Go Programming Language

Tan Cher Wah

cherwah@nus.edu.sg

Go Programming Language

- Developed at Google (2007)
- Supports Object-Oriented style of programming
- No Type Hierarchy
- Statically-Typed
- Garbage Collection
- Rich Standard Library

Included Features in Go

- Type Inference
- Type Embedding (Composition)
- Interfaces
- Statically-linked native binaries (no external dependencies)
- In-built Concurrency Support (simultaneous execution)

Installing Go

- Go to official Go website - https://go.dev/dl
- Choose the appropriate installer for your operating system
- Follow the prompts in the installer to complete the installation
- Once the installation is complete, open a new terminal window and type "**go version**" (without the quotes). You should see "go version goX.Y.Z" where X.Y.Z is the version number.

Setup Visual Studio Code for Go

- In VS Code, bring up the **Extensions view** by clicking on the Extensions icon in the Activity Bar. Or use keyboard shortcut - Ctrl+Shift+X (for Windows) or Cmd+Shift+X (for Mac)
- Search for the **Go** (by Go Team at Google) **extension**, then select **install**
- On your local machine, create a new folder (any name) to host your Go files
- Create a main.go file in that newly created folder (see next slide)
- In VS Code's **Explorer view**, open that folder via Open Folder
- You should now be able to run or debug your main.go file

Program Entry Point

The **entry point** for a Go program is the **main()** function

```
package main

func main() {
    //  program logic here
}
```

Primitive Data Types

Primitive data types are the **most basic data types** in a programming language and are **building blocks** for more complex data structures

- bool (e.g. true, false)
- int, uint (e.g. 5)
- float32, float64 (e.g. 5.1)
- string (e.g. "hello")
- byte (alias for uint8)
- rune (alias for int32) - represents a Unicode code point

Variables

The **type** of a variable can be **explicitly declared** or **inferred**

```go
// explicitly declared
var i int = 1
var j float64 = 2.0
var k string = "hello"

// inferred
i2 := 1
j2 := 2.0
k2 := "hello"
```

Type Conversion

Casting can be achieved by specifying the **destination-type** in front of the target variable

```
// inferred 'int' type
i := 42

// cast from 'int' to 'float'
f := float64(i)

// cast from 'float; to unsigned int
u := uint(f)
```

Constants

Constants are defined by the **const** keyword

```
const Pi = 3.14
const Truth = true

fmt.Println("Value of Pi =", Pi)
fmt.Println("Go rules? -", Truth)
```

Conditionals

Caveat: The parser **only** recognises **K&R brace-style**!

```go
x := 1

if x == 0 {
    fmt.Println("x is 0.")
} else if x == 1 {
    fmt.Println("x is 1.")
} else {
    fmt.Println("x is not 0 or 1.")
}
```

In Go, if-else statements do not require **brackets** around **conditions**

For Loop (I)

A **for-loop** to iterate through a range of numbers

```
sum := 0

for i := 1; i < 5; i++ {
    sum++
}
```

For Loop (II)

A **for-loop** that behaves like a **while-loop** (in other languages)

```go
sum := 0

for sum < 5 {
    sum *= 2
}
```

For Loop (III)

A **for-loop** without conditions behaves like an **infinite loop**

```
sum := 0

for {
    sum *= 2

    if sum >= 5 {
        break
    }
}
```

For Loop (IV)

A **for-loop** to get the **index** and **value** of each element in the list

```
strs := []string{"hello", "world"}

for i, str := range strs {
    fmt.Println(i, str)
}
```

Acts like a **foreach** statement in other programming languages

Scope

Inner Code Blocks can access variables defined in **Outer Code Blocks**; but not the other way round

```go
{
    v := 1

    {
        v2 := 2

        // okay
        fmt.Println(v)
    }

    // error!!
    fmt.Println(v2)
}
```

Strings

Strings are **immutable** in Golang; can be read but not modified

```go
s1 := "hello"
s2 := "world"

// this is indexing first byte
// (not indexing first character)
s1[0] = "a"      // cannot compile!

greeting := s1 + " " + s2   // new string
fmt.Println(greeting)
```

