Introduction to Go Language

Programming

THE BEST OF ALL WORLDS

Go Programming language combines the freedom and syntax of C languages and the usability and abstraction of Python.

Freedom and syntax: pointers, compilations, blocks, and more.

Usability and abstraction: garbage collection, large assortment of standard libraries, string manipulation, and more.

Easy to use, such as Python, but sophisticated enough for complex development, similar to C / C++.

GO LANGUAGE

Go

Go Language (golang) is an open source language developed by Robert Griesemer, Rob Pike, and Ken Thompson. It was launched at the Google Open Source Blog on November 10, 2009. Go was initially developed for requirements set by Google. Go was created partly from frustration in using C++.

Go Language is made available under the BSD 3-Clause "New" or "Revised" License.

"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 PROJECTS

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

LOGISTICS

Class Hours:

- Start time is 9am
- End time is 4:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

Lunch:

- Lunch is 11:45am to 1pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

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

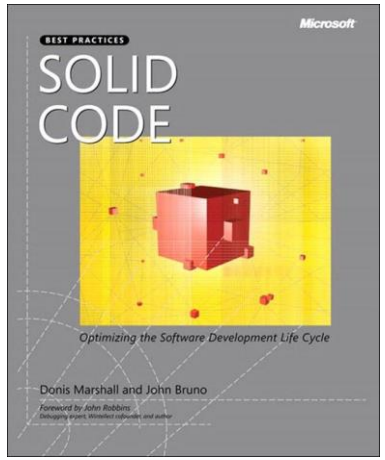
Miscellaneous

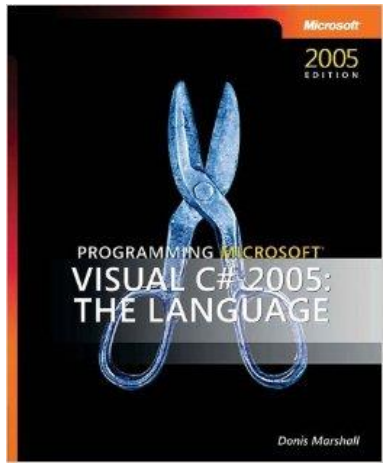
- Courseware
- Bathroom
- Fire drills

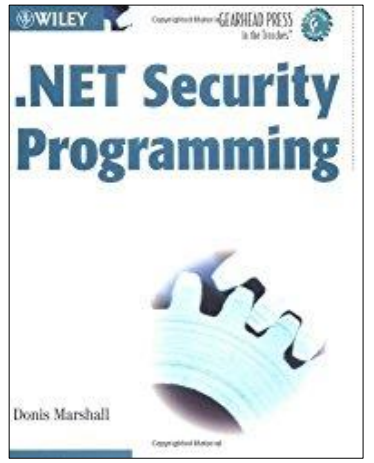
DONIS MARSHALL

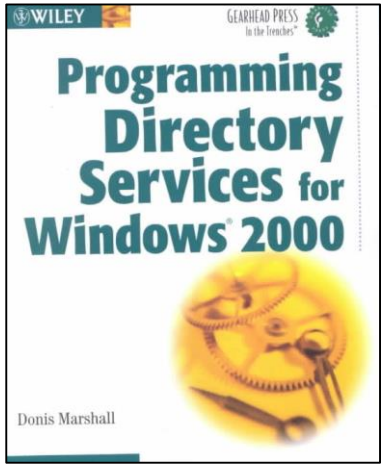
GO Language Practitioner
Microsoft MVP
Microsoft Certified
C++ Certified
Author

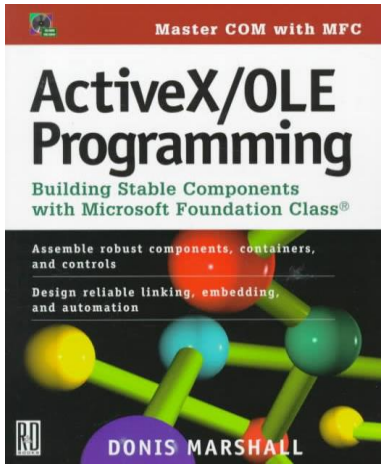
donis@innovationinsoftware.com

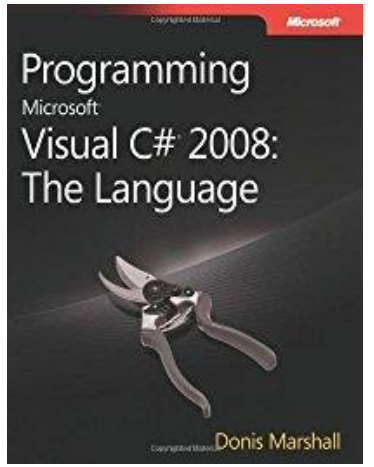












INTRODUCTION TO GO LANGUAGE

Go is C-like, small, and a strongly typed language. The language supports low-level capabilities such as pointers and concurrency; while supporting higher level features such as garbage collection and collections. Go Language can be used as a procedural, functional, object-oriented language, or some combination.

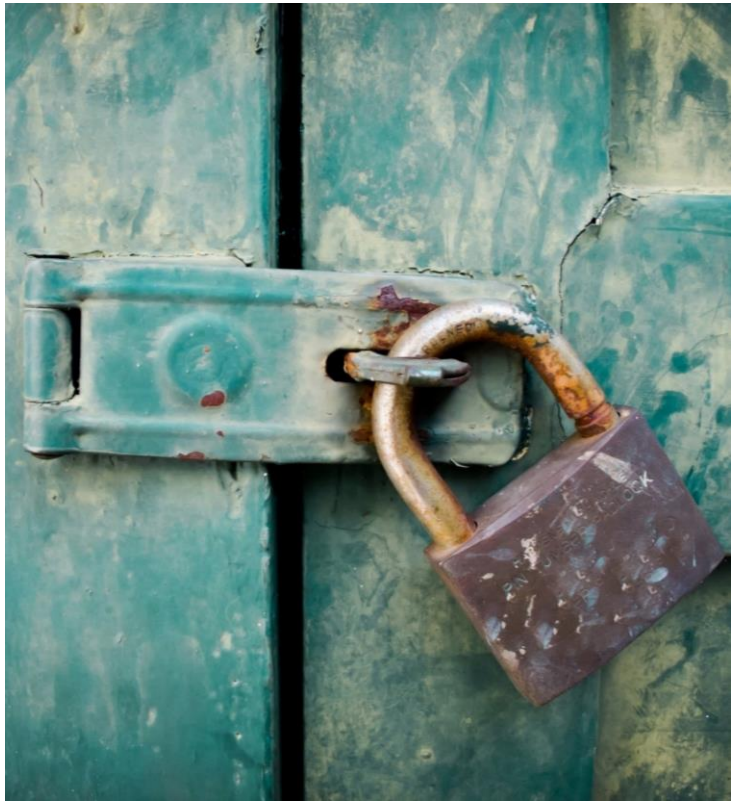
Code from the Go Language is compiled into binary unlike Python language, which is an interpreted language. This allows for faster execution, performance optimization, cross-platform deployment, and more. Go Language scales efficiently making it easier to develop large applications. One example is the better handling of package dependencies.

Robust library of open source packages that can be used with the Go Language. Many of the standard packages are deployed here:

https://golang.org/pkg/

Go offers many features including Unicode strings, various data structures, garbage collection, concurrency, file handling, and networking.

NO INHERITANCE



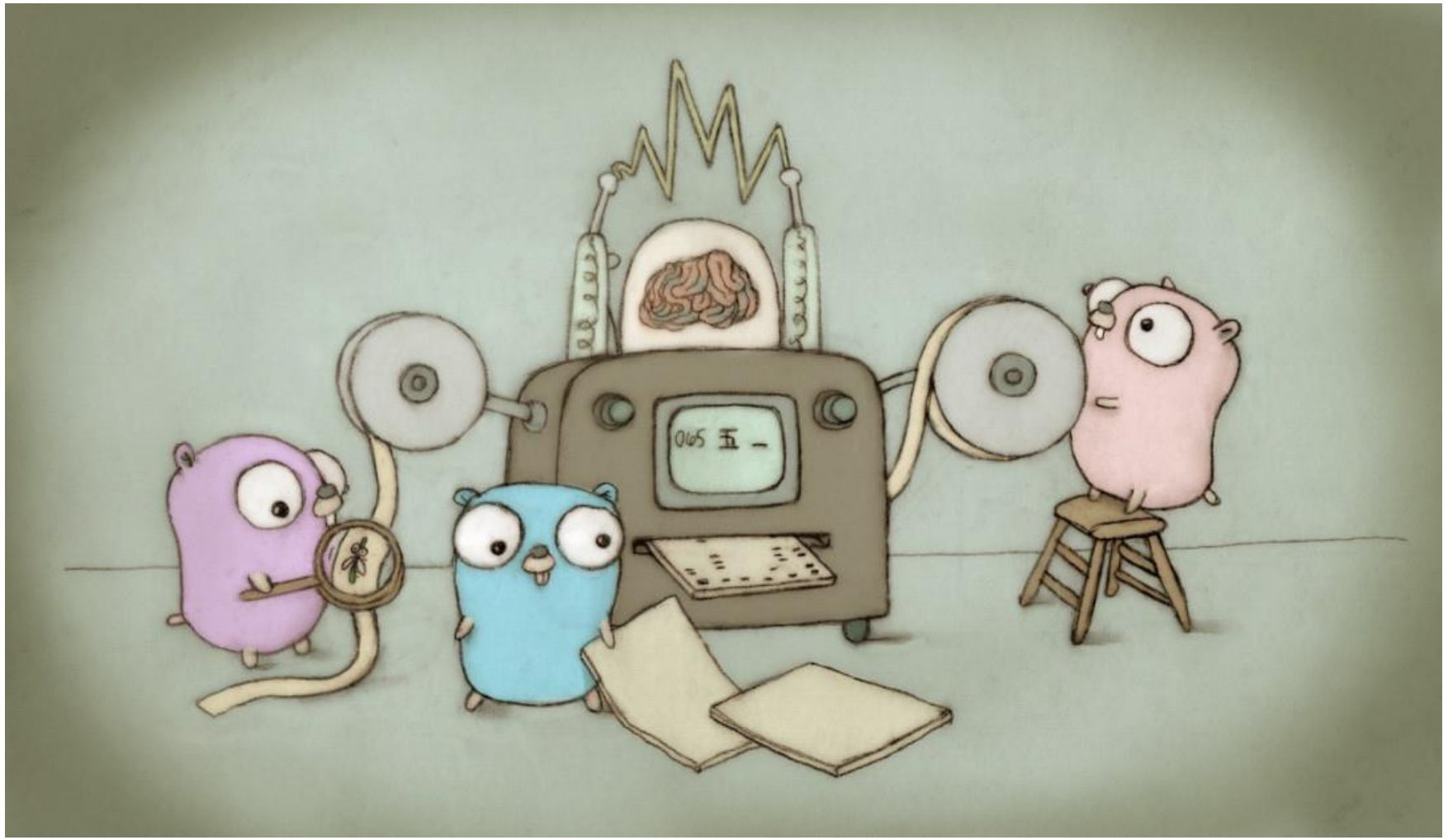
Go Language closes the door on classes and inheritance. Instead, composition is used for object oriented programming. Traditionally, there has been an overreliance on inheritance in modern languages, such as C++, Java, and Python.

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

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

http://bit.ly/2H3mHxT

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 with GitHub; not just source code.

Git is an 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.

The Go Programming 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.

