 40, 50}*

*// Create a new slice.*

*// Contains a length of 2 and capacity of 4 elements.*

*newSlice := slice[1:3]*

After the slicing operation performed, we have two slices that are sharing the same underlying array. Take look at fig.



The original slice views the underlying array as having a capacity of five elements, but the view of *newSlice* is different. For *newSlice*, the underlying array has a capacity of four elements. *newSlice* can’t access the elements of the underlying array that are prior to its pointer. As far as *newSlice* is concerned, those elements don’t even exist.

Calculating the length and capacity for any new slice is performed using the following formula.

*For slice[i:j] with an underlying array of capacity k*

*Length: j - i*

*Capacity: k - i*

You need to remember that you now have two slices sharing the same underlying array. Changes made to the shared section of the underlying array by one slice can be seen by the other slice

* + A slice can only access indexes up to its length. Trying to access an element outside of its length will cause a runtime exception. The elements associated with a slice’s capacity are only available for growth. They must be incorporated into the slice’s length before they can be used.

*slice := []int{10, 20, 30, 40, 50}*

*// Create a new slice.*

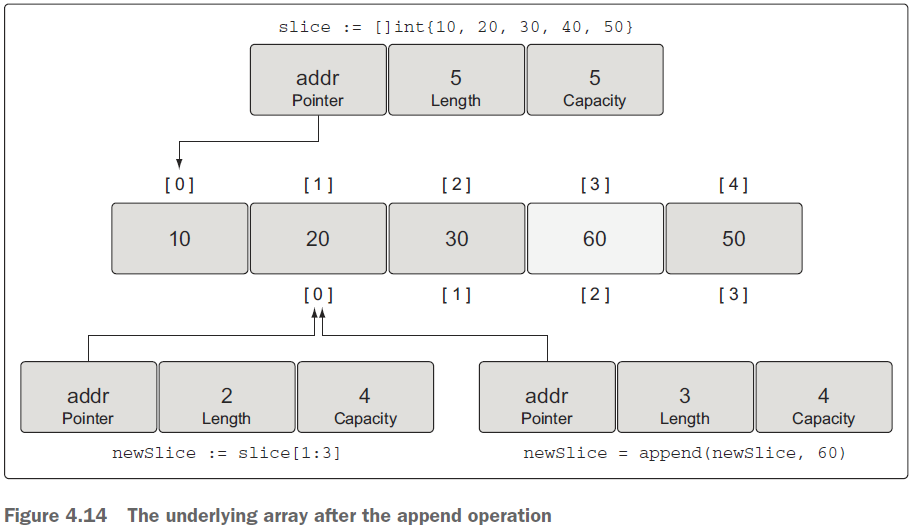
*// Contains a length of 2 and capacity of 4 elements.*

*newSlice := slice[1:3]*

*// Allocate a new element from capacity.*

*// Assign the value of 60 to the new element.*

*newSlice = append(newSlice, 60)*



* + When there’s no available capacity in the underlying array for a slice, the append function will create a new underlying array, copy the existing values that are being referenced, and assign the new value.

*// Create a slice of integers.*

*// Contains a length and capacity of 4 elements.*

*slice := []int{10, 20, 30, 40}*

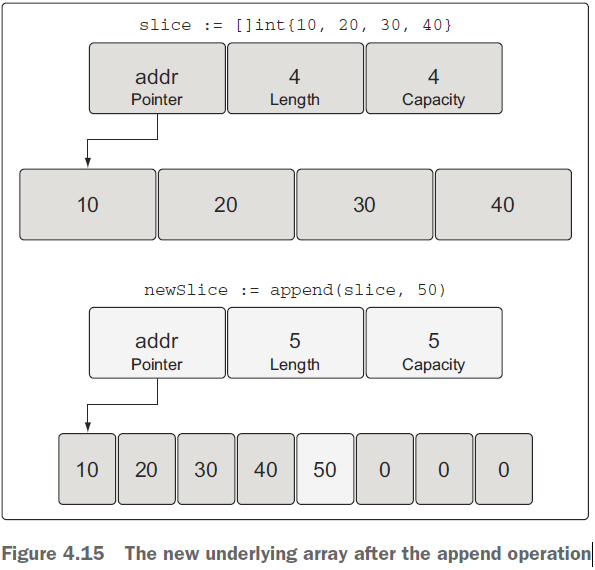
*// Append a new value to the slice.*

*// Assign the value of 50 to the new element.*

*newSlice := append(slice, 50)*

After this append operation, newSlice is given its own underlying array, and the capacity of the array is doubled from its original size (see figure).

