Go Foundation

Programming

LOGISTICS

Class Hours:

- Instructor will set class start and end times.
- There will be regular breaks in class.

Telecommunication:

- Turn off or set electronic devices to silent (not vibrate)
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

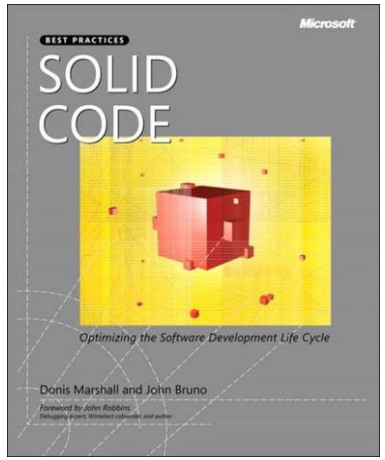
Miscellaneous:

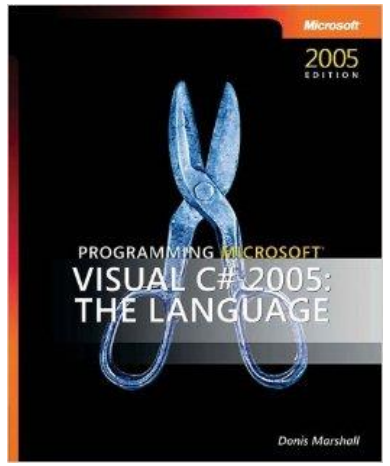
- Courseware
- Bathroom
- Fire drills

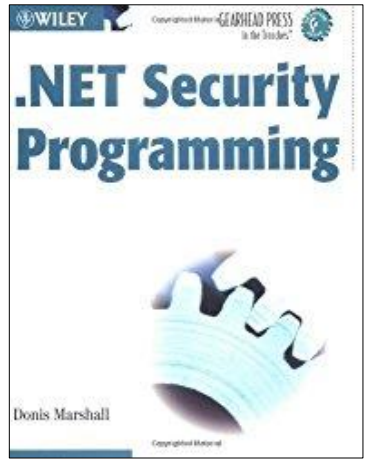
DONIS MARSHALL

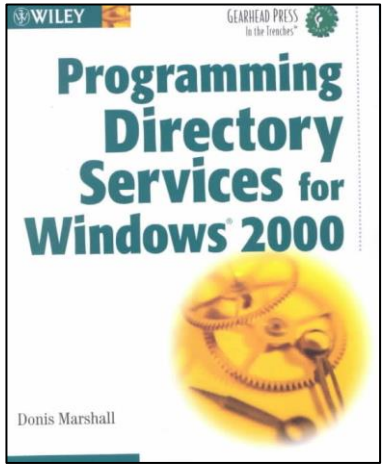
Go Language Practitioner
Microsoft MVP
Microsoft Certified
C++ Certified
Author

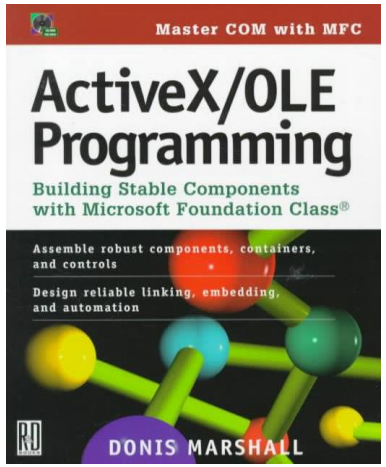
dmarshall@innovationinsoftware.com

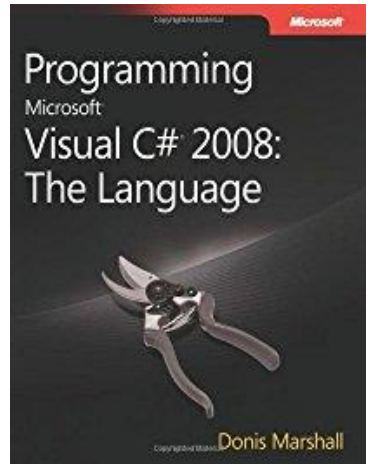












INTRODUCE YOURSELF

Time to introduce yourself:

- Name
- What is your role in the organization
- Indicate Go experience

THE BEST OF ALL WORLDS

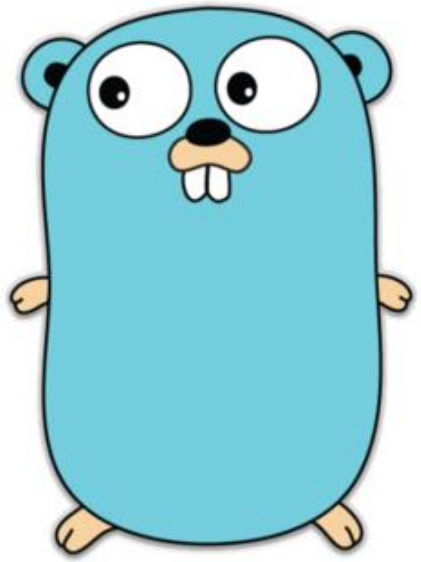
Go Programming language combines the freedom and syntax of a native language plus the usability and abstraction of a managed language.

Ease of use, similar to Python, but detailed enough for complex development, similar to C / C++.

For example:

- Garbage collection (managed)
- Pointers (native)

GO LANGUAGE

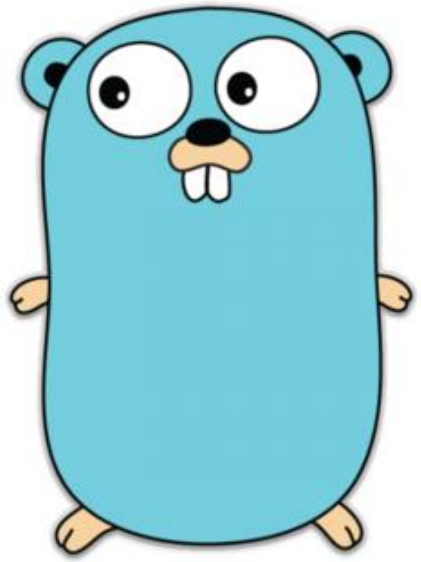


Go

Go Language (golang) is an open source language developed by Robert Griesemer, Rob Pike, and Ken Thompson. Go was created partly from frustration with of C++.

"When the three of us got started, it was pure research. The three of us got together and decided that we hated C++. We started off with the idea that all three of us had to be talked into every feature in the language, so there was no extraneous garbage put into the language for any reason." – Ken Thompson

GO LANGUAGE - 2



Go

Go was initially developed from requirements set by Google. Go Language was announced in the Google Open Source Blog on November 10, 2009.

https://bit.ly/39dTE8t

The Google Open Source Blog says that "Go combines the development speed of working in a dynamic language like Python with the performance and safety of a compiled language like C or C++."

POPULARITY

- Bitcoin
- Ethereum
- Terraform
- Kubernetes
- Docker
- Etcd
- Revel
- InfluxDB

The visibility and popularity of Go exploded with the release of transformative products developed with the language. Most notably, blockchain and container technologies, such as Bitcoin and Docker, created considerable interest in the language.

Here are some of the products developed with the Go language.

INTRODUCTION TO GO LANGUAGE

Go (Golang) is C-like, small, and a strongly typed language. The language supports low-level capabilities such as pointers and concurrency; while offering higher level features such as garbage collection and collections. Go Language can be used as either a procedural or object-based language.

Code from the Go Language is compiled into binary unlike Python language, which is an interpreted language. This allows for faster execution, performance optimization, and more. Go Language scales efficiently making it easier to develop large applications.

There are a variety of libraries available for Go Language:

https://golang.org/pkg/

These libraries offer many features including string manipulation, time functions, concurrency, file handling, and networking.

NO INHERITANCE

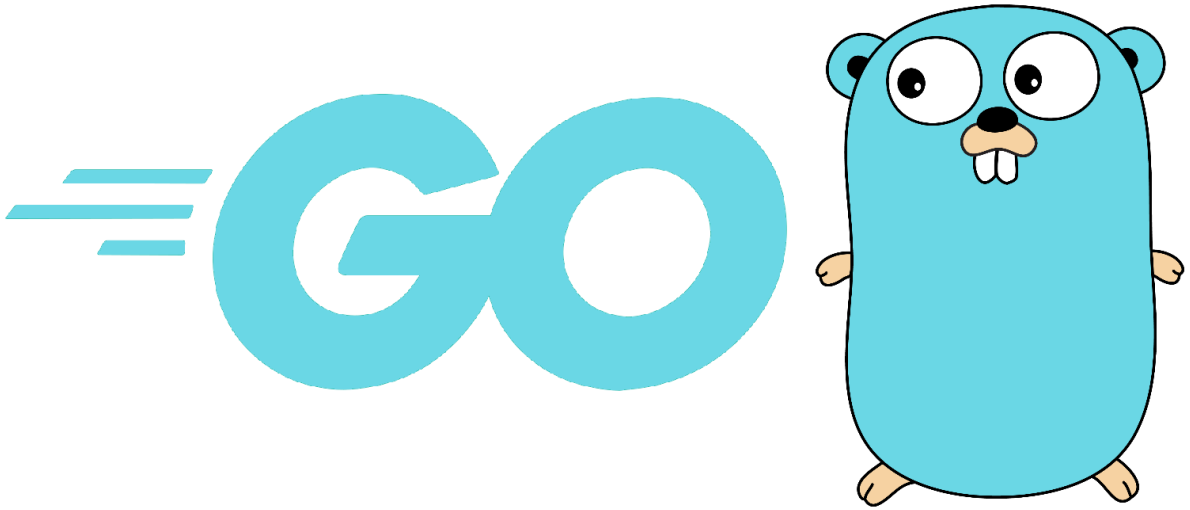
Go Language does not use classes and inheritance. Instead, composition is used for object-based programming. Traditionally, there has been an overreliance on inheritance in modern languages, such as C++, Java, and Python. For this reason, Golang features a different approach.

In the Go Language, object programming is implemented with structures and methods.

This is an article on the benefits of composition over inheritance.

http://bit.ly/2H3mHxT

GO LANG 1.18



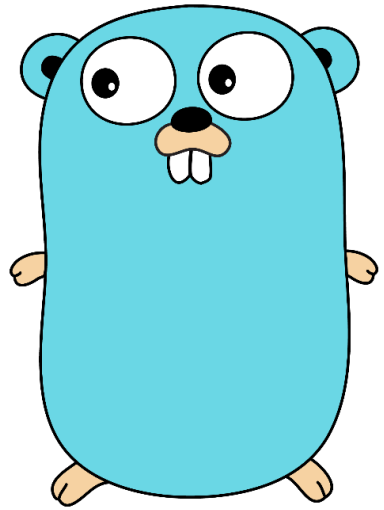
Go 1.18 was released on March 15, 2022 and represents a major release with new features.

https://bit.ly/36U9md3

Three features dominate this release:

- Generics
- Fuzzing
- Workspace mode

GO LANG 1.19



Go 1.18 was released on March 15, 2022 and represents a major release with new features.

https://bit.ly/36U9md3

Three features dominate this release:

- Generics
- Fuzzing
- Workspace mode

MODERN GO - GENERICS

Generics is a popular feature in many languages, such as C++, Java, Rust, and many others. Generics is the most anticipated new feature is Go 1.18.

With generics, you can create parameterized functions and types, with type parameters. Type parameters are placeholders for specific types that are defined at runtime.

Generics provide unprecedented flexibility. For example, a function can be called with types as arguments.

The bonus to generics is increased performance.

MODERN GO - FUZZING

Go is now the first popular programming language to include fuzzing as a native feature.

Fuzzing is the ability to iteratively exercise code while randomizing the input. Fuzzing is considered a security feature but also a testing feature that increases code coverage.

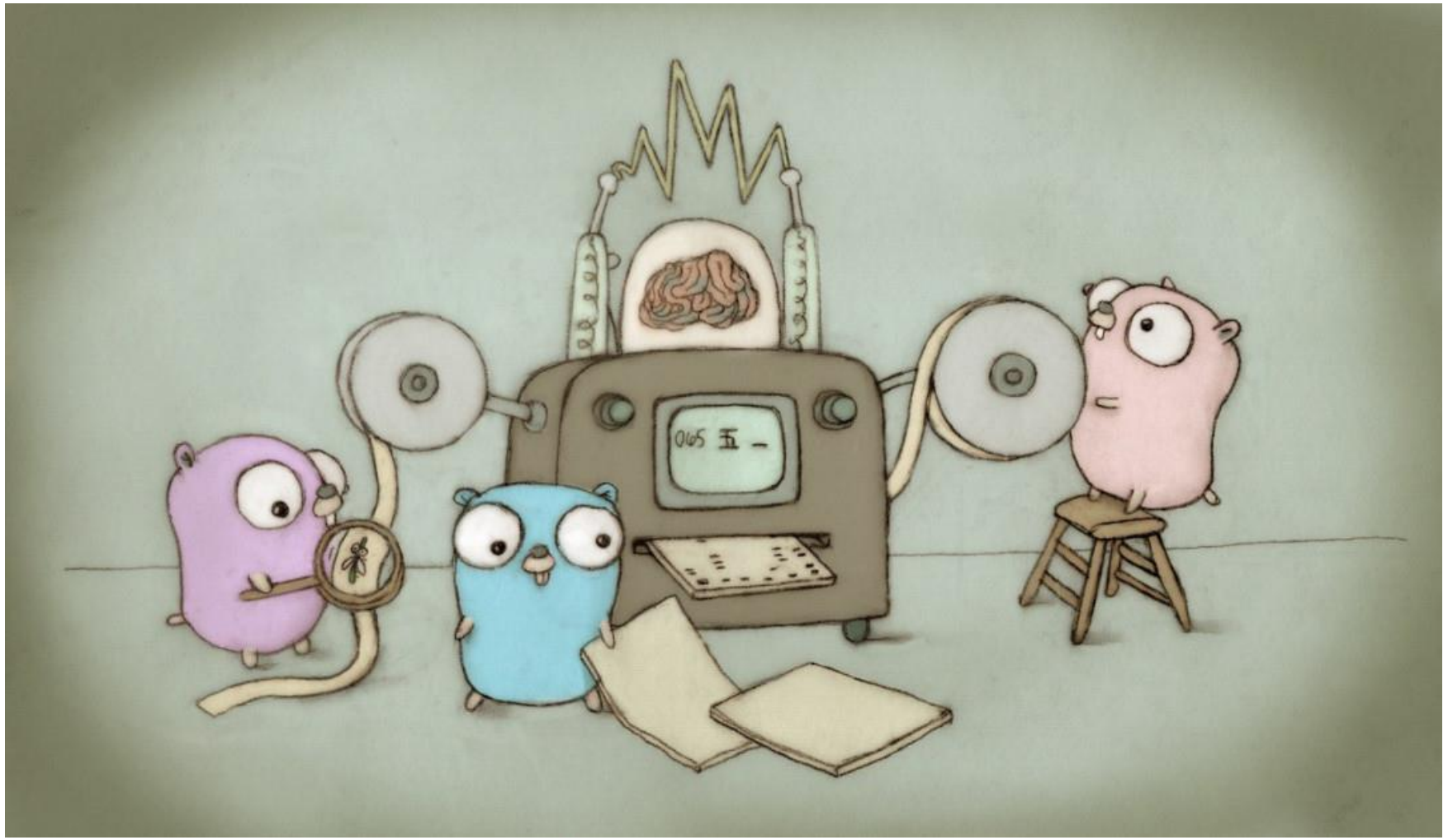
This is a powerful alternative to manually executing a function with different values.

MODERN GO - TOOLBOX

The latest versions of Go also add several miscellaneous features, such as:

- Any
- ~ Tilde operator
- Strings.Cut
- And many more

GO LANGUAGE ENVIRONMENT



PLATFORM AGNOSTIC

Go Language is platform agnostic. You can download Go Language for Mac, Linux, or the Windows environment.