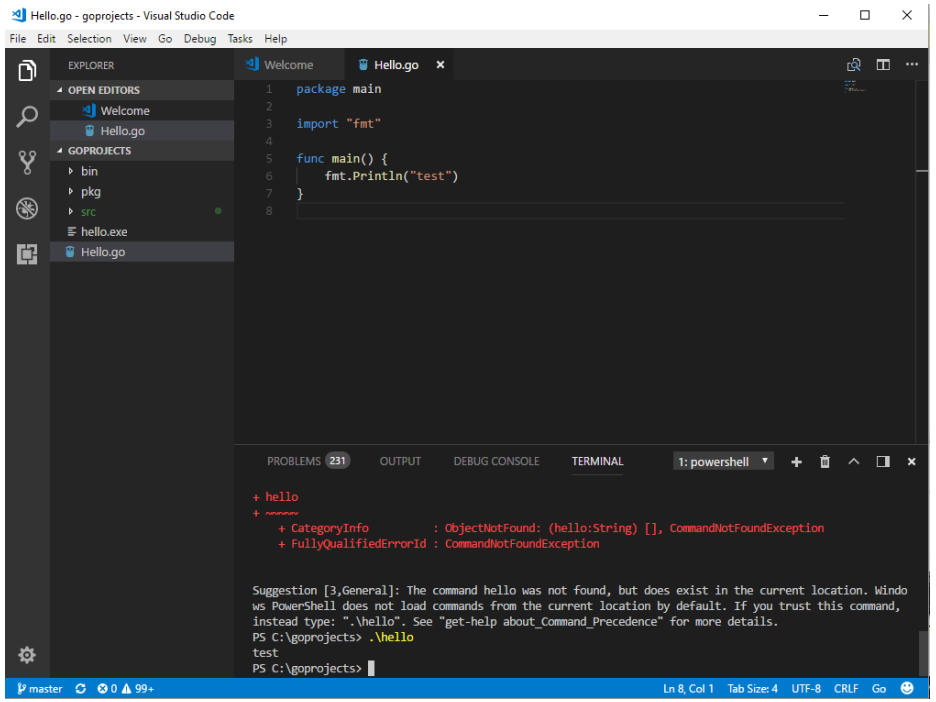
https://bit.ly/2LhPy34

CODING

Go Language supports UTF-8 text format, which is supported by many editors. For this reason, formal IDEs are not required to edit, compile, or execute a Go program. You only need the Go Language, 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:

- VSCode
- IntelliJ
- Emacs
- Eclipse

VSCODE



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

The instructor will use VSCode as the IDE. It is free and available in most environments; and has a Go Language extension.

Download VSCode here:

https://code.visualstudio.com/download

THE GO PLAYGROUND

The Go Playground is an online web-based environment for Go Language development. A great place to practice your programming expertise for 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://play.golang.org/

Lab 1- Setup

STEP 1

Here are the steps for creating an environment to develop Go projects. These are generic steps and specific details may vary based on your operating system and architecture.

You can skip some steps if completed previously. For example, no need to install Git if already installed unless you want a more recent version.

Speaking of Git, Step 1 is installing Git. Install Git from this web page.

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

At this page, you will also find documentation on installing GIT for each environment.

STEP 2

Next step is installing the Go Language.

Here is the webpage to install the Go Language for a variety of target environments.

https://golang.org/doc/install

You will also find documentation on installing for each environment at this page.

After the installation is completed, inspect the PATH, GOROOT, and GOPATH environment variables. The PATH should include an entry identifying the GO binary directory. GOROOT references the installation directory. GOPATH is where the source code for your projects and packages are located.

Test the Go command (go version) from the command prompt / terminal as a smoke test of a successful installation.

STEP 3

Delve is a debugger for debugging Go applications. In addition, Delve interestingly was developed using the Go Language.

Instructions for installing Delve are located here:

http://nanxiao.me/en/a-brief-intro-of-delve/

This webpage also provides basic documentation on debugging Go Language source code using Delve.

STEP 4

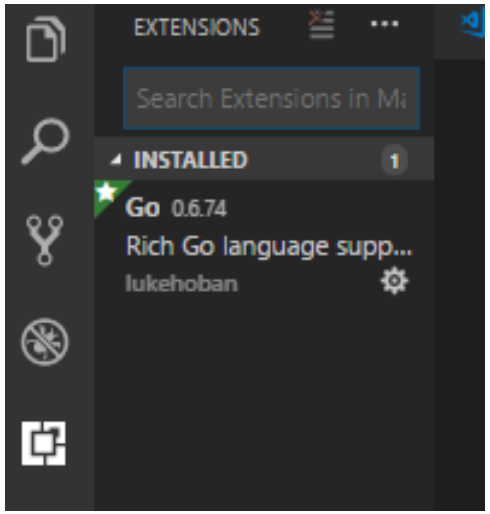
Installing VSCode is the final step. Here is the installation page for deploying VSCode to various environments.

https://code.visualstudio.com/docs/setup/setup-overview

After installation, start the VSCode application.

- From the left panel, install the Go extension.
- If extensions are not visible, select Extensions from the View menu. You can also select the Extension icon ( ) from the vertical toolbar.
- After installing, make sure to activate the Go extension.

You are ready to begin developing code using the Go Language!



The Basics

Go Programming

GETTING STARTED

Hard to get started without a *start*. That would be considered a Yogi Berra-ism.

http://nyp.st/2nWXCw7

This chapter provides a start on the Go Language. You will learn how to create procedural, object oriented, and networking programs using Go Language and with the help of annotated source code.

With this foundation, we can then backfill some of the concepts introduced in later modules.

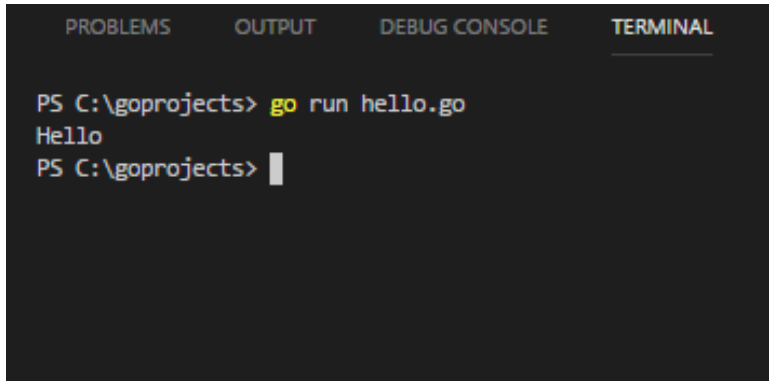
PROCEDURAL

The founders of Go Programming understand the lasting benefits of procedural programming. Object oriented programming is not always the most ideal or straightforward solution.

Every introduction, and some advanced, programming classes have an “Hello, world!” program or some variation.

HELLO.GO

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])
}
```

HELLO.GO - EXPLAINED

build an executable versus a shared package with main as the entry point method (1)

import these shared packages (2)

declare hello variable as a slice of strings(3)

declare index as an integer type(4)

convert command-line argument from string to integer(5)

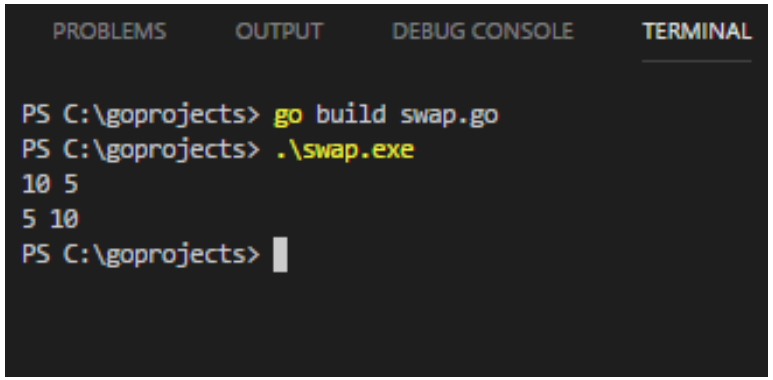
If index not within range, set to 1 (6)

print the correct hello based on command-line argument (7)

```
package main          1

import (
                "fmt"
                "os"          2
                "strconv"
)
var hello = []string{"Hello", "Hola",
                "Bon Jour", "Ciao", "こんにちは"}     3
func main() {
                var index = 1          4
                if len(os.Args) > 1 {
                                index, _ = strconv.Atoi(os.Args[1])     5
                }

                if(index < 1 || index > len(hello)) {          6
                                index=1
                }
                fmt.Println(hello[index-1])          7
}
```

SWAP.GO

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
}
```

SWAP.GO - EXPLAINED

build an executable versus a shared package with main as the entry point method (1)

import fmt package (2)

short variable declaration for a and b with int type implied (3)

call swap1 function and pass arguments by value (4)

call swap2 function and pass arguments by pointer (5)

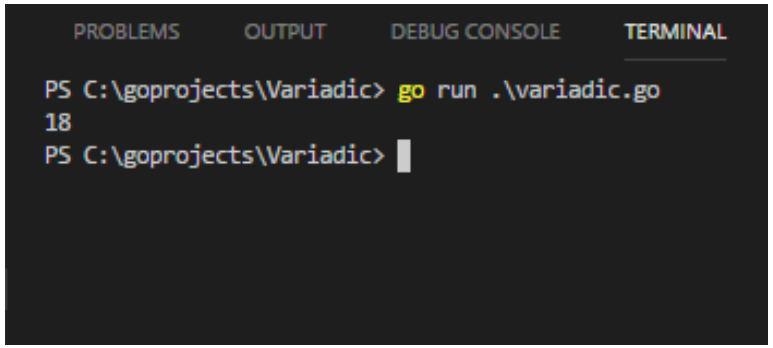
define swap1 function: two integer arguments, return swapped two integer arguments (6)

define swap2 function: two integer pointer arguments. Swap values at pointers. (7)

```
package main                                    1

import "fmt"                                    2

func main() {
    a := 5                                      3
    b := 10
    a, b = swap1(a, b)                          4
    fmt.Println(a, b)

    swap2(&a, &b)                               5
    fmt.Println(a, b)
}

func swap1(x int, y int) (int, int) {
    return y, x                                 6
}

func swap2(x *int, y *int) {
    *x, *y = *y, *x                             7
}
```

VARIADIC.GO

Go Language supports variable length argument list for functions.




```
package main

import "fmt"

func main() {

    t1 := total(1, 5, 9, 3)
    fmt.Println(t1)
}

func total(values ...int) int {
    var result int
    for _, item := range values {
        result += item
    }
    return result
}
```

VARAIDIC .GO - EXPLAINED

build an executable versus a shared package with main as the entry point method (1)

import fmt package (2)

call total function with variable number of arguments (3)

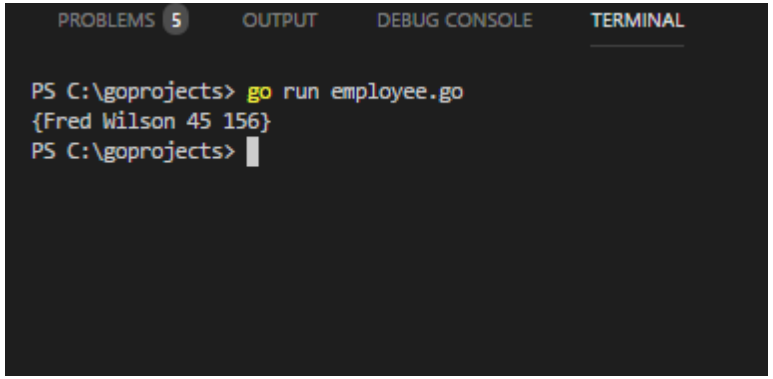
function signature for variadic function (4)

sum the arguments (5)

```
package main          1

import "fmt"          2

func main() {

    t1 := total(1, 5, 9, 3)   3
    fmt.Println(t1)
}

func total(values ...int) int {     4
    var result int
    for _, item := range values {
        result += item              5
    }
    return result
}
```

EMPLOYEE.GO

Creating custom / composite types (struct) is easy in Go Language.




```
package main

import "fmt"

type employee struct {
    firstName string
    lastName string
    age int
    weight int
}

func main() {
    fred := employee{firstName: "Fred",
            lastName: "Wilson", age: 45, weight: 156}
    fmt.Println(fred)
}
```

EMPLOYEE.GO - EXPLAINED

build an executable (1)

import fmt package (2)

define the employee struct composite type (3)

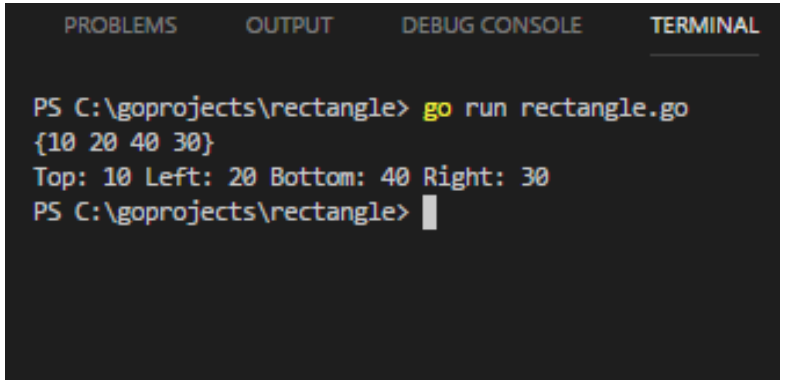
declare variable fred as a struct (employee) and initialize fields (4)

display fred variable (5)

```
package main          1

import "fmt"          2

type employee struct {
    firstName string
    lastName string          3
    age int
    weight int
}

func main() {                4
    fred := employee{firstName: "Fred",
            lastName: "Wilson", age: 45, weight: 156}
    fmt.Println(fred)        5
}
```

