Unit-1 : Basic

## Basic Syntax

A Go file consists of the following parts:

- Package declaration
- Import packages
- Functions
- Statements and expressions

Example

- ```
  package main
  import ("fmt")

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

Example explained

- **Line 1:** In Go, every program is part of a package. We define this using the package keyword. In this example, the program belongs to the main package.
- **Line 2:** import ("fmt") lets us import files included in the fmt package.
- **Line 3:** A blank line. Go ignores white space. Having white spaces in code makes it more readable.
- **Line 4:** func main() {} is a function. Any code inside its curly brackets {} will be executed.
- **Line 5:** fmt.Println() is a function made available from the fmt package. It is used to output/print text. In our example it will output "Hello World!".

**Note: In Go, any executable code belongs to the main package.**

**** comment is the same as the java and c**

Terminal Command
  ```

go run hello.go

## 1.2 Variable and Declaration

Go Variable Types

In Go, there are different **types** of variables, for example:

- int- stores integers (whole numbers), such as 123 or -123
- float32- stores floating point numbers, with decimals, such as 19.99 or -19.99
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool- stores values with two states: true or false

Declaration

1. by using the var keyword
   var *variablename type = value*

2. With the := sign:

   *variablename := value*

3.Variable Declaration With Initial Value

```
package main
import ("fmt")

func main() {
  var student1 string = "John" //type is string
  var student2 = "Jane" //type is inferred
  x := 2 //type is inferred

  fmt.Println(student1)
  fmt.Println(student2)
  fmt.Println(x)
}
```

** **inferred** from the value (means that the compiler decides the type of the variable, based on the value).

**Note:** It is not possible to declare a variable using :=, without assigning a value to it.

4.Variable Declaration With Initial Value

```
package main
import ("fmt")
```

```
func main() {
  var student1 string = "John" //type is string
  var student2 = "Jane" //type is inferred
  x := 2 //type is inferred

  fmt.Println(student1)
  fmt.Println(student2)
  fmt.Println(x)
}
```

5.Variable Declaration Without Initial Value

```
package main
import ("fmt")

func main() {
  var a string
  var b int
  var c bool

  fmt.Println(a)
  fmt.Println(b)
  fmt.Println(c)
}
```

6. Value Assignment After Declaration

```
package main
import ("fmt")

func main() {
  var student1 string
  student1 = "John"
  fmt.Println(student1)
}
```

**Difference Between var and :=**

| var | := |
|---|---|
| **Can be used inside and outside of functions** | **Can only be used inside functions** |

| Variable declaration and value assignment can be done separately | Variable declaration and value assignmen in the same line) |
| --- | --- |

Unit:2 Conditional Sentences

Example of the conditional Statement

```
package main
import "fmt"
func main() {
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }
    if 8%2 == 0 || 7%2 == 0 {
        fmt.Println("either 8 or 7 are even")
    }
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

2. if-else statement
   it is similar to the java and c

also the loop an switch case is the same as the java and c

**Function**

**Create a Function**

- Use the func keyword.
- Specify a name for the function, followed by parentheses ().
- Finally, add code that defines what the function should do, inside curly braces {}.

Example

Syntax

```
func FunctionName() {
  // code to be executed
}
```

**Call a Function**

Functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

In the example below, we create a function named "myMessage()". The opening curly brace ( { ) indicates the beginning of the function code, and the closing curly brace ( } ) indicates the end of the function. The function outputs "I just got executed!". To call the function, just write its name followed by two parentheses ():

Example

```
package main
import ("fmt")

func myMessage() {
  fmt.Println("I just got executed!")
}

func main() {
  myMessage() // call the function
}
```

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
- If the function name consists of multiple words, techniques introduced for multi-word variable naming can be used

**Tip:** Give the function a name that reflects what the function does!

**Function With Parameter Example**

The following example has a function with one parameter (fname) of type string. When the familyName() function is called, we also pass along a name (e.g. Liam), and the name is used inside the function, which outputs several different first names, but an equal last name:

```
package main
import ("fmt")

func familyName(fname string) {
  fmt.Println("Hello", fname, "Refsnes")
}

func main() {
  familyName("Liam")
  familyName("Jenny")
  familyName("Anja")
}
```

**Return Values**

If you want the function to return a value, you need to define the data type of the return value (such as int, string, etc), and also use the return keyword inside the function.

Syntax

```
func FunctionName(param1 type, param2 type) type {
  // code to be executed
  return output
}
```

**Function Return Example**

```go
package main
import ("fmt")

func myFunction(x int, y int) int {
  return x + y
}

func main() {
  fmt.Println(myFunction(1, 2))
}
```

# Named Return Values

Example

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):

```
package main
import ("fmt")

func myFunction(x int, y int) (result int) {
 result = x + y
 return
}

func main() {
 fmt.Println(myFunction(1, 2))
}
```

**Multiple Return Values**

```
package main
import ("fmt")

func myFunction(x int, y string) (result int, txt1 string) {
 result = x + x
 txt1 = y + " World!"
 return
}
```

```
func main() {
  fmt.Println(myFunction(5, "Hello"))
}
```

**Type-casting in Golang**

By Sutirtha Chakraborty / January 16, 2020

Sometimes we need to convert one data-type to another. In this post, we will go into detail about type-casting in the Go programming language.

**Types in Go**

There are many data-types in Go. The numbers, floating points, boolean and string.

- Number: int, int32, int64, uint32, uint64 etc
- Floating points: float32, float64, complex64, complex128
- Boolean: bool
- string: string

**What is type-casting?**

Type-casting means converting one type to another. Any type can be converted to another type but that does not guarantees that the value will remain intact or in fact preserved at all as we will see in this post.

Backward Skip 10sPlay VideoForward Skip 10s

**Why is it important?**

Type-casting is an important concept in general programming. It converts one type to another and whenever we need some other types for the expression type casting helps.

**Type-casting syntax**

The syntax for general type casting is pretty simple. just use that other type name as a function to convert that value.

**v := typeName(otherTypeValue)**

e.g. **i := int(32.987)** // casting to integer

**Implicit type conversions**

Unlike other languages, Go **doesn't support** implicit type conversion. Although when dividing numbers, implicit conversions happen depending on the scenario. So we need to be very careful about what type to use where.

**Basic Type-casting**

Anywhere we need to change the type of the variable, the typecasting is needed. Sometimes typecasting may not be direct as discussed before. Here is an example of direct type conversion.

```
package main
import (
   "fmt"
)
func main() {
   var a int = 42
  f := float64(a)
   fmt.Println(f)      // 42
11}
```

**Conversions between string and int**

There is a package called **strconv** which can be used to convert between string and int values. Below is the code on how to do that.

```
1  package main
2
3  import (
4     "fmt"
5     "strconv"
6  )
7
8  func main() {
9     var s string = "42"
```

```
10    v, _ := strconv.Atoi(s)      // convert string to int
11
12    fmt.Println(v)    // 42
13
14    var i int = 42
15    str := strconv.Itoa(i)       // convert int to string
16
17    fmt.Println(str) // 42
18}
```

**Conversions between int and float**

When conversion between int and floats happen the numbers may lose precision. Here are type conversions between int and floats.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      f := 12.34567
9      i := int(f)  // loses precision
10     fmt.Println(i)     // 12
11
12     ii := 34
13     ff := float64(ii)
14
15     fmt.Println(ff)    // 34
```

```
16}
```

As you can see in the code above, the conversion of a float to an int always **loses precision**.

