Unit : 2 Variables and Primitive Data Types

- Declaration
- Initialization
- Primitive types
- Pointers
- Type conversion

Variables

- A variable is a piece of storage containing data temporarily to work with it
- The most general form to declare a variable in Golang uses the `var` keyword, an explicit type, and an assignment
  - var name type = expression

Declaration with initialization

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

Execute | Share main.go STDIN

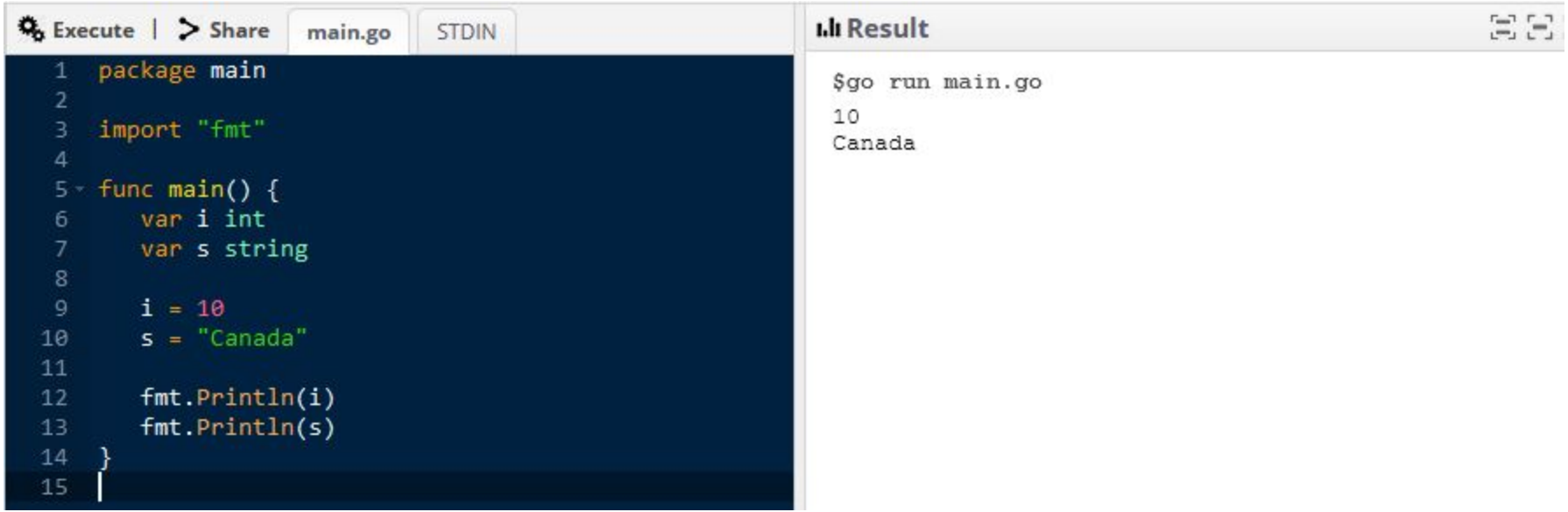
```go
package main

import "fmt"

func main() {
    var i int = 10
    var s string = "Canada"

    fmt.Println(i)
    fmt.Println(s)
}
```