RECTANGLE.GO

Like a class, you can assign functions to a struct, which is then called a method. This is object-oriented programming in Go Language.




```
package main

import "fmt"

func main() {
    r1 := rectangle{10, 20, 40, 30}
    r1.Print()
    r1.Draw()
}

type rectangle struct {
    top int
    left int
    bottom int
    right int
}

func (r rectangle) Draw() {
    fmt.Printf("Top: %d Left: %d Bottom: %d Right: %d",
        r.top, r.left, r.bottom, r.right)
}

func (r rectangle) Print() {
    fmt.Println(r)
}
```

RECTANGLE.GO - EXPLAINED

build an executable (1)

import fmt package (2)

declare a rectangle and initialize the fields (3)

Print is a rectangle method and called using the dot syntax (4b)

Draw is a rectangle method and called using the dot syntax (4a)

define the rectangle type (5)

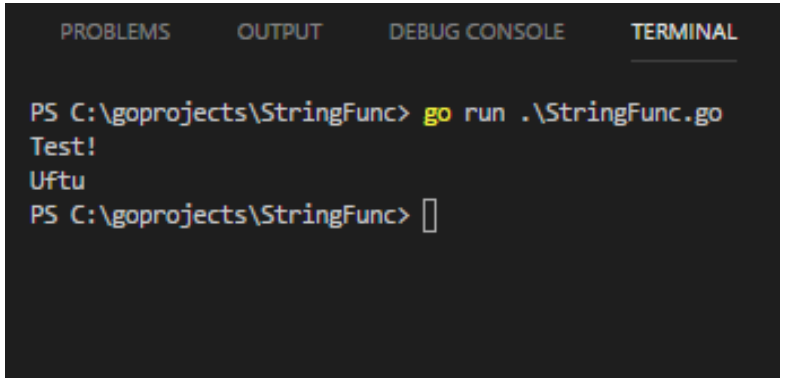
define a Draw method for rectangle struct (6)

define a Print method for rectangle struct (7)

```
package main                                    1

import "fmt"                                    2

func main() {
    r1 := rectangle{10, 20, 40, 30}                    3
    r1.Print()
    r1.Draw()                                   4
}

type rectangle struct {
    top int
    left int                                        5
    bottom int
    right int
}

func (r rectangle) Draw() {                         6
    fmt.Printf("Top: %d Left: %d Bottom: %d Right: %d",
        r.top, r.left, r.bottom, r.right)
}

func (r rectangle) Print() {                        7
    fmt.Println(r)
}
```

STRINGFUNC.GO

Functions are first class entities in Go Language. As such, functions can be used as parameters, and return variables.




```
package main

import "fmt"

func main() {
    fmt.Println(stringModify("Test", addExclamation))
    fmt.Println(stringModify("Test", right1))
}

type modifier func(string) string

func stringModify(s string, f1 modifier) string {
    return f1(s)
}

func addExclamation(s string) string {
    return s + "!"
}

func right1(s string) string {
    var temp string
    for _, char := range s {  // byte array | only ANSI
        temp = temp + string(char+1)
    }
    return temp
}
```

STRINGFUNC.GO - EXPLAINED

build an executable(1)

import fmt package (2)

call function to modify string (3)

define a function type for a function accepting a string parameter and returning a string (4)

define stringModify function that calls a function to modify a string (5)

define addExclamation function that modifies a string: add "!" (6)

define right1 function that modifies a string: shift bytes right one code point (7)

access individual bytes of a string and increase each code point by one (8)

```
package main                                                    1

import "fmt"                                                    2

func main() {
    fmt.Println(stringModify("Test", addExclamation))
    fmt.Println(stringModify("Test", right1))                   3
}

type modifier func(string) string                               4

func stringModify(s string, f1 modifier) string {
    return f1(s)                                                5
}

func addExclamation(s string) string {                          6
    return s + "!"
}

func right1(s string) string {                                  7
    var temp string
    for _, char := range s {  // byte array | only ANSI
        temp = temp + string(char+1)                            8
    }
    return temp
}
```

GETURL.GO

Go Language is multi-purpose and ideal for general and system programming. This program makes a networking HTTP request and displays the response.


```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            os.Exit(1)
        }
        fmt.Printf("%s\n\n", url)
        fmt.Printf("%s\n\n", b)
    }
}
```

GETURL.GO - EXPLAINED

build an executable (1)

import required packages (2)

Get function makes a HTTP request; returns response and error object (3)

if error object not nil, an error has occurred and program exits (4)

ReadAll function reads the body text from the resp object (5)

closes connection with server (6)

if error object not nil, an error has occurred and program exits (7)

Display response data (8)

```
package main                                    1

import (
    "fmt"
    "io/ioutil"                                 2
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)              3
        if err != nil {                         4
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)     5
        resp.Body.Close()                       6
        if err != nil {
            os.Exit(1)                          7
        }
        fmt.Printf("%s\n\n", url)
        fmt.Printf("%s\n\n", b)                 8
    }
}
```

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 and 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

The Basics

Go Programming

TYPES

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

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

This module also reviews various operations, including Boolean, arithmetic, and bitwise that apply to standard types.

STANDARD TYPES

Type	Description
bool	true or false
byte	Int8
float32	32-bit floating point variable. Mantissa reliable to 7-bits.
float64	64-bit floating point variable. Mantissa reliable to 15-bits.
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--

POP QUIZ: WHAT IS THE ANSWER?

```
package main

import "fmt"

func main() {

    fmt.Println(10*10+10/10)

}
```

10 MINUTES

MAXIMUM SIZE

The maximum or minimum 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 sets aside memory for a specific type and value. You can declare variables using various syntax. A variable declaration consists of an identifier, type, and value. The type and value are optional. The type can be inferred, which is different than dynamic. The value can default to some variation of zero.

The identifier is essentially a label for the memory location where the data resides.

VARIABLES - 2

Variables are memory labels. You can used pointers to access the underlying address. More about pointers later.

```
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 methods to declare a variable.

declare a typed variable of a specific type (1)

declare a variable with a default value (2)

declare a variable with type inference (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 Language programmers.

Short variable declarations must be done within a function and not at file scope.

```
package main

func main() {

    a := 1

    b, c := 5, 6

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

}
```

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
- Methods that start 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 relies 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. However, there are three ways to create something similar.

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. First, use the bitwise left shift (1 <<) of iota on the starting flag. The remaining flags can then default to their proper bitwise representation. Second, combine the flags with the bitwise or (|).

```
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)
}
```

BOOLEAN

There are two Boolean values: true and false. Unlike C++, 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 the side affects of short circuiting by full evaluating Boolean expressions before the Boolean operation.

The Boolean operators can be used with similar types: Booleans, numbers, structs, and custom type. 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. Casting from smaller to larger types is safe. The results of casting from larger to smaller types is undefined.

You can also use integers with the bitwise operator.

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))

}
```

WANT SOMETHING BIG

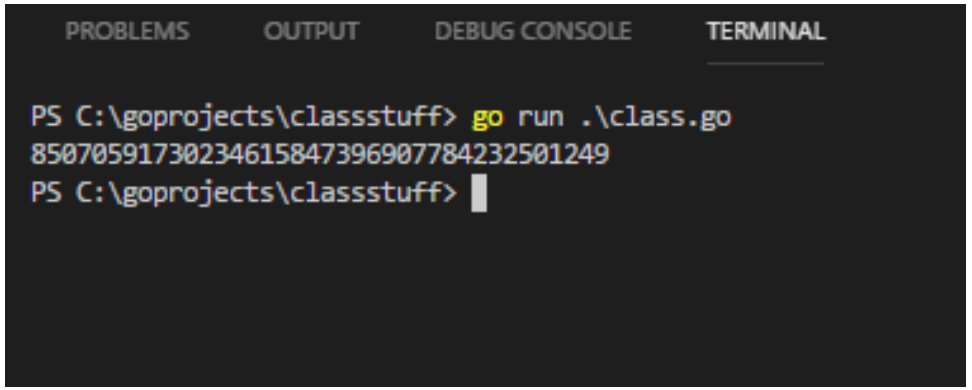
Do you need a bigger version of an integer or float? You will find it in math/big package.

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

Note: using big types to avoid *unknown* overflows is not recommended.

WANT SOMETHING BIG - 2




```
package main

import (
    "fmt"
    "math"
    "math/big"
)

// (Public) Returns F(n).
func main() {

    fmt.Println(mul(big.NewInt(math.MaxInt64),
           big.NewInt(math.MaxInt64)))
}

func mul(x, y *big.Int) *big.Int {
    return big.NewInt(0).Mul(x, y)
}
func sub(x, y *big.Int) *big.Int {
    return big.NewInt(0).Sub(x, y)
}
func add(x, y *big.Int) *big.Int {
    return big.NewInt(0).Add(x, y)
}
```

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    // 1010
    j := i >> 1   // 0101

    k := i | j  // 1111


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

}
```

10 MINUTES

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 rounding. You perform manual rounding using the math.modf function, which returns the whole and fractional parts.

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

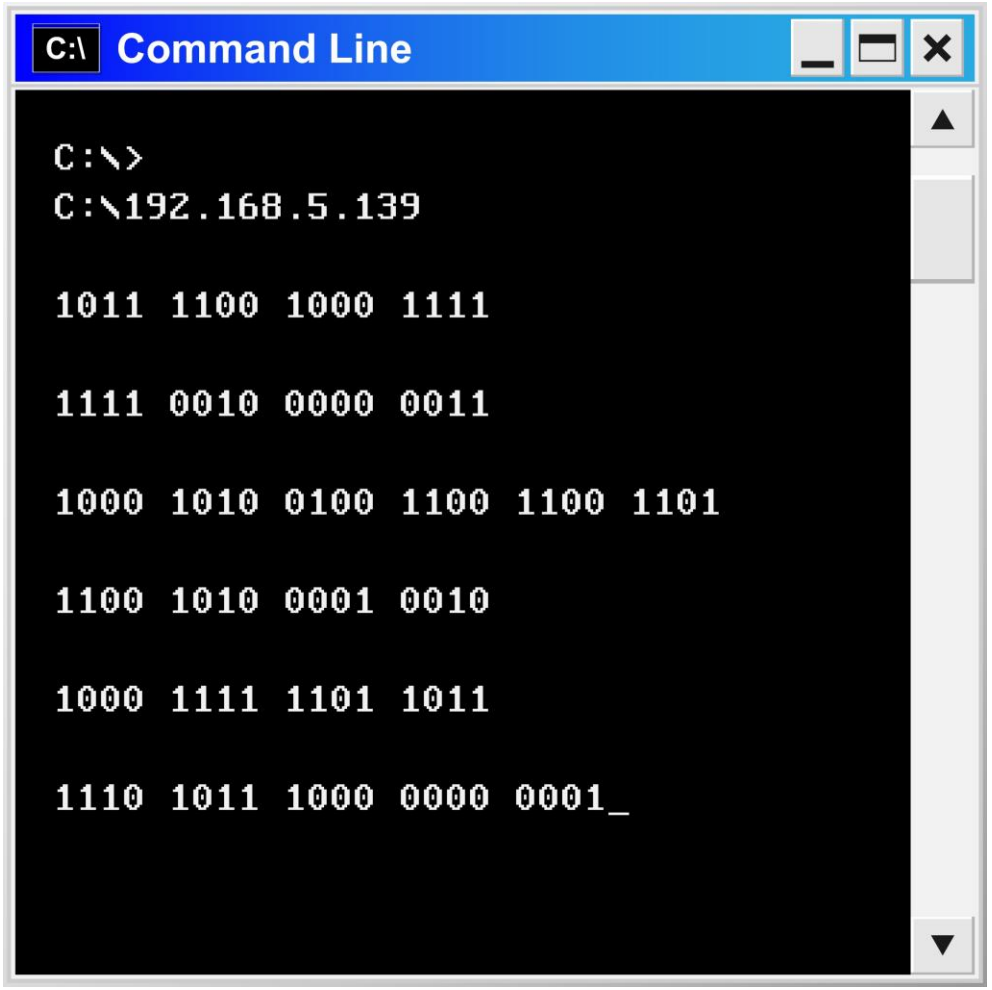
INTERESTING

```
1   package main
2
3   func main() {
4
5       var float32 = 10
6
7       print(float32)
8
9
10
11
12      var a float32
13  }
14
```

float32 is not a type

float32 int

CONSOLE READ



BUFFER READ

You can read from the console using a stream, which can be stdin (i.e., console / terminal).