**Strings and bytes conversion**

Strings are slices of bytes. So both can be converted to each other with minimal effort. Here is the simplest way to do that.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      var s string = "Hello World"
9      var b []byte = []byte(s)     // convert ty bytes
10
11     fmt.Println(b)  // [72 101 108 108 111 32 87 111 114 108 100]
12
13     ss := string(b)             // convert to string
14
15     fmt.Println(ss)     // Hello World
16}
```

**Type conversion during division**

During division, types are converted implicitly. Here are some examples.

```
1  package main
2
3  import (
4      "fmt"
```

```
5  )
6
7  func main() {
8      a := 6/3        // both are int, a is int
9      f := 6.3/3      // float and int, f is float
10
11     fmt.Println(a, f)     // 2 2.1
12}
```

**Drawbacks of type-casting in Golang**

Type conversion is a great way to change the original data type. But unfortunately, in some cases, it **loses precision**. It should be done sparingly. The needs of type conversion lie in the fact Go **doesn't support implicit type conversions**. So in many cases, it should be done separately.

**Short Variable Declaration (:= Operator) in Golang**

By Sutirtha Chakraborty / January 20, 2020

The usage of variables is essential in any programming language. In this post, we will use the shorthand syntax of declaring variables in Go.

**Variable declarations in Go**

There are multiple ways of declaring variables. The shorthand syntax allows us to declare and initialize the variable both at the same time. This not only reduces code but creates concise syntax. First, let's see the regular syntax of declaring variables.

**var variableName variableType**

PlayNext

Mute

Current Time 0:00

/

Duration 1:45

Loaded: 3.80%

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

Now, we can assign a value to it of type **variableType**.

The shorthand syntax merges two steps into one. Below is the syntax.

**variableName := initialValueOfTheVariable**

The operator above (**:=**) is called the short declaration operator.

**Type inferencing**

After declaring a variable with the short declaration syntax the variable gets a type. So, if it is reassigned to another type then it throws an error.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      a := 42
9      fmt.Println(a)
10     a = "Hi" // cannot use "Hi" (type string) as type int in assignment
11 }
```

The variable type gets inferred by the Go compiler after the declaration happens so it cannot be reassigned with a new type.

**Declaring multiple variables**

Multiple variables can be declared using the shorthand syntax of variable declaration. To declare multiple variables, the order should be maintained on both sides of the operator. The declaration below shows that.

**var1, var2, var3, ... := value1, value2, value3, ...**

**Declaring functions**

Functions can be declared with the same short syntax as well. Go has great support for functions and allows support anonymous functions in Go code. The code below shows how to declare functions using the shorthand syntax.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      // declare the function
9      f := func() {
10         fmt.Println("Inside a function")
11     }
12
13     // call using the name
14     f()   // Inside a function
15}
```

**Advantages of short declaration**

The short declaration of variables has some advantages. It enables the code to be short and concise. Also, it stops assigning wrong types to a variable.

A function can return multiple values in Go. So, in that case, multiple assignments can be made easily. Error handling is a use case of this particular type.

**Drawbacks of short declaration**

The short declaration has some drawbacks also. It cannot be used in the Global scope. It must be inside a function. To declare outside the function scope **var, const, func** must be used accordingly.

## Array

**Go Arrays**

Arrays are used to store multiple values of the same type in a single variable, instead of declaring separate variables for each value.

Declare an Array

In Go, there are two ways to declare an array:

1. With the var keyword:

Syntax

var *array_name* = [*length*]*datatype*{*values*} // here length is defined

or

var *array_name* = [...]*datatype*{*values*} // here length is inferred

2. With the := sign:

Syntax

*array_name* := [*length*]*datatype*{*values*} // here length is defined

or

*array_name* := [...]*datatype*{*values*} // here length is inferred

**Note:** The *length* specifies the number of elements to store in the array. In Go, arrays have a fixed length. The length of the array is either defined by a number or is inferred (means that the compiler decides the length of the array, based on the number of *values*).

Array Examples

Example

This example declares two arrays (arr1 and arr2) with defined lengths:

```
package main
import ("fmt")

func main() {
  var arr1 = [3]int{1,2,3}
```

```
  arr2 := [5]int{4,5,6,7,8}

  fmt.Println(arr1)
  fmt.Println(arr2)
}
```

Result:

```
[1 2 3]
[4 5 6 7 8]
```

Example

This example declares two arrays (arr1 and arr2) with inferred lengths:

```
package main
import ("fmt")

func main() {
  var arr1 = [...]int{1,2,3}
  arr2 := [...]int{4,5,6,7,8}

  fmt.Println(arr1)
  fmt.Println(arr2)
}
```

Result:

```
[1 2 3]
[4 5 6 7 8]
```

Example

This example declares an array of strings:

```
package main
import ("fmt")

func main() {
  var cars = [4]string{"Volvo", "BMW", "Ford", "Mazda"}
  fmt.Print(cars)
}
```

Result:

[Volvo BMW Ford Mazda]

Access Elements of an Array

You can access a specific array element by referring to the index number.

In Go, array indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

Example

This example shows how to access the first and third elements in the prices array:

```
package main
import ("fmt")

func main() {
  prices := [3]int{10,20,30}

  fmt.Println(prices[0])
  fmt.Println(prices[2])
}
```

Result:

10
30

Change Elements of an Array

You can also change the value of a specific array element by referring to the index number.

Example

This example shows how to change the value of the third element in the prices array:

```
package main
import ("fmt")

func main() {
  prices := [3]int{10,20,30}

  prices[2] = 50
```

```
  fmt.Println(prices)
}
```

Result:

```
[10 20 50]
```

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.

**Tip:** The default value for int is 0, and the default value for string is "".

Example

```
package main
import ("fmt")

func main() {
  arr1 := [5]int{} //not initialized
  arr2 := [5]int{1,2} //partially initialized
  arr3 := [5]int{1,2,3,4,5} //fully initialized

  fmt.Println(arr1)
  fmt.Println(arr2)
  fmt.Println(arr3)
}
```

Result:

```
[0 0 0 0 0]
[1 2 0 0 0]
[1 2 3 4 5
```

Initialize Only Specific Elements

It is possible to initialize only specific elements in an array.

Example

This example initializes only the second and third elements of the array:

```
package main
import ("fmt")

func main() {
  arr1 := [5]int{1:10,2:40}

  fmt.Println(arr1)
}
```

Result:

[0 10 40 0 0]

Example Explained

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

The len() function is used to find the length of an array:

Example

```
package main
import ("fmt")

func main() {
  arr1 := [4]string{"Volvo", "BMW", "Ford", "Mazda"}
  arr2 := [...]int{1,2,3,4,5,6}

  fmt.Println(len(arr1))
  fmt.Println(len(arr2))
}
```

Result:

4
6

**Slice**

Go Slices

Slices are similar to arrays, but are more powerful and flexible.

Like arrays, slices are also used to store multiple values of the same type in a single variable.

However, unlike arrays, the length of a slice can grow and shrink as you see fit.

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*}

Syntax

*slice_name* := []*datatype*{*values*}

A common way of declaring a slice is like this:

myslice := []int{}

The code above declares an empty slice of 0 length and 0 capacity.

To initialize the slice during declaration, use this:

myslice := []int{1,2,3}

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:

- len() function - returns the length of the slice (the number of elements in the slice)
- cap() function - returns the capacity of the slice (the number of elements the slice can grow or shrink to)

Example

This example shows how to create slices using the []*datatype*{*values*} format:

```
package main
import ("fmt")
```

```
func main() {
  myslice1 := []int{}
  fmt.Println(len(myslice1))
  fmt.Println(cap(myslice1))
  fmt.Println(myslice1)

  myslice2 := []string{"Go", "Slices", "Are", "Powerful"}
  fmt.Println(len(myslice2))
  fmt.Println(cap(myslice2))
  fmt.Println(myslice2)
}
```

Result:

```
0
0
[]
4
4
[Go Slices Are Powerful]
```

In the example above, we see that in the first slice (myslice1), the actual elements are not specified, so both the length and capacity of the slice will be zero. In the second slice (myslice2), the elements are specified, and both length and capacity is equal to the number of actual elements specified.

## Create a Slice From an Array

You can create a slice by slicing an array:

### Syntax

```
var myarray = [length]datatype{values} // An array
myslice := myarray[start:end] // A slice made from the array
```

### Example

This example shows how to create a slice from an array:

```
package main
import ("fmt")
```

```
func main() {
  arr1 := [6]int{10, 11, 12, 13, 14,15}
  myslice := arr1[2:4]

  fmt.Printf("myslice = %v\n", myslice)
  fmt.Printf("length = %d\n", len(myslice))
  fmt.Printf("capacity = %d\n", cap(myslice))
}
```

Result:

```
myslice = [12 13]
length = 2
capacity = 4
```

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 (remember that array indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.). The slice can grow to the end of the array. This means that the capacity of the slice is 4.

If myslice started from element 0, the slice capacity would be 6.

Create a Slice With The make() Function

The make() function can also be used to create a slice.

Syntax

*slice_name* := make([]*type*, *length*, *capacity*)

**Note:** If the *capacity* parameter is not defined, it will be equal to *length*.

Example

This example shows how to create slices using the make() function:

```
package main
import ("fmt")

func main() {
  myslice1 := make([]int, 5, 10)
```

```
  fmt.Printf("myslice1 = %v\n", myslice1)
  fmt.Printf("length = %d\n", len(myslice1))
  fmt.Printf("capacity = %d\n", cap(myslice1))

  // with omitted capacity
  myslice2 := make([]int, 5)
  fmt.Printf("myslice2 = %v\n", myslice2)
  fmt.Printf("length = %d\n", len(myslice2))
  fmt.Printf("capacity = %d\n", cap(myslice2))
}
```

Result:

```
myslice1 = [0 0 0 0 0]
length = 5
capacity = 10
myslice2 = [0 0 0 0 0]
length = 5
capacity = 5
```

Access Elements of a Slice

You can access a specific slice element by referring to the index number.

In Go, indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

Example

This example shows how to access the first and third elements in the prices slice:

```
package main
import ("fmt")

func main() {
  prices := []int{10,20,30}

  fmt.Println(prices[0])
  fmt.Println(prices[2])
}
```

Result:

10
30

Change Elements of a Slice

You can also change a specific slice element by referring to the index number.

Example

This example shows how to change the third element in the prices slice:

```
package main
import ("fmt")

func main() {
  prices := []int{10,20,30}
  prices[2] = 50
  fmt.Println(prices[0])
  fmt.Println(prices[2])
}
```

Result:

10
50

Append Elements To a Slice

You can append elements to the end of a slice using the append()function:

Syntax

*slice_name* = append(*slice_name*, *element1*, *element2*, ...)

Example

This example shows how to append elements to the end of a slice:

```
package main
import ("fmt")

func main() {
  myslice1 := []int{1, 2, 3, 4, 5, 6}
  fmt.Printf("myslice1 = %v\n", myslice1)
```

```
 fmt.Printf("length = %d\n", len(myslice1))
 fmt.Printf("capacity = %d\n", cap(myslice1))

 myslice1 = append(myslice1, 20, 21)
 fmt.Printf("myslice1 = %v\n", myslice1)
 fmt.Printf("length = %d\n", len(myslice1))
 fmt.Printf("capacity = %d\n", cap(myslice1))
}
```

Result:

```
myslice1 = [1 2 3 4 5 6]
length = 6
capacity = 6
myslice1 = [1 2 3 4 5 6 20 21]
length = 8
capacity = 12
```

Append One Slice To Another Slice

To append all the elements of one slice to another slice, use the append()function:

Syntax

*slice3* = append(*slice1*, *slice2*...)

**Note:** The **'...'** after *slice2* is **necessary** when appending the elements of one slice to another.

Example

This example shows how to append one slice to another slice:

```
package main
import ("fmt")

func main() {
 myslice1 := []int{1,2,3}
 myslice2 := []int{4,5,6}
 myslice3 := append(myslice1, myslice2...)

 fmt.Printf("myslice3=%v\n", myslice3)
 fmt.Printf("length=%d\n", len(myslice3))
 fmt.Printf("capacity=%d\n", cap(myslice3))
}
```

Result:

```
myslice3=[1 2 3 4 5 6]
length=6
capacity=6
```

Change The Length of a Slice

Unlike arrays, it is possible to change the length of a slice.

Example

This example shows how to change the length of a slice:

```
package main
import ("fmt")

func main() {
  arr1 := [6]int{9, 10, 11, 12, 13, 14} // An array
  myslice1 := arr1[1:5] // Slice array
  fmt.Printf("myslice1 = %v\n", myslice1)
  fmt.Printf("length = %d\n", len(myslice1))
  fmt.Printf("capacity = %d\n", cap(myslice1))

  myslice1 = arr1[1:3] // Change length by re-slicing the array
  fmt.Printf("myslice1 = %v\n", myslice1)
  fmt.Printf("length = %d\n", len(myslice1))
  fmt.Printf("capacity = %d\n", cap(myslice1))

  myslice1 = append(myslice1, 20, 21, 22, 23) // Change length by appending items
  fmt.Printf("myslice1 = %v\n", myslice1)
  fmt.Printf("length = %d\n", len(myslice1))
  fmt.Printf("capacity = %d\n", cap(myslice1))
}
```

Result:

```
myslice1 = [10 11 12 13]
length = 4
capacity = 5
myslice1 = [10 11]
length = 2
```

```
capacity = 5
myslice1 = [10 11 20 21 22 23]
length = 6
capacity = 10
```

## Memory Efficiency

When using slices, Go loads all the underlying elements into the memory.

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.

Syntax

copy(*dest*, *src*)

The copy() function takes in two slices *dest* and *src*, and copies data from *src* to *dest*. It returns the number of elements copied.

Example

This example shows how to use the copy() function:

```
package main
import ("fmt")

func main() {
  numbers := []int{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
  // Original slice
  fmt.Printf("numbers = %v\n", numbers)
  fmt.Printf("length = %d\n", len(numbers))
  fmt.Printf("capacity = %d\n", cap(numbers))

  // Create copy with only needed numbers
  neededNumbers := numbers[:len(numbers)-10]
  numbersCopy := make([]int, len(neededNumbers))
  copy(numbersCopy, neededNumbers)

  fmt.Printf("numbersCopy = %v\n", numbersCopy)
  fmt.Printf("length = %d\n", len(numbersCopy))
```

```
  fmt.Printf("capacity = %d\n", cap(numbersCopy))
}
```

Result:

```
// Original slice
numbers = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
length = 15
capacity = 15
// New slice
numbersCopy = [1 2 3 4 5]
length = 5
capacity = 5
```

The capacity of the new slice is now less than the capacity of the original slice because the new underlying array is smaller.

## Go Maps

### Go 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 :=

Syntax

var *a* = map[KeyType]ValueType{*key1*:*value1*, *key2*:*value2*,...}
*b* := map[KeyType]ValueType{*key1*:*value1*, *key2*:*value2*,...}

Example

This example shows how to create maps in Go. Notice the order in the code and in the output

```
package main
import ("fmt")

func main() {
  var a = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}
  b := map[string]int{"Oslo": 1, "Bergen": 2, "Trondheim": 3, "Stavanger": 4}

  fmt.Printf("a\t%v\n", a)
  fmt.Printf("b\t%v\n", b)
}
```

Result:

```
a   map[brand:Ford model:Mustang year:1964]
b   map[Bergen:2 Oslo:1 Stavanger:4 Trondheim:3]
```

**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:

Syntax

```
var a = make(map[KeyType]ValueType)
b := make(map[KeyType]ValueType)
```

Example

This example shows how to create maps in Go using the make()function.