Result

```
$go run main.go
10
Canada
```

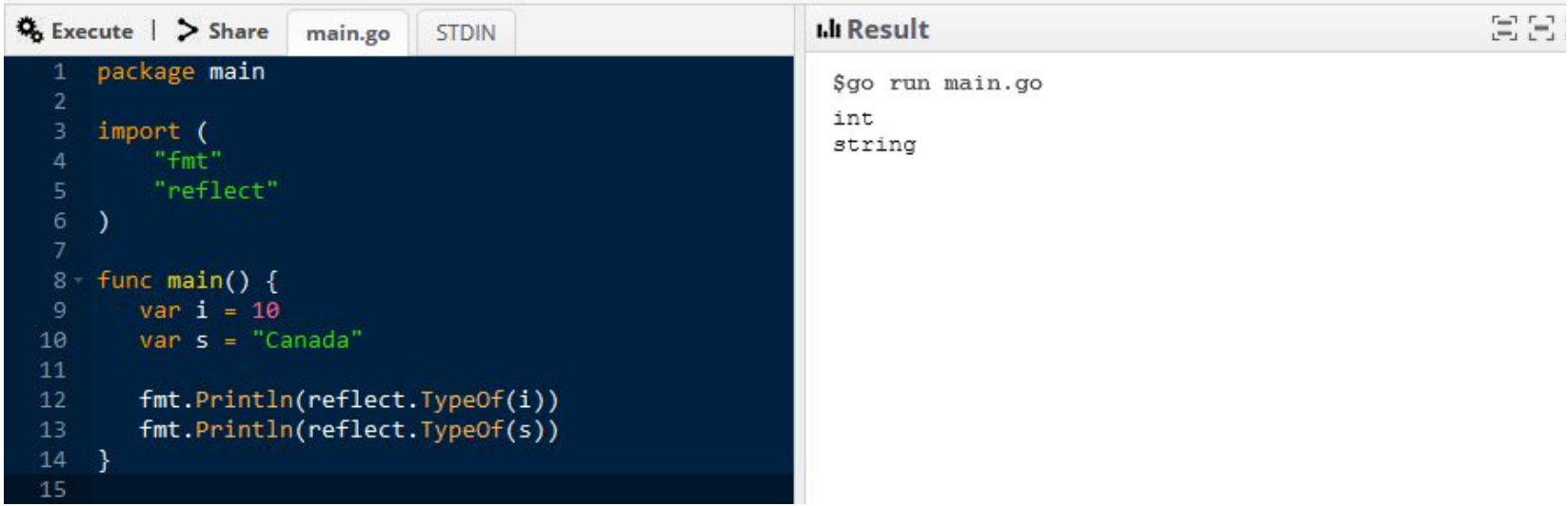
Declaration without initialization

- The keyword var is used for declaring variables followed by the desired name and the type of value the variable will hold



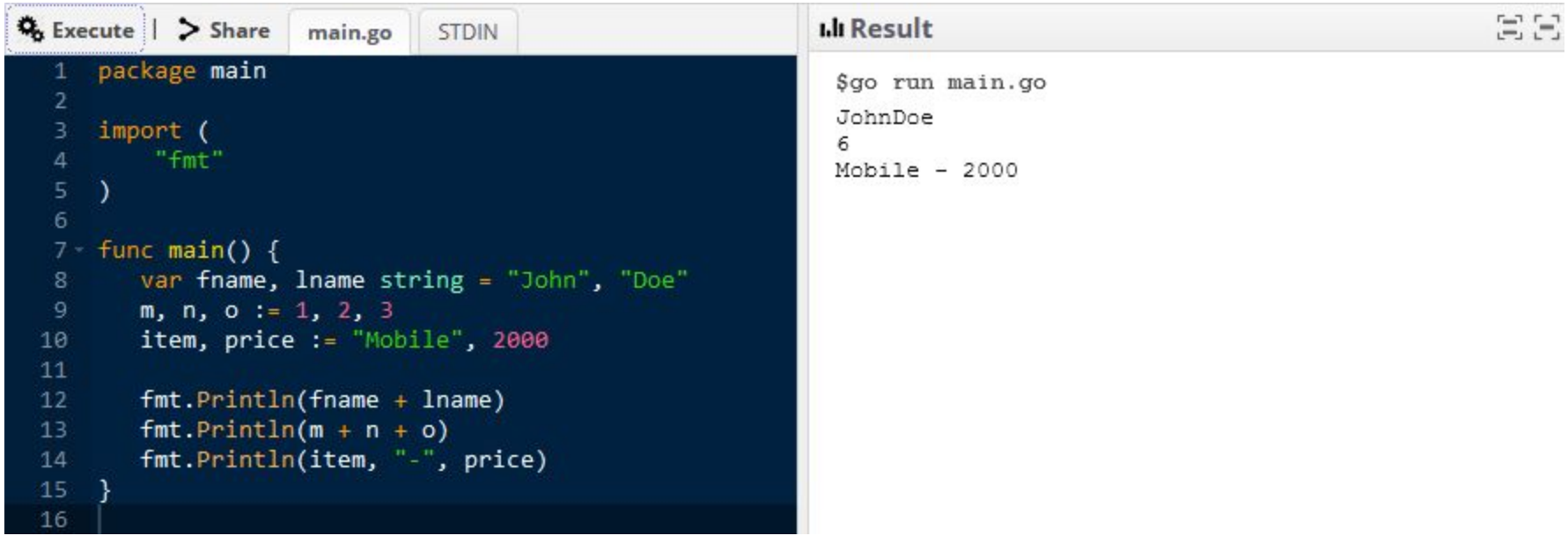
Declaration with type inference

- You can omit the variable type from the declaration, when you are assigning a value to a variable at the time of declaration
- The type of the value assigned to the variable will be used as the type of that variable