```
package main

import (
    "bufio"
    "fmt"
    "os"
)
// This is not perfect code!

func main() {
    reader := bufio.NewReader(os.Stdin)
    for {
        fmt.Print("Enter text: ")
        text, _ := reader.ReadString('\n')
        fmt.Println(text)
        if len(text) == 2 {
            print("done")
            break
        }
    }
}
```

SCAN FUNCTIONS

Scanln defaults to reading from the console / terminal. Empty reads do not leave residual characters in the input variable.

```
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

THE GAME

User gets 5 guesses at a number. Here are the steps of the Guessing Number game:

1. Prompt and then enter number to guess (< 100)
2. Prompt for guess
3. Enter guess
4. If correct, end game. List number of guesses and congratulations.
5. After 5th incorrect guess:
    A. Display correct number
    B. Display game over message
    C. End game
6. Display “high” or “low” depending on guess
7. Go to Step 2

Bonus: calculate target number using random number generation and a dynamic seed.

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 done. If false, the if block is not done.

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 done. 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. *Expression switches* switch on an expression, including string expressions.

- 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 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, you do not continue between cases within a switch statement, such as in C++. You 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 a type value where execution jumps to a matching statement. The type value should be of interface {} type.

- There is no fall through between case statements.
- The default statement is the else statement of the switch

Interface {} 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 syntaxes 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.

EXCEPTIONAL EVENTS

WHY DOES GO NOT HAVE EXCEPTIONS?

We believe that coupling exceptions to a control structure, as in the try-catch-finally idiom, results in convoluted code. It also tends to encourage programmers to label too many ordinary errors, such as failing to open a file, as exceptional.

Go takes a different approach. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. A canonical error type, coupled with Go's other features, makes error handling pleasant but quite different from that in other languages.

Go also has a couple of built-in functions to signal and recover from truly exceptional conditions. The recovery mechanism is executed only as part of a function's state being unwound after the scope of an exception has been identified.

DEFER

The defer statement defers the execution of a function or method until the surrounding function or method is preparing to exit. The defer method is called before the return results have been evaluated.

You can have more than one defer statement. They are execute Last In First Out (LIFO).

In the defer function or method perform cleanup that must be completed in all circumstances, 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

Exception handling in the Go Language uses the panic and deferred functions to recover.

The idiomatic way for handling known problems is to return an error type as the sole value or one of many return values (typically the error code is the last value).

Your program should handle known problems gracefully. Exceptions, as unexpected events, use the panic function.

Call the recover function in a deferred method to suppress a pending panic. In the deferred method, you should take the opportunity to log the error and implement a remedy.

PANIC / RECOVER - 2

Call the panic method to essentially throw an exception.

1. Execution in the current function immediately stops
2. Deferred methods in functions are called
3. Panic walks the call stack
4. The stack walk ends when a recover is located on the call stack in a deferred method
5. If no recover is found, the program 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 EXCEPTION

You can check for a specific error versus suppressing all errors. Check the error message to determine the error. Handling unknown errors can pose a risk.

The recover function returns the error object as an interface {}. For that reason, you do not have immediate access to the *error* interface and the Error function, which returns the string message. Use type assertion to cast to the appropriate type.

If error not handled, repanic the error to continue up the call stack.

SPECIFIC EXCEPTION - EXAMPLE

```
package main
import (

    "fmt"

    "os"

)
func main() {
    defer func() {

        err := recover()  // interface {}

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


        if 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 type. Error types adhere to the Error 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, 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 4 – Factorial

CALCULATE FACTORIALS

A factorial is the product of multiplying successful values. For example, 5! is 1*2*3*4*5. The result is 120.

Create an application that generates the result of a factorial using a for loop. Read the factorial from the command line and then calculate and display the result.

Provide exception handling if factorial less than 0 or greater than 100: use the panic, recover, and defer commands. If exception occurs, display error message and exit.

Collections

arrays, slices, and maps

5

ARRAYS

Arrays in the Go Language are a fixed length mutable 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 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 arrays 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 variable-length collection containing elements of the same type. Slices can be shrunk by slicing or enlarged using the append function. Every slice has an underlying hidden array, which defines the capacity. You can create multi-dimensional slices by assigning slices to slices.

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.

.

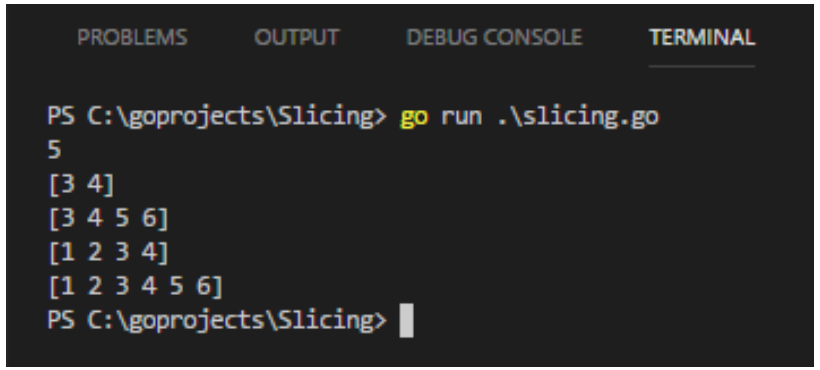
```
package main

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 OPERATIONS

Slice operations can be applied to either arrays or slices. Here are the 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 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 data between two slices, be careful. You are references 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. 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

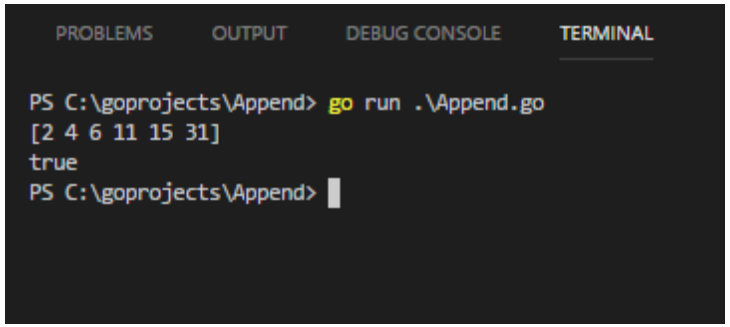
```
package main

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

You can create a custom sort using the sort.Sort method and providing a sort routine.

This is the sort interface.

```
type Interface 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.

Maps are reference types.

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

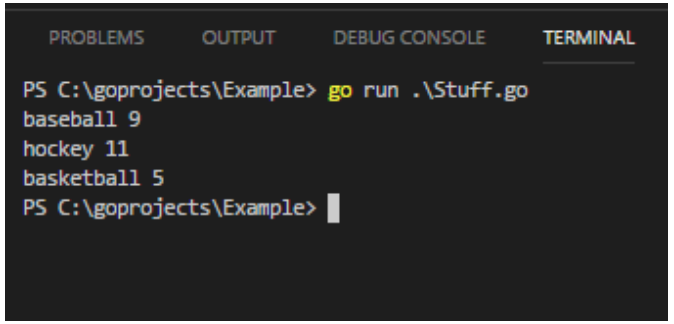
- Delete a value

  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)
    }

}
```

NIL SLICES

It is okay to add elements to a nil slice with the exception of a map slice. For a map slice, adding elements to a nil instance will cause a panic to be raised.

.

```
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. You can confirm the default value using the right most return value.

```
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

When listing the elements of a composite literal, the elements, even the last member, should conclude with a comma. This is part of the semicolon rule.

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 uses a simple rule to insert semicolons automatically as it scans, so the entered text is mostly free of them.

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

You can add semicolons to separate statements on a single line of source code.

SEMICOLON RULE - 2

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

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

.

One affect of semicolon insertion is that an opening brace cannot be alone on a single line.

POP QUIZ: WHY IS THE FINAL COMMA NECESSARY?

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

5 MINUTES

Lab 5 – AIRPORT

AIRPORT ITINERARY

Create a map of airport codes. As a command prompt, accept a airport itinerary (airport codes) as command arguments and display the trip. Try these itineraries:

BKK LAX JFK

LHR ATL ORD

SIN LA DFW

For example, "BKK LAX JFK" would return:

Bangkok to Los Angeles to New York

Make sure program has error detection.

Airportc	Code
Atlanta	ATL
Beijing	PEK
Chicago	ORD
London	LHR
Tokyo	HND
Los Angeles	LAX
Paris	CDG
Dallas	DFW
Frankfurt	FRA
Denver	DEN
Hong Kong	HKG
Madrid	MAD
Dubai	DXB
New York	JFK
Amsterdam	AMS
Jakarta	CGK
Bangkok	BKK
Singapore	SIN
Guangzhou	CAN
Shanghai	PVG

AIRPORT CODES

You can copy the airport codes from a number of websites online, such as.

http://www.nationsonline.org/oneworld/major_world_airports.htm

Functions

procedure, functional, anonymous

FUNCTIONS

Functions are a fundamental building block in most languages. This is the bedrock of procedural, modular programming, and object oriented programming. In Go Language, functions are first class citizens and can be used as variables, parameters, and return values.

Here is the syntax for a function.

```
func functionName(optional parameters) optional return type {
        body
}

func functionName(optional parameters) (optional return types) {
        body
}
```

FUNCTIONS EXPLAIN - PARAMETERS

- A function has zero or more parameters. If more than one parameter, provide a comma separated list.
- For variadic functions, the last function parameter is ellipses (...). Variadic functions have a variable length parameter list. The parameter is passed into the function as a slice.

.

```
package main

import "fmt"

func main() {

    funcA(5, "cool")
    funcB(5, true, false, false)
}

func funcA(a int, b string) {
    fmt.Println(a, b)
}

func funcB(a int, b ...bool) {
    fmt.Println(b)
}
```

FUNCTIONS EXPLAIN – RETURN VALUES

A function can return zero or more values. If more than one return value, provide a comma separated list within parentheses "()".

- The return values can be either named or unnamed; but not a mixture.
- If unnamed, all return values must be specified.
- If named, you can return all values or none; but not mixture.
- Functions that have a return value must have at least one return statement or call panic as the final statement.

.

FUNCTIONS EXPLAIN – RETURN VALUES - 2

```
package main

import "fmt"

func main() {

    fmt.Println(funcA())
    fmt.Println(funcB())
    fmt.Println(funcC())
    fmt.Println(funcD())
    fmt.Println(funcE())
    fmt.Println(funcF())
}

func funcA() int {
    return 5
}

func funcB() (int, int) {
    return 5, 10
}
```

```
func funcC() (a int, b int) {
    a = 12
    b = 18
    return a, b
}

func funcD() (a int, b int) {
    a = 20
    b = 30
    return
}

func funcE() (a int, b int) {
    a = 12
    b = 18
    return b, a
}

func funcF() (a int, b int) {
    a = 12
    b = 18
}
```

POP QUIZ: DOES THIS COMPILE?

```
func doesThisWork() int {

    if true {

        return 5

    }

}
```

5 MINUTES

RESERVED FUNCTIONS

The Go Language reserves two function names: init and main. Both are called automatically and should not be called directly. The init and main functions should have no parameters or return values.

The init function is optional; but the main function is required for an executable. Here is the order of invocation:

1. The init function is called on each imported package first
2. If several packages are imported, init functions are called in textual order for each.
3. The init function is called in the executable.
4. The main function is called in the executable.

EXIT

Call the Exit function in the os package to immediately exit a Go Language program and return a status.

Note: defer methods will not be called. For that reason, only use os.Exit in critical situations.

If running the application from the command prompt, *go run* will display the exit code.

GetExitCodeProcess - IPC

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

CLOSURE EXAMPLE 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

GENERIC FUNCTION

Languages, such as Java and C++, support generic or templated functions. Go Language does not have this feature.

In C++, the templated swap function can be called with various types and prevents redundant code, which is of course is a bad practice.

How is this done indirectly in the Go Language?

```
// C++ code


template <class T>
void swap(T& x,T& y)
{
      T temp;
      temp=x;
      x=y;
      y=temp;
}


swap(5, 20)


swap('a', 'b')
```

GENERIC FUNCTION - 2

Using the interface type, you can create something similar to a generic method. Be warned – this is potentially unsafe. An empty interface is the anything type in the Go Language. Similar to the object type in C# or Java.

