Introduction to Programming in Go

1. Install Go
2. Verify that you can compile and run (Hello-World.go) (5pts)
3. Echo command line arguments (echo1.go) (5pts)
4. Parse command line arguments (echo2.go) (5pts)
5. Read and write JSON file (read-json1.go) (10pts)
6. Test a simple library package (15pts)
7. Printing (print1.go) (10pts)
8. Copy a File (copy1.go, copy2.go) (20pts)
9. Hash and a simple test in Go (test1.go) (30pts)
10. Summary

100 Points total.

Item #8 and #9 will be the first auto-graded items. I will use automated tests to check that the code works.

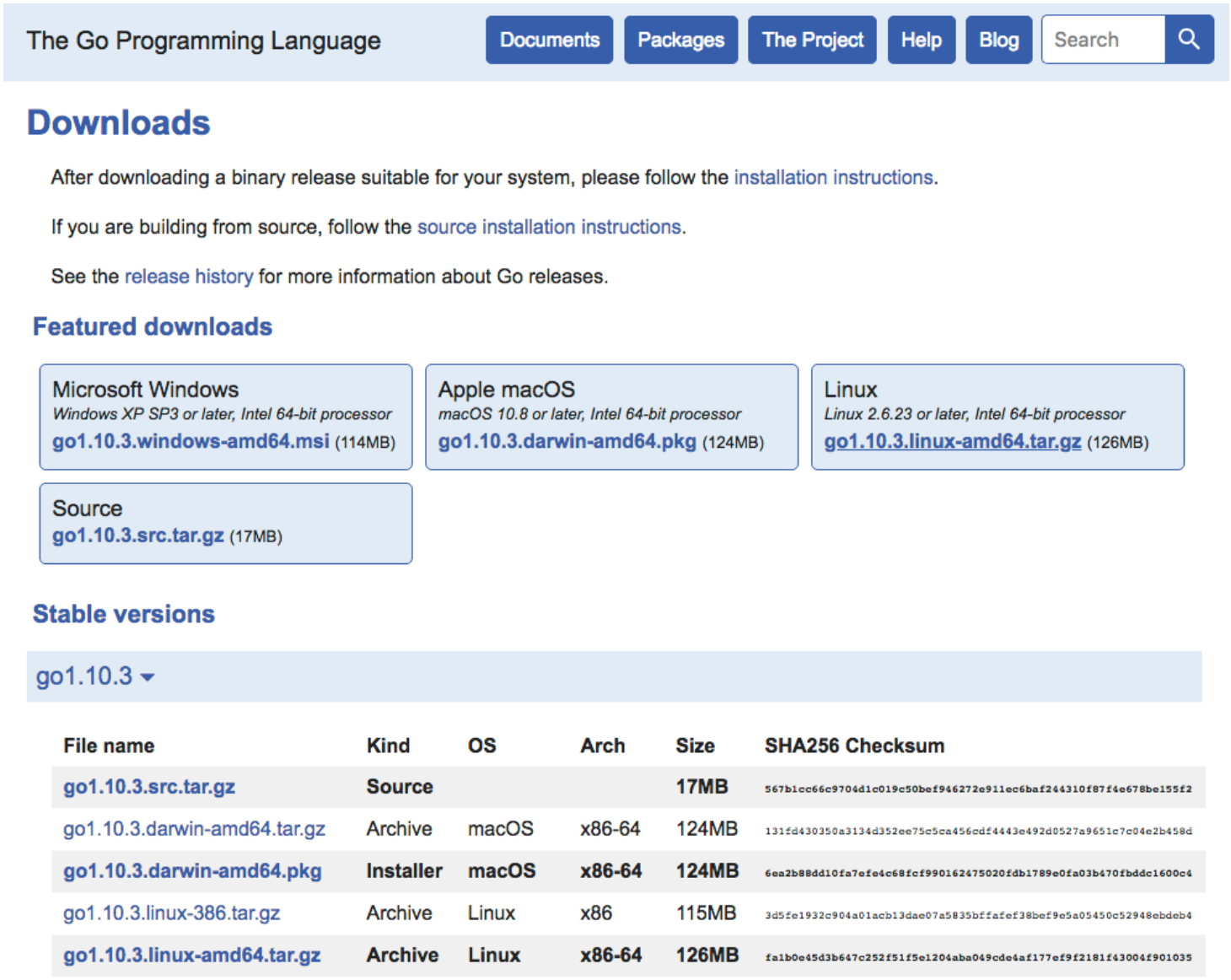
Note: Not all the assignments will be this long. Don't Panic!