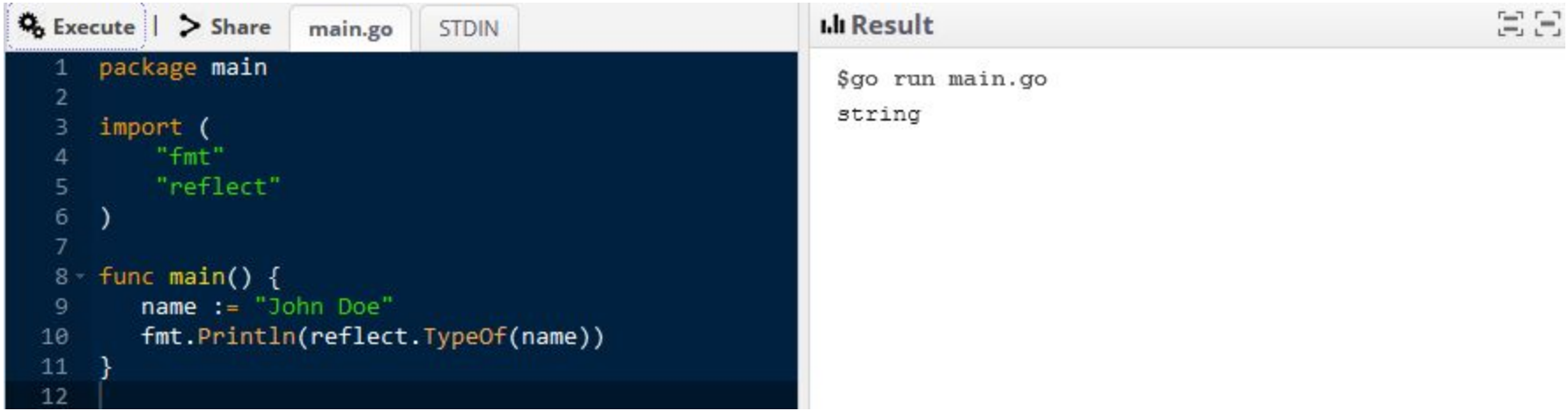
Declaration of multiple variables

- Golang allows you to assign values to multiple variables in one line



Short Variable Declaration in Golang

- The := short variable assignment operator indicates that short variable declaration is being used
- There is no need to use the var keyword or declare the variable type



Scope of Golang Variables Defined by Brace Brackets

- Golang uses lexical scoping based on code blocks to determine the scope of variables
- Inner block can access its outer block defined variables, but outer block cannot access inner block defined variables

Scope of Golang Variables Defined by Brace Brackets

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
)

var s = "Japan"

func main() {
   fmt.Println(s)
   x := true

   if x {
      y := 1
      if x != false {
         fmt.Println(s)
         fmt.Println(x)
         fmt.Println(y)
      }
   }
   fmt.Println(x)
}
```

Result

```
$go run main.go
Japan
Japan
true
1
true
```

Golang Variable Declaration Block

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
)

var (
    product = "Mobile"
    quantity = 50
    price = 50.50
    inStock = true
)

func main() {
   fmt.Println(product)
   fmt.Println(quantity)
   fmt.Println(price)
   fmt.Println(inStock)
}
```

Result

```
$go run main.go
Mobile
50
50.5
true
```

Primitive data types in Golang

- In golang, we have **int**, **float**, **byte**, **string**, **rune** & **bool**(boolean if you are coming from other programming languages).
- **Int**
- Int is one of the numeric types which represents a set of integers
- There are various kinds of integer types associated with int
- We have uint or unsigned integers, complex numbers, and int itself
- Along with them are the different sets of numbers that they contain

Primitive data types in Golang

- For example, int8 contains all the 8-bit integers from -128 to +127,
  - int16 contains all the 32-bit integers from -32768 to 32767
- By default, when we write int, the bits associated with it depends on the architecture of your computer
- It can be either 32 or 64 bits

Primitive data types in Golang

- **Floats**
- Floats are also numeric types
- They represent the decimal numbers
- We have some special formatting on the floats which are known as verbs
- Verbs are used to set the precision of the floating-point numbers
- We will cover verbs when we will work on a problem where we have to calculate the GPA of a student

Primitive data types in Golang

- In general, verbs are not only associated with floats. Verbs are the different formatting styles that we can apply when using the **Printf method from package fmt**
- **String**
- A string type represents a series of string values or characters if you may
- String type is usually used when we want to name things out or to write sentences

Primitive data types in Golang

- There are various needs that strings can satisfy but in layman's terms, they are used for writing characters as we write words and sentences
- **Boolean**
- When the outcome of our input can either be true or false, we use Boolean

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

Pointers

- 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 −

Pointers

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
   var a int = 10
   fmt.Printf("Address of a variable: %x\n", &a)
}
```

Result

```
$go run main.go
Address of a variable: c42000e1f8
```

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 −
  - var var_name *var-type
- Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable

What Are Pointers?

- 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 −
  - `var ip *int        /* pointer to an integer */`
  - `var fp *float32    /* pointer to a float */`

What Are Pointers?

- 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

How to Use Pointers?

- There are a few important operations, which we frequently perform with pointers:
  - we define pointer variables,
  - assign the address of a variable to a pointer, and
  - access the value at the address stored in the pointer variable
- 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

How to Use Pointers?

➢ The following example demonstrates how to perform these operations

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
   var a int = 20 /* actual variable */
   var ip *int    /* pointer variable */

   ip = &a  /* store address of a in pointer */

   fmt.Printf("Address of a variable: %x\n", &a)

   fmt.Printf("Address stored in ip variable:
       %x\n", ip)

   fmt.Printf("Value of *ip variable: %d\n", *ip)
}
```

Result

```
$go run main.go
Address of a variable: c4200c0018
Address stored in ip variable: c4200c0018
Value of *ip variable: 20
```