```
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
}
```

FUNCTION POINTERS

You can declare a function pointer with the type keyword. This defines a function type based on the function signature. The resulting function type can be used as a variable, parameter, and 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))
}
```

SCOPE

Scope defines where a variable is visible. Variables are either global or local. Local variables can hide a global variable that has the same identifier. The local variable is a shadow variable when this occurs.

```
package main

import (
    "fmt"
)

var i = 5

func main() {

    i := 4.5

    {
        i := "test"
        fmt.Println(i)
    }
    fmt.Println(i)
}
```

SELECT STATEMENT

In the Go language, coroutines are equivalent to threads and are asynchronous function calls. Every program has a primary thread, where main is the entry point. A coroutine is a separate and parallel path of execution of the designated function.

As a form of interthread communication, threads can communicate with each other via channels.

You can block and wait on communication via a select statement. A select block is similar to a switch block except:

- You are switching on channels
- The select statement will block waiting for data in the channel

Note: Threading is a major topic and something covered in depth in the advanced class.

This slide is just to introduce select as another transfer of control statement.

SELECT STATEMENT - EXAMPLE

```
package main

import (
    "fmt"
    "time"
)

func main() {

    channel1 := make(chan string)
    channel2 := make(chan string)

    go func() { // thread
        time.Sleep(2 * time.Second)
        channel1 <- "go routine1"
    }()  // thread is invoked
    go func() { // thread
        time.Sleep(4 * time.Second)
        channel2 <- "go routine2"
    }() // thread is invoked
}
```

```
for i := 0; i < 2; i++ {
    select {
    case message1 := <-channel1:
        fmt.Println("Channel1", message1)
    case message2 := <-channel2:
        fmt.Println("Channel2", message2)
    }
}
```

FUNCTION OVERLOADING

GO Language does not support function overloading.

```
package main

import (
    "fmt"
)

func FuncA() {

}

func FuncA(a int) {
    fmt.Println(a)
}

func main() {

    FuncA(5)
}
```

Lab 6 – MATRIX

MATIX OPERATIONS

Create an array 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.

Strings

slices, characters, and bytes

STRINGS

Go Language supports Unicode strings. Strings in Go Language are also immutable. Strings are slices and the slice operations apply.

When using the append function, the target string may change and not be replaced. As with an slice, this depends on the capacity of the underlying array.

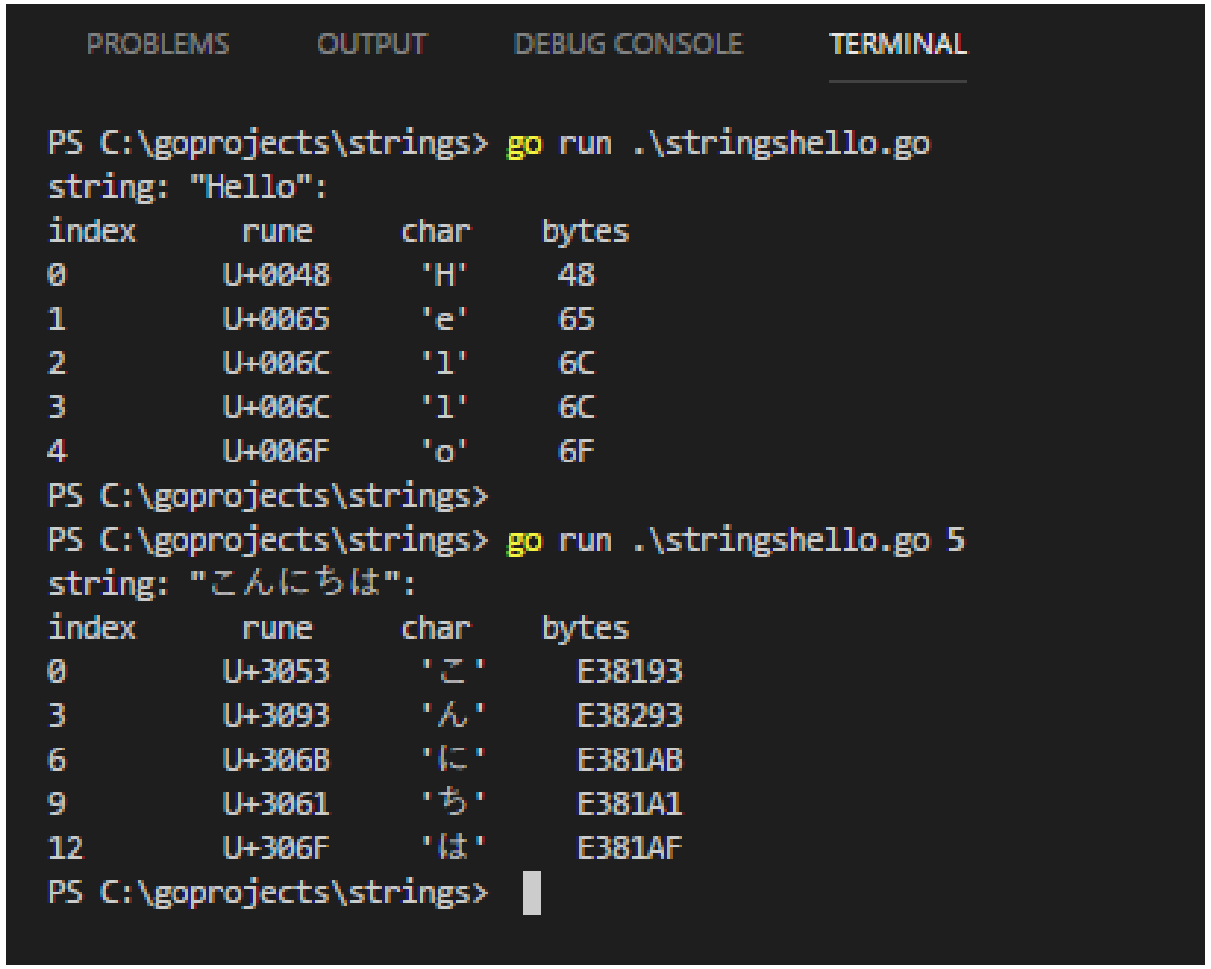
Go Language provides substantial support for string manipulation. There several standard methods but also additional functionality in several packages, especially the Strings, Strconv, and Unicode packages.

Strings default to an empty string; not a nil string.

STRING EXAMPLE

```
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])
    temp := hello[index-1]
    fmt.Printf("string: \"%s\":\n", temp)
    fmt.Println("index rune char bytes")
    for index, char := range temp {
        fmt.Printf("%-2d %U '%c' %X\n",
            index, char, char,
            []byte(string(char)))
    }
}
```

STRING EXAMPLE - 2



UNICODE

Unicode is a multiple byte character set. There is a codepoint (rune) for each character around the world. The code points range from 0x0 to 0x10FFFF, which is also unicode.MaxRune.

Unicode is supported by various character encodings. UTF-8 is the most popular of those encodings and used by Go Programming. UTF-8 uses one byte for the first 128 code points and up to four characters for the remaining code points.

Because of the mixed character lengths, the challenge with UTF-8 is getting the length or byte location within a string.

In Go Language, each code point is called a rune which is synonymous with a int32.

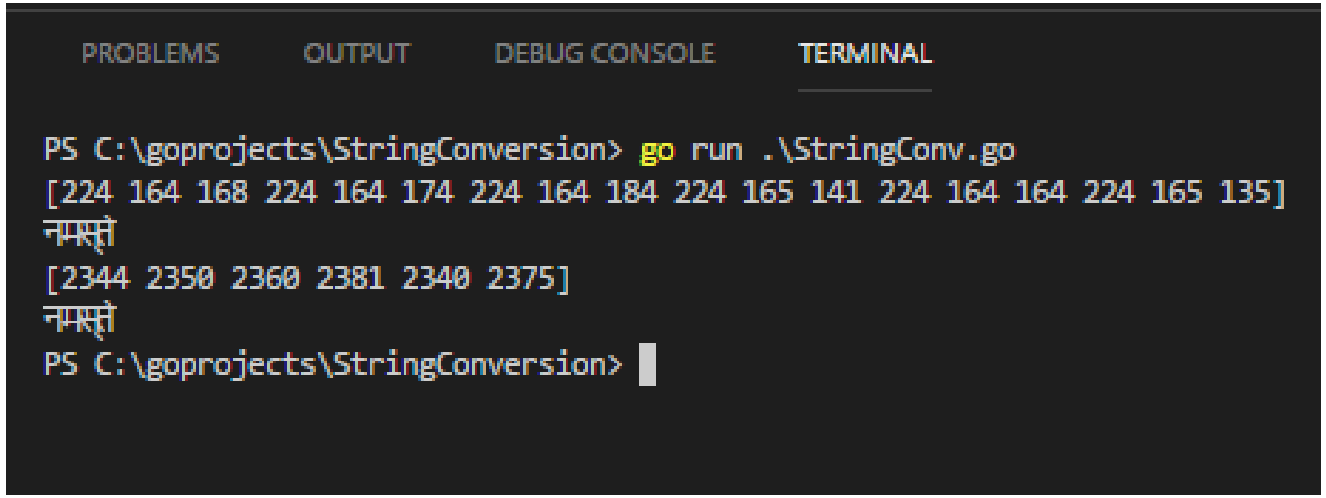
STRINGS AS BYTES

```
package main

import "fmt"

func main() {
    str := "नमस्ते"

   bytes := []byte(str)
    fmt.Println(bytes)

    str2 := string(bytes)
    fmt.Println(str2)

    runes := []rune(str)
    fmt.Println(runes)

    str3 := string(runes)
    fmt.Println(str3)
}
```

You can convert between strings and something else:

- String to bytes
- Bytes to string
- Strings to rune
- Rune to strings
- Strings to characters

STRINGS AS BYTES - 2



STRING LITERALS

Literals are defined within quotes. Double quote is the interpreted literal. Back ticks (`) are for raw string literals and multi-line strings.

Escape sequence	Description
\\	Backslash
\000	Unicode character
\”	Single quote*
\’	Double quote*
\a	Bell
\b	Backspace
\f	Formfeed
\n	Linefeed
\r	Carriage return
\t	Tab
\uhhhh	Unicode character
\v	Vertical tab
\xhh	Unicode character

STRING EXAMPLE

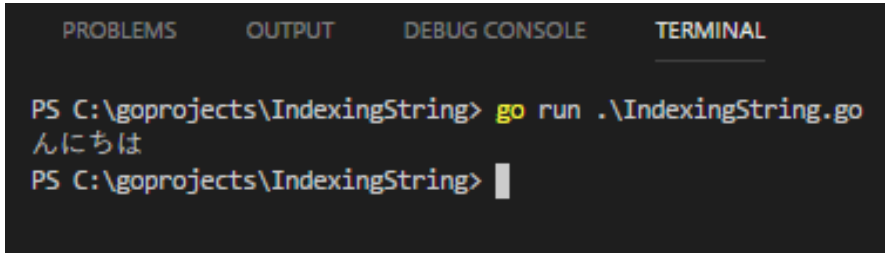

```
package main

import "fmt"

func main() {

    fmt.Println("test1\t1\t2\ntest2\t3\t4\n")
    fmt.Println(`test1\t1\t2\ntest2\t3\t4
        .....
        cool!`)
    fmt.Println("\u3041")
    fmt.Println("\x78")
}
```

STRINGS ARE SLICES

Strings are slices and sliced along bytes. For ASCII (lets say for English strings), there is a one-to-one mapping. This makes accessing ASCII strings straightforward. This however potentially poses a challenge for other languages where character length is greater than a byte.

There are two consistent options: convert string to a rune slice or use these functions: strings.Index or strings.LastIndex.

STRING EXAMPLE




```
package main

import (
    "fmt"
    "strings"
)

var hello = "こんにちは "

func main() {
    indexb := strings.Index(hello, "ん")
    indexe := strings.LastIndex(hello, " ")
    fmt.Println(hello[indexb:indexe])
}
```

STRING OPERATIONS

These are the standard string operations.

Strings are slices. All of the standard slice operations also apply to strings.