Install Go

Go runs on Windows, Mac and Linux (and a bunch of other systems too). To install you will need to download the Go compiler for your system. If you already have Go installed you will may have to delete the old version, then install the new version. Go is super-stable - there is no reason to keep an old version of the compiler around.

Search google for "download golang". Go is usually referred to as "golang" when you search for it.

You should find: https://golang.org/dl/. The page should look like



For Windows there is a `.msi` installer that is very good. It will automatically install go into `C:\go\` and setup your permissions. You can choose a different path for the install. Follow the instructions on the download site for Windows.

Install at least version 1.10.3 of the language. On mac you download .pkg and click on it. On windows the .msi and run it (double click). On Linux... Follow the instructions on the download page for your flavor of Linux.

You will need to use an editor. I don't usually use an IDE. I do use vi (vim). Pick a text editor that you like.

If you need to create a portable USB drive with Go and VIM on it see me after class.

You will need a github.com account. Go and create one if you do not already have one. A free account will work.

If you already had Go installed.

You only have to do this if you have an old version of Go installed.

On a Mac go is by default installed in /usr/local/go, on windows in C:\go. You will need to delete the old version of go.

Mac and Linux

```
$ cd /usr/local
$ ls
$ sudo rm -rf go
```

Windows

The latest version of the Go installer for windows will offer to automatically delete an old version from the system. Choose this in the installer.

Submit:

1. run `go version` at the command line and capture the output. Submit a 1 line file with the output in it. You can cut paste or pipe the output to a file. You should see a version around `go1.10` or `go1.10.3`.

(1.2) Verify that you can compile and run (Hello-World.go) (5pts)

For this class you will need to have `git` installed. On Mac you should be able to use `brew` to install git.

```
brew install git
```

On Windows download from https://git-scm.com/download/win and install git. This will put a funny 4 square icon on your desktop that says 'bash' in it. Click on that to bring up a `bash` shell.

Let's put our code in the right place from the very start. On a mac that is `~/go/src/github.com/[Your Github Account]/`. In this case go to that directory - or build it. Then create a directory called `hello-world`. I used my username in this. You need to use your github.com username.

```
cd
mkdir -p ./go/src/github.com/pschlump
```

```
cd ./go/src/github.com/pschlump
mkdir hello-world
cd hello-world
```

In Windows you should be in your home directory xyzzy or a similar directory. In the bash shell from git the same commands as on Mac should work (the did for me a few minutes ago).

Cut and paste - or type in (probably better for you) the following program. These instructions are also on the Go download page. Put it into a file called `main.go` .

```
package main

// Your Name - it is important if you want to get credit for your assignment.
// Assignment 1.2 hello world - This is important too.  If you want credit.

import "fmt"

func main() {
        fmt.Printf("Hello Wonderful World\n")
}
```

To run it:

```
go build
./hello-world
```

or on Windows

```
go build
./hello-world.exe
```

If you get an error about access an un-exported function, then you failed to capitalize the first letter in a method call. Capital letters tell Go that the function is exported. `fmt.printf` will cause this error. `fmt.Printf` will work.

Submit:

1. A copy of your code.
2. Change the `fmt.Pr...` to `fmt.Println("Hello, 世界")` - re run the code and save the output. What is 世界 ?

References

1. A Tour of Go
2. Hello World Explained

You may want to work through the A Tour of Go. That will help with this class.

Mac and Linux

This is the set of commands for setting up your Go environment.

```
$ export GOPATH=/Users/<Your Github Username>/go
$ mkdir -p ~/go/src/github.com/<Your Github Username>
$ cd ~/go/src/github.com/<Your Github Username>
$ mkdir hello-world
$ cd hello-world
$ vi hw.go
$ go build
$ ./hello-world
Hello Wonderful World
```

PC / Windows

You are probably in a directory something like `C:\home\your-name` . I also have a copy of Vi for windows installed. You can use your favorite text editor instead. I installed vim using the gvim81.exe from https://www.vim.org/download.php#pc.

```
$ mkdir hello-world
$ cd hello-world
$ vi hw.go
$ go build
$ hello-world.exe
Hello Wonderful World
```

You will want to create a directory for each main program and under that directories for each package that you create in Go.