Here is the page for downloading Go Language for your environment. It also documents operating system compatibility.

https://golang.org/doc/install

Operating system	Architectures
FreeBSD 9.3 or later	amd64, 386
Linux 2.6.23 or later with glibc	amd64, 386, arm, arm64,s390x, ppc64le
macOS 10.8 or later	amd64
Windows XP SP2 or later	amd64, 386

GIT

GitHub is a web-based hosting service for version control of files. Often a repository for open source code projects. You can store many types of documents within GitHub; not just source files.

Git is a free and open source tool for distributed version control – typically using GitHub as the source repository. It is lightweight and fast. Teams have preferred GIT and distributed version control in lieu of centralized version control and tools such as Subversion, Perforce, and ClearCase.

For example, the Go Language repository is stored in Github at:

https://github.com/golang/go

Git is typically installed with the Go Programming language. This link explains how other source control programs can be used instead.

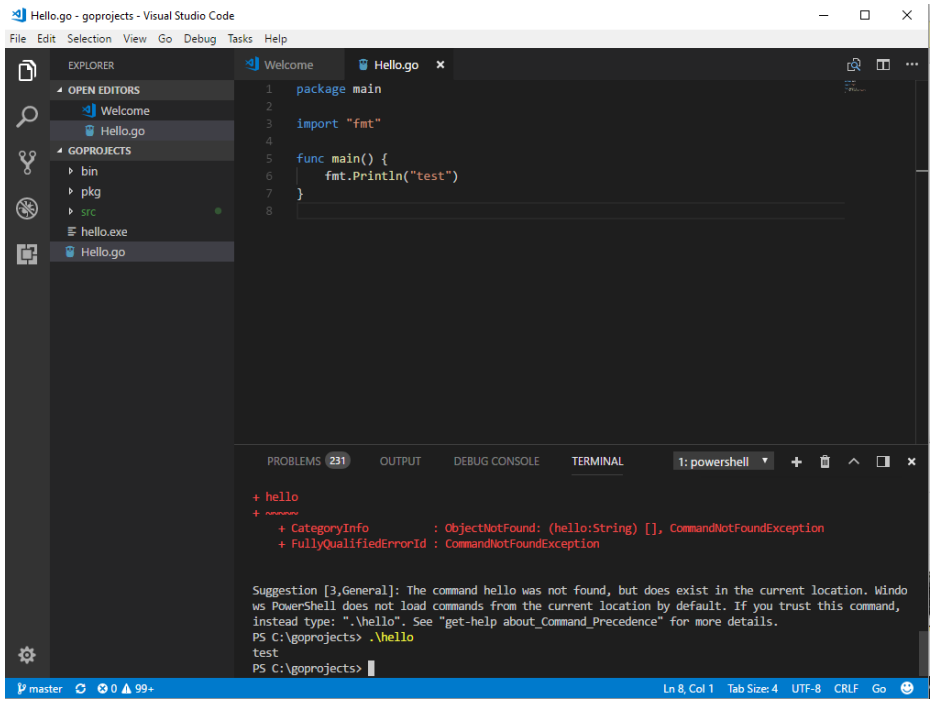
https://bit.ly/2LhPy34

CODING

Basic text editors can be used to code Go programs. For this reason, formal IDEs are not required to edit, compile, or execute a Go program. You only need the Go Language artifacts, a text editor, and a terminal / command prompt. However, an IDE can make creating a Go program more convenient. Features such as syntax checking and intellisense, which are common to formal IDEs, are helpful. Popular IDEs that support Go Language include:

- VS Code
- IntelliJ
- Emacs
- Eclipse

VS CODE



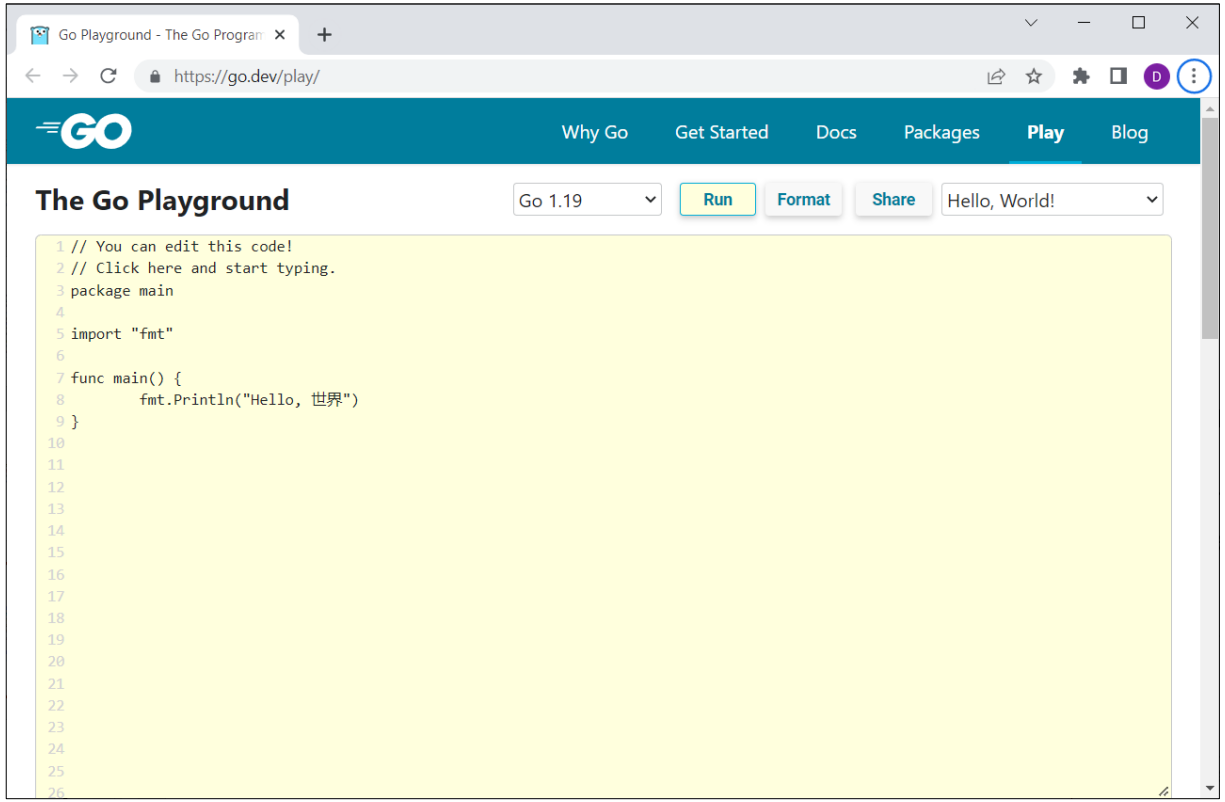
In class, you can use any tool for Go development – whether an IDE or a simple text editor. However, you are expected to understand your tool.

The instructor will use VS Code as the IDE. It is free and available in most environments. VS Code has an extension for the Go Language.

Download VS Code here:

https://code.visualstudio.com/download

THE GO PLAYGROUND



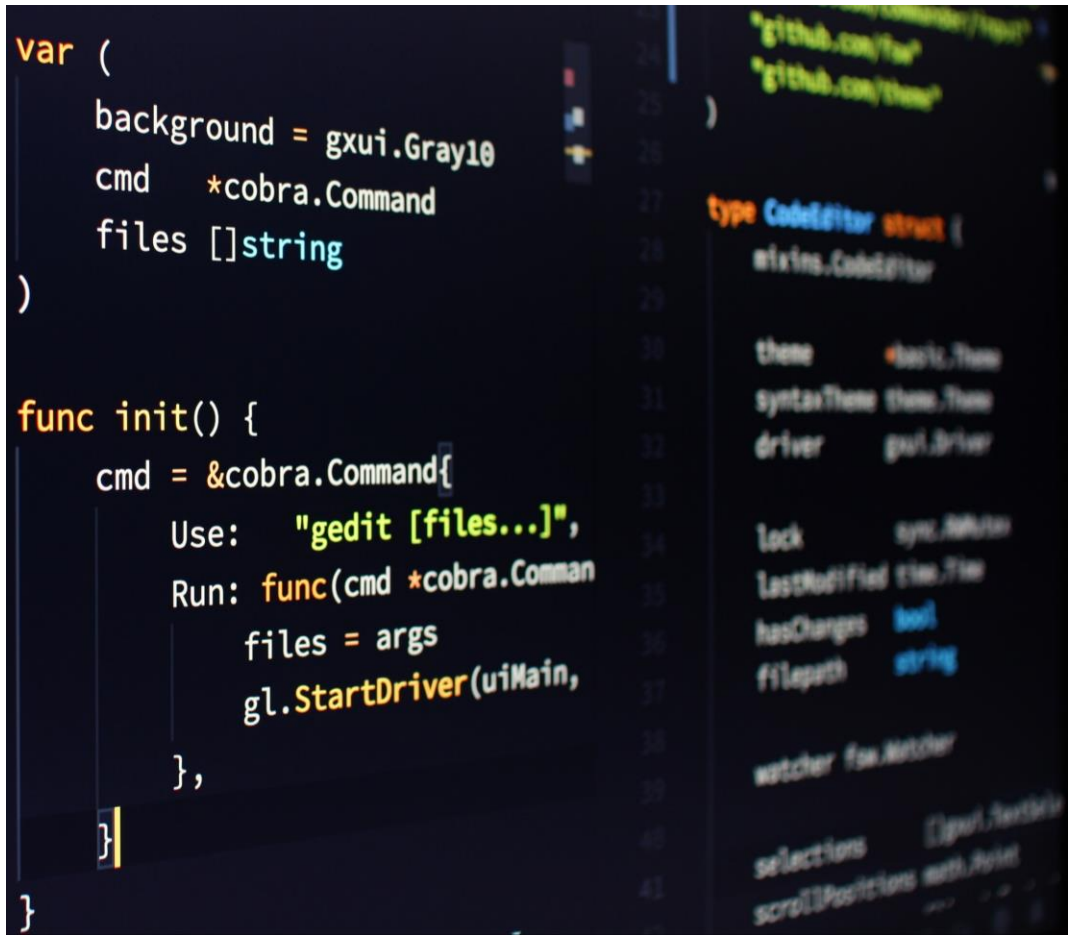
The Go Playground is an online web-based environment for Go Language development. A great sandbox for practicing the Go Language. Now you can develop in Go anywhere and 24/7.

You can use the Go Playground in class as the IDE.

The Go Playground is found at:

https://go.dev/play

CODE



The code presented in this course may not be robust. For example, panic and error handling is not included in some of the sample code. Instead, the emphasis is demonstrating a relevant topic and clarity.

Before using sample code in your applications, be sure to harden the code to proper quality standards.

COURSE AGENDA

1. Types
2. Operators
3. Functions
4. Generics
5. C2Go
6. Closures
7. Arrays / Slices
8. Maps
9. Interfaces
10. Duck typing
11. Concurrency
12. Modules

LABS

Learning is better when hands-on. Some of the modules have a companion lab reinforcing a kinesthetic learning experience.

- Labs review and reinforce important concepts
- Don’t be surprised - many of the labs extend the topics introduced in the module
- The labs offer an opportunity for real-world experience
- Pair programming can be effective when working on the labs

YOUR CLASS!

Yes, this is your class. What does this mean? You define the value.

- What is the most important ingredient of class – your participation!
- Your feedback and questions are always welcomed.
- There is no protocol in class. Speak up anytime!
- We welcome your comments during and after class. Just email dmarshall@innovationinsoftware.com.

Lab completed

The Basics

Go Programming

INTRODUCTION

This module is a quick tour of the Go language. You will learn the basics of a Go application with examples.

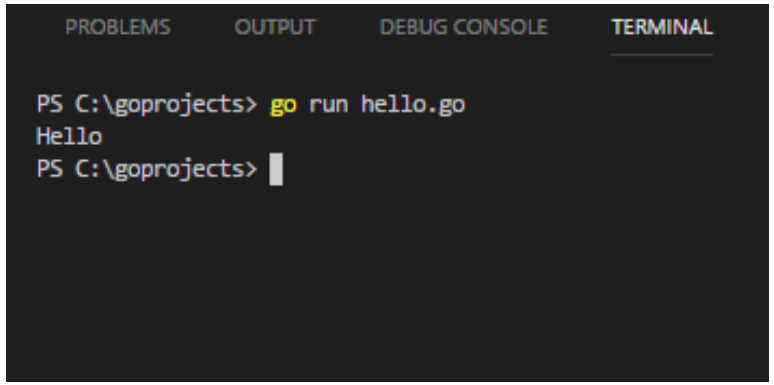
With this foundation, we can build upon that knowledge in later modules.

Most programming classes have an “Hello, world!” program or some variation. This course is no different.

This is a hands-on class. Feel free to participate using your chosen IDE.

FIRST PROGRAM

More versatile version of the standard "hello, world" application.




```
package main

import (
	"fmt"
	"os"
	"strconv"
)
var hello = []string{"Hello", "Hola",
	"Bon Jour", "Ciao", "こんにちは"}

func main() {
	var index = 1
	if len(os.Args) > 1 {
		index, _ = strconv.Atoi(os.Args[1])
	}

	if(index < 1 || index > len(hello)) {
		index=1
	}
	fmt.Println(hello[index-1])
}
```

FIRST PROGRAM - 2

```
package main

import (
            "fmt"
            "os"
            "strconv"
)
```

The *package main* statement indicates the program is an executable, not a library.

The import statement imports the listed libraries.

- Fmt library. fmt.Format function
- Os library. os.Args variable
- Strconv library. strconv.Atoi function

FIRST PROGRAM - 3

```
var hello = []string{"Hello", "Hola",
                     "Bon Jour", "Ciao", "こんにちは"}
```

The *var* statement declares a variable.

The hello variable is a slice, dynamic array. It consists of a sequence of Unicode strings.

FIRST PROGRAM - 4

```
func main() {
        var index = 1
}
```

The function *main* is an entry point for an executable. Functions are preceded with the *func* keyword. It has no parameters and returns nothing.

This is where your code starts executing. Your primary path of execution.

FIRST PROGRAM - 5

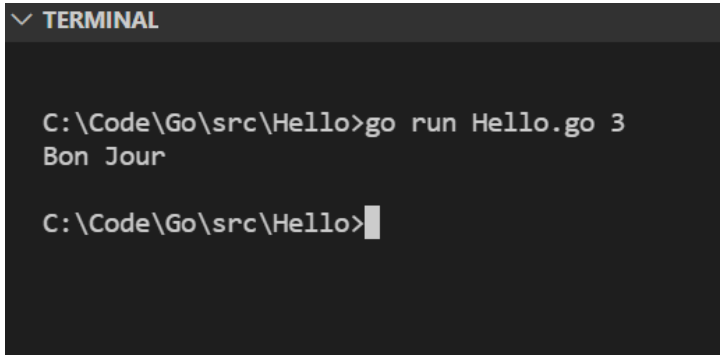
```
        if len(os.Args) > 1 {
                index, _ = strconv.Atoi(os.Args[1])
        }

        if(index < 1 || index > len(hello)) {
                index=1
        }
        fmt.Println(hello[index-1])
}
```

Go supports a variety of transfer of control statements, which evaluate Boolean expressions (true and false).

Here is pseudocode translation:

```
If(len(cmdline arguments) > 1 {
      index, ret=
            convert string to integer(second cmldline arg)
If(index<1 || index > len(hello slice)
      index=1
print hello[index-1]
```

FIRST PROGRAM - 6

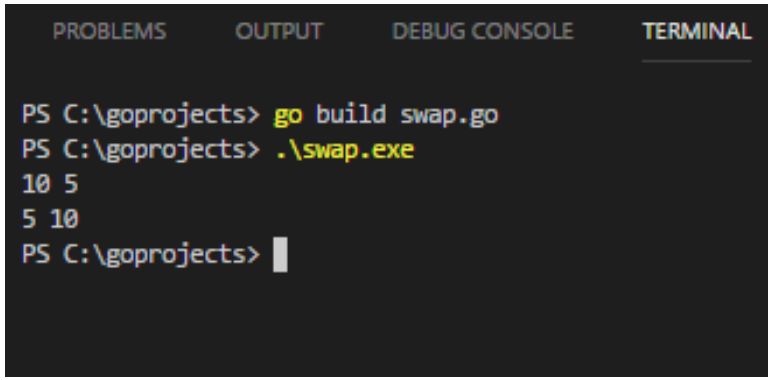


Here are instruction to run the hello application using the VS Code terminal window.