The comparison operators for strings are:

<, <=, ==, ++, !=, >, and >=

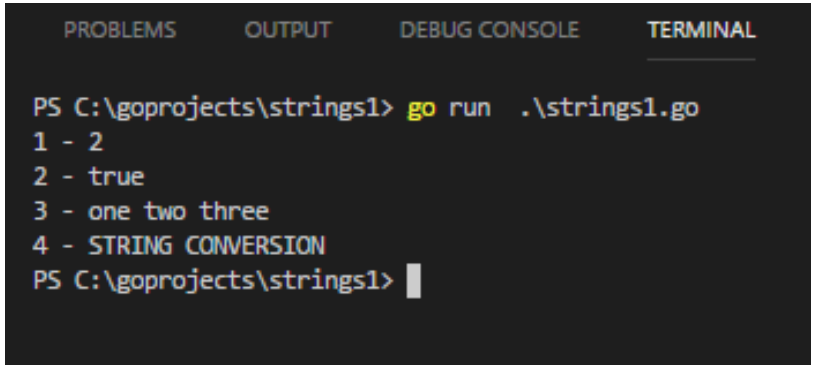
Operation	Description
L1+=L2	Append string L2 to L1
L1+L2	Concatenate L2 to L1
L1[n]	Access nth raw byte
L1[b:e]	String from index b to e-1
L1[b:]	String from index to end
L1[:e]	String from 0 to index
len(str)	Number of bytes
len([] rune(L1))	Number of characters
[] bytes(L1)	Convert string to slice of bytes
string([] byte)	Convert [] byte to string

STRINGS PACKAGE

The strings package contains a myriad of functions that are commonly used for string manipulations. Here are some of the prevalent functions.

Operation	Description
Contains(s, t)	Returns true if t in s
Count(s, t)	Count of t in s
EqualFold(s1, s2)	Non case sensitive equality
Fields(s)	Returns string slice split along whitespace
HasPrefix(s, t)	True if s begins with t
HasSuffix(s, t)	True if s ends with t
Index(s, t)	Returns index of first t in s
Join([] s, c)	Joins strings with c delimiter
LastIndex(s, t)	Last occurrence of t in s
Repeat(s, i)	Repeat s times i number of times
ToLower(s)	Convert string to lowercase
ToUpper(s)	Convert string to uppercase
Trim(s, t)	Trim t from either end of s
TrimSpace(s)	Trim spaces from either end of s

STRINGS PACKAGE - 2



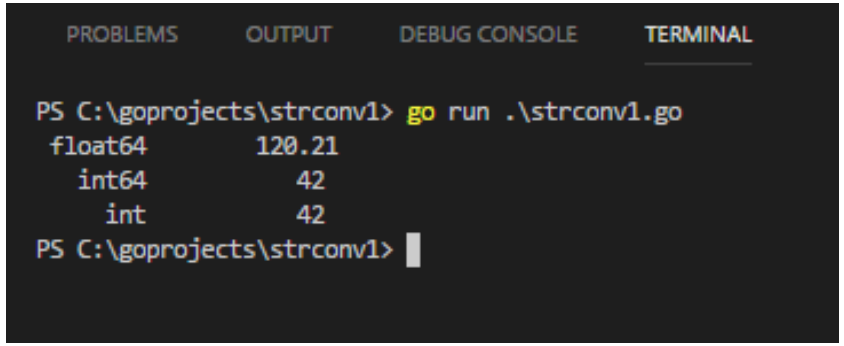

```
package main

import (
    "fmt"
    "strings"
)

func main() {

    test := "String conversion"
    test2 := []string{"one", "two", "three"}
    fmt.Printf("1 - %d\n", strings.Count(test, "i"))
    fmt.Printf("2 - %t\n", strings.HasPrefix(test, "Str"))
    fmt.Printf("3 - %s\n", strings.Join(test2, " "))
    fmt.Printf("4 - %s\n", strings.ToUpper(test))
}
```

STRCONV PACKAGE

The strconv package has several functions for converting to and from strings.

Operation	Description
Atoi(s)	Convert string to int
FormatBool(b)	Return “true” or “false”
FormatFloat(f, fmt, prec, bits)	Return float as string
FormatInt(i, base)	Return int as a string
FormatUInt(u, base)	Return uint as a string
IsPrint(c)	True if c is printable
Itoa(i)	Convert int to string
ParseBool(s)	Return s as bool
ParseFloat(s, bits)	Return s as float
ParseInt(s, base, bits)	Return s as int
ParseUint(s, base, bits)	Return s as uint

STRCONV PACKAGE - 2




```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    x, _ := strconv.ParseFloat("120.21", 64)
    fmt.Printf("%8T\t%5v\n", x, x)
    y, _ := strconv.ParseInt("42", 10, 0)
    fmt.Printf("%8T\t%5v\n", y, y)
    z, _ := strconv.Atoi("42")
    fmt.Printf("%8T\t%5v\n", z, z)
}
```

UNICODE PACKAGE

The Unicode package has a variety of functions mostly to confirm the status of a Unicode character.

Operation	Description
IsControl	True if control character
IsDigit	True if decimal digit
IsLetter	True if letter
IsLower	True if lowercase letter
IsPrint	True if printable character
IsPunct	True if punctuation character
IsSpace	True if whitespace character
IsSymbol	True if symbol character
IsUpper	True if uppercase character
ToLower	Convert to lowercase character
ToUpper	Convert to uppercase character

FORMAT COMMANDS

The format commands are in the fmt package. Many of these functions have already been demonstrated. Some write to stdout, a string, or stream. Here are the more common functions:

Operation	Description
Errorf(format, args…)	Return error info
Fprint(writer, args…)	Write the args to a writer stream
Fprintf(writer, format, args…)	Write the args to a writer stream in the specified format
Fprintln(writer, args…)	Same as Fprintf but terminated with a linefeed
Print(args…)	Write args to stdout
Printf(format, args…)	Write the args to the stdout in the specified format
Println(args…)	Same as Println but terminated with a linefeed
Sprint(args…)	Return a string consisting of args
Sprintf(format, args…)	Return a string consisting of args in the specified format
Sprintln(args…)	Return a string consisting of args terminated with a linefeed

APPEND VERSUS JOIN

```
package main

import (
    "bytes"
    "fmt"
)

func main() {

    strings := []string{"one", "two", "three", "four"}
    var buffer bytes.Buffer

    for _, valuestring := range strings {
        buffer.WriteString(valuestring)
    }
    fmt.Println(buffer.String())
}
```

The compound operation += is convenient for combining strings. The string.Join is often a better alternative for performance. Writing into a byte buffer is another solution that may be quicker.

Lab 7- String functions

STRING FUNCTIONS

Create three string functions

ReverseText(string): reverses text within a string.

Test with: 1) Python has superior string manipulation.

ReverseWords(input string, delimiter string): reverses words within a phrase. The delimiter parameter indicates the separator between words.

Test with: 1) Python has superior string manipulation

2) This.is.cool

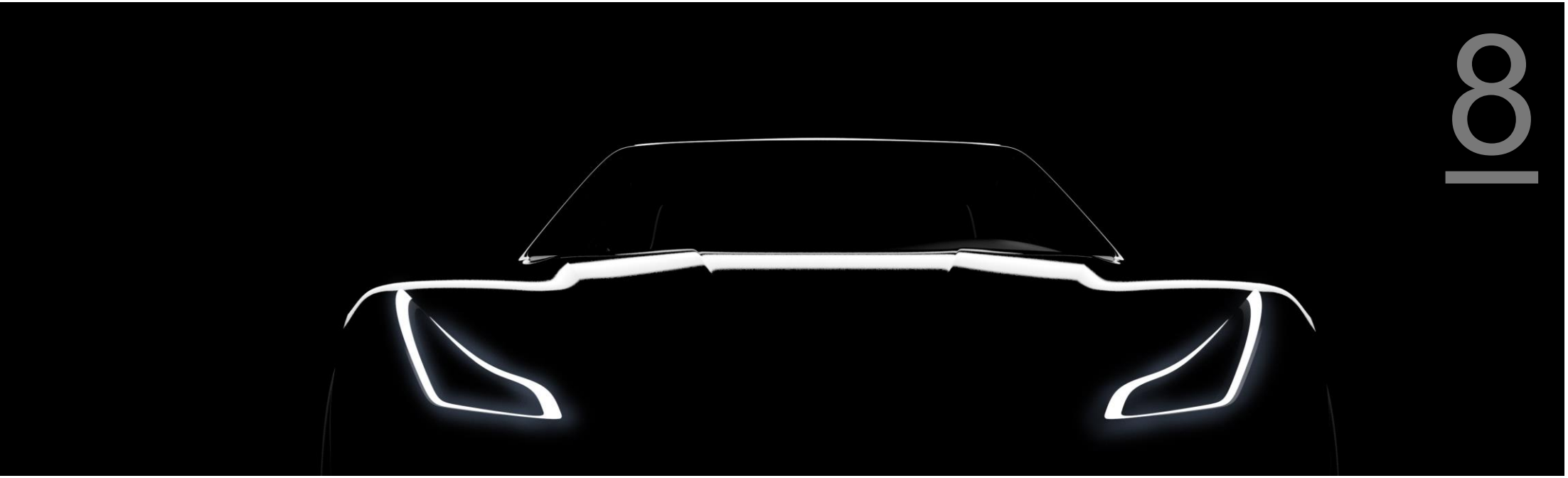
ReverseSentence(input string, delimiters [] string): Reverse sentences where delimiters contains all the possible delimiters.

Test with: 1) Hello. This is a good day!

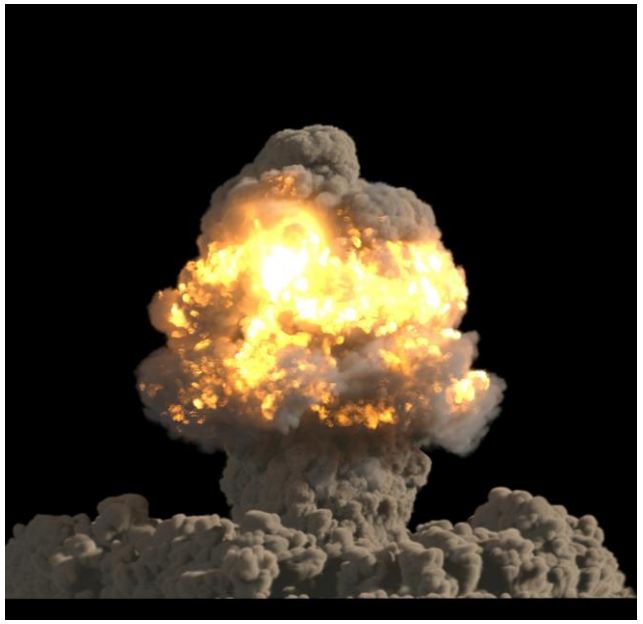
2) What is the answer to life, the universe and everything? 42!

Structures

types, object oriented, composition



NEW APPROACH



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 when helpful.

TERMINOLOGY

Emblematic of the change in perspective is the change of terminology in the Go Language as pertains to object oriented concepts:

- No inheritance
- Class is type
- Instance is value
- Member functions are methods
- *this* is receiver
- Polymorphism is duck typing

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.

The contrast between the two alternatives is best highlighted with the classic Employee example.

C++ INHERITANCE

Here is standard C++ code for inheritance. What is wrong with this code – especially for Fred?

What if Fred wanted a promotion?

What if Fred decided to retire?

Is this truly a *is a kind of* relationship?

```
class Person {

};

class Salaried : public Person {

};

class Hourly : public Person {

};

class Executive: public Person {

};

int main() {
    Executive Bob;
    Salaried Sally;
    Hourly Fred;

    return 0;
}
```

C++ COMPOSITION

Here is the solution for the employee scenario using composition. There are a few benefits including.

- Lazy evaluation
- Extensibility
- Interface driven

From a strategic perspective, this solution can be fully implemented in Go Language.

```
class Person  {
    void SetEmployment(IEmployee *e);
    Employee *pEmployed;
}

class IEmployee {
};

class Salaried : public IEmployee {
};

class Hourly : public IEmployee {
};
. . . . .
int main() {
    Person Bob;
    Person Sally;
    Person Fred;

    return 0;
}
```

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
- Improves readability
- 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
}
```

METHOD

A method is a function that is called on a custom type. The value of the type is usually passed into the method.

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 type 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
}
// Person.city

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

    fmt.Println(c.street, c.city,
        c.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.

```
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 main() {

    var c Person

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

METHOD OVERRIDING

You can override methods inherited from embedded types. Simply implement the same method on the surrounding custom type. If wanted, you can in addition 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. For that reason, a interface is abstract. There is no implementation. However, you can associate types with an interface for the implementation. The related type must implement the entire interface to avoid a compiler error when used as the interface.

This is useful in filtering values passed as a parameter, type of field, return from a method or function, or membership in a collection.