The append operation is clever when growing the capacity of the underlying array. Capacity is always doubled when the existing capacity of the slice is under 1,000 elements. Once the number of elements goes over 1,000, the capacity is grown by a factor of 1.25, or 25%. This growth algorithm may change in the language over time.



* + There’s a third index option we haven’t mentioned yet that you can use when you’re slicing. This third index gives you control over the capacity of the new slice. The purpose is not to increase capacity, but to restrict the capacity. As you’ll see, being able to restrict the capacity of a new slice provides a level of protection to the underlying array and gives you more control over append operations

*source := []string{"Apple", "Orange", "Plum", "Banana", "Grape"}*

*// Slice the third element and restrict the capacity.*

*// Contains a length of 1 element and capacity of 2 elements.*

*slice := source[2:3:4]*

If you attempt to set a capacity that’s larger than the available capacity, you’ll get a runtime error.

the new slice references the Plum element and has capacity up to the Banana element.

We can apply the same formula that we defined before to calculate the new slice’s length and capacity

*For slice[i:j:k]*

*Length: j - i*

*Capacity: k – i*

* + The built-in function append is also a variadic function. This means you can pass multiple values to be appended in a single slice call. If you use the ... operator, you can append all the elements of one slice into another.

*s1 := []int{1, 2}*

*s2 := []int{3, 4}*

*// Append the two slices together and display the results.*

*fmt.Printf("%v\n", append(s1, s2...))*

* + Iterating over slices : Since a slice is a collection, you can iterate over the elements. Go has a special keyword called range that you use in conjunction with the keyword for to iterate over slices

*slice := []int{10, 20, 30, 40}*

*// Iterate over each element and display each value.*

*for index, value := range slice {*

*fmt.Printf("Index: %d Value: %d\n", index, value)*

*}*

It’s important to know that range is making a copy of the value, not returning a reference. If you use the address of the value variable as a pointer to each element, you’ll be making a mistake.

* + If you don’t need the index value, you can use the underscore character to discard the value.

*for \_, value := range slice {*

*fmt.Printf("Value: %d\n", value)*

*}*

* + The keyword range will always start iterating over a slice from the beginning. If you need more control iterating over a slice, you can always use a traditional for loop.

*slice := []int{10, 20, 30, 40}*

*// Iterate over each element starting at element 3.*

*for index := 2; index < len(slice); index++ {*

*fmt.Printf("Index: %d Value: %d\n", index, slice[index])*

*}*

* + There are two special built-in functions called len and cap that work with arrays, slices, and channels. For slices, the len function returns the length of the slice, and the cap function returns the capacity
  + **Multidimensional slices -** read topic in book
  + **Passing slice between functions** : Passing a slice between two functions requires nothing more than passing the slice by value. Since the size of a slice is small, it’s cheap to copy and pass between functions.

*slice := make([]int, 1e6)*

*slice = foo(slice)*

*func foo(slice []int) []int {*

*...*

*return slice*

*}*

On a 64-bit architecture, a slice requires 24 bytes of memory. The pointer field requires 8 bytes, and the length and capacity fields require 8 bytes respectively. Since the data associated with a slice is contained in the underlying array, there are no problems passing a copy of a slice to any function. Only the slice is being copied, not the underlying array.

* **Map internals and fundamentals :** 
  + A map is a data structure that provides you with an unordered collection of key/value pairs
  + The strength of a map is its ability to retrieve data quickly based on the key. A key works like an index, pointing to the value you associate with that key
  + Maps are collections, and you can iterate over them just like you do with arrays and slices. But maps are unordered collections, and there’s no way to predict the order in which the key/value pairs will be returned. Even if you store your key/value pairs in the same order, every iteration over a map could return a different order. This is because a map is implemented using a hash table. for internal architecture of Go's hashtable for implementing Maps, see pg 103 of 'Go in Action'.
  + **Creating and initializing :**

There are several ways you can create and initialize maps in Go. You can use the builtin function make, or you can use a map literal.