- Open the Terminal window from the View menu.
- From the command line prompt, change to the hello directory.
- Run the application using the go run command.

SECOND PROGRAM

Swap two values by-value and then by-pointer.




```
package main

import "fmt"

func main() {
    a := 5
    b := 10
    a, b = swap1(a, b)
    fmt.Println(a, b)

    swap2(&a, &b)
    fmt.Println(a, b)
}

func swap1(x int, y int) (int, int) {
    return y, x
}

func swap2(x *int, y *int) {
    *x, *y = *y, *x
    return
}
```

SECOND PROGRAM - 2

```
package main

import "fmt"

func main() {
}
```

This program is an executable – package main.

Imports the fmt library.

The main function is the entry point for the executable.

SECOND PROGRAM - 3

```
func swap1(x int, y int) (int, int) {
    return y, x
}
```

The swap1 function accepts to integer parameter by value. The function also returns to integers.

Notice the parameters are returned swapped.

SECOND PROGRAM - 4

```
func swap2(x *int, y *int)
{
    *x, *y = *y, *x
    return
}
```

The swap2 function accepts two integer parameters by pointer (*). Pointers provide direct access to the memory of a variable. In an expression however, the asterisk dereferences the pointer and returns the value.

The assignment (=) statement assigns a value to one or more variables.

The return statement ends the function. If no value is return from a function, the return statement is optional.

SECOND PROGRAM - 5

```
func main() {
    a := 5
    b := 10
    a, b = swap1(a, b)
    fmt.Println(a, b)
}
```

In main, two variables are declared with the short declaration statements.

The swap1 function is called and two values are returned and assigned to variables a and b. The parameters are passed by value. Changes within the swap1 function are lost unless returned.

SECOND PROGRAM - 6

```
func main() {
    a := 5
    b := 10
    a, b = swap1(a, b)
    fmt.Println(a, b)

    swap2(&a, &b)
    fmt.Println(a, b)
}
```

The swap2 function accepts a pointer to the variables a and b, by pointer. For that reason, changes in the function are preserved outside the scope of swap2.

The values are swapped within the swap2 functions and variable a and b are updated.

Lab 2- Fibonacci

FIBONACCI

The Fibonacci series is a set of numbers where each element of the series is the total of the previous two items. The first two numbers are of the series are 0 and 1. The origin of the name is Filius Bonacci (i.e., Fibonacci). He was an Italian mathematician during the middles ages that died in 1250.

Here is a partial sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, and so on

GENERATE FIBONACCI

- Create a function that displays a segment of the Fibonacci series.
- The function has two parameters: starting and ending point
- The function returns the Fibonacci series between the starting and ending points inclusively
- An integer slice and the append method might be helpful
- Maximum ending point is 100,000
- In main:
  - Read beginning and ending values from command-line
  - Call Fibonacci function
  - Display results

Fibonacci 100 300

Lab completed

The Basics

Go Programming

INTRODUCTION

Go Language supports a robust assortment of types. You can also create custom and aggregate types, which are discussed in a later module. This module reviews standard numeric and Boolean types, which are implicit to the language.

Types are used to reserve memory for an application, which is assigned an identifier. Identifiers label memory and provide a reference to a value in a program.

This module also reviews various operations, including Boolean, arithmetic, and bitwise operations.

STANDARD TYPES

Type	Description
bool	true or false
byte	int8
float32	32-bit floating point variable.
float64	64-bit floating point variable.
int	Depends on implementation
int8	-128 to 127
int16	-32,768 to 32,767
int32	-2,147,483,648 to -2,147,483,647
int64	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,807
rune	int32

Type	Description
string	Unicode string with UTF-8 encoding
uint	Depends on implementation
uint8	0 to 255
uint16	0 to 65,535
uint32	0 to 4,294,967,295
uint64	0 to 18,446,744,073,709,551,615

ARITHMETIC OPERATORS

These are the arithmetic operators in Go Language. This assumes L1 and R1 are compatible types.

Type	Description
+	Addition: L1 + R1
-	Subtraction: L1 + R1
+	Positive: +L1
-	Negation: -L1
*	Multiplication: L1 * R1
/	Division: L1 / R1
%	Remainder: L1 % R1
++	Postfix increment: L1++
--	Postfix decrement: L1--

MAXIMUM SIZE

The minimum and maximum size of many standard types are available in the math package.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("MaxFloat32\t", math.MaxFloat32)
    fmt.Println("MaxFloat64\t", math.MaxFloat64)
    fmt.Println("MaxInt16\t", math.MaxInt16)
    fmt.Println("MaxInt32\t", math.MaxInt32)
    fmt.Println("MaxInt8\t", math.MaxInt8)
    fmt.Println("MinInt16\t", math.MinInt16)
    fmt.Println("MinInt32\t", math.MinInt32)
    fmt.Println("MaxInt64\t", math.MaxInt64)
}
```

VARIABLES

Declaring a variable allocates a specific amount of memory for a declared type and value. You can declare variables using the var statement. A variable declaration consists of an identifier, type, and value. The type and value are optional and often implied. The type can be inferred from the value, which is different than dynamic. The value can default to some variation of zero. The type and value cannot both be inferred.

The identifier is essentially a label for the memory location where the value of the variable resides.

VARIABLES - 2

Variables reference a value in memory. You can use the ampersand (&) to get the memory address (pointer) of a variable. More about pointers later.

Pointers are a valid type. You can assign pointers to variables of this type.

```
package main

import "fmt"

func main() {
    var xyz int = 5
    var pxyz *int

    pxyz = &xyz
    fmt.Println(pxyz, *pxyz)
}
```

```
0xc04200e090 5
```

DECLARATIONS

Here are various syntax to declare a variable.

declare a variable of a specific type (1)

declare a variable with a default value (2)

declare a variable with type inferred (3)

declare multiple variables of the same type (4)

declare multiple variables of different types (5)

```
package main

func main() {

    var a int = 15          1

    var b int               2

    var c = 15              3

    var f, g int = 5, 6     4

    var (
        h = 10              5
        i = "test"
        j int
    )
}
```

SHORT VARIABLE DECLARATIONS

You can declare variables using type inference with short variable declarations (:=), which is an abbreviated syntax. This is the preferred syntax of professional Go programmers.

Short variable declarations must be done within a function and not at file scope (i.e., global).

```
package main

func main() {

    a := 1

    b, c := 5, 6

    d, e, f := 1, true, 4.56

}
```

NEW

In Go, both primitive and non-primitive types—such as slices, maps, structs, and interfaces—when allocated with the new keyword may either be placed on the stack or the heap depending on escape analysis.

The new keyword will return a pointer to the new value, wherever it resides.

Important: Go language has garbage collection for heap memory.

```
var a int
pa := new(int)
fmt.Printf("Type (pa) = %T\n", pa)
fmt.Printf("a = %v pa=%v", a, pa)
```

```
pa=nil
```

IDENTIFIERS

Identifiers are used to identify entities in the Go Language, such as types, variables, functions, and so on.

- Identifiers are case sensitive
- Can consist of letters, digits, and some special characters are allowed.
- First character must be a letter
- _ is the blank identifier for unused return values
- Entities starting with uppercase are considered public / exported

DEFAULT VALUES

Uninitialized variables are assigned a default value, which is a variation of zero.

- Numeric types : 0
- Boolean types : false
- Strings : “”

```
package main

import "fmt"

func main() {
        var i int
        var f float64
        var b bool
        var s string
        fmt.Printf("%v %v %v %q\n",
                i, f, b, s)
}
```

CONSTANTS

Constants are read only variables at run time. You can define constants with literals or expressions involving other constant entities.

You define a constant with the const keyword as a prefix to the declaration; instead of var.

```
package main

func main() {
    const a int = 5

    const b int = 10 * a

    // Does not work
    var d int = 10
    const c int = 10 * d
}
```

UNTYPED CONST

```
1   package main
2
3   func main() {
4       const a int = 5
5
6       const b = 5
7
8       var c float32 = a
9
10      var d float32 = b
11  }
12
```

Untyped const variables rely on type inference. But there is another important difference. Typed const can only be used with other entities of the same type. Untyped const can be used in an expression with entities of *similar* type.

ENUMERATION

The Go Language does not support enum types. Enum types are typically used for flags in C based languages. However, there are three approaches to create something similar in Go.

Declare a constant for each flag (1)

abbreviation of syntax 1 (2)

use the iota keyword to increment flags starting at zero for the first flag (3)

```
const Bold = 1
const Underline = 2
const Italics = 3

const (
    Small = 0
    Medium = 1
    Large = 2
    XLarge = 3
)

const (
    South = iota
    East
    North
    West
)
```

1

2

3

BITWISE

You can create a bitwise enumeration using the bitwise operators.

You can even use bitwise operators with the iota expression. For example, the left shift (1 <<) works with iota.

You can then use bitwise operators (| and &) with the bitwise value.

```
package main

import "fmt"

func main() {

    type BitFlag int
    const (
        Bold BitFlag = 1 << iota
        Underline
        Italics
    )

    result := Underline | Italics

    fmt.Println("Flags", Bold, Underline, Italics)
    fmt.Println("Result", result)
}
```

BITWISE OPERATORS

Operator	Description
^	Bitwise complement: ^L
&	Bitwise And: L1 & L2
|	Bitwise Or: L | R
^	Bitwise Xor: L ^ R
&^	Bitwise clear: L &^ R
<<	Left shift: L << R
>>	Right shift: L >> R
…	Compound operators

POP QUIZ: WHAT IS THE RESULT?

```
package main

import "fmt"

func main() {

    i := 10

    j := i >> 1

    k := i | j


    fmt.Println("j", j, "k", k)

}
```

10 MINUTES

BOOLEAN

There are two Boolean values: true and false. There is strict type checking of Boolean types. Most notable, zero and non-zero integral values cannot be cast to Boolean types.

There are a variety of Boolean operators as listed next. As an efficiency, the Boolean operators will short circuit when possible. Prevent side affects of short circuiting by fully evaluating Boolean expressions before the Boolean operation. For example, do not call a function within a Boolean expression. Call beforehand and use the function results in the Boolean expression.

The Boolean operators can be used with similar types: Booleans, numbers, structs, and arrays. However, you cannot use the Boolean operators to compare slices.

POP QUIZ: WHICH EXPRESSIONS WILL SHORT CIRCUIT?

```
func AFunc() bool {
    return true
}

func BFunc() bool {
    return false
}
```

```
var bool1 = AFunc() || BFunc()      1
var bool2 = AFunc() || true         2
var bool3 = false && BFunc()        3
var bool4 = AFunc()                 4
var bool5 = BFunc() || AFunc() || false  5
```

10 MINUTES

BOOLEAN OPERATORS

Operator	Description
==	Equality: L == R
!=	Non-equality: L != R
!	Not: !L
||	Or: L || R
&&	And: L && R
<	Less than: L < R
>	Greater than: L > R
<=	Less than or equal: L <= R
>=	Greater than or equal: L >= R

INTEGER

Go Language offers 11 different integral types – both signed and unsigned. The byte type is a synonym for uint8, while rune is a synonym for int32. Depending on your implementation, int may be 32 bit or 64 bit. Except for reading and writing persistent data, Go Language developers typically use int.

Expressions require the same integer types. You can cast to create compatible types.

SIZE OF INT

Get the size of the int type for your implementation using the Sizeof method, which is in the unsafe package.

```
package main

import "fmt"
import "unsafe"

func main() {

    var i int = 1

    fmt.Printf("Size of i is: %d",
           unsafe.Sizeof(i))

}
```

OVERFLOW

What happens when the value of an integral variable exceeds its bounds? Integral variables are circular, and the value simply wraps. For example, from the maximum positive value to the maximum minimum value.

```
package main


import "math"

import "fmt"


func main() {

    var value int64 = 0

    value = math.MaxInt64

    value++

    fmt.Println("value =", value)

}
```

```
C:\Code\Go\src\overflow>go run overflow.go
value = -9223372036854775808
```

OVERFLOW - 2

For overflows, the overflow package is helpful and has methods that check for overflow.

Here is the statement to download the library.

```
go get github.com/johncgriffin/overflow
```

The overflow.Add method is an excellent example of handling an overflow. It performs an addition operation. If there is an overflow, the second return value returns false. The first return value contains the results.

```
// go get github.com/johncgriffin/overflow


package main
import "math"
import "fmt"


import "github.com/JohnCGriffin/overflow"


func main() {
    value, ok := overflow.Add(math.MaxInt64, 1)
    if !ok {
        fmt.Println("overflow occured")
    }
    fmt.Println("value ", value)
}
```

```
C:\Code\Go\src\overflow2>go run overflow.go
overflow occured
value  -9223372036854775808
```

WANT SOMETHING BIG

Do you need a bigger version of an integer or float? You will find it in math/big package.

The big package contains the types Int and Float, which as essentially unlimited in size. Using these types is an effective way to avoid known overflows.

You can indirectly cast int and float64 to big versions (i.e., Int and Float) using the NewInt and NewFloat functions respectively.

WANT SOMETHING BIG – 2

This is example code for a big type: Int.

- Create a big integer, bigval, on the heap using the new method.
- Create two big integers, op1 and op2, using the big.NewInt method. Initialize both with a maximum 64-bit integer.
- Multiply the two big integers, op1 and op2, with bigval.Mul. Place the results in bigval and op3.
- Display the results.

```
bigval := new(big.Int)

bigval.SetInt64(123)

fmt.Println("bigval = ", bigval)

op1 := big.NewInt(math.MaxInt64)
op2 := big.NewInt(math.MaxInt64)
op3 := bigval.Mul(op1, op2)
fmt.Println(bigval, op1, op2, op3)
```

FLOAT

Floating point data is stored in IEEE-754 format.

https://en.wikipedia.org/wiki/IEEE_754

Unlike integers, there is no generic float type. You must specify either float32 or float64. The float64 type is the most common floating point type.

When cast to an integer (i.e., int(f)), the fractional part of a float is discarded or truncated. There is no automatic rounding.

CASTING

Explicit casting is supported in the Go Language. Assuming a compatible type, you can cast using the type name.

```
package main

import "fmt"

func main() {
    var a = 12.55
    var b = int(a)
    var c = float64(b)
    var d = []byte("test")
    var e = []rune("test")

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

COMMENTS

Go Language supports C++ style comments. Single line comments are (//). This is a comment to end of line. Multi-line comments are (/* */).

There are two philosophies on commenting source code: traditional versus agile.

LANGUAGE BUILDING BLOCKS

KEYWORDS

break	case	chan
const	continue	default
defer	else	fallthrough
for	func	go
goto	if	import
interface	map	package
range	return	select
struct	switch	type
var		

IDENTIFIERS

append	bool	byte
cap	close	complex
complex64	complex128	copy
delete	error	false
float32	float64	image
int	int8	int16
int32	int64	iota
len	make	new
nil	panic	print
println	real	recover
rune	string	true
uint	uint8	uint16
uint32	uint64	uintptr

SCAN FUNCTIONS

Scanln defaults to reading from the console / terminal. Notice that Scanln accepts a pointer to a string. This is an out parameter. If the string is empty, the user entered a blank line for input.

```
package main

import (
    "fmt"
)

func main() {

    for {
        var s string
        fmt.Print("Enter text: ")
        fmt.Scanln(&s)
        fmt.Println(s)
        if len(s) == 0 {
            print("done")
            break
        }
    }
}
```

Lab 3 – Guess number

GUESS A NUMBER

In this program, a user guesses a random number between 1 and 5 inclusively. You will seed the random number using the current time.

If the user guesses the random number correctly, display a winning message:

1. In main function, get the current time using the time package: time.Now method. The time.Now function returns a “time” type.
2. Create a random number using math/rand package. Create a new random number generator using the rand.NewSource method, which returns a “source” type. From Step 1, use the Second attribute on the time variable to obtain the seed for the NewSource method.
3. Generate a random number using the generator. Call source.Intn and get a random number between 0 and 4.
4. Add 1 to the random number.