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 −

Nil Pointers in Go

Execute | Share | main.go | STDIN

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     var ptr *int
7
8     fmt.Printf("Value of ptr is : %x\n", ptr)
9  }
10
```

Result

```
$go run main.go
Value of ptr is : 0
```

➢ When the above code is compiled and executed, it produces the following result −

- Value of ptr is : 0

Go Pointers in Detail

Sr.No	Concept & Description
1	Go - Array of pointers You can define arrays to hold a number of pointers.
2	Go - Pointer to pointer Go allows you to have pointer on a pointer and so on.
3	Passing pointers to functions in Go Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function

Type Conversion

- The process of converting a value from one data type to another is known as type conversion
- Converting from one datatype to another is a frequent task we programmers do

Example :

- **How to Convert string to integer type in Go?**
- Golang includes strings as a built-in type
- Let's take an example, you may have a string that contains a numeric value "100"
- However, because this value is represented as a string, you can't perform any mathematical calculations on it
- You need to explicitly convert this string type into an integer type before you can perform any mathematical calculations on it

Example :

- In order to convert string to integer type in Golang, you can use the following methods.
- Atoi() Function
- You can use the strconv package's Atoi() function to convert the string into an integer value
- Atoi stands for ASCII to integer
- The Atoi() function returns two values: the result of the conversion, and the error (if any)

Example :

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "reflect"
    "strconv"
)

func main() {
   strVar := "100"

   intVar, err := strconv.Atoi(strVar)
   fmt.Println(intVar, err, reflect.TypeOf(intVar))

   intVar1, err := strconv.ParseInt(strVar, 0, 8)
   fmt.Println(intVar1, err, reflect.TypeOf(intVar1))

   intVar1, err = strconv.ParseInt(strVar, 0, 16)
   fmt.Println(intVar1, err, reflect.TypeOf(intVar1))

   intVar1, err = strconv.ParseInt(strVar, 0, 32)
   fmt.Println(intVar1, err, reflect.TypeOf(intVar1))

   intVar1, err = strconv.ParseInt(strVar, 0, 64)
   fmt.Println(intVar1, err, reflect.TypeOf(intVar1))
}
```

Result

```
$go run main.go
100 <nil> int
100 <nil> int64
100 <nil> int64
100 <nil> int64
100 <nil> int64
```

Example :

- Using fmt.Sscan
- The fmt package provides sscan() function which scans string argument and store into variables
- This function read the string with spaces and assign into consecutive Integer variables

Example :

➢ **Example**

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
   strVar := "100"
   intVal := 0
   _, err := fmt.Sscan(strVar, &intVal)
   fmt.Println(intVal, err, reflect.TypeOf(intVal))
}
```

Result

```
$go run main.go
100 <nil> int
```

Constants

- Constants

Constants

- As the name **CONSTANTS** suggests means fixed, in programming languages also it is same i.e., once the value of constant is defined it cannot be modified further
- There can be any basic data types of constant like an integer constant, a floating constant, a character constant, or a string literal
- **How to declare?**
- Constants are declared like variables but in using a ***const*** keyword as a prefix to declare constant with a specific type

Constants

- It cannot be declared using **:=** syntax
- **Example:**

Execute | Share main.go STDIN

```
1  package main
2
3  import "fmt"
4
5  const Pi = 3.14
6
7  func main() {
8     const World = "abc"
9
10    fmt.Println("Hello", World)
11    fmt.Println("Happy", Pi, "Day")
12
13    const Truth = true
14    fmt.Println("Go rules!", Truth)
15 }
16
```

Result

```
$go run main.go
Hello abc
Happy 3.14 Day
Go rules! true
```

Untyped and Typed Numeric Constants

- Typed constants work like immutable variables can inter-operate only with the same type and untyped constants work like literals can inter-operate with similar types
- Constants can be declared with or without a type in Go
- Following is the example which show typed and untyped numeric constants that are both named and unnamed

```
const untypedInteger           = 123
const untypedFloating typed    = 123.12
const typedInteger  int                = 123
const typedFloatingPoint   float64   = 123.12
```

List of constant in Go

- Numeric Constant (Integer constant, Floating constant, Complex constant)
- String literals
- Boolean Constant

Numeric Constant

- Numeric constants are ***high-precision values***
- As Go is a statically typed language that does not allow operations that mix numeric types
- You can't add a ***float64*** to an ***int***, or even an ***int32*** to an ***int***
- Although, it is legal to write ***1e6*time.Second*** or ***math.Exp(1)*** or even ***1<<('\t'+2.0)***
- In Go, constants, unlike variables, behave like regular numbers
- Numeric constant can be of 3 kinds i.e., integer, floating-point, complex

Integer Constant

- A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal
- An integer literal can also have a *suffix* that is a combination of U(upper case) and L(upper case), for unsigned and long, respectively
- It can be a *decimal, octal, or hexadecimal constant*
- An int can store at maximum a 64-bit integer, and sometimes less

Some examples of *Integer Constant*

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
212          /* Legal */
215u         /* Legal */
0xFeeL       /* Legal */
078          /* Illegal: 8 is not an octal digit */
032UU        /* Illegal: cannot repeat a suffix */
```