```
package main
import ("fmt")

func main() {
  var a = make(map[string]string) // The map is empty now
  a["brand"] = "Ford"
  a["model"] = "Mustang"
  a["year"] = "1964"
                        // a is no longer empty
  b := make(map[string]int)
```

```
  b["Oslo"] = 1
  b["Bergen"] = 2
  b["Trondheim"] = 3
  b["Stavanger"] = 4

  fmt.Printf("a\t%v\n", a)
  fmt.Printf("b\t%v\n", b)
}
```

Result:

```
a   map[brand:Ford model:Mustang year:1964]
b   map[Bergen:2 Oslo:1 Stavanger:4 Trondheim:3]
```

**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.

Syntax

var *a* map[KeyType]ValueType

**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 causes a runtime panic.

Example

This example shows the difference between declaring an empty map using with the make()function and without it.

```
package main
import ("fmt")

func main() {
  var a = make(map[string]string)
  var b map[string]string

  fmt.Println(a == nil)
  fmt.Println(b == nil)
}
```

Result:

false
true

**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:

Syntax

*value* = *map_name*[key]

Example

```
package main
import ("fmt")

func main() {
  var a = make(map[string]string)
  a["brand"] = "Ford"
  a["model"] = "Mustang"
  a["year"] = "1964"

  fmt.Printf(a["brand"])
}
```

Result:

Ford

Update and Add Map Elements

Updating or adding an elements are done by:

Syntax

*map_name*[key] = value

Example

This example shows how to update and add elements to a map.

```
package main
import ("fmt")

func main() {
  var a = make(map[string]string)
  a["brand"] = "Ford"
  a["model"] = "Mustang"
  a["year"] = "1964"

  fmt.Println(a)

  a["year"] = "1970" // Updating an element
  a["color"] = "red" // Adding an element
```

```
 fmt.Println(a)
}
```

Result:

```
map[brand:Ford model:Mustang year:1964]
map[brand:Ford color:red model:Mustang year:1970
```

Remove Element from Map

Removing elements is done using the delete() function.

Syntax

delete(*map_name*, key)

Example

```
package main
import ("fmt")

func main() {
 var a = make(map[string]string)
 a["brand"] = "Ford"
 a["model"] = "Mustang"
 a["year"] = "1964"

 fmt.Println(a)

 delete(a,"year")

 fmt.Println(a)
}
```

Result:

```
map[brand:Ford model:Mustang year:1964]
map[brand:Ford model:Mustang]
```

Check For Specific Elements in a Map

You can check if a certain key exists in a map using:

Syntax

*val*, *ok* :=*map_name*[key]

If you only want to check the existence of a certain key, you can use the blank identifier (_) in place of val.

Example

```
package main
import ("fmt")

func main() {
  var a = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964", "day":""}

  val1, ok1 := a["brand"] // Checking for existing key and its value
  val2, ok2 := a["color"] // Checking for non-existing key and its value
  val3, ok3 := a["day"]   // Checking for existing key and its value
  _, ok4 := a["model"]    // Only checking for existing key and not its value

  fmt.Println(val1, ok1)
  fmt.Println(val2, ok2)
  fmt.Println(val3, ok3)
  fmt.Println(ok4)
}
```

Result:

```
Ford true
 false
 true
true
```

Example Explained

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*.

Maps Are References

Maps are references to hash tables.

If two map variables refer to the same hash table, changing the content of one variable affect the content of the other.

Example

```
package main
import ("fmt")

func main() {
  var a = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}
  b := a

  fmt.Println(a)
  fmt.Println(b)

  b["year"] = "1970"
  fmt.Println("After change to b:")

  fmt.Println(a)
  fmt.Println(b)
}
```

Result:

```
map[brand:Ford model:Mustang year:1964]
map[brand:Ford model:Mustang year:1964]
After change to b:
map[brand:Ford model:Mustang year:1970]
map[brand:Ford model:Mustang year:1970
```

**Iterate Over Maps**

You can use range to iterate over maps.

Example

This example shows how to iterate over the elements in a map. Note the order of the elements in the output.