GUESS A NUMBER

5. Prompt the user to enter a number between 1 and 5 inclusive.
6. Convert user response to an integer using Atoi, found in the strconv package.
7. If there is an error converting the number, display error and quit.
8. Compare input value to random number.
    a) If equal, display a message that the user has guessed correctly.
    b) If not equal, display an appropriate message and the correct value (random number).

Lab completed

PANIC

defer

WHY DOES GO NOT HAVE EXCEPTIONS?

Some believe that the control structure of exception handling, as in the try-catch-finally idiom, can contribute to convoluted code. It also tends to encourage reactive code, versus proactive error handling.

For error handling, Go functions should be proactive. Typically, a function returns two values: return value and error value. If no error occurs, error is set to nil. Otherwise, an error value is returned.

Sometimes error handling is not possible. For exceptional events, Go uses panics instead of exceptions.

Important. You an create an error object with the fmt.Errorf function.

DEFER

The defer statement helps manage panics.

The defer statement defers execution of a function. A deferred function executes at the end of the current block (i.e., scope).

You can have more than one defer statement at the same scope. They are executed Last In First Out (LIFO).

In the deferred function, cleanup can be performed when there is a panic, such as closing a file.

```
package main

import "fmt"

func main() {

    defer fileCleanup()
    defer releaseConnections()

    fmt.Println("in main...")
}

func fileCleanup() {
    fmt.Println("doing file cleanup")
}

func releaseConnections() {
    fmt.Println("releasing connection")
}
```

DEFER IN BLOCK

Deferred methods are associated with a function not a block.

```
package main

import (
    "fmt"
)

func main() {

    {
        defer fmt.Println("test")
    }
    fmt.Println("test2")
}
```

PANIC / RECOVER

In the Go Language, recover is called in a deferred method to handle an unplanned error (panic). If desired, you can call the panic function to force a panic.

The idiomatic way for handling known problems is to return an error object. If multiple return values, the last return value should be the error.

```
func handler() {
    err := recover()
    if err != nil {
        fmt.Println("***", err.(error))
    }
}

func main() {
    defer handler()
    err := fmt.Errorf("my error")
    panic(err)
}
```

PANIC / RECOVER – 2

Here are the steps to handle a panic.

1. Panic raised – either implicitly or panic function
2. Execution in the current function immediately stops
3. Stack walk occurs
4. For each stack frame, including the current function, deferred methods are called before exiting.
5. If the recover method is called within a deferred method, the panic is considered recovered.
6. Check the error object returned from the recover method.
7. Execution continues within the calling function of the current stack frame.
8. If recover functoin not called during the stack unwind, the application will eventually terminate.

PANIC / RECOVER - 3

```
package main

import "fmt"

func main() {
    funcA()
    fmt.Println("finishing...")
}
func funcC() {
    fmt.Println("funcC")
    panic("Error!")

}

func funcB() {
    defer defer1()
    funcC()
}
```

```
func funcA() {
    defer defer2()
    funcB()
}
func defer1() {
    fmt.Println("defer1")
}

func defer2() {
    s := recover()
    fmt.Println("recovered from", s)
}
```

CHECKING FOR A SPECIFIC PANIC

Handling unknown errors can pose a risk. You can check for a specific panic by inspecting the panic message, which is a String.

In a defer function, the recover method probably returns the error object as an *any* type. For that reason, you do not have immediate access to the *error* interface. The error object implements the String method to implicitly return the error message.

If the error is not handled in the deferred method, repanic the error to continue walking the call stack. Repanic using the panic statement.

SPECIFIC PANIC - EXAMPLE

```
package main
import (

    "fmt"

    "os"

    "strings"

)
func main() {
    defer func() {

        err := recover()  // any {}

        var message = err.(error).Error()


         if strings.HasPrefix(message, "runtime error: index out of range") {

            fmt.Fprintf(os.Stderr, "Exception handled\n")

        } else if message != "" {

            fmt.Fprintf(os.Stderr, "Unplanned Exception: %v\n", message)

            panic(err)

        } else {

            fmt.Fprintf(os.Stderr, "Exception unknown\n")

            panic(err)

        }

    }()


    stuff := []int{1, 2, 3, 4}

    stuff[6] = 12
}
```

ERROR NEW

Many functions return an error object. Error types implement the Error and Stringer interface which is:

```
type error interface {
    Error() string
}
```

You can create an instance of the error object using the errors.New function. The function accepts a string and returns and error object.

```
func Something(f float64) (float64, error) {
      if f < 0 {
          return 0, errors.New(
                 "some error")
      }
      // implementation
      return answer, nil
}
```

CUSTOM ERROR TYPE

You can create a custom error interface by inheriting the error interface and adding whatever is necessary. For a network error for example, you might include a timeout attribute method..

```
package net


type NetworkError interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Lab completed

Control

if, for, switch, and panic

BRANCHING

Branching and arcs are necessary for real world applications. Top-down programming is not practical in all scenarios. However, too much reliance on branching can add to the complexity of your application.

Go has the basic control structures, such as the if and for statement. However, some common control elements are missing:

- while
- do..while
- ternary operator

IF STATEMENT

The if statement evaluates a bool expression. If true, the if block is executed. If false, the if block is not run.

Here is the syntax:

```
if optionalStatement; booleanExpression {
        optionalStatements
}
```

```
package main

import "fmt"

func main() {
    if a:=5; a > 4 {
        fmt.Println(a)
    }
}
```

IF ELSE STATEMENT

The if statement evaluates a bool expression. If true, the if block is executed. If false, the else block is performed.

Here is the syntax:

```
if optionalStatement; booleanExpression {
          optionalStatements
} else {
          optionalStatements
}
```

```
package main

import "fmt"

func main() {

    if a := 5; a > 4 {
        fmt.Println(a)
    } else {
        fmt.Println("cool!")
    }
}
```

IF ELSE IF STATEMENT

The if else if construct is a nested if statement. The construct is more efficient and cleaner than a series of mutually exclusive if statements.

```
package main

import "fmt"

func main() {
    a := 5

    b := 10

    if a > b {

        fmt.Println("a")

    }

    if b > a {

        fmt.Println("b")

    }

    if b == a {

        fmt.Println("a equal b")

    }


    // Alternative
    if a > b {

        fmt.Println("a")

    } else if b > a {

        fmt.Println("b")

    } else {

        fmt.Println("a equal b")

    }

}
```

SWITCH STATEMENT

Switch statements switch on a value where execution jumps to a matching statement. The switch statement transfers execution based on an expression, including string expressions.

- The case expressions must be consistent.
- There is no automatic fall through between case statements.
- The default statement is the else statement of the switch

.

The syntax for a expression switch is:

```
switch optionalStatement; valueExpression {
            case expressioncommalist1:
                        optionalStatements
            case expressioncommalistn:
                        optionalStatements
            default:
                        optionalStatements
}
```

SWITCH STATEMENT - 2

Here are a couple of examples of expression switches.

```
package main

import "fmt"

func main() {

    val := "str"
    switch val {
    case "test":
        fmt.Println("no")
    case "str":
        fmt.Println("good")
    }

    val2 := true
    switch val2 {
    case true:
        fmt.Println("true")
    case false:
        fmt.Println("false")
    default:
        fmt.Println("impossible!")
    }
}
```

SWITCH WITH EXPRESSIONS

You can use expressions as a case in a switch block. They do not have to be a constant.

```
package main

import (
    "fmt"
)

func main() {

    a := 5
    b := 10

    val := 9
    b--
    switch val {
    case a:
        fmt.Println(a)
    case b:
        fmt.Println(b)
    case a + b:
        fmt.Println(a + b)
    }
}
```

SWITCH CONTINUATION

By default, execution does not continue between cases within a switch statement, such as in C++. You can explicitly continue to the next case statement with the fallthrough keyword. See the opposite code.

```
package main

import "fmt"

func main() {
    const (
        Critical = iota
        Error
        Warning
    )
    val := Critical

    switch val {
    case Critical:
        fmt.Println("Critical")
        fallthrough
    case Error:
        fmt.Println("Error")
        fallthrough
    case Warning:
        fmt.Println("Warning")
    }
}
```

TYPE SWITCH

Type switch statements switch on the type of a value where each case is a specific type. The value should be an interface {} type.

Interface {} is discussed in a later module.

.

The syntax for a type switch is:

```
switch optionalStatement; typeExpression.(type) {
            case typecommalist1:
                        optionalStatements
            case typecommalistn:
                        optionalStatements
            default:
                        optionalStatements
}
```

TYPE SWITCH - 2

Here is an example of the type switch.

```
package main

import "fmt"

type MyStruct struct {
    a int
    b int
}

func main() {
    var xyz MyStruct

    var val interface{} = xyz
    val.a=5 // error!!!
    switch val.(type) {
        case string:
            fmt.Println("string")
        case int, int16, int32, int64:
            fmt.Println("int")
        case MyStruct:
            fmt.Println("MyStruct")
    }

    fmt.Println(xyz)
}
```

FOR LOOP

There are several syntax in the Go Language for the for loop.

Here are the various syntaxes:

```
for {
		// infinite loop
		optionalStatements
}

for optionalPrestatement; boolExpression; optionalPostStatement {
		// C style for loop
		optionalStatements
}

for booleanExpression {
		// while loop
		optionalStatements
}

for index, value:=range collection {
		// iteration
}
```

BREAK / CONTINUE

The break statement terminates a for, switch, or select statement.

The continue statement continues to the next iteration of a for loop.

Collections

arrays, slices, and maps

INTRODUCTION

Go language has excellent support for sequences.

Arrays and slices are the basic sequences. Arrays are fixed length, while slices are dynamically sized. Both arrays and slices contain scalar values of the same type.

Map is a more complex sequence and a built-in type. Maps are often referred to as a dictionary or lookup table. Maps are also presented in this module.

Sequences are iterable where elements are accessible individually using indexes or for loops.

This module presents the differences between arrays and slices. This includes sub-slices, appending, and sorting.

ARRAYS

Arrays in the Go Language are a fixed length collection of elements of the same type. You can create either single- or multi- dimensional arrays.

Elements of an array are accessible via the index operator ([]). Treat the index as an offset from the beginning of the array. The first element of the array is located at index 0.

There are various syntaxes for declaring an array:

```
[length] type
[length] type { value1, value2, valuen}
[…] type { value1, value2, valuen}
```

ARRAYS - 2

Sample code for creating and manipulating arrays.

.

```
package main

import "fmt"

var a1 [5]int
var a2 = [5]int{1, 2, 3, 4, 5}
var a3 = [...]int{1, 2, 3, 4, 5}
var a4 = [2][3]int{{1, 2, 5}, {3, 8, 4}}

func main() {
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
    fmt.Println(a4)
    for i := 0; i < len(a2); i++ {
        fmt.Println(a2[i])
    }
}
```

FOR LOOP

You can iterate elements of an array using a for..range loop.

.

```
package main

import "fmt"

func main() {

    a1 := [...]int{11, 12, 13, 14, 15}
    for a, b := range a1 {
        fmt.Println(a, b)
    }
}
```

SLICES

A slice is a dynamic collection of elements that are the same type.

- Every slice has an underlying array, which defines the capacity.
- Using the interface type interface {} , you can actually store anything in a slice or array.

Here are methods to create a slice:

```
make([] type, length, capacity)
make([] type, length)
[] type {}
[] type {value1, value2, valuen}
```

SLICES - 2

Sample code for creating and manipulating slices, even a multi dimensional slice.

.

```
package main

import "fmt"

var a1 = make([]int, 5, 10)
var a2 = []int{1, 2, 3, 4, 5}
var a3 = [][]int{{1, 2, 5}, {3, 8, 4}}

func main() {
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
}
```

SLICE CAPACITY

Slice capacity is either set implicitly or explicitly, using the make keyword.

When the capacity of slice is exceeded, the underlying array is replaced with a new array that is twice the size. The old array is then copied into the new array. Basically, the capacity is doubled whenever necessary.

What is the result of this example code?

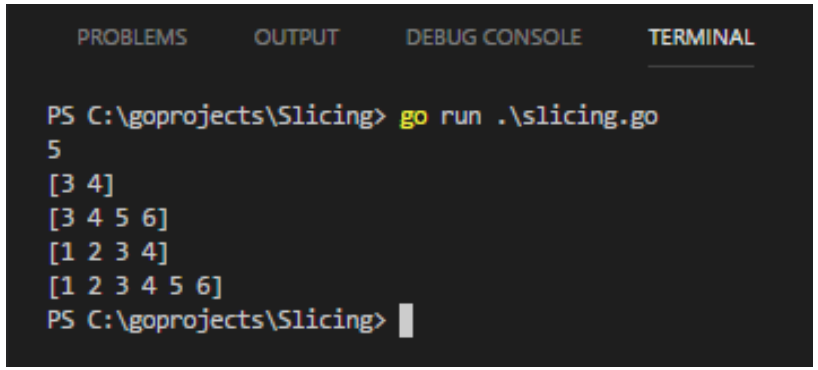
```
s1 := make([]int, 4, 5)
fmt.Println("Len ", len(s1), "Capacity ", cap(s1))


s1 = append(s1, 5)
s1 = append(s1, 5)
fmt.Println("Len ", len(s1), "Capacity ", cap(s1))


s1 = append(s1, 5)
s1 = append(s1, 5)
s1 = append(s1, 5)
s1 = append(s1, 5)
s1 = append(s1, 5)
s1 = append(s1, 5)
fmt.Println("Len ", len(s1), "Capacity ", cap(s1))
```

SLICE OPERATIONS

Slice operations can be applied to either arrays or slices. Here are some slice operations.

Operation	Description
s[n]	Return item at index
s[n:m]	Slice from n to m-1
s[n:]	Slice from n to end
s[:m]	Slice beginning to m-1
s[:]	Slice from beginning to end
cap(s)	Return the capacity
len(s)	Return the length

SLICE OPERATIONS - 2




```
package main

import "fmt"

func main() {
    myslice := []int{1, 2, 3, 4, 5, 6}

    a := myslice[4]

    b := myslice[2:4]

    c := myslice[2:]

    d := myslice[:4]

    e := myslice[:]

    fmt.Println(a)

    fmt.Println(b)

    fmt.Println(c)

    fmt.Println(d)

    fmt.Println(e)
}
```

APPEND

Arrays are fixed length and cannot be resized. However, slices are dynamically sized and can be expanded at runtime, but not shortened. Call the append method to expand a slice.

For an array, convert an array to a slice – then expand the slice. This syntax converts an array to a slice:

slice:=array[:]

APPEND OPERATIONS

an executable (1)

import fmt package (2)

create a slice of six integers (3)

create a slice of three integers (4)

append three elements to a slice(5)

append a slice (6)

append a partial slice (7)

display results (8)

```
package main          1

import "fmt"          2

func main() {
    myslice := []int{1, 2, 3, 4, 5, 6}        3
    myslice2 := []int{20, 21, 22}             4
    a := append(myslice, 7, 8, 9)             5
    b := append(myslice, myslice2...)         6
    c := append(myslice, myslice2[:2]...)     7
    fmt.Println(a)
    fmt.Println(b)        8
    fmt.Println(c)
}
```

POP QUIZ: WHAT IS DISPLAYED?

```
package main

import (
    "fmt"
)

func main() {

    slice1 := []int{1, 2, 3}
    slice2 := slice1[1:3]

    slice2[0] = 5

    fmt.Println(slice1)
}
```

5 MINUTES

COPY BETWEEN SLICES

When assigning a slice or sub-slice, be careful. You are referencing the underlying array and will create a dependency. This is a bug that is hard to isolate in code.

Use the copy function instead.

```
package main

import (
    "fmt"
)

func main() {

    slice1 := []int{1, 2, 3}
    slice2 := make([]int, 2)
    copy(slice2, slice1[1:3])

    slice2[0] = 5

    fmt.Println(slice1)
}
```

ARRAY INDEX VALUE PAIR

You can create index value pairs, which are different from maps. Index value pairs are restricted to *int key, type value*.

Here is the syntax:

```
[size]int{0: value, 1:value,
    n: value, …}
```

Indexes and value, or both, are optional. If not provided, the index and value will be extrapolated.

Be careful because the index value pair will extrapolate additional elements implicitly.