Complex constant

- Complex constants behave a lot like floating-point constants
- It is an *ordered pair* or *real pair* of integer constant(or parameter) And the constant are separated by a comma, and the pair is enclosed in between parentheses
- The first constant is the real part, and the second is the imaginary part
- A complex constant, COMPLEX*8, uses *8 bytes* of storage
- **Example:**

```
(0.0, 0.0) (-123.456E+30, 987.654E-29)
```

Floating Type Constant

- A floating type constant has an *integer part, a decimal point, a fractional part, and an exponent part*
- Can be represent floating constant either in decimal form or exponential form
- *While* representing using the decimal form, you must include the decimal point, the exponent, or both
- And while representing using the *exponential* form, must include the integer part, the fractional part, or both

Floating Type Constant

➢ Following are the examples of Floating type constant:

```
3.14159          /* Legal */

314159E-5L       /* Legal */

510E             /* Illegal: incomplete exponent */

210f             /* Illegal: no decimal or exponent */

.e55             /* Illegal: missing integer or fraction */
```

String Literals

- Go supports two types of string literals i.e., the ” ” (double-quote style) and the ‘ ‘ (back-quote)
- Strings can be *concatenated* with **+** and **+=** operators
- A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters
- And this is of untyped
- The zero values of string types are blank strings, which can be represented with ” ” or ” in literal

String Literals

- String types are all comparable by using operators like **==, !=** and (for comparing of same types)
- **Syntax**

```
type _string struct {
    elements *byte // underlying bytes
    len      int   // number of bytes
}
```

Example

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    const A = "LGH"
    var B = "LearnGoHere"

    // Concat strings
    var C = A + " " + B
    C += "!"
    fmt.Println(C)

    // Compare strings
    fmt.Println(A == "LGH")
    fmt.Println(B < A)

}
```

Result

```
$go run main.go
LGH LearnGoHere!
true
false
```

Boolean constant

- Boolean constants are similar to string constants
- It applies the same rules as string constant
- The difference is only that it have two untyped constants true and false

Example

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    const trueConst = true

    // Type definition using type keyword
    type myBool bool
    var defaultBool = trueConst          // allowed
    var customBool myBool = trueConst    // allowed

    // defaultBool = customBool          // not allowed
    fmt.Println(defaultBool)
    fmt.Println(customBool)

}
```

Result

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

Collection Types

- Arrays
- Slices
- Maps
- Structs

Arrays

- Basic type declarations such as
  - integers, floating-point numbers, boolean, strings, and constants form the basic building blocks of any programming language
- The Go language also includes complex types to accommodate real and imaginary components,
  - each of which are nothing but float32 and float64 types, respectively
- Arrays and slices are two of the most common composite types to represent data in Go

Arrays in Go

- Go arrays are fixed length data structures of zero or more elements of a particular type stored in memory
- The elements are characterized by their homogeneity, which simply means the same type of n number of elements can be stored in an array
- The functionalities with arrays in Go are quite similar to arrays in C/C++

Arraya in Go

- There is an indexing operator [] for declaring an array and the values start at the 0 position
- Therefore, the array [0] indicates the first element and array[len(array)-1] indicates the last element
- Note that there is a built-in versatile function called len() that returns the size of the type (in this case array size)
- The len() function can also be used to get the size of a string literal or any other composite types such as slices or map

Arrays to Go

- As with arrays, it is possible to create both one dimensional or
  - multidimensional arrays where the access values always start with indexes 0 to its size-1
- This means that if the size of the array is 10, then the starting index value would be array [0] and the last value would be at array[len(array)-1]
- Arrays are mutable; if we put array[index] at the left of an assignment it indicates that we are assigning some value to the array at the location indicated by the index

Arrays to Go

- Similarly, if we put array[index] at the right of the assignment it indicates that we want to get the array value at the specific index location.
- Example