```
package main
import ("fmt")

func main() {
```

```
 a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}

 for k, v := range a {
  fmt.Printf("%v : %v, ", k, v)
 }
}
```

Result:

two : 2, three : 3, four : 4, one : 1,

**Iterate Over Maps in a Specific Order**

Maps are unordered data structures. If you need to iterate over a map in a specific order, you must have a separate data structure that specifies that order.

Example

```
package main
import ("fmt")

func main() {
 a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}

 var b []string           // defining the order
 b = append(b, "one", "two", "three", "four")

 for k, v := range a {       // loop with no order
  fmt.Printf("%v : %v, ", k, v)
 }

 fmt.Println()

 for _, element := range b {  // loop with the defined order
  fmt.Printf("%v : %v, ", element, a[element])
 }
}
```

Result:

two : 2, three : 3, four : 4, one : 1,
one : 1, two : 2, three : 3, four : 4,

Try it Yourself »

# Struct

Go Structures

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:

Syntax

```
type struct_name struct {
  member1 datatype;
  member2 datatype;
  member3 datatype;
  ...
}
```

Example

Here we declare a struct type Person with the following members: name, age, job and salary:

```
type Person struct {
  name string
  age int
  job string
  salary int
}
```

**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:

Example

```
package main
import ("fmt")

type Person struct {
  name string
  age int
  job string
  salary int
}

func main() {
  var pers1 Person
  var pers2 Person

  // Pers1 specification
  pers1.name = "Hege"
  pers1.age = 45
  pers1.job = "Teacher"
  pers1.salary = 6000

  // Pers2 specification
  pers2.name = "Cecilie"
  pers2.age = 24
  pers2.job = "Marketing"
  pers2.salary = 4500

  // Access and print Pers1 info
  fmt.Println("Name: ", pers1.name)
  fmt.Println("Age: ", pers1.age)
  fmt.Println("Job: ", pers1.job)
  fmt.Println("Salary: ", pers1.salary)

  // Access and print Pers2 info
  fmt.Println("Name: ", pers2.name)
  fmt.Println("Age: ", pers2.age)
```

```
  fmt.Println("Job: ", pers2.job)
  fmt.Println("Salary: ", pers2.salary)
}
```

Result:

```
Name: Hege
Age: 45
Job: Teacher
Salary: 6000
Name: Cecilie
Age: 24
Job: Marketing
Salary: 4500
```

Pass Struct as Function Arguments

You can also pass a structure as a function argument, like this:

Example

```
package main
import ("fmt")

type Person struct {
  name string
  age int
  job string
  salary int
}

func main() {
  var pers1 Person
  var pers2 Person

  // Pers1 specification
  pers1.name = "Hege"
  pers1.age = 45
  pers1.job = "Teacher"
  pers1.salary = 6000
```

```
  // Pers2 specification
  pers2.name = "Cecilie"
  pers2.age = 24
  pers2.job = "Marketing"
  pers2.salary = 4500

  // Print Pers1 info by calling a function
  printPerson(pers1)

  // Print Pers2 info by calling a function
  printPerson(pers2)
}

func printPerson(pers Person) {
  fmt.Println("Name: ", pers.name)
  fmt.Println("Age: ", pers.age)
  fmt.Println("Job: ", pers.job)
  fmt.Println("Salary: ", pers.salary)
}
```

Result:

```
Name: Hege
Age: 45
Job: Teacher
Salary: 6000
Name: Cecilie
Age: 24
Job: Marketing
Salary: 4500
```