```
example1 := [...]int{1: 2, 2: 3, 3: 4}
for index, value := range example1 {
    fmt.Println(index, value)
}
```

ARRAY INDEX VALUE PAIR

Here are examples of index value pairs.

```
example2 := [...]int{10: 4}
for index, value := range example2 {
    fmt.Println(index, value)
}

example3 := [...]int{8: 7, 10: 4}
for index, value := range example3 {
    fmt.Println(index, value)
}

example4 := [...]int{9, 10, 10: 4}
for index, value := range example4 {
    fmt.Println(index, value)
}
```

ARRAY / SLICE COMPARISON

You can compare (=) arrays of the same type.

However, you cannot compare slices with a couple of exceptions:

- Slices can be compared to nil
- You can compare []byte slices with bytes.Equal

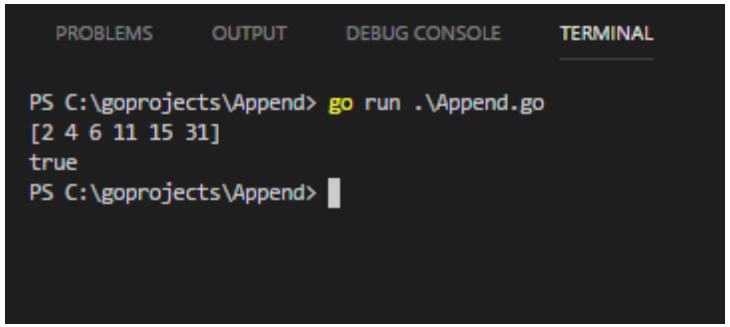
```
package main

import (
    "fmt"
)

func main() {

    array1 := [3]int{1, 2, 3}
    array2 := [3]int{1, 2, 3}

    if array1 == array2 {
        fmt.Println("equal")
    } else {
        fmt.Println("not equal")
    }

    slice1 := []int{1, 2, 3}
    slice2 := []int{1, 2, 3}

    if slice1 == slice2 {
        fmt.Println("equal")
    } else {
        fmt.Println("not equal")
    }
}
```

SORT OPERATIONS

You can sort slices based on type, such as sorting an int slice. Sort operations are naturally located in the Sort package. Here are the basic sort operations.

Operation	Description
Float64(fs)	Sorts []float64 in ascending order
Float64sAreSorted(fs)	Returns true if []float64 is sorted
Ints(is)	Sorts []int in ascending order
IntsAreSorted(is)	Returns true if []int is sorted
SearchFloat64s(fs, f)	Returns index position of f in fs
SearchInts(is, i)	Returns index position of i in is
SearchStrings(ss, s)	Returns index position of s in ss
Strings(ss)	Sorts []string in ascending order
StringsAreSorted(ss)	Returns true if []string is sorted

SORT OPERATIONS - 2




```
package main

import "fmt"
import "sort"

func main() {
    myslice := []int{11, 2, 31, 4, 15, 6}

    sort.Ints(myslice)

    fmt.Println(myslice)

    fmt.Println(sort.IntsAreSorted(myslice))
}
```

CUSTOM SORT

Create a custom sort using the sort.Sort method and providing a sort routine.

This is the sort interface.

```
type sort interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

```
package main

import "sort"
import "fmt"

type byLength []string

func (s byLength) Len() int {
    return len(s)
}

func (s byLength) Less(i, j int) bool {
    return len(s[i]) < len(s[j])
}


func (s byLength) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

func main() {
    fruits := []string{"peach", "banana", "kiwi"}
    sort.Sort(byLength(fruits))
    fmt.Println(fruits)
}
```

s

MAPS

Maps in Go Language are an unordered collection of key, value pairs where the keys are unique. The key should support the == and != operators. Most notably, slices cannot be used as keys.

Map keys must be of the same type. You can store values of different types as a map value using the interface{} type.

There are various methods to create a map:

```
make(map[keyType] valueType, initialCapacity)

make(map[keyType] valueType)

map[keyType] valueType{}

map[keyType] valueType {key1:value1, key2:value2, keyn:value2}
```

MAP BEHAVIOR

Maps support various behaviors:

- Add a key, value

  mymap[key]=value

- Change a value

  mymap[key]=value

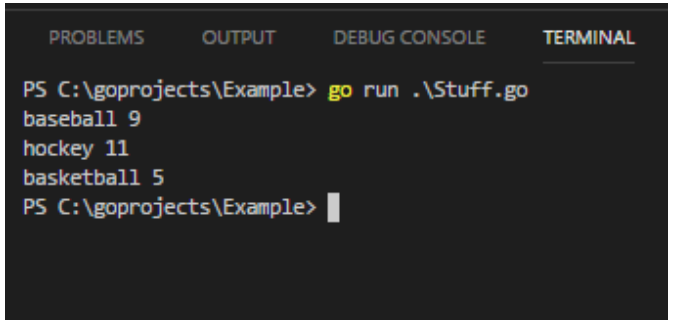
- Delete a value

  delete(mymap, key)

- Get a map value

  value, exists=mymap[key]

MAP EXAMPLE




```
package main

import "fmt"

func main() {
    teams := map[string]int{"basketball": 5,
        "football": 11, "baseball": 9}

    teams["hockey"] = 11

    delete(teams, "football")

    for team, players := range teams {
        fmt.Println(team, players)
    }

}
```

EMPTY SLICE

You can add elements to an empty slice. However, you cannot add elements to an empty map. For an empty map, adding elements causes a panic.

.

```
var fslice []float64
fslice = append(fslice, 1.0)
fmt.Println(fslice)

var mymap map[string]int
mymap["test"] = 1
```

MAP ENTRY OKAY

When accessing a map entry, a value is always returned. If not available, a variation of zero is returned.

There is a syntax that confirms if a default value has been returned. The first return value is the result. With the second return value, you can confirm whether a default value was returned: true or false.

```
package main

import (
    "fmt"
)

func main() {

    map1 := map[string]int{"a": 1, "b": 2}

    value, exist := map1["c"]

    fmt.Println(value)
    if !exist {
        fmt.Println("not okay!")
    }
}
```

CAP AND MAPS

You cannot use the cap function with maps – even if the capacity is specified.

.

```
var2 := make(map[string]int, 9)
fmt.Println(cap(var2))

var3 := make([]int, 5, 10)
fmt.Println(cap(var3))
```

SEMICOLON RULE

Like many languages Go Language statements have a semi-colon terminator, which is inserted implicitly. The lexer adds the semi-colon.

Like C, Go's formal grammar uses semicolons to terminate statements, but unlike in C, those semicolons do not appear in the source. Instead the lexer inserts semicolons. For that reason, the source code is mostly free of semicolons.

If a statement concludes with a token before a linefeed, a semi-colon is implicitly added as the terminator.

Semicolons are sometimes required. The best example are substatements, such as included in a for statement.

SEMICOLON RULE – 2

You must insert semicolons manually when placing more than one statement on a line, such as a for loop.

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

POP QUIZ: WHY IS THE FINAL COMMA NECESSARY?

```
var m4=map[string] int {
        "a":1,
        "b":2,
}
```

5 MINUTES

Functions

generics and lambda

CLOSURES

Anonymous functions are closures. They are also called function literals. Closures are a great solution where a named function is not required or late binding is helpful. You can use closures to initialize function parameters, return values, and more.

Closures capture constants and variables available at the scope where it is created; but only if it references them. This occurs even if the constant or variable is out of scope before the closure executes.

From other languages, closures are similar to a lambda or function pointer.

CLOSURES - 2

declare local variables (1)

declare closure and assign to a variable (2a)

closure captures two variables: a, b (2a)

execute closure (3)

create and execute closure in-place (4)

```
package main

import "fmt"

func main() {

    a := 5
    b := 10

    x := func() int {
        return a * b
    }

    fmt.Println(x())
    fmt.Println(func() int {
        return 42
    }())
}
```

1

2

3

4

CLOSURES - 3

Another example of a closure, where the inner function is capturing data of the outer function.

```
package main

import "fmt"

func AFunc() func() {
    a := 5
    return func() {
        a++
        fmt.Println(a)
    }
}

func main() {
    f := AFunc()
    f()
}
```

POP QUIZ: WHAT DOES THIS CODE DO?

```
package main

import "fmt"

func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}

func main() {
    f := squares()
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())
    fmt.Println(f())
}
```

10 MINUTES

FUNCTION POINTERS

You can declare a function pointer type with the type keyword. This is based on the function signature. The resulting function type can be used as a parameter or return type.

When initialized, you can invoke a function through a function pointer using the call operator "()".

```
package main

import "fmt"
import "os"

type fptr func(int) int

func main() {

    if len(os.Args) == 1 {
        funcExecute(double)
    } else {
        funcExecute(triple)
    }
}

func double(a int) int {
    return a * a
}
func triple(a int) int {
    return a * a * a
}

func funcExecute(f fptr) {
    fmt.Println(f(5))
}
```

BENEFITS

Generics is more than language syntax, but a change in the style of programming. Eventually, this feature will probably touch most aspects of the language. The positive impact of generics exceeds the benefits of parametric polymorphism. Here are the main benefits:

- DRY (Don't Repeat Yourself)
- Extensibility
- Type parameters type checked at compile type
- Reduction of casting

BEFORE GO 1.18

Using the interface{} type, you can create something similar to a generic method.

With reflection, you can make this approach safe.

```
package main

import "fmt"
import "reflect"

func main() {

    a, b := swap(10, 5.4)
    c, d := swap("dog", "cat")

    fmt.Println("a", a, "b", b)
    fmt.Println("c", c, "d", d)
}

func swap(first interface{},
        second interface{}) (interface{},
            interface{}) {
    ft := reflect.TypeOf(first).Kind()
    st := reflect.TypeOf(second).Kind()

    if ft != st {
        return nil, nil
    }

    return second, first
}
```

BEFORE GO 1.18 – ANOTHER EXAMPLE

Here is sample code that adds two integers and returns the result.

```
// Add two integer values
func add(a int, b int) int {
    return a + b
}
```

BEFORE GO 1.18 – ANOTHER EXAMPLE - 2

In this example, the add method is implemented for both the integers and strings. However, this requires writing two separate methods. For this reason, the solution is not extensible.

```
func add_i(a int, b int) int {
      return a + b
}
func add_s(a string, b string) string {
      return a + b
}
```

INTERFACE {}

The empty interface{} type is the “any” type in Go. This type is a blank canvas that can be used for anything. We have already discussed the any type, which replaces the empty interface{} type.

Empty interface{} types can be used as placeholders in structures and functions. You simply place the interface type anywhere a placeholder is desired.

ANY

Any type has replaced the empty interface type “interface {}” as the base type of everything. This makes Go more consistent with other languages. It is similar to the object type in many other languages.

The new *any* type and empty interface{} type are interchangeable.

```
package main

import "fmt"

func total(values ...any) {
	for index, value := range values {
		fmt.Printf("%d %v\n", index, value)
	}
}

func main() {
	total("AB", 12, 24)
}
```

BEFORE GO 1.18 – ANOTHER EXAMPLE - 3

When Version 1 is compiled, the following error message is received:

```
invalid operation: operator + not defined on
          a (variable of type interface{}
```

The reason for the error is that the empty interface is abstracted and has minimal understanding of the underlying type. Without that knowledge, the Go compiler cannot determine when the operator + is supported.

Resolve the problem with a type cast, as show here as shown in Version 2.

```
// Version 1

func add(a interface{}, b interface{}) interface{} {
      return a + b
}

// Version 2

func add(a interface{}, b interface{}) interface{} {
      return a.(int) + b.(int)
}
```

FINAL VERSION

This version of the add function resolves the previous concerns with the addition of reflection and a type switch.

Reflection is used to assure that both parameters are the same type.

The type switch matches the parameter value to a specific type. You can then perform the proper addition operation.

Future types, such as user define types (UDT), are not implicitly supported.

```
func add(a interface{}, b interface{}) interface{} {
	if reflect.TypeOf(a) != reaflect.TypeOf(b) {
		panic("incompatible types")
	}
	switch a.(type) {
	case int:
		return a.(int) + b.(int)
	case string:
		return a.(string) + b.(string)
	default:
		panic("not implemented")
	}
	return 0
}
```

GENERICS VERSION

Generic functions declare type parameters, within square brackets, between the function name and parameter list. Type parameters become placeholders for specialization within the function.

You can have multiple type parameters that are comma delimited. By convention, T is typically the name of the first type parameter.

You can apply type parameters to function parameters, return types, and declarations. Type substitution is inferred when the function is called.

Here is the add function with generics.

```
// add method with generics
func add[T any](a T, b T) T {
    return a + b
}
func main() {
    fmt.Println(add(4, 2))
}
```

invalid operation: operator + not defined
on a (variable of type T constrained by any)!

CONSTRAINTS

Constraints can limit the set of types. For example, constrain types to those that support the plus operation. With that assurance, this application compiles without errors.

Equal3, a generic function, implements the transitive principle to determine the equality of three values. This requires the equals operator (==). However, there is no guaranty that all types support this operator.

Constraints are either types or interfaces. The comparable interface can be used as a constraint, replacing any, and limits (constrains) the set to types that support the equals operator.

```
func equal3[T comparable](a T, b T, c T) bool {
     return (a == b) == (b == c)
}
```

ORDERED CONSTRAINT

What constraint requires supports for the plus operator? There are additional constraints available in the constraints package.

The Ordered constraint is included in this package and assures support of the plus operator.

```
go get golang.org/x/exp/constraints
```

Here is the revised add function using the Ordered constraint.

```
func add[T constraints.Ordered](a T, b T) T {
	return a + b
}

func main() {
	fmt.Println(add(4, 2))
	fmt.Println(add("first", "second"))
}
```

GENERIC TYPES

Generic types are generalized structures with type parameters. The type parameters are type placeholders defined when the structure is instantiated.

Unlike generic functions, the type parameter cannot be inferred with structures. You must provide the specialization.

```
type dog struct {}
type cat struct {}

type herd[T any] struct {
        animals []T
}
func main() {
        myherd:=herd[dog]{animals:[]dog{dog{}, dog{}, dog{}}}
        fmt.Println(myherd)
}
```

MULTIPLE CONSTRAINTS

You can apply multiple constraints, types or interfaces, to a type parameter using the pipe "|". You can include one or more pipes in the constraint. The constraints represent an *or* condition.

Any value that adheres to one of the constraints satisfies the requirement.

```
package main

import "fmt"

func myfunc[T int | float64](value T) {
	fmt.Println(value)
}

func main() {
	myfunc(42)
}
```

OVERVIEW

The release of generics is an important milestone for the Go Community. Congratulations to the community! Generics is robust and ready to be integrated into your applications.

Generics promotes code that is more concise and extensible. The result is applications that are more maintainable and requiring less refactoring.

Hard to overstate the impact that generics will have on Go development. This will radically change how Go applications are designed and developed.

Lab 5 – MATRIX

MATRIX OPERATIONS

Create an array or slice of functions that represent math operations that return integer types. Each takes two parameters and returns the result.