Execute | Share main.go STDIN

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     var intArray[5] int
7     fmt.Printf("%T %d %v", intArray, len(intArray),
          intArray)
8  }
9
```

Result

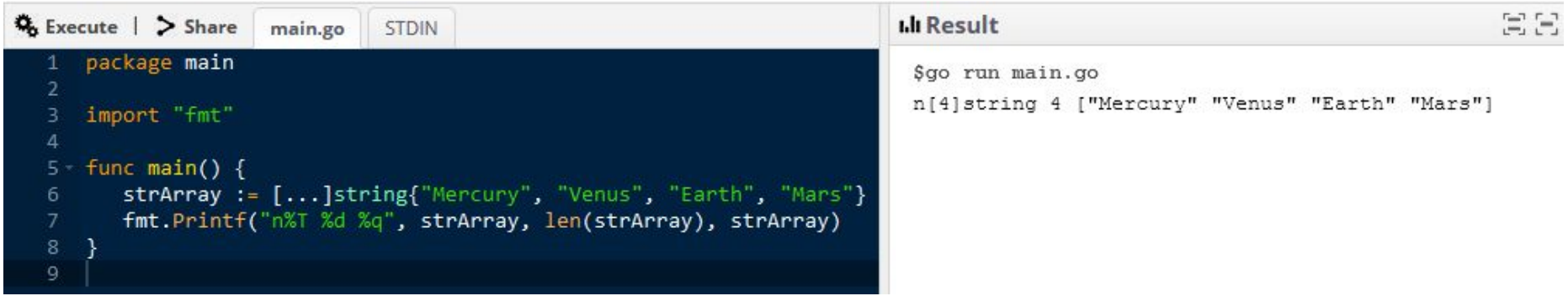
```
$go run main.go
[5]int 5 [0 0 0 0 0]
```

Arrays to Go

- In the code above we have declared an integer array of size 5 and print in the standard output its type, length and the value it contains
  - Note that we have not initialized the array but the output shows that it has been already initialized with the value 0
- Interestingly, we can create an array using the ellipse operator (…) as shown in the next slide.

Arrays to Go

- When we use ellipse, operator Go internally calculates the array size
- The operator has been overloaded for the purpose
- The size is statically counted and does not in any way mean that it creates a dynamic array



Few Things To Note About Arrays

- An array declared is array initialized, this means that Go guarantees that array items if not explicitly initialized,
  - partly or fully, then it automatically initializes it to their zero values at the time of creation
- Array length can be retrieved by the len() function
  - Since arrays have a fixed-length, which is the same as its capacity,
    - the cap() function returns the same result as the len() function

Array Example

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    A := [3][3]int{
        {1, 1, 0},
        {1, 1, 1},
        {1, 0, 0},
    }
    B := [3][3]int{
        {0, 1, 0},
        {1, 0, 1},
        {1, 0, 1},
    }
    var C [3][3]int
    for i := 0; i < 3; i++ {
        for j := 0; j < 3; j++ {
            for k := 0; k < 3; k++ {
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
            }
        }
    }
    fmt.Println("A = ", A)
    fmt.Println("B = ", B)
    fmt.Println("C = ", C)
}
```

Result

```
$go run main.go
A =  [[1 1 0] [1 1 1] [1 0 0]]
B =  [[0 1 0] [1 0 1] [1 0 1]]
C =  [[1 1 1] [2 1 2] [0 1 0]]
```

Slices in Go

- Slices are more versatile than arrays in Go
- They are more flexible to operate on
- Unlike arrays, slices are dynamic in nature in the sense that here the length of the array is determined during creation
- We can use the built-in make function to create a slice
- Slices are comparatively more efficient while passing as a function argument because they are always passed as a reference

Slices in Go

- The length of a slice can be resized after creation while arrays are of fixed length
- All standard API libraries internally use slices rather than array
- Although slices store items of the same type, with the use of interface we can store items of any type in an interesting way

A few things to note about slices

- The size of the slice can be extended or reduced after creation. It can be shrunk by slicing them or extended by built-in function append()
- Slice length can be retrieved by the len() function
- Slices are passed by reference
- Built-in make function can be used to create a slice

Slice Example

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "math/rand"
)

func main() {
   randInt := make([]int, 3)

   // insert some random data
   for i := 0; i < 3; i++ {
      randInt[i] = rand.Intn(100)
   }

   for i := 0; i < 3; i++ {
      fmt.Printf("%dt", randInt[i])
   }

}
```

Result

```
$go run main.go
81t87t47t
```

Matrix multiplication using slices

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    A := [3][3]int{
        {1, 1, 0},
        {1, 1, 1},
        {1, 0, 0},
    }
    B := [3][3]int{
        {0, 1, 0},
        {1, 0, 1},
        {1, 0, 1},
    }
    C := make([][]int, 3)
    for i := range C {
        C[i] = make([]int, 3)
    }

    for i := 0; i < 3; i++ {
        for j := 0; j < 3; j++ {
            for k := 0; k < 3; k++ {
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
            }
        }
    }
    fmt.Println("A = ", A)
    fmt.Println("B = ", B)
    fmt.Println("C = ", C)
}
```

Result

```
$go run main.go
A =  [[1 1 0] [1 1 1] [1 0 0]]
B =  [[0 1 0] [1 0 1] [1 0 1]]
C =  [[1 1 1] [2 1 2] [0 1 0]]
```

Arrays and Slices - Conclusion

- Both arrays and slices are typically used to store items of the same type
- Arrays are characterized by its rigidity in their implementation
- Once created the size remains fixed, and they are passed by value to a function
- However, we may use pointers if we are interested in passing arrays by reference
- Slices are always passed by reference to functions

What is a map?

- A map is a built in type in Go that is used to store key-value pairs
- Let's take the example of a startup with a few employees
- For simplicity, let's assume that the first name of all these employees is unique
- We are looking for a data structure to store the salary of each employee
- A map will be a perfect fit for this use case
- The name of the employee can be the key and the salary can be the value

How to create a map?