Character Encoding

A **character** has a Unicode "**code point**" as **value**, and is **encoded** in **UTF-8** (UTF-8 is an **Encoding Standard**, Unicode is a **Character Set**)

```go
s := "世界"

for i, value := range s {
    fmt.Printf("index: %d, char: %c, unicode: %U\n",
        i, value, value)
}
```

↓

```
index: 0, char: 世, unicode: U+4E16
index: 3, char: 界, unicode: U+754C
```

Rune

A **Rune** contains a Unicode **code-point value** of a character and uses the **int32** data-type

```
s := "a世界b"

for i, value := range s {
    fmt.Printf("index: %d, char: %c, unicode: %U, rune: %d, value: %d\n",
        i, value, value, rune(value), value)
}
```

```
index: 0, char: a, unicode: U+0061, rune: 97, value: 97
index: 1, char: 世, unicode: U+4E16, rune: 19990, value: 19990
index: 4, char: 界, unicode: U+754C, rune: 30028, value: 30028
index: 7, char: b, unicode: U+0062, rune: 98, value: 98
```

Array

An Array is a **fixed-size**, **ordered** collection of elements of the **same type**

```go
// array pre-initialized with values
ints := [3]int{1, 2, 3}

// ok to modify index 0, 1 and 2
ints[1] = 0

// error! - out of bounds
ints[4] = 4
```

```go
// start off with an empty array
strs := [2]string{}

// add new entries
strs[0] = "Hello"
strs[1] = "World"

fmt.Println(strs)
```

Output

```
[Hello World]
```

Slice

A Slice allows us to **extract** a sequence of elements; it is built on top of an underlying array and allows for **dynamic resizing**

```go
// an array of integers
x := [5]int{1, 2, 3, 4, 5}

// create a slice of the array
sub_x := x[1:3]

fmt.Println(sub_x)
```

Output

```
[2 3]
```

Updating a Slice

Updating a Slice also **updates** the array that **it points to**

```go
// an array of integers
x := [5]int{1, 2, 3, 4, 5}

// creating a slice of the array
sub_x := x[1:3]

fmt.Println("sub_x =", sub_x)

// updating the slice
sub_x[0] = 0
sub_x[1] = 0

fmt.Println("sub_x =", sub_x)
fmt.Println("x =", x)
```

Output

```
sub_x = [2 3]
sub_x = [0 0]
x = [1 0 0 4 5]
```

Appending to a Slice

Unlike an Array, we can **add** more elements to a Slice

```go
// create a slice of 3 items
list := []int{1, 2, 3}

fmt.Println("before =", list)

// add 4, 5, 6 to slice
list = append(list, 4, 5, 6)

fmt.Println("after =", list)
```

Output

```
before = [1 2 3]
after = [1 2 3 4 5 6]
```

Map

A Map is a **dictionary** with a **key/value** pair storage

```go
// initialising an empty dictionary
// where the 'key' is a string
// and the 'value' is a float64
m := map[string]float64{}

// adding a key-value pair
m["pi"] = 3.1416

// retrieving value using key
fmt.Println(m["pi"])
```

```go
// initialising with values
m := map[string]float64{
    "pi": 3.14,
    "e":  2.71,
}

// retrieving value using key
fmt.Println(m["e"])
```

Functions

Functions are defined by the **func** keyword; the function named **main** is the **entry point** of a Go program

```go
package main

import "fmt"

func main() {
    callme()
}

func callme() {
    fmt.Println("You called?")
}
```

Output

```
You called?
```

Structures

Structures are defined by the **struct** keyword; there are **NO Classes** in Go, only **Structures**!

```
type person struct {
    name string
    age  int
}

func main() {
    // create a new Person object
    john := person{
        name: "john",
        age:  23,
    }
}
```

Methods

Methods are linked to Structures by specifying the **struct-type** in front of a **function name**

```go
type person struct {
    name string
    age  int
}

func main() {

    // create a new Person object
    john := person{
        name: "John",
        age:  23,
    }

    john.self_intro()
}
```

```go
func (j person) self_intro() {
    fmt.Printf("Hi! My name is %s and
       I'm %d years old.\n", j.name, j.age)
}
```