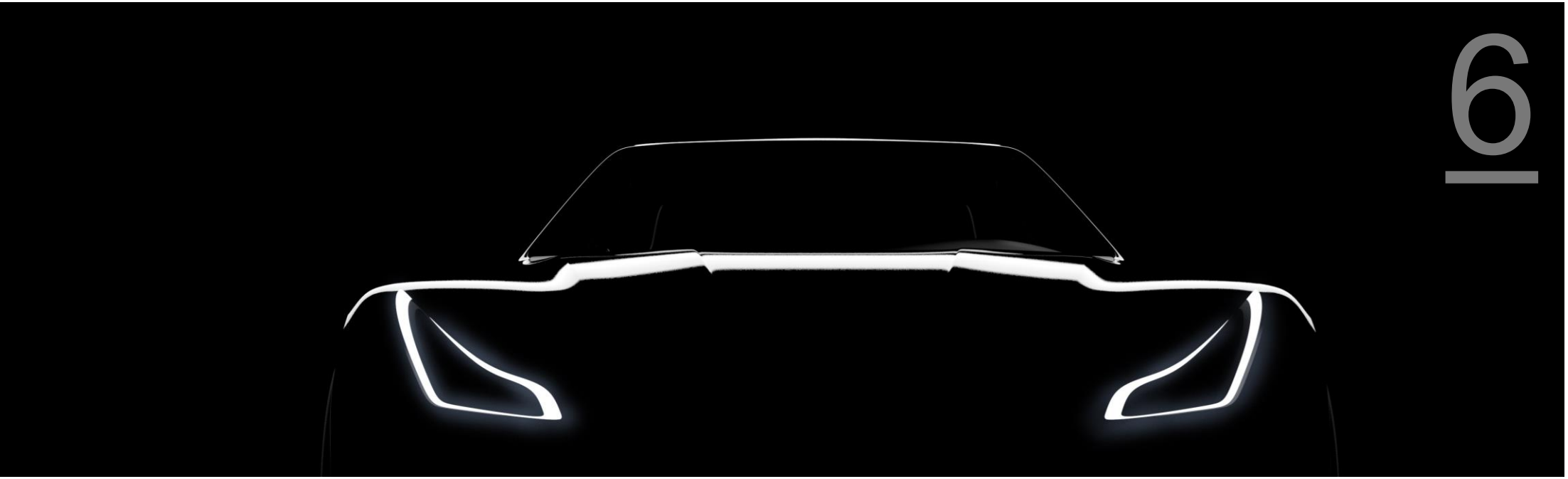
	Operations	Param 1	Param 2	Answer
Addition	a	4	5	?
Division	d	0	3	?
Multiplication	m	4	8	?
Subtraction	s	1	4	?

Perform each operation (row) in the most efficient way. When completed, display the results.

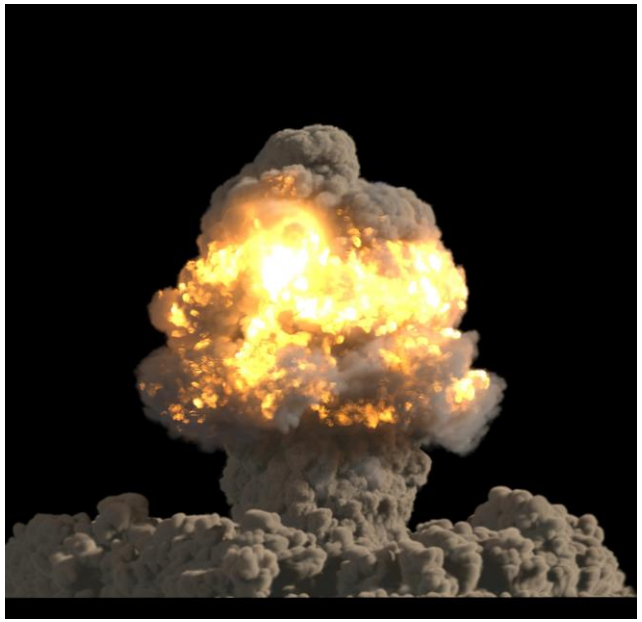
Lab completed

Structures

types, object oriented, composition



INTRODUCTION



The fathers of the Go Language were not afraid to blow up ingrained and well-established patterns of behavior. The triumvirate did just that to object-oriented programming with Go Language. That is one reason that the Go Language is so exciting. It is the demolition of traditional approaches in favor of something entirely new, exciting, and helpful.

INHERITANCE VERSUS COMPOSITION

At schools, books, and online, there has been a long standing bias towards inheritance and against composition. This has led to archaic solutions that are at best neither maintainable nor extensible.

Inheritance implies an "is a kind of" relationship between two types, code reuse, and tight coupling. Composition implies an "is a part of" relationship between two types, code reuse, and loose coupling. The loose coupling often makes composition more maintainable and extensible than a similar solution using inheritance. Composition is also referred to as embedding or containment.

CUSTOM TYPES

You can define custom types with the type keyword. This is similar to typedef in C / C++. Custom types are used in several ways:

- Structs
- Function pointers
- Interfaces
- Aliases

The syntax is:

type typename typeSpecification

- typename is the identifier
- typeSpecification indicates the type, such as struct, int, and so on.

CUSTOM TYPES - 2

Here is some sample code of defining custom types and declaring values of that type. Declaring a value for a custom type is the same syntax as a standard type.

```
package main


type Count = int
type StringInt map [string] int
type Fptr func(int) (int, int)
type MyStruct struct{
    data1 int
    data2 int
}

func main() {

    var a Count
    var b StringInt
    var c MyStruct
}
```

STRUCTS

Structures are aggregates types and can consist of multiple fields.

There are various methods for initializing a new structure:

- Name of structure
- Field initialization with colon
- Named field initialization

Access fields of a structure value using the dot syntax.

```
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    fredwilson := person{"Fred", 45}
    sallyjohnson := person{age: 45, name: "Sally"}
    fmt.Println(fredwilson)

    sallyjohnson.age += 15
    fmt.Println(sallyjohnson.name,
        sallyjohnson.age)
}
```

METHOD

A method is a function that is called on a custom type, most often a struct. The value of the target type is usually passed into the method as the receiver.

The method signature is similar to a function with the addition of a receiver:

func (receiver) identifier(parameters) return

The method can be called on a value of that type using the .dot syntax. The value is implicitly passed into the method as the receiver. The receiver can be passed by value or by pointer.

METHOD - 2

define a custom type (struct) (1)

declare a method for the custom type (2)

declare a method for the custom type (3)

create a value for the custom type (4)

call the method on the value (5)

```
package main

import "fmt"

type MyStruct struct {
    data1 int                    1
    data2 int
}

func (s MyStruct) Add() int {
    return s.data1 + s.data2     2
}

func (s MyStruct) Multiply() int {
    return s.data1 * s.data2     3
}

func main() {

    var c = MyStruct{5, 10}      4
    fmt.Println(c.Add())         5
    fmt.Println(c.Multiply())
}
```

POP QUIZ: WHAT IS THE RESULT?

```
package main

import "fmt"

type MyStruct struct {
    data1 int
    data2 int
}

func (s MyStruct) Increment() {
    s.data1++
    s.data2++
}

func main() {

    var c = MyStruct{5, 10}
    c.Increment()
    fmt.Println(c)
}
```

10 MINUTES

POINTER TO RECEIVER

Unless otherwise stipulated, receivers are passed by value into a method. Therefore, the method receives a copy of the receiver. Changes made to the copy will not persist to the original receiver. Pass the receiver by pointer to change the original value.

Note: When the receiver is passed by pointer, the method *still* uses the dot syntax and not pointer notation.

```
package main

import "fmt"

type MyStruct struct {
    data1 int
    data2 int
}

func (s *MyStruct) Increment() {
    s.data1++
    s.data2++
}

func main() {

    var c = MyStruct{5, 10}
    c.Increment()
    fmt.Println(c)
}
```

EMBEDDING

You can embed a struct inside a struct. Embedded types are unnamed. The affect is similar to inheritance and the members of the embedded value will be first class member values of the outer type.

```
package main
import "fmt"

type Address struct {
    street string
    city   string
    state  string
}
type Person struct {
    first string
    last  string
    Address
}

func main() {
    var bob = Person{"Bob", "Wilson",
        Address{"200 Broad St", "Phoenix",
            "AZ"}}
    fmt.Println(bob.first, bob.last,
        bob.street, bob.city,
        bob.state)
}
```

EMBEDDING - 2

Methods from embedded types are also available as first class members of the surrounding custom type. Those methods can be called directly on the surrounding type.

```
type Address struct {
    street string
    city   string
    state  string
}
type Person struct {
    first string
    last  string
    Address
}

func (a Address) getaddress() string {
    return fmt.Sprintf("%s\n%s\n%s",
        a.street, a.city, a.state)
}

func main() {

    var obj Person

    fmt.Println(obj.getaddress())
}
```

METHOD OVERRIDING

You can override methods inherited from embedded types. Simply implement the same method on the surrounding custom type. If wanted, you can delegate to the overridden method using the fully qualified type.

```
package main

import "fmt"

type Address struct {
    street string
    city string
    state string
}
type Person struct {
    first string
    last string
    Address
}

func (a Address) get() string {
    return fmt.Sprintf("%s\n%s\n%s",
        a.street, a.city, a.state)
}

func (a Person) get() (string, string) {
    return a.first + " " + a.last + "\n",
        a.Address.get()
}

func main() {

    var c = Person{"Bob", "Wilson",
        Address{"200 Broad St", "Phoenix",
            "AZ"}}

    fmt.Println(c.get())
}
```

INTERFACES

An interface is a custom type that consists of method signatures, not implementation. For that reason, an interface is abstract. Custom types are responsible for implementing interface.

The related type must implement the entire interface (the interface functions). If not, the type cannot be used in the context of the interface.

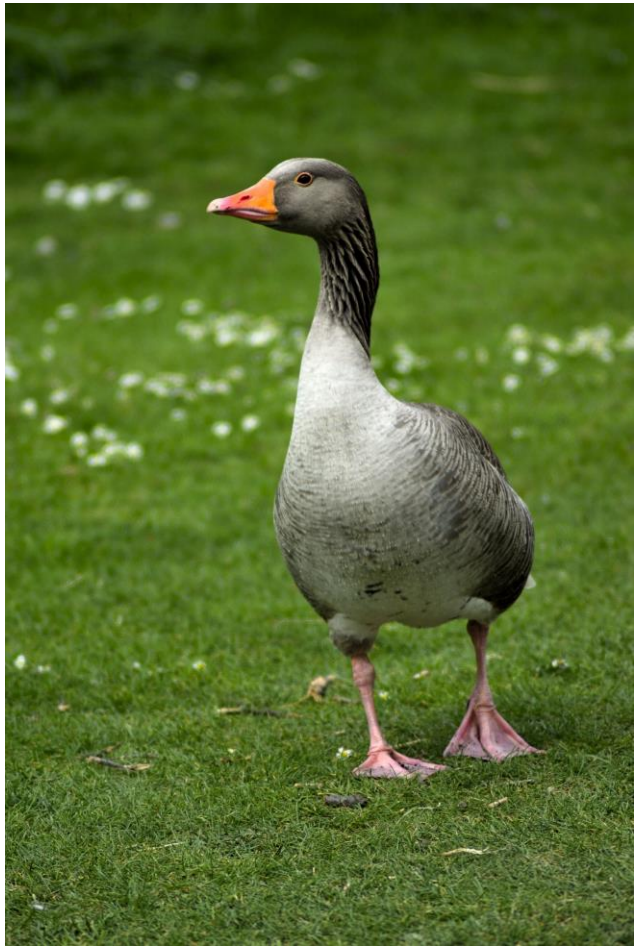
Benefit of interfaces:

- Hiding
- Membership
- Duck typing

INTERFACES - 2

```
package main

type Vehicle interface {
    Start(a int, b int) (int, float64)
    Turn()
    Stop()
}

type Auto struct {
}

func (a Auto) Start() {

}
func (a Auto) Turn() {

}

func (a Auto) Stop() {

}

type Train struct {
}

func (a Train) Start() {

}

func (a Train) Stop() {

}

func turnLeft(v Vehicle) {

}

func main() {
    var a Auto
    var b Train

    turnLeft(a)

    turnLeft(b)

}
```

DUCK TYPING



Go has duck typing instead of polymorphism. Duck typing is implemented with interfaces.

If a struct implements a duck interface, it must be a type of duck. All other structs that implement the duck interface are also assumed to be some type of duck. This may not be true! Without inheritance, that type of relationship is not assured.

POLYMORPHISM

Duck typing is how polymorphism is implemented in the Go. Polymorphism typically involves these four attributes:

- Related types
- Same methods
- Different behavior
- Derived pointer (or reference) to base class type

Duck typing can reasonably assure three of these four attributes. *Related types are not guaranteed however*. This can cause interesting outcomes.

DUCK TYPING EXAMPLE

```
package main

import "fmt"

type IPerson interface {
    talk()
    walk()
    eat()
}

type Human struct {
}

type Martian struct {
}

func (Human) talk() {
    fmt.Println("talking")
}

func (Human) walk() {
    fmt.Println("walking")
}

func (Human) eat() {
    fmt.Println("eating sushi")
}
func (Martian) talk() {
    fmt.Println("talking")
}

func (Martian) walk() {
    fmt.Println("walking")
}

func (Martian) eat() {
    fmt.Println("eating Humans")
}

func invite2Party(partygoers ...IPerson) {

    for _, person := range partygoers {
        person.eat()
    }
}
```

BAD PARTY!

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\goprojects\Example> go run stuff.go
eating sushi
eating sushi
eating Humans
eating sushi
PS C:\goprojects\Example>
```

Lab 6- Shapes

SHAPES

The goal of this lab is to draw a variety of shapes. Each shape is a type of geometric object that at a minimum able to draw themselves.

1. Define an interface iGeoshape with two methods: draw and print. The methods have no parameters or return values.
2. Implement a custom Rectangle type. Define two methods for Rectangle: draw and print. Stub both methods to display an appropriate message.
3. Implement a custom Circle type. Define two methods for Circle: draw and print. Stub both methods to display a message.
4. Create a render function that accepts one parameter – a variable length list of iGeoshape values. In the render function, call draw on each iGeoshape value.

SHAPES - 2

5. In main, create values for three rectangles and two circles.
6. Call render function with the five values.

Lab completed

Goroutines

Synchronization

I AM A THREAD. WHAT AM I?

A thread is a separate path of execution, concurrent or parallel, through an application.

Asynchronous function call:

- I have parameters
- I have a return value
- When the entry point function exits, I am done.

I own a stack, which includes one or more stack frames, starting at the entry point method, and each successive synchronous function call. Stack frames include:

- Parameters
- Local variables
- Instruction pointer
- And more

PARALLELISM VERSUS CONCURRENCY

There are two major objectives of multithreaded applications.

Parallelism

- Performance
- Maximum utilization of cores
- Scalability

Concurrency

- Responsiveness
- User interface events

WHAT HAPPENED TO THE SINGLE PROCESSOR COMPUTER?

Single processor computers have virtually disappeared from the marketplace. They are now as extinct as the dinosaur. At your local computer store, only multi-core computers are being sold – guaranteed.

During the past five years, it has become harder, if not impossible, to meet Moore's Law in an affordable computer. Vendors have sought to resolve the problem by adding more processor cores to every computer.

Unfortunately, applications do not instantly see increased throughput. There are many factors that contribute to whether an actual performance benefit is realized, including rewriting software to recognize the additional cores.

CONCURRENCY TERMS

Moore's Law. Gordon Moore defined Moore's Law in 1965. Despite popular myth, Moore's law does not directly predict that computer throughput will double every year. It predicts that the number of transistors on a microchip will double every two years. Moore's Law held true for many decades until recently.

Amdahl's Law. Amdahl's Law predicts the performance of a multicore application. Named after Gene Amdahl, it was first presented in 1967. The calculation recognizes that even the most ambitious program cannot be fully parallelized.

Embarrassingly parallel. Embarrassingly parallel refers to a solution that is decomposed into tasks that run almost entirely parallel. Requires removing dependencies, minimizing serialization, and other hurdles to parallelization. The goal is maximum throughput and utilization of cores.

PROCESS

The process owns the resources contained within the application, such as:

- Process memory
- Process priority
- Threads
- Handles
- Exit value

Processes start with a single thread, the primary thread. This is typically the main function in most programming languages. When the primary thread finishes, typically the application exits.

THREAD BEHAVIOR

The process owns the resources of the application. Threads execute within a process performing tasks and utilizing those resources.

Here are attributes of operating system threads:

- Scheduled independently
- Concurrent or parallel
- Assigned a thread priority
- Have a context, which includes the instruction pointer
- Own a stack
- Threads can be in one of three states:
    a. Waiting: blocked for some reason, not requesting processor attention.
    b. Runnable: available for processor time, but not running.
    c. Executing: currently executing on a processor core.

THREAD BEHAVIOR - 2

Threads are typically either I/O or CPU bound:

- I/O Bound. I/O bound threads regularly contact the I/O system to perform a tasks. When accessing an I/O resource, such as reading a file, a thread yields the CPU. This provides opportunities for other threads to run.
- CPU bound. CPU bound threads complete tasks with in-memory resources. CPU bound threads rely almost entirely on the CPU and are task driven, such as performing a calculation. They are more performant in general than I/O bound threads.

Running multiple threads, either I/O or CPU bound, may require coordination, thread synchronization.

COMMUNICATING SEQUENTIAL PROCESS (CSP)

In 1978, Charles Antony Richard Hoare introduced the concept of Communicating Sequential Process (CSP) in the paper of the same title:

http://www.usingcsp.com/cspbook.pdf

Communicating Sequential Process (CSP) is the model for concurrency in the Go Language, where data creation and consumption are the primary goal. Execution is data driven versus functionally driven.

For example, *task a* starts *task b.* Task b executes an algorithm and wants to return the result to task a. Task b writes the result into a channel. At some point, task a may block to wait for the results of task b. Task a reads the result from the channel. This is the basis of channels, an important synchronization component in Go.

RUNTIME METHODS

These methods, in the runtime package, can provide helpful information on processors:

- NumCPU. Provides the number of available processor cores for the Go application.
- GOMAXPROCS. Changes the available logical processors. Returns the previous setting.
- NumGoRoutine. Provides the number of running goroutines.