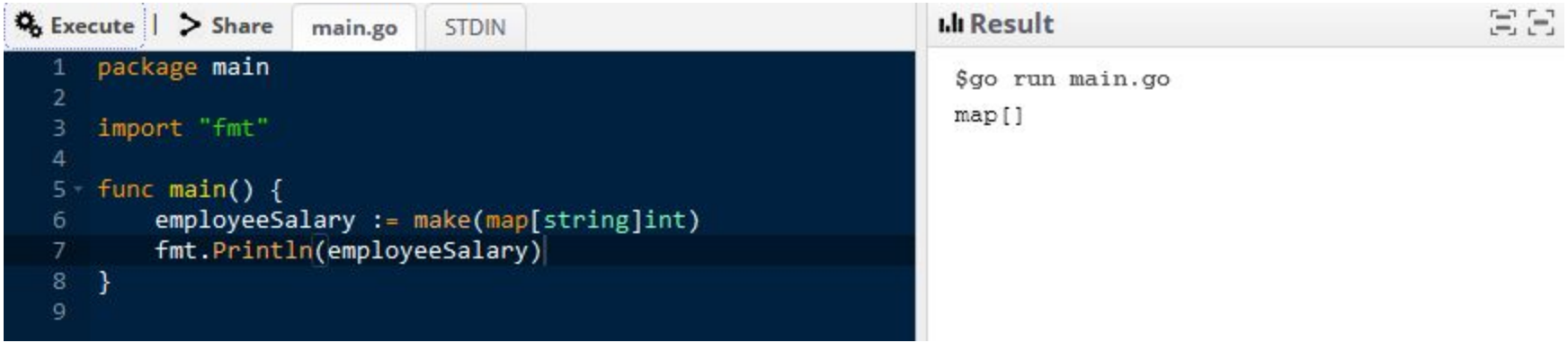
- A map can be created by passing the type of key and value to the make function
- The following is the syntax to create a new map

  make(map[type of key]type of value)

  employee Salary := make(map[string]int)
- The above line of code creates a map named employeeSalary which has string keys and int values

How to create a map?

- The program above creates a map named `employeeSalary` with `string` key and `int` value
- Since we have not added any elements to the map, it's empty



Adding items to a map

- The syntax for adding new items to a map is the same as that of arrays
- The program below adds some new employees to the `employeeSalary` map

Adding items to a map

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    employeeSalary := make(map[string]int)
    employeeSalary["steve"] = 12000
    employeeSalary["james"] = 15000
    employeeSalary["mike"] = 9000

    fmt.Println("map contents: ", employeeSalary)
}
```

Result

```
$go run main.go
map contents:  map[steve:12000 james:15000 mike:9000]
```

Zero value of a map

- The zero value of a map is nil
- If you try to add elements to a nil map, a run-time panic will occur
- Hence the map has to be initialized before adding elements

Execute | Share main.go STDIN

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var employeeSalary map[string]int
7      employeeSalary["steve"] = 12000
8
9      fmt.Println("map contents: ", employeeSalary)
10 }
11
```

Result

```
$go run main.go
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
        /home/cg/root/3160718/main.go:7 +0x59
exit status 2
```

Retrieving value for a key from a map

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    employeeSalary := map[string]int{
        "steve": 12000,
        "james": 15000,
        "mike": 9000,
    }
    employee := "mike"
    salary := employeeSalary[employee]
    fmt.Println("Salary of", employee, "is", salary)
}
```

Result

```
$go run main.go
Salary of mike is 9000
```

Deleting items from a map

- *delete(map, key)* is the syntax to delete key from a map
- The delete function does not return any value

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    employeeSalary := map[string]int{
        "steve": 12000,
        "james": 15000,
        "mike": 9000,
    }
    fmt.Println("map before deletion", employeeSalary)
    delete(employeeSalary, "steve")
    fmt.Println("map after deletion", employeeSalary)
}
```

Result

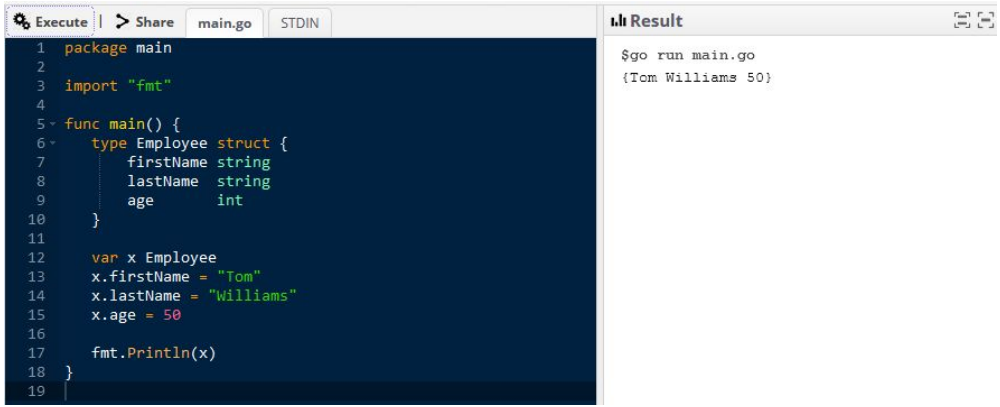
```
$go run main.go
map before deletion map[steve:12000 james:15000 mike:9000]
map after deletion map[james:15000 mike:9000]
```