Output

```
Hi! My name is John and I'm 23 years old.
```

Pointers

A **Pointer** stores a **memory address** that **points to** an **actual value**

```go
func main() {

    // create a new Person object
    john := person{
        name: "John",
        age:  23,
    }

    ptr := &john

    fmt.Printf("Memory address: %p\n", ptr)
    fmt.Printf("Memory address: %p\n", &john)
}
```

Output

```
Memory address: 0xc00019c000
Memory address: 0xc00019c000
```

Modifying Values via Pointers

Values of in-memory structs can be **modified** via Pointers

```go
func main() {

    // create a new Person object
    john := person{
        name: "John",
        age:  23,
    }

    ptr := &john

    // before changes
    fmt.Println(john)

    ptr.age = 24

    // after changes
    fmt.Println(john)
}
```

Output

```
{John 23}
{John 24}
```

Copying of Structs

Modifying a **copy** of an object **does not affect** the original object

```go
func main() {

    // create a new Person object
    john := person{
        name: "John",
        age:  23,
    }

    // make a copy
    john2 := john

    // before change
    fmt.Println(john)

    john2.age = 25

    // after change
    fmt.Println(john2)
    fmt.Println(john)
}
```

Output

```
{John 23}
{John 25}
{John 23}
```

Pass By Value

Function Arguments are **passed by value**, meaning a function receives a **copy** of the argument's value, **not a reference** to the argument itself

```go
package main

import "fmt"

func main() {
    x := 42
    fmt.Println("Before:", x)
    passByValue(x)
    fmt.Println("After:", x)
}

func passByValue(x int) {
    x = 43
}
```

Output

```
Before: 42
After: 42
```

Function **Parameters** are the **names** listed in a function's definition.
Function **Arguments** are the **real values** passed to the function.

Pass By Reference

In Go, you can **pass** a **reference** to a value by using a **pointer** (i.e. a variable that stores the memory address of the actual value)

```
package main

import "fmt"

func main() {
    x := 42
    fmt.Println("Before:", x)
    passByReference(&x)
    fmt.Println("After:", x)
}

func passByReference(x *int) {
    // de-reference the pointer
    // to access the value
    *x = 43
}
```

Output

```
Before: 42
After: 43
```

Interfaces

In Go, an **Interface** is a **type** that **defines** a **set of methods**

```go
type shape interface {
    area() float64
}

type rectangle struct {
    width, height float64
}

func (r rectangle) area() float64 {
    return r.width * r.height
}
```

```go
package main

import "fmt"

func main() {
    r := rectangle{width: 10, height: 5}
    fmt.Println("Area:", calArea(r))
}

// implements "shape" interface
func calArea(s shape) float64 {
    return s.area()
}
```

Output

```
Area: 50
```

Composition (Embedding)

While there is no inheritance in Go, you can use **composition** (or **embedding**) to **reuse fields and methods** of an existing struct in a new struct

```go
type person struct {
    name string
    dob  int
}

type employee struct {
    person
    company string
}

func (e employee) intro() {
    fmt.Printf("I am %s, born in %d
        and work at %s.\n",
        e.name, e.dob, e.company)
}
```

```go
func main() {
   e := employee{
        person: person{
            name: "John",
            dob:  1980,
        },
        company: "NUS",
    }

    e.intro()
}
```

Output

```
I am John and I born in 1980.
I am John, born in 1980 and work at NUS.
```

Coroutines

Go Coroutines are a lightweight, **concurrent execution paths** within a single process. They allow us to perform multiple tasks **simultaneously**

```
func main() {

    // start first thread
    go func1()

    // start second thread
    go func2()

    // let both threads have a chance
    // to run and finish
    time.Sleep(5 * time.Second)
}
```

```
func func1() {
    for i := 1; i <= 50; i++ {
        fmt.Println("go1: " + strconv.Itoa(i))
    }
}

func func2() {
    for i := 1; i <= 50; i++ {
        fmt.Println("go2: " + strconv.Itoa(i))
    }
}
```

Output

```
go2: 1
go2: 2
go2: 3
go1: 1
go1: 2
go1: 3
go1: 4
...
go2: 49
go2: 50
go1: 48
go1: 49
go1: 50
```