```
package main


import (
	"fmt"
	"runtime"
)


func main() {
	fmt.Println(runtime.NumCPU())
	fmt.Println(runtime.GOMAXPROCS(0))
	fmt.Println(runtime.NumGoroutine())
}
```

GOSCHED

The Gosched method, in the runtime package, cooperatively yields the logical processor from the current goroutine. This is an opportunity for another goroutine to receive processor time. However, execution may return to the original goroutine.

POP QUIZ: WHAT IS THE RESULT?

```
func main() {
    runtime.GOMAXPROCS(1)
    go func() {
        for i := 1; i < 100; i = i + 2 {
            fmt.Printf("%d ", i)
        }
    }()
    go func() {
        for i := 2; i < 100; i = i + 2 {
            fmt.Printf("%d ", i)
        }
    }()

    time.Sleep(time.Second)
}
```

5 MINUTES

LIGHTWEIGHT

Goroutines are less expensive than threads:

- Synchronization with channels is lighter than similar activity with threads.
- Startup and termination of goroutines are less expensive than threads.
- 2k stack size with a stack check. The stack is growable at 2x .
- The Go scheduler is a user mode process

This makes goroutines considerably more scalable than a thread. , Potentially millions of instances are possible.

SYNCHRONIZATION

Accessing shared resources from parallel code can be unpredictable and require synchronization.

Shareable resources include:

- Global memory
- Files
- Functionality
- And more

When porting existing code, special attention must be given to shared data and functionality.

SYNCHRONIZATION - 2

Channels are effective for data synchronization between parallel processes. However, memory and execution synchronization are required at times. For that reason, Go offers a variety of other synchronization objects:

- Mutex
- Cond
- Semaphore
- Once
- And more

YOUR FIRST GOROUTINE

This is a simple program. The go keyword runs a function asynchronously (goroutine).

In main, funca is run asynchronously.

```
package main

import "fmt"

func funca() {
     fmt.Println(
          "In Goroutine funca")
}

func main() {
     go funca()
}
```

POP QUIZ: PROBLEM

What is the problem with the code on the previous slide?

5 MINUTES

YOUR SECOND GOROUTINE

Synchronization has been added to the previous code. This prevents the race condition between main and funca, both goroutines. This code uses WaitGroup, which is a wrapper for a synchronization counter. WaitGroup is in the sync package:

- WaitGroup.Add: set wait count
- WaitGroup.Done: decrement wait count
- WaitGroup.Wait: block until wait count is zero.

When greater than zero, a WaitGroup object is non-signaled. At zero, the WaitGroup object is signaled.

```
package main

import (
    "fmt"
    "sync"
)

func funca() {
    defer wg.Done()
    fmt.Println(
        "In Goroutine funca")
}

var wg sync.WaitGroup

func main() {
    wg.Add(1)
    go funca()
    wg.Wait()
}
```

WAITGROUP

WaitGroup is typically used to wait on one or more goroutines to complete execution. The count is set in the starting goroutine, either individually or in aggregate, WaitGroup.Add method. Decrement the count in the new goroutine, using WaitGroup.Done.

Calling WaitGroup.Wait is blocking if the WaitGroup count is greater than 0.

```
package main
// Example 1
var wg sync.WaitGroup

func main() {

    wg.Add(2)
    go func() {
        wg.Done()
    }()
    go func() {
        wg.Done()
    }()

    wg.Wait()
}
```

CHANNEL BASICS

Channels provide interthread communication, where data is transferred between two goroutines. Channels are type safe and associated with a specific type. For example, this is the declaration of a channel that accepts string data:

```
c1 chan string
```

Insert data into the channel:

```
c1 <- "message"
```

Retrieve data from the channel:

```
m:=<- c1
```

This is an unbuffered channel, which is blocking. Unbuffered channels can contain 1 item.

CHANNEL BASICS - 2

The default channel is an unbuffered channel.

Unbuffered channels are blocking:

- Reading from an empty unbuffered channel is blocking.
- Writing into a channel is blocking.

CHANNEL BASICS - 3

In main, the sample code creates an integer channel. The channel is captured in a goroutine. In the goroutine, insert data into the channel (10).

- In main, read from the channel and block, unless data is already in channel.
- Display the value read from the channel.

```
package main

import "fmt"

func main() {

    channel := make(chan int)

    go func() {
        channel <- 10
    }()

    value := <-channel
    fmt.Println(value)
}
```

UNIDIRECTIONAL CHANNELS

A channel can be unidirectional or bidirectional, the default.

A unidirectional channel is either a receive or send channel, but not both.

Here is how you define a send only (write only) channel:

channel chan <- string

Here is how you define a read only (read only) channel:

channel <- chan string

A bidirectional channel can be coerced into a unidirectional channel.

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

channel := make(chan int)
var rchannel <-chan int = channel
var wchannel chan<- int = channel

go func() {
    defer wg.Done()
    fmt.Println(<-rchannel)
    fmt.Println(<-rchannel)
    // rchannel <- 1  not compile
}()

go func() {
    defer wg.Done()
    wchannel <- 4
    wchannel <- 2
}()

wg.Wait()
```

CLOSE

Close a channel with the close method.

```
close(channel)
```

Once closed, a channel can be read from. However, the closed channel cannot be written into. Writing into a closed channel causes a panic.

```
channel := make(chan int)
go func() {
    channel <- 5
    time.Sleep(time.Millisecond * 5)
    channel <- 15
}()

go func() {
    fmt.Println(<-channel)
    close(channel)
}()

time.Sleep(time.Second * 10)
fmt.Println(<-channel)
```

BUFFERED CHANNEL

Unbuffered channels can hold a single item. Buffered channels however can hold more than one item. For example, here is a buffered channel for four integers.

```
channel:=make(chan int, 4)
```

Buffered channels are asynchronous with two exceptions. Buffered channels block:

- Reading from an empty buffered channel
- Writing into a full buffered channel (at capacity)

FOR RANGE

Items in a channel are iterable using a for range statement, as shown in the sample code.

The channel will be iterated until no data is present in the channel.

```
queue:=make(chan bool, 5)
go func() {
    queue<-true
    queue<-true
    queue<-false
    queue<-true
    queue<-false

    close(queue)
}()

go func(){
    for item:=range queue {
        fmt.Println(item)
    }
    fmt.Println("after")
}()
```

SYNCHRONIZATION

Channels can be used purely for synchronization to coordinate activities between goroutines. Typically, an any channel or bool channel are used for this purpose. A goroutine reads from the channel and blocks until another goroutine writes into the channel, which triggers the event.

When used in this scenario, the actual data is immaterial. For this reason, the value is discarded.

<-channel

```
done:=make(chan bool)

go func(){
    fmt.Println("closure - start")
    time.Sleep(time.Second)
    done<-true
}()

<-done
fmt.Println("main - end")
```

SELECT STATEMENT

The select statement is useful as a message pump for channels.

Select statements are similar to the switch statement. However, each arm is *case <-channel* instead of *case expression*. The arms represent the participating channels contributing events to the queue.

Writing to a channel creates an event that the select statement recognizes. Read data from the channel to receive the event details.

SELECT STATEMENT - 2

When all of the participating channels are empty, the select statement blocks. It blocks until an event occurs within one of the channels.

Essentially, each channel is a mailbox. When there is mail, open the mail and respond.

SELECT - EXAMPLE

Here is sample code for the select statement. In this code there are three channels;

- walk: to move forward a certain distance
- run: to run a certain distance
- turn: to change direction (degree)

Each event represents a step in a longer journey. The steps represent either a distance or a change of direction.

In our example, there are five steps.

```
type distance int
type degree int

var wg = sync.WaitGroup{}

func main() {

    walk := make(chan distance)
    run := make(chan distance)
    turn := make(chan degree)

    wg.Add(2)
    go func() {
        defer wg.Done()
        walk <- 5
        walk <- 20
        turn <- 45
        run <- 5
        walk <- 0
    }()
```

SELECT – EXAMPLE - 2

This code handles the journey.

The message pump is contained in a for loop, which includes the select statement.

1. Block on the select until an event happens (step in the journey).
2. After handling the step, iterate.
3. Block again, wait for the next step
4. If already queued, handle the next step.

Notice the walk channel. If the distance walked is zero, the journey is finished. The application simply returns.

```
go func() {
    defer wg.Done()
    for {
        select {
        case walking := <-walk:
            if walking == 0 {
                fmt.Printf("person has stopped\n")
                return
            }
            fmt.Printf("walking %d blocks\n", walking)
        case running := <-run:
            fmt.Printf("running %d blocks\n", running)
        case turning := <-turn:
            fmt.Printf("turning %d degrees\n", turning)
        }
    }
}()

wg.Wait()
}
```

POP QUIZ: CANCELLATION

How do you cancel a running goroutine? What are some of the concerns?

10 MINUTES

CANCELLATION

Cancellation of a goroutine is an important consideration, which presents some subtle challenges.

When the primary goroutine exits for example, the other goroutines are also terminated. You have a choice – an orderly exit or disarray. If you prefer disarray, skip ahead a couple of slides. Everyone else should use the cancellation model as described next. With this model, goroutines can receive a cancellation notification. Once this notification is obtained, the goroutine is then responsible for an orderly shutdown.

Golang does not provide an implementation of a cancellation policy. However, the implementation provided here is the best practice.

CANCELLATION - CHALLENGES

Here are the challenges:

- Must be able to broadcast a cancellation to all goroutines.
- Polling is expensive.
- Can goroutines check on cancellations at a convenient time, something cooperative but alertable?
- Once notified, goroutines must be given an opportunity for an orderly exit.
- If necessary, set an expiration.

CLOSE

The solution relies on magic related to the *close* method. When a channel is closed (i.e., close method), *all waiting goroutines* are immediately signaled and released.

```
done := make(chan int)

go func() {
    <-done
    fmt.Println("finishing 1")
}()
go func() {
    <-done
    fmt.Println("finishing 2")

}()
time.Sleep(time.Millisecond * 5)
go func() {
    fmt.Println("unblocking...")
    close(done)
}()
```

CANCELLATION – STEPS

Here are steps for a successful cancellation policy:

1. Create a global read-only channel (done channel). Alternatively, pass into goroutines as a parameter or capture.
2. Include the done channel in the select statement for the related handlers / routines.
3. For cancellation, close the done channel.
    a. Set a timeout, if necessary
    b. Continue
4. In the done event, cleanup resources related to the active channels and exit.

CANCELLATION – EXAMPLE CODE

```
package main

import (
    "fmt"
    "time"
)

func funca() {
    for {
        select {
        case <-dosomething:
        case <-stop:
            //       cleanup
            fmt.Println("exiting...")
            return
        }
    }
}
```

```
var stop = make(chan struct{})
var dosomething = make(chan struct{})

func main() {

    go funca()
    close(stop)

    time.Sleep(time.Millisecond * 5)

}
```

Lab 6 – Banking

BANKING

In this lab, manage customer bank accounts: deposits and withdrawals. Bank transactions, withdrawal and deposit, are handled in a message pump. The message pump will monitor the withdrawal and deposit channels for channels.

The message pump will update bank balances based on each transaction.

STARTUP

1. Create a project and source file for an executable.
2. Import the fmt package
3. Create a structure called transaction. The members are:
   - name as a string
   - balance as an unsigned integer
4. Create a map of customers:

```
"Bob":    0
"Ted":    0
"Carol":  0
"Alice":  0
```

5. Create a deposit and withdrawal channel. The channels accept transaction structures.
6. Create a done channel that accepts Booleans.

CONTROLLER

In main, define the controller within a closure (anonymous method). These steps implement the controller:

1. Add an infinite for loop
2. Add select statement within the loop :
    a. For the deposit channel:
        - Get the transaction from the channel
        - Confirm that the customer exists in the map
        - If not, apply a panic with a proper message
        - Update the customer balance (add the transaction amount)

CONTROLLER - 2

b. For the withdrawal channel:
   - Get the transaction from the channel
   - Confirm that the customer exists in the map
   - If not, apply a panic with the proper message
   - Calculate the balance (subtract the transaction amount) and save to a temporary variable
   - If new balance less than 0, apply a panic. Customers are not allowed to overdraw their account.
   - Update the customer balance in the map

c. For the done channel, print “exiting” and exit the infinite loop.

3. Run the controller

TRANSACTIONS

Here is an example of how to create a deposit transaction for Bob.

```
deposit <- transaction{"Bob", 10}
```

In main, create these transactions.

- Deposit 10 for Bob
- Deposit 40 for Ted
- Withdrawal 5 from Bob
- Deposit 25 for Alice
- Withdrawal 10 from Alice
- Deposit 10 for Bob

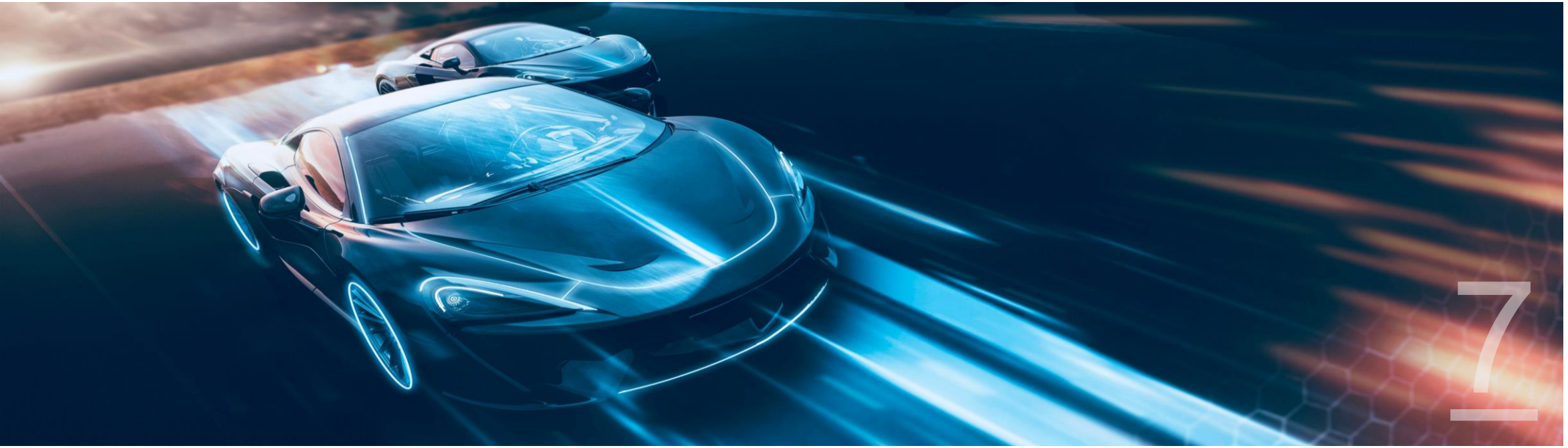
Notify the controller that the transactions are done.