Struct

- A struct is a user-defined type that represents a collection of fields
- It can be used in places where it makes sense to group the data into a single unit rather than having each of them as separate values
- For instance, an employee has a firstName, lastName and age
- It makes sense to group these three properties into a single struct named Employee

Declaring a struct

- The below snippet declares a struct type Employee with fields firstName, lastName and age
- The below Employee struct is called a **named struct** because it creates a new data type named Employee using which Employee structs can be created




```go
package main

import "fmt"

func main() {
    type Employee struct {
        firstName string
        lastName  string
        age       int
    }

    var x Employee
    x.firstName = "Tom"
    x.lastName = "Williams"
    x.age = 50

    fmt.Println(x)
}
```

Declaring a struct

- This struct can also be made more compact by declaring fields that belong to the same type in a single line followed by the type name
- In the above struct firstName and lastName belong to the same type string and hence the struct can be rewritten as

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Employee struct {
    firstName, lastName string
    age int
}

func main() {
    emp8 := Employee{
        firstName: "steve",
        lastName:  "jobs",
        age:       60,
    }
    fmt.Println("Employee emp8: ", emp8)
}
```

Result

```
$go run main.go
Employee emp8:  {steve jobs 60}
```

Creating named structs

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
   // create struct specifying field names
   emp1 := Employee{
       firstName: "Same",
       age:       25,
       salary:    500,
       lastName:  "Anderson",
   }

   // create struct without specifying field names
   emp2 := Employee{"Thomas", "John", 29, 800}
   fmt.Println("Employee 1", emp1)
   fmt.Println("Employee 2", emp2)
}
```

Result

```
$go run main.go
Employee 1 {Same Anderson 25 500}
Employee 2 {Thomas John 29 800}
```

Creating anonymous structs

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    // create anonymous struct specifying field names
    emp3 := struct {
        firstName string
        lastName  string
        age       int
        salary    int
    }{
        firstName: "Sam",
        lastName:  "Anderson",
        age:       31,
        salary:    5000,
    }

    fmt.Println("Employee 3", emp3)
}
```

Result

```
$go run main.go
Employee 3 {Sam Anderson 31 5000}
```

Accessing individual fields of a struct

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
   emp4 := Employee{
       firstName: "Sam",
       age:       25,
       salary:    500,
       lastName:  "Anderson",
   }

   // access struct fields using .
   fmt.Println("Employee 4 first name", emp4.firstName)
   fmt.Println("Employee 4 last name", emp4.lastName)
   fmt.Println("Employee 4 age", emp4.age)
   fmt.Println("Employee 4 salary", emp4.salary)
}
```

Result

```
$go run main.go
Employee 4 first name Sam
Employee 4 last name Anderson
Employee 4 age 25
Employee 4 salary 500
```

Zero value of a struct

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
   var emp5 Employee // zero value struct
   fmt.Println("Employee 5 first name", emp5.firstName)
   fmt.Println("Employee 5 last name", emp5.lastName)
   fmt.Println("Employee 5 age", emp5.age)
   fmt.Println("Employee 5 salary", emp5.salary)
}

```

Result

```
$go run main.go
Employee 5 first name
Employee 5 last name
Employee 5 age 0
Employee 5 salary 0
```

Pointers to a struct

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
   // pointer to a struct
   emp6 := &Employee{
       firstName: "Sam",
       lastName:  "Anderson",
       age:       55,
       salary:    10000,
   }
   fmt.Println("Employee 6 first name", (*emp6).firstName)
   fmt.Println("Employee 6 salary", (*emp6).salary)
}
```

Result

```
$go run main.go
Employee 6 first name Sam
Employee 6 salary 10000
```

Nested structs

- It is possible that a struct contains a field which in turn is a struct.

  These kinds of structs are called nested structs

Nested structs

Execute | Share main.go STDIN

```go
package main

import "fmt"

type Address struct {
    city    string
    state   string
}

type Person struct {
    name    string
    age     int
    address Address
}

func main() {
   p := Person{
      name: "Thomas",
      age:  25,
      address: Address{
          city: "Chicago",
          state: "Illinois",
      },
   }

   fmt.Println("Person name", p.name)
   fmt.Println("Person age", p.age)
   fmt.Println("Person city", p.address.city)
   fmt.Println("Person state", p.address.state)
}
```

Result

```
$go run main.go
Person name Thomas
Person age 25
Person city Chicago
Person state Illinois
```