*// Create a map with a key of type string and a value of type int.*

*dict := make(map[string]int)*

*// Create a map with a key and value of type string.*

*// Initialize the map with 2 key/value pairs.*

*dict := map[string]string{"Red": "#da1337", "Orange": "#e95a22"}*

The map key can be a value from any built-in or struct type as long as the value can be used in an expression with the == operator.

* + Slices, functions, and struct types that contain slices can’t be used as map keys. This will produce a compiler error.

*// Create a map using a slice of strings as the key.*

*dict := map[[]string]int{}*

*Compiler Exception:*

*invalid map key type []string*

There’s nothing stopping you from using a slice as a map value. This can come in handy when you need a single map key to be associated with a collection of data

*// Create a map using a slice of strings as the value.*

*dict := map[int][]string{}*

* + **Working with maps** :
  + Assigning a key/value pair to a map is performed by specifying a key of the proper type and assigning a value to that key.

*// Create an empty map to store colors and their color codes.*

*colors := map[string]string{}*

*// Add the Red color code to the map.*

*colors["Red"] = "#da1337"*

* + You can create a nil map by declaring a map without any initialization. A nil map can’t be used to store key/value pairs. Trying will produce a runtime error.

*var colors map[string]string*

*// Add the Red color code to the map.*

*colors["Red"] = "#da1337"*

*// Runtime Error: panic: runtime error: assignment to entry in nil map*

* + Checking existence of key :

*// Retrieve the value for the key "Blue".*

*value, exists := colors["Blue"]*

*// Did this key exist?*

*if exists {*

*fmt.Println(value)*

*}*

The other option is to just return the value and test for the zero value to determine if the key exists. This will only work if the zero value is not a valid value for the map

*// Retrieve the value for the key "Blue".*

*value := colors["Blue"]*

*// Did this key exist?*

*if value != "" {*

*fmt.Println(value)*

*}*

* + A
  + When you index a map in Go, it will always return a value, even when the key doesn’t exist. In this case, the zero value for the value’s type is returned
  + **Iterating over map** :

*// Create a map of colors and color hex codes.*

*colors := map[string]string{*

*"AliceBlue": "#f0f8ff",*

*"Coral": "#ff7F50",*

*"DarkGray": "#a9a9a9",*

*"ForestGreen": "#228b22",*

*}*

*// Display all the colors in the map.*

*for key, value := range colors {*

*fmt.Printf("Key: %s Value: %s\n", key, value)*

*}*

* + If you want to remove a key/value pair from the map, you use the built-in function delete.

*delete(colors, "Coral")*

* + **Passing maps between functions** :
  + Passing a map between two functions doesn’t make a copy of the map. In fact, you can pass a map to a function and make changes to the map, and the changes will be reflected by all references to the map

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Go’s type system*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **User-defined types** :
  + Go allows you the ability to declare your own types. When you declare a new type, the declaration is constructed to provide the compiler with size and representation information, similar to how the built-in types work
  + There are two ways to declare a userdefined type in Go. The most common way is to use the keyword struct, which allows you to create a composite type. Each field in a struct is declared with a known type, which could be a built-in type or another userdefined type.

*// user defines a user in the program.*

*type user struct {*

*name string*

*email string*

*ext int*

*privileged bool*

*}*

*// declaring variables of type user*

*var bill user*

* + The value that the variable represents is always initialized. The value can be initialized with a specific value or it can be initialized to its zero value, which is the default value for that variable’s type. For numeric types, the zero value would be 0; for strings it would be empty; and for Booleans it would be false. In the case of a struct, the zero value would apply to all the different fields in the struct.
  + If the variable will be initialized to something other than its zero value, then use the short variable declaration operator with a struct literal

*// Declare a variable of type user and initialize all the fields.*

*lisa := user{*

*name: "Lisa",*

*email: "lisa@email.com",*

*ext: 123,*

*privileged: true,*

*}*

The struct literal can take on two forms for a struct type. Above example shows 1st form. order doesn't matter in above example. the other way can be follows

*// Declare a variable of type user.*

*lisa := user{"Lisa", "lisa@email.com", 123, true}*

The order of the values does matter in this case and needs to match the order of the fields in the struct declaration.