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)

}
```

INTERFACE {}

Interface {} is a special interface that contains an empty set of methods. Every value at a minimum has no methods. For that reason, every value is an interface {} type. As an interface {} value, you are limited. For example, you cannot call any methods on the value. If necessary, the underlying methods can be accessed via type assertion or type switch.

Interface {} is the ideal type when any type is acceptable or required for a parameter, return, or member of a collection.

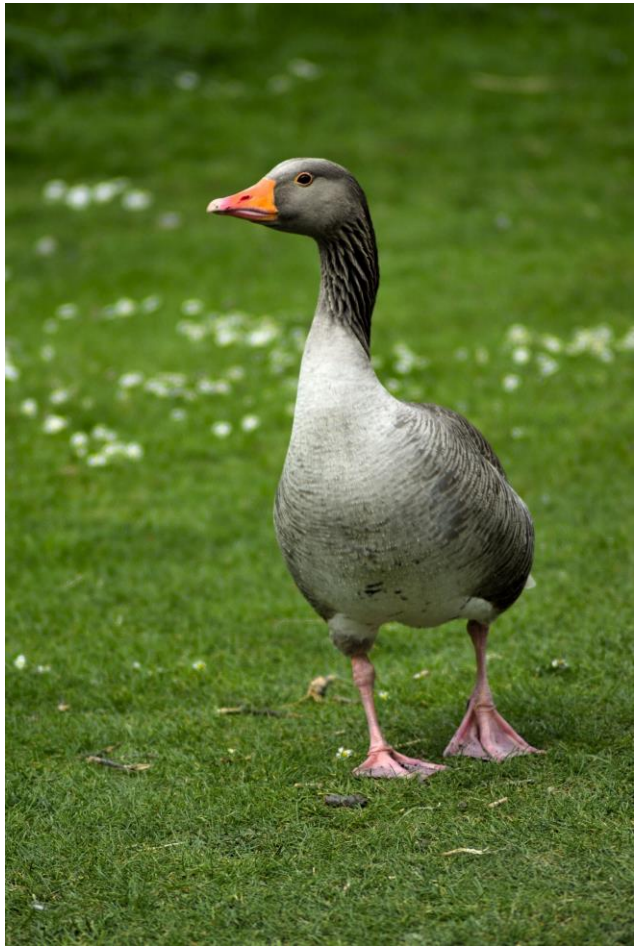
STRUCTURES

The Go Language supports C style structures.

Once defined, you can declare a value from a structure like any type.

Hard to make structures behave like a standard type. For example, there is no operator overloading. Some consider operator overloading to be syntactic candy anyway. You have to build out the behavior of the structure using methods.

DUCK TYPING



In Go Programming, duck typing is done with interfaces. If a value implements a duck interface, it must be a duck. All other values that implement the interface are also assumed to be equally ducks. This may not be true!

POLYMORPHISM

Duck typing is how polymorphism is implemented in the Go Language. 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 is not guaranteed however*. This can lead to interesting results.

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 8- 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.

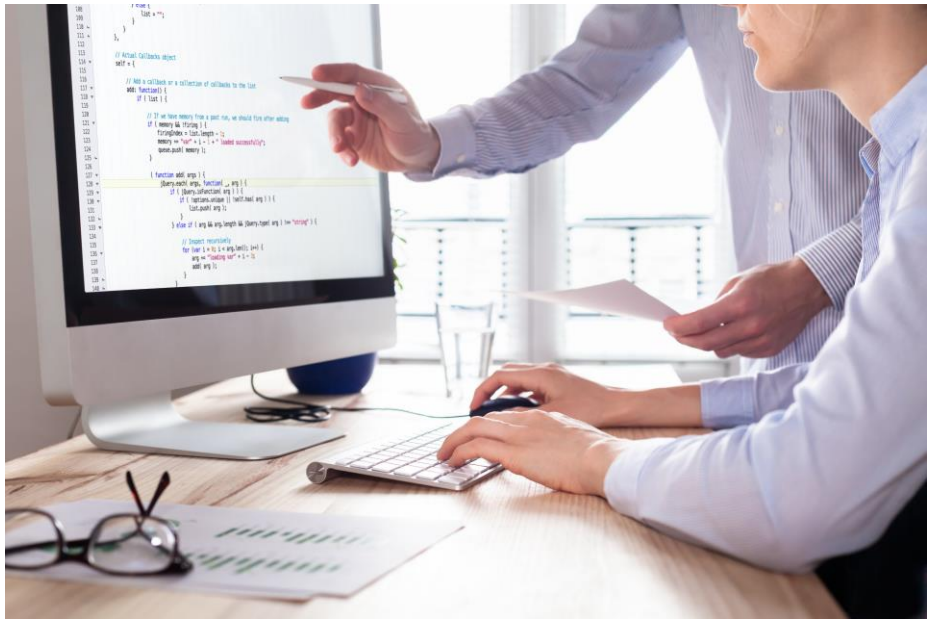
SHAPES - 2

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

Debugging

Walkthrough

WALKTHROUGH - DEBUGGING



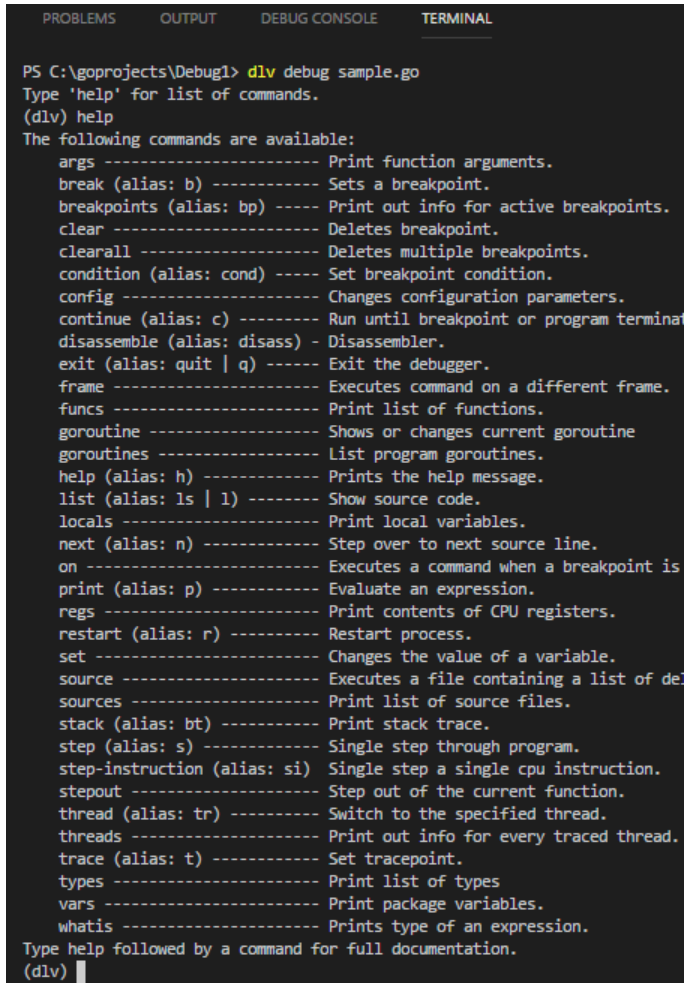
This is a walkthrough of the Delve debugger. Of course, the delve debugger must be installed. Here is the link again:

http://nanxiao.me/en/a-brief-intro-of-delve/

DEBUG - 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()
    }
}
```

WALKTHROUGH – START DEBUGGING



Start debugging:

dlv debug sample.go

The debugger will attached and immediately stop the application. This provides you an opportunity to setup the debugging session.

Other helpful commands:

- Exit to quit
- Restart to start a new debugging session

WALKTHROUGH – BREAKPOINTS

```
(dlv) break main.main
Breakpoint 1 set at 0x4b610a for main.main() C:/goprojects/Debug1/sample.go:48
(dlv) break sample.go:54
Breakpoint 2 set at 0x4b611b for main.main() C:/goprojects/Debug1/sample.go:54
(dlv) breakpoints
Breakpoint unrecovered-panic at 0x42d030 for runtime.startpanic() C:/Go/src/runtime/panic.go:577 (0)
        print runtime.curg._panic.arg
Breakpoint 1 at 0x4b610a for main.main() C:/goprojects/Debug1/sample.go:48 (0)
Breakpoint 2 at 0x4b611b for main.main() C:/goprojects/Debug1/sample.go:54 (0)
(dlv) break walk
Command failed: Location "walk" ambiguous: main.human.walk, main.martian.walk, main.(*human).walk, main.(*martian
(dlv) break main.martian.walk
Breakpoint 3 set at 0x4b5eca for main.martian.walk() C:/goprojects/Debug1/sample.go:33
(dlv) breakpoints
Breakpoint unrecovered-panic at 0x42d030 for runtime.startpanic() C:/Go/src/runtime/panic.go:577 (0)
        print runtime.curg._panic.arg
Breakpoint 1 at 0x4b610a for main.main() C:/goprojects/Debug1/sample.go:48 (0)
Breakpoint 2 at 0x4b611b for main.main() C:/goprojects/Debug1/sample.go:54 (0)
Breakpoint 3 at 0x4b5eca for main.martian.walk() C:/goprojects/Debug1/sample.go:33 (0)
(dlv)
```

Setting breakpoints is typically the first and most important step in debugging. After setting, you then can navigate between breakpoints.

Set break with the breakpoint command.

Breakpoints command will list the breakpoints.

WALKTHROUGH – NAVIGATION

```
(dlv) continue
> main.main() C:/goprojects/Debug1/sample.go:48 (hits goroutine(1):1 total:1) (PC: 0x4b610a)
Warning: debugging optimized function
    43:         for _, person := range partygoers {
    44:                 person.eat()
    45:         }
    46: }
    47:
=>  48: func main() {
    49:         var Bob human
    50:         var Sally human
    51:         var Fred martian
    52:         var Amanda human
    53:
(dlv) step
> main.main() C:/goprojects/Debug1/sample.go:54 (hits goroutine(1):1 total:1) (PC: 0x4b611b)
Warning: debugging optimized function
    49:         var Bob human
    50:         var Sally human
    51:         var Fred martian
    52:         var Amanda human
    53:
=>  54:         invite2Party(Bob, Sally, Fred, Amanda)
    55: }
(dlv)
```

You can run the application between breakpoints with the continue command.

Other helpful commands:

- Step (step in)
- Next (step over)
- Stepout (step out)

WALKTHROUGH - STACK

```
(dlv) locals
(no locals)
(dlv) args
(no args)
(dlv) regs
   Rip = 0x00000000004b611b
   Rsp = 0x000000c042063f00
   Rax = 0x00000000004b60f0
   Rbx = 0x000000000055e008
   Rcx = 0x000000c042024000
   Rdx = 0x00000000004f00c8
   Rdi = 0x00000000000001e0
   Rsi = 0x0000000000000001
   Rbp = 0x000000c042063f78
    R8 = 0x0000000000000200
    R9 = 0x0000000000000008
   R10 = 0x0000000000000000
   R11 = 0x000000c04206e0d7
   R12 = 0x000000c041ffc8f3
   R13 = 0x0000000000000000
   R14 = 0x00000000000000d0
   R15 = 0x00000000000000d0
Eflags = 0x0000000000000204     [PF IF IOPL=0]
    Cs = 0x0000000000000033
    Fs = 0x0000000000000053
    Gs = 0x000000000000002b
   TLS = 0x0000000000278000

(dlv)
```

Examining the stack is easy with commands such as:

- Locals
- Args
- Regs

WALKTHROUGH – MARTIAN EATS

```
(dlv) restart
Process restarted with PID 29240
(dlv) clearall
Breakpoint 1 cleared at 0x4b5e1a for main.martian.talk() C:/goprojects/Debug1/sample.go:29
Breakpoint 2 cleared at 0x4b610a for main.main() C:/goprojects/Debug1/sample.go:48
Breakpoint 3 cleared at 0x4b611b for main.main() C:/goprojects/Debug1/sample.go:54
(dlv) break eat
Command failed: Location "eat" ambiguous: main.human.eat, main.martian.eat, main.(*human).eat, main.(*martian).eat
(dlv) break main.martian.eat
Breakpoint 4 set at 0x4b5f7a for main.martian.eat() C:/goprojects/Debug1/sample.go:37
(dlv) continue
eating sushi
eating sushi
> main.martian.eat() C:/goprojects/Debug1/sample.go:37 (hits goroutine(1):1 total:1) (PC: 0x4b5f7a)
Warning: debugging optimized function
    32:
    33: func (martian) walk() {
    34:         fmt.Println("walking")
    35: }
    36:
=>  37: func (martian) eat() {
    38:         fmt.Println("eating humans")
    39: }
    40:
    41: func invite2Party(partygoers ...iPerson) {
    42:
```

Let us restart the debugging session. Now we want to break only when the Martian eats!