WaitGroup

sync.WaitGroup provides a way to **wait** for a **collection of Goroutines** to **complete**

```go
func main() {
    wg := sync.WaitGroup{}

    // two goroutines to wait on
    wg.Add(2)

    go func1(&wg)
    go func2(&wg)

    // wait for completion
    wg.Wait()
}
```

A Timer is no longer necessary; wg.Wait() will wait for both coroutines to complete.

```go
func func1(wg *sync.WaitGroup) {
    // invoked at end of function
    defer wg.Done()

    for i := 1; i <= 50; i++ {
        fmt.Println("go1: " + strconv.Itoa(i))
    }
}

func func2(wg *sync.WaitGroup) {
    // invoked at end of function
    defer wg.Done()

    for i := 1; i <= 50; i++ {
        fmt.Println("go2: " + strconv.Itoa(i))
    }
}
```

Output

```
go2: 1
go2: 2
go2: 3
go1: 1
go1: 2
go1: 3
go1: 4
...
go2: 49
go2: 50
go1: 48
go1: 49
go1: 50
```

Channels

Channels provide a mechanism for **passing values** (FIFO order) between Goroutines

```go
func main() {
    wg := sync.WaitGroup{}
    wg.Add(2)

    chan_ma := make(chan int)
    chan_ab := make(chan int)

    go go1(chan_ma, chan_ab, &wg)

    var _ = <-chan_ma
    go go2(chan_ab, &wg)

    wg.Wait()
}
```

Synchronisation between the two coroutines' alternate outputs (take turns) was achieved with the use of channels.

```go
func go1(ma chan int, ab chan int, wg *sync.WaitGroup) {
    defer wg.Done()

    ma <- 1

    for i := 1; i <= 50; i++ {
        fmt.Println("go1: " + strconv.Itoa(i))
        ab <- 1
        var _ = <-ab
    }
}

func go2(ab chan int, wg *sync.WaitGroup) {
    defer wg.Done()

    for i := 1; i <= 50; i++ {
        var _ = <-ab
        fmt.Println("go2: " + strconv.Itoa(i))
        ab <- 1
    }
}
```

Output

```
go1: 1
go2: 1
go1: 2
go2: 2
go1: 3
go2: 3
go1: 4
go2: 4
...
go1: 48
go2: 48
go1: 49
go2: 49
go1: 50
go2: 50
```

Go modules

- Go modules are a way to manage dependencies of a Go project
- A Go module is a collection of related Go packages
- A module is defined by a 'go.mod' file, which specifies the module's name, version, and dependencies
- When creating a new Go project, start by creating a module, then one of more packages within that module

Go packages

- A Go package is a collection of Go source files that are organised together in a directory and can be imported and used by other Go programs
- A package can define types, functions, and variables that can be used by other packages
- A **package** that is named **main** will be compiled into an **executable file** (instead of a library)
- The **main()** function, inside the **package** named **main**, is the program's **entry point**

Visibility in Go packages

- A **package** is the **smallest unit** of **private** encapsulation in Go
- All identifiers defined within a package are **visible** within the **package**
- When a **package** is being **imported**, only its **exported identifiers** can be accessed
- An identifier is **exported** if it **begins** with a **Capital letter**

Package (Example)

Accessing StopWatch.running from a different package is not allowed as the variable begins a lowercase

```go
package timer

import "time"

type StopWatch struct {
    start   time.Time
    running bool
}

func (s *StopWatch) Start() {
    if !s.running {
        s.start = time.Now()
        s.running = true
    }
}
```

```go
package main

import "patterns_in_go/timer"

func main() {
    clock := new(timer.StopWatch)
    clock.Start()

    // error!! **not exported**
    if clock.running {

    }
}
```

Go Commands

- **go mod init <module_name>** to create a Go module (outputs a **go.mod** file with module-name)
- **go mod tidy** cleans up unused dependencies and adds missing dependencies
- **go work init** to work with **multiple** Go modules (outputs a **go.work** file to contain multiple module-names)
- **go work use <mod1> <mod2> …** to specify the module-names to work within a workspace (aka directory)

References

- A Tour of Go
- An Introduction to Programming in Go
- Go Go-To Guide
- How to use Go Modules
- Working with multiple Go Modules

THE END