* + When declaring a struct type, you’re not limited to just the built-in types. You can also declare fields using other user-defined types

*type admin struct {*

*person user*

*level string*

*}*

*// Declare a variable of type admin.*

*fred := admin{*

*person: user{*

*name: "Lisa",*

*email: "lisa@email.com",*

*ext: 123,*

*privileged: true,*

*},*

*level: "super",*

*}*

* + - A second way to declare a user-defined type is by taking an existing type and using it as the type specification for the new type. These types are great when you need a new type that can be represented by an existing type. The standard library uses this type declaration to create high-level functionality from the built-in types. In following code snippets, Duration is a type that represents the duration of time down to the nanosecond. The type takes its representation from the built-in type int64. In the declaration of Duration, we say that int64 is the base type of Duration. Even though int64 is acting at the base type, it doesn’t mean Go considered them to be the same. Duration and int64 are two distinct and different types

*package main*

*type Duration int64*

*func main() {*

*var dur Duration*

*dur = int64(1000)*

*}*

a variable named dur of type Duration is declared and set to its zero value, then next line produces the following compiler error when the program is built

*prog.go:7: cannot use int64(1000) (type int64) as type Duration in assignment*

The compiler is clear as to what the problem is. Values of type int64 can’t be used as values of type Duration. In other words, even though type int64 is the base type for Duration, Duration is still its own unique type. Values of two different types can’t be assigned to each other, even if they’re compatible. The compiler doesn’t implicitly convert values of different types.

* **Methods :**
  + Take look at the example <https://github.com/goinaction/code/blob/master/chapter5/listing11/listing11.go>
  + Methods provide a way to add behavior to user-defined types. Methods are really functions that contain an extra parameter that’s declared between the keyword func and the function name. refer the above code from link.
  + Lines 16 and 23 of code show two different methods. The parameter between the keyword func and the function name is called a ***receiver*** and binds the function to the specified type. When a function has a receiver, that function is called a ***method***.
  + the program declares a struct type named user and then declares a method named notify.
  + There are two types of receivers in Go: ***value receivers*** and ***pointer receivers.***
  + the notify method is declared with a value receiver. The receiver for notify is declared as a value of type user. When you declare a method using a value receiver, the method will always be operating against a copy of the value used to make the method call.
  + You can also call methods that are declared with a value receiver using a pointer type. To support the method call, Go adjusts the pointer value to comply with the method’s receiver. *(\*lisa).notify()*
  + You can also declare methods with pointer receivers. declaration of the changeEmail method, which is declared with
  + a pointer receiver. This time, the receiver is not a value of type user but a pointer of type user. When you call a method declared with a pointer receiver, the value used to make the call is shared with the method.
  + You can also call methods that are declared with a pointer receiver using a value type. Once again, Go adjusts the value to comply with the method’s receiver to support the call. *(&bill).notify()*
* **The nature of types**
  + **Built-in types**
  + Built-in types are the set of types that are provided by the language. We know them as the set of numeric, string, and Boolean types. These types have a primitive nature to them. Because of this, when adding or removing something from a value of one of these types, a new value should be created. Based on this, when passing values of these types to functions and methods, a copy of the value should be passed.
  + Strings, just like integers, floats, and Booleans, are primitive data values and should be copied when passed in and out of functions or methods
  + **Reference types**
  + Reference types in Go are the set of slice, map, channel, interface, and function types. When you declare a variable from one of these types, the value that’s created is called a ***header*** value. Technically, a string is also a reference type value. All the different header values from the different reference types contain a pointer to an underlying data structure. Each reference type also contains a set of unique fields that are used to manage the underlying data structure. You never share reference type values because the header value is designed to be copied. The header value contains a pointer; therefore, you can pass a copy of any reference type value and share the underlying data structure intrinsically.
  + Code snippet shows a type called IP which is declared as a slice of bytes. Declaring a type like this is useful when you want to declare behavior around a built-in or reference type.

*type IP []byte*

*func (ip IP) MarshalText() ([]byte, error) {*

*if len(ip) == 0 {*

*return []byte(""), nil*

*}*

*if len(ip) != IPv4len && len(ip) != IPv6len {*

*return nil, errors.New("invalid IP address")*

*}*

*return []byte(ip.String()), nil*

*}*

* + **Struct types**
  + Struct type basically
  + Struct type can behave either primitive nature OR non-primitive depends upon how it is used by developers.
  + **Interfaces**
  + Read in detail on 109 from go in action.