ClearAll will clear the existing breakpoints.

WALKTHROUGH – THREADS / VARIABLES

```
(dlv) threads
  Thread 4968 at :0
  Thread 27156 at :0
  Thread 27580 at :0
  Thread 30120 at :0
* Thread 30796 at 0x4b5f88 C:/goprojects/Debug1/sample.go:38 mai
  Thread 32444 at :0
  Thread 33976 at :0
  Thread 34300 at :0
(dlv) var uni
Command failed: command not available
(dlv) threads
  Thread 4968 at :0
  Thread 27156 at :0
  Thread 27580 at :0
  Thread 30120 at :0
* Thread 30796 at 0x4b5f88 C:/goprojects/Debug1/sample.go:38 mai
  Thread 32444 at :0
  Thread 33976 at :0
  Thread 34300 at :0
(dlv) vars uni
unicode/utf8.first = [256]uint8 [...]
unicode/utf8.acceptRanges = [5]unicode/utf8.acceptRange [...]
time.unitMap = map[string]int64 [...]
unicode.White_Space = (*unicode.RangeTable)(0x55b220)
unicode.initdone· = 2
unicode.C = (*unicode.RangeTable)(0x558ea0)
unicode.Cc = (*unicode.RangeTable)(0x5591a0)
unicode.Cf = (*unicode.RangeTable)(0x5593e0)
unicode.Co = (*unicode.RangeTable)(0x559720)
unicode.Cs = (*unicode.RangeTable)(0x559a20)
unicode.L = (*unicode.RangeTable)(0x5583e0)
unicode.Ll = (*unicode.RangeTable)(0x559c60)
unicode.Lm = (*unicode.RangeTable)(0x559f20)
unicode.Lo = (*unicode.RangeTable)(0x55a260)
unicode.Lt = (*unicode.RangeTable)(0x55a620)
unicode.Lu = (*unicode.RangeTable)(0x55a860)
unicode.M = (*unicode.RangeTable)(0x55a360)
```

The threads and vars command are often helpful in examining an application.

WALKTHROUGH – THREADS

```
(dlv) threads
  Thread 4968 at :0
  Thread 27156 at :0
  Thread 27580 at :0
* Thread 30120 at :0
  Thread 30796 at 0x4b5f88 C:/goprojects/Debug1/sample
  Thread 32444 at :0
  Thread 33976 at :0
  Thread 34300 at :0
(dlv) thread 30120
Switched from 30120 to 30120
(dlv) regs
   Rip = 0x00007ffc354efef4
   Rsp = 0x000000000360fb98
   Rax = 0x0000000000000004
   Rbx = 0x0000000000000000
   Rcx = 0x00000000000000cc
   Rdx = 0x0000000000000000
   Rdi = 0x00000000000000cc
   Rsi = 0x00000000ffffffff
   Rbp = 0x000000000360fd40
    R8 = 0x0000000000000000
    R9 = 0x0000000000000000
   R10 = 0x0000000000000000
   R11 = 0x0000000000000000
   R12 = 0x0000000000000000
   R13 = 0x0000000000000000
   R14 = 0x0000000000000000
   R15 = 0x0000000000000000
Eflags = 0x0000000000000246     [PF ZF IF IOPL=0]
    Cs = 0x0000000000000033
    Fs = 0x0000000000000053
    Gs = 0x000000000000002b
   TLS = 0x00000000002c5000

(dlv) list
Stopped at: 0x7ffc354efef4
=>no source available
Command failed: open : The system cannot find the file
(dlv)
```

Changing the thread context is often useful and affects commands such as regs, locals, args, list, and so on.

Use the thread command to change thread context.

Input / Output

Files

DATA RULES!

Data is the fuel for professional applications. Hard to have a sophisticated application without data storage and manipulation. The term "data" can reference a variety of resources, including file input-output, JSON, STDIN, STDOUT, TCP/IP, and even printing. Each refers to a stream of information flowing from a source to a destination.

Of course, nowadays most data resides in the cloud or backed there.

FILES

Files exist in a variety of types and sizes:

- Configuration files
- System files
- JSON
- XML files
- And more

FILE INPUT / OUTPUT

Import the io/ioutil package to access file input/output functions. This package includes functions for file and directory manipulation and management.

- WriteFile: write bytes to a file
- ReadFile: read bytes to a file

You will need to convert data between bytes and another format when necessary. When writing text for example, you need to convert between bytes and string format.

WriteFile and Readfile is document based input / output where data is transferred as a single instance.

FILE PERMISSIONS

In Go, Unix permissions are used for file permissions. There are three groups of permissions:

- The owner of the file
- Anyone in the same group of the file
- Everyone else

Of course, setting correct permissions is essential for proper application security.

Permission	Description
0000	No permissions
0700	Read, write, and execute for owner
0770	Read, write, and execute for the owner and the file group
0777	Read, write, and execute for everybody
0111	Execute for everybody
0222	Write for everybody
0333	Write and execute for everybody
0444	Read for everybody
0555	Read and execute for everybody
0644	Read and write for the owner; everyone else is read
0666	Read and write for everybody
0740	Read, write, and execute for owner; read for the file group

WRITE FILE

Use the WriteFile function to write bytes to a file.

The syntax for the function is:

```
func WriteFile(filename string, data
        []byte, perm os.FileMode) error
```

Here is sample code for writing text to a file. Notice that the text is converted into bytes using []byte.

```
package main

import (
    "io/ioutil"
    "log"
)

func main() {
    s := "Hello, File!"
    err := ioutil.WriteFile("hello.txt",
        []byte(s), 0644)
    if err != nil {
        log.Fatal(err)
    }
}
```

WRITE FILE- EXPLAINED

Import io/ioutil package (1)

initialize string (2)

write to hello.txt file (3)

convert string to bytes (3b)

read / write permissions for the owner; everybody else is read (3c)

if write operation fails, log err to stderr and exit the program (4)

```
package main

import (
    "io/ioutil"   1
    "log"
)

func main() {
    s := "Hello, File!"   2
    err := ioutil.WriteFile("hello.txt",   3
        []byte(s), 0644)
    if err != nil {   4
        log.Fatal(err)
    }
}
```

READ FILE

Use the ReadFile function to read bytes from a file.

The syntax for the function is:

```
func ReadFile(filename string) (
        []byte, error)
```

Here is sample code for reading text from a file. Notice that bytes are converted to a string using the string function.

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
)

func main() {

    fileBytes, err := ioutil.ReadFile("hello.txt")
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(fileBytes)
    fileString := string(fileBytes)
    fmt.Println(fileString)
}
```

READ FILE- EXPLAINED

Import io/ioutil package (1)

read bytes from hello.txt file (2)

if write operation fails, log err to stderr (3)

display file bytes (4)

convert bytes to a string (5)

display string (6)

```
package main

import (
    "fmt"
    "io/ioutil"   1
    "log"
)

func main() {

    fileBytes, err := ioutil.ReadFile("hello.txt")  2
    if err != nil {
        log.Fatal(err)   3
    }

    fmt.Println(fileBytes)   4
    fileString := string(fileBytes)   5
    fmt.Println(fileString)   6
}
```

READING AND WRITING STRUCTURES

Structures must be converted to and from bytes to read and write to a file using the ReadFile and WriteFile functions respectively.

In the Json package, the Marshal and Unmarshal methods convert a structure to and from bytes.

The syntax is:

```
func Marshal(v interface{}) ([]byte, error)

func Unmarshal(data []byte, v interface{}) error
```

These methods will use Json (JavaScript Object Notation) encoding.

WRITE STRUCTURE - EXAMPLE

```
package main
import (
	"encoding/json"
	"fmt"
	"io/ioutil"
	"log"
)
type Person struct {
	Id   int
	Name string
	Age  int
}
```

```
func main() {
	me := Person{
		Id:   1,
		Name: "Me",
		Age:  64}

	b, err := json.Marshal(me)

	fmt.Println(b, err)

	err2 := ioutil.WriteFile("hello3.bin",
		b, 0644)
	if err2 != nil {
		log.Fatal(err)
	}
}
```

READ STRUCTURE - EXAMPLE

```
package main

import (
	"encoding/json"
	"fmt"
	"io/ioutil"
	"log"
)

type Person struct {
	Id   int
	Name string
	Age  int
}
```

```
func main() {
    me := Person{}

    fileBytes, err4 := ioutil.ReadFile("hello3.bin")
    if err4 != nil {

    }

    json.Unmarshal(fileBytes, &me)
    fmt.Println(me)
}
```

READING AND WRITING STRUCTURES (GOB)

You can also read and write structures using GOB, which is the Go language serialization format. GOB data is self-describing. Pointers are not serialized. However the data at the pointer is saved. The effect is that the data is flatten as part of the serialization.

WRITE STRUCTURE (GOB)

The advantage to GOB is simplicity. To serialize to a file:

- os.Create to get file handle
- gob.NewEncoder to obtain a GOB encoder
- gob.Encode to write stream to target resource
- Don't forget to close the file

Here is the syntax:

```
func Create(name string) (*File, error)
func NewEncoder(w io.Writer) *Encoder
func (enc *Encoder) Encode(e interface{}) error
```

```
package main

import (
    "encoding/gob"
    "os"
)

type Person struct {
    Name string
    Age int32
}

func main() {
    filename := "buffer.gob"
    bob := Person{"Bob Johnson", 35}

    file, err := os.Create(filename)
    if err == nil {
        encoder := gob.NewEncoder(file)
        encoder.Encode(bob)
    }
    file.Close()
}
```

READ STRUCTURE (GOB)

To deserialize from a file to an object:

- os.Create to get file handle
- gob.NewDecoder to obtain a GOB encoder
- gob.Decode to read stream into target object
- Don't forget to close the file

Here is the syntax:

```
func Create(name string) (*File, error)  ***
func NewEncoder(w io.Writer) *Encoder
func (dec *Decoder) Decode(e interface{}) error
```

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
)

type Person struct {
    Name string
    Age int32
}

func main() {
    filename := "buffer.gob"
    var bob = new(Person)
    file, err := os.Open(filename)
    if err == nil {
        decoder := gob.NewDecoder(file)
        err = decoder.Decode(bob)
    }
    file.Close()

    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(bob.Name, "\t", bob.Age)
    }

}
```

APPENDING TO A FILE

You cannot use io/ioutil package to append to a file. The ioutil functions were created as a convenience and kept simple.

```
package main

import (
    "os"
)

func main() {

    filename := "hello.txt"
    f, err := os.OpenFile(filename,
        os.O_APPEND|os.O_WRONLY, 0600)
    if err != nil {
        panic(err)
    }

    defer f.Close()

    text := "42"
    if _, err = f.WriteString(text); err != nil {
        panic(err)
    }
}
```

Packages

custom packages

PACKAGES

The Go Language provides a lot of extra functionality through third party packages available at:

https://goperfd.appspot.com/

You can create your own packages, executables or libraries, that are shared with others.

WHAT IS A PACKAGE

In this course, package have been single file applications. However, that is not required. You can take a more modular approach.

Packages can spread across multiple files that are in the same directory. Just make the name of each package the same. For an executable, the package files should start with “package main”.

Shared packages should be placed in a GOPATH source directory (i.e, %gopath%\src. To avoid name conflicts, it is recommend to create subdirectory for each unique package within the src directory. Remember to indicate the hierarchal structure when importing the package.

MULTI-FILE PACKAGE

Packages can be, and are often, multifile. Compared to creating a monolithic code file. The best practice is one structure / type per file.

Place the code files of the same package in the same directory. The package statement and name should be at the top of each related file.

```
package main

func main() {
    FuncA()
}
```

```
package main

import "fmt"

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

THE IMPORT STATEMENT EXTRAS

- "_" ignore import
- Prefix – create an alias
- "." implicit namespace

```
package main

import . "./Lib"
import _ "fmt"

func main() {
    FuncC()
}
```

EXPORTED FUNCTIONS

Exported function names should start with an uppercase character. Conversely, the initial character of private methods are lowercase. In addition, public entities should be commented.

Global cross package variables are similarly defined.

```
package mypackage

import "fmt"

// FuncA comment
func FuncA() {
    fmt.Println("FuncA")
    funcb()
}

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

LOCAL PACKAGE

You can have a local or private library that is consumed only by your applications.

Here are the steps:

1. In the application directory, create a subdirectory using the name of the library package. If the library name is stuff.go, create a subdirectory called stuff.
2. Copy the library into that directory.
3. The package command of the library should be:

    package libraryname

4. In the client application, import the package as:

    import "./libraryname"