Display the customer balances.

Lab completed

Modules

Dependency Management



DEPENDENCY MANAGEMENT

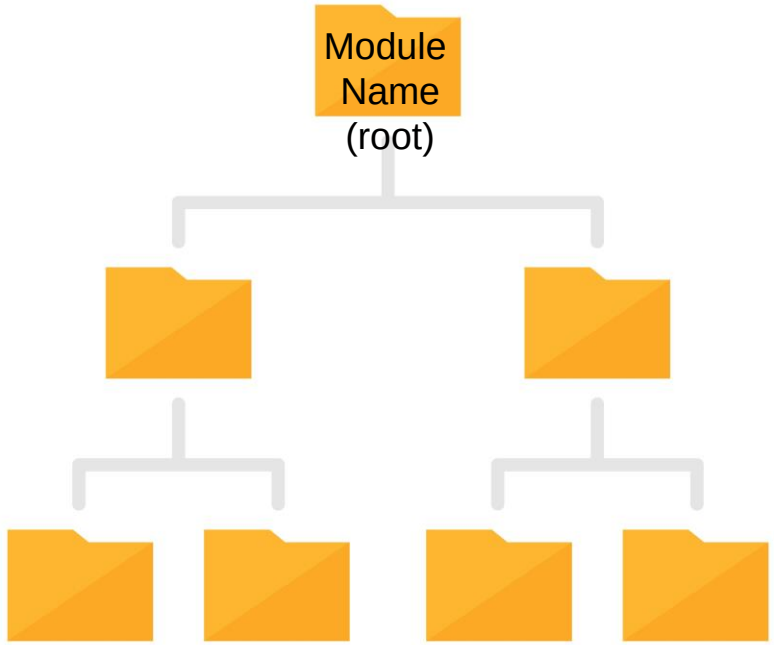
Dependency management was recently improved in Go Language with the modules feature and can prevent some common dependency / versioning problems. This feature appeared first in Go 1.11. Starting in Go 1.13 the dependency model was fully adopted.

What is a module? A module is a collection of Go packages residing in the same file hierarchy with a module file at the root.

The primary goal of modules are:

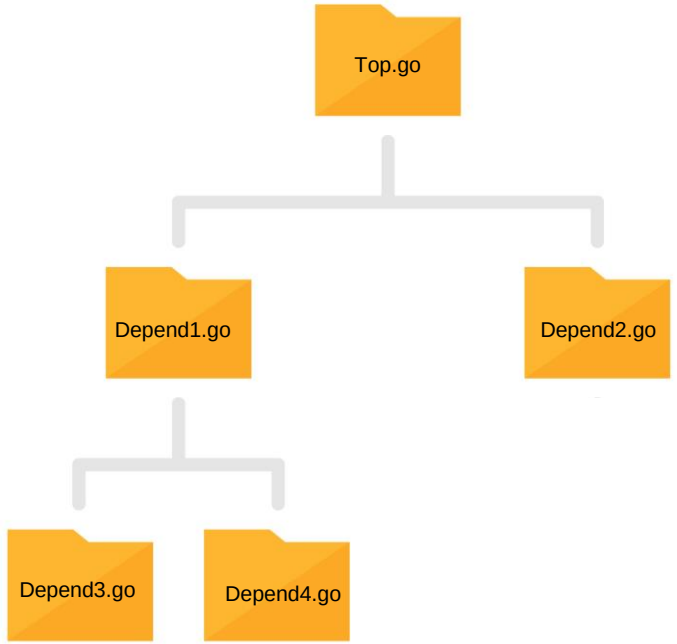
- Resolving DLL Hell with shared libraries
- Managing multi-package application development

MODULE



A module is a file directory database that consist of multiple local packages for a Go application. For a module, create a go.mod file in the root directory.

MODULE - 2



Managing related packages is easier when arranged in a file hierarchy. Storing all the packages of a single application in a single directory could be cumbersome and prevent effective versioning.

MODULE

```
C:\Code\Go\src\sample>go mod init sample.com
go: creating new go.mod: module sample.com

C:\Code\Go\src\sample>type go.mod
module sample.com

go 1.14
```

Create a “module” file at the root of the module file hierarchy. Without this file, the module does not exist.

This command creates a module file, go.mod:

go init mod *rootname*

For a module with no dependencies, the go.mod file has the name of the module and the version of the go environment.

The rootname can be anything. As a best practice however, the root name should be the application location.

MULTI-PACKAGE MODULE

Consider the file directory for Go applications a database where each directory is a repository containing one or more packages. Use file system directories to organize packages and avoid developing a monolithic application. This is especially useful for applications comprised of many local packages.

The path for packages in the module are relative to the root name. For example, here is an import statement for a package within a module:

```
import “root/package1/package2”
```

MULTI-PACKAGE MODULE - 2

In this example, this is the main package for a module.

The import references library packages that are also contained in the same module.

The path to the libraries is relative to the root name for each module. The root name here is “multi”. This must have been set using the go mod init command.

```
// root directory
package main

import "multi/h1"
import "multi/h2"

func main() {
      h1.FuncA()
      h2.FuncB()
}
```

MULTI-PACKAGE MODULE - 3

Here is the code for both libraries, each in separate files and directories. Remember, functions named with initial caps are exported and externally visible.

- The h1 package exports the FuncA function
- The h2 package exports the FuncB function

```
// h1 subdirectory
package h1
import "fmt"

func FuncA() {
    fmt.Println("FuncA")
}

// h2 subdirectory
package h2

import "fmt"

func FuncB() {
    fmt.Println("FuncB")
}
```

DEPENDENCY

Here is sample code of a package, within a module, that has a dependency. In this example, "rsc.io/quote" is the dependency.

The quote package is imported into this application and found in pkg.go.dev. Pkg.go.dev is the open source package library for the Go community.

https://pkg.go.dev/

```
// hello.go
package hello
import "rsc.io/quote"

func Hello() string {
    return quote.Hello()
}

// hello_test.go
package hello
import "testing"

func TestHello(t *testing.T) {
    want:="Hello, world."
    if got:=Hello(); got !=want {
        t.Errorf("Hello) %q, want %q", got, want)
    }
}
```

SEMANTIC VERSION NUMBER

```
C:\Code\Go\src\go_hello>go list -m all
donis.xyz/hello
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
rsc.io/quote v1.5.2
rsc.io/sampler v1.3.0
```

Go modules have a semantic version number, which has three parts.

- Major
- Minor
- Patch

The current standard is Semver 2.0. Here is the format:

vMajor.Minor.Patch

SEMANTIC VERSION NUMBER - 2

```
C:\Code\cool>go list -m all
testmod2
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
rsc.io/quote v1.5.2
rsc.io/sampler v1.3.0
```

Detailed explanation of the semantic version number is beyond the scope of this course. More information is here:

https://bit.ly/2kUbR7l

https://bit.ly/2NTu5mV

LIST DEPENDENCIES

```
C:\Code\hellomod>go list -m all
example/hellomod
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
rsc.io/quote v1.5.2
rsc.io/sampler v1.3.0
```

As mentioned, list the dependencies of a module with this command:

`go list –m all`

The list will provide.

1. The first item in the list is the module.
2. Other modules and dependencies

The golang.org/x/text module has routines for handling internationalization / localization in Go, which includes encoding and text handling

DEPENDENCY FILES

Importing dependencies impacts two files: go.mod and go.sum

- The go.mod file list the dependencies, version number, and status
- A go.sum file is created and contains the hashes for the imported modules. The hashes are used to validate the integrity of dependencies.

```
C:\Code\Go\src\go_hello>type go.mod
module donis.xyz/hello

go 1.14

require rsc.io/quote v1.5.2

C:\Code\Go\src\go_hello>type go.sum
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nkL0i8=
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c/go.mod h1:NqM8EUOU14njkJ3fqMW+pc6Ldnwhi/IjpwHt7yyuwOQ=
rsc.io/quote v1.5.2 h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
rsc.io/quote v1.5.2/go.mod h1:LzX7hefJvL54yjefDEDHNONDjII0t9xZLPXsUe+TKr0=
rsc.io/sampler v1.3.0 h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
rsc.io/sampler v1.3.0/go.mod h1:T1hPZKmBbMNahiBKFy5HrXp6adAjACjK9JXDnKaTXpA=
```

GO MOD INIT DETAILS

- If the go.mod exists, the go mod init sub-command will not recreate the file or update the version of dependencies.
- By default, imports the latest version of packages
- Can import a chain of dependencies
- The imported package is shared in the GOPATH/pkg/mod directory.
- The go.mod file will be updated as the source files in the module reference other dependences
- You can update go.mod manually, such as updating version information.

IMPORT THE LATEST VERSION

```
C:\Code\Go\src\module_get>go mod init donisexample
go: creating new go.mod: module donisexample

C:\Code\Go\src\module_get>go get github.com/dustin/go-humanize
go: github.com/dustin/go-humanize upgrade => v1.0.0

C:\Code\Go\src\module_get>type go.mod
module donisexample

go 1.14

require github.com/dustin/go-humanize v1.0.0 // indirect

C:\Code\Go\src\module_get>type go.sum
github.com/dustin/go-humanize v1.0.0 h1:VSnTsYCnlFHaM2/igO1h6X3HA71jcobQuxemgkq4zYo=
github.com/dustin/go-humanize v1.0.0/go.mod h1:HtrtbFcZ19U5GC7JDqmcUSB87Iq5E25KnS6fMYU6eOk=
```

You can import or update to the latest version of a module using the get command. Specify the module without a version information.

GET A SPECIFIC VERSION

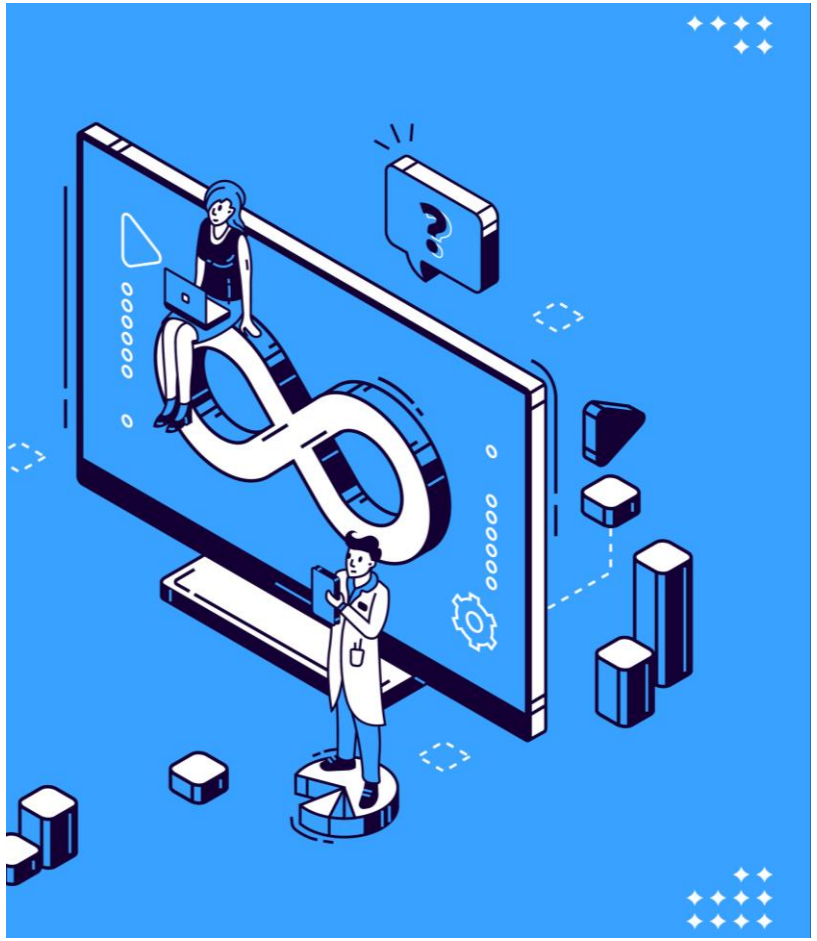
```
C:\Code\hellomod>go list -m -versions rsc.io/sampler
rsc.io/sampler v1.0.0 v1.2.0 v1.2.1 v1.3.0 v1.3.1 v1.99.99

C:\Code\hellomod>go get rsc.io/sampler@v1.2.1
go: finding rsc.io/sampler v1.2.1
go: finding rsc.io/quote v1.5.1
go: finding rsc.io/quote v1.5.0
go: finding rsc.io/quote v1.4.0
go: finding rsc.io/sampler v1.0.0
go: downloading rsc.io/sampler v1.2.1
go: extracting rsc.io/sampler v1.2.1

C:\Code\hellomod>go get rsc.io/sampler@latest
go: finding rsc.io/sampler v1.99.99
go: downloading rsc.io/sampler v1.99.99
go: extracting rsc.io/sampler v1.99.99
```

You can also select a specific version with the *go get* subcommand. Simply specify the semantic version number at the end of the specification.

```
go get modulepath@v1.2.3
```

The "latest" version number imports the latest version without specifying the semantic version.

INTRODUCTION



Workspaces help resolve real world problems when referencing dependencies. It is especially useful as an alternative to the replace directive.

Both the replace directive and workspaces are related to Go modules.

WALKTHROUGH – MODULES

In this walkthrough, you will explore the *go mod* commands and the replacement directive.

HELLO PACKAGE

The Hello package exports a Hello method that displays “Hello, world!”.

Create:

- Directory named Hello
- Sub-directory Hello: Hello/Greeting
- File within Greeting directory: greeting.go
- Create a module file. Set the module path following the best practice:

  github.com/*baseaddress*/*project*/*module*

  `("github.com/donismarshall/lab/greeting")`

SOURCE FILE

Here is the source code for the greeting.go source file.

```
package greeting

import "fmt"

func Hello() {
      fmt.Println("Hello, world!")
}
```

USER PACKAGE

The User package will invoke the Hello function in the Greeting package.

Create:

- Directory named User
- Sub-directory Hello: Hello/User
- File within User directory: user.go
- Create a module file. Name the module *user*.

SOURCE FILE

Here is the source code for the user.go source file.

```
package main

import (
     "github.com/donismarshall/lab/greeting"
)

func main() {
     greeting.Hello()
}
```

FINAL STEPS

Import and update dependencies for the user application.

```
go mod tidy
```

The greeting package has not been deployed to the github. Because of this, you will receive this error message.

```
go: finding module for package github.com/donismarshall/lab/greeting
user imports
        github.com/donismarshall/lab/greeting: cannot find module providing pa
ckage github.com/donismarshall/lab/greeting: module github.com/donismarshall/l
ab/greeting: git ls-remote -q origin in c:\code2\go\pkg\mod\cache\vcs\c201cab2
3d009a0f8fb4438523362cbfd42470d58166c634ca02395c4ed188f6: exit status 128:
        remote: Repository not found.
        fatal: repository 'https://github.com/donismarshall/lab/' not found
```

Resolve this error with the replace director. Redirect the github reference to a local reference.

```
go mod edit -replace "github.com/remaining path"=../greeting
```

```
go mod edit -replace "github.com/donismarshall/lab/greeting"=../greeting
```

Next, execute the command “go mod tidy”

You should now be able to run the program successfully.

REPLACE DIRECTIVE

When the application is released into beta or production, the module must be updated to undo the effects of the replace directive. Unfortunately, this is sometimes forgotten.

Workspaces to the rescue!

WORKSPACE MODE

The workspace file (go.work) removes the most important problem with go.mod- the replace directive. You must remember to update the go.mod file before releasing the product. If not, you might release the wrong or local version of dependency.

With the workspace mode, you can remove the replace directive from go.mod.

For github, include go.work in the .gitignore file. This means the go.work file will not be released into production. For this reason, the correct version is selected, from go.mod, when the package is deployed.

WORKSPACE MODE – 2

1. Remove the replace directive from the go.mod file
2. At the root directory of the client application, create a workplace file.

```
go work init location1 ,location2, … locationn
go work init ..\greeting
```

Identify the *location* of the dependent modules

When released:

1. Add the go.work file to .gitignore.
2. Build the application: go build.
3. You can run your application.