If you have a different text editor that you like better than vim - then install it - use it (Microsoft Word is not an editor! It will not create files that you can compile with Go.)

I worked through the rest of the homework on a Windows 10 system with very little Windows related difficulty. Using the git-bash your `C:\` drive is `/c/` , your `E:\` drive is `/e/` . You use `/` for all the directory separators.

(1.3) Echo command line arguments (echo1.go) (5pts)

Echo should take the command line arguments (after the 0th one, the name of the command) and print them out. Cut and paste the following code. Get it to run. Compile with `go build`. Run it with `go run <your file>.go Arguments`.

```
package main

// Your Name - it is important if you want to get credit for your assignment.
// Assignment 1.3 echo command line arguments.

import (
        "fmt"
        "os"
)

func main() {
        ags := os.Args[1:]
        for ii, ag := range ags {
                if ii < len(ags) {
                        fmt.Printf("%s ", ag)
                } else {
                        fmt.Printf("%s", ag)
                }
        }
        fmt.Printf("\n")
}
```

1. `import` is a list
2. `os.Args[1:]` is a slice of an array
3. `range`
4. no semicolons
5. `:=` declares variables.

Submit:

1. A copy of your code with your name in it.
2. An example (cut and paste) of running this with the arguments `aa BB cc`.

References

1. os Package

(1.4) Parse command line arguments (echo2.go) (5pts)

Cut and paste the following code. Get it to work. Run it. Save the output - edit the code and change `Your Name` . Add your email address.

Follow THIS link, JSON in Go. There is a section with an "Example (CustomMarshalJSON)" - Click on the example to open the example. In the example an custom Go type called `Animal` is created (Around line 10). Custom code allows saving of the `Animal` type in a JSON file and re-loading the `Animal` type from the file. In the JSON file it will be "zebra" "gopher" - in the Go code it will be an integer constant (like a C++ `enum`). The Constant named values are created around lines 12 to 15. This process is called marshal/unmarshal. The functions are called UnmarshalIJSON and MarshalIJSON. You should cut-paste the code, put it into a directory and run it.

Create the echo2 directory and `cd` into it. You should be in a directory `~/go/src/github.com/<YourGithubUsername>/echo2` .

Edit main.go and put this code in it:

```
package main

// Your Name - it is important if you want to get credit for your assignment.
// Your Email Address
// Assignment 1.4 echo command line arguments and parse arguments.

import (
        "encoding/json"
        "flag"
        "fmt"
        "io/ioutil"
        "os"
)

type ConfigData struct {
        Name  string
        Value string `json:"Year"`
}

func main() {
        var Cfg = flag.String("cfg", "cfg.json", "config file for this call")

        flag.Parse() // Parse CLI arguments to this program, --cfg <name>.json

        fns := flag.Args()
        if len(fns) == 0 {
                fmt.Fprintf(os.Stderr, "Usage: ./echo2 [-cfg cfg.json] arg1 ...
                os.Exit(1)
        }

        if Cfg == nil {
```

```
                        fmt.Printf("--cfg is a required parameter\n")
                        os.Exit(1)
                }

                gCfg, err := ReadConfig(*Cfg)
                if err != nil {
                        fmt.Fprintf(os.Stderr, "Unable to read confguration: %s error %
                        os.Exit(1)
                }

                fmt.Printf("Congiguration: %+v\n", gCfg)
                fmt.Printf("JSON: %+v\n", IndentJSON(gCfg))

                for ii, ag := range fns {
                        if ii < len(fns) {
                                fmt.Printf("%s ", ag)
                        } else {
                                fmt.Printf("%s", ag)
                        }
                }
                fmt.Printf("\n")
        }

        func ReadConfig(filename string) (rv ConfigData, err error) {
                var buf []byte
                buf, err = ioutil.ReadFile(filename)
                if err != nil {
                        return ConfigData{}, err
                }
                err = json.Unmarshal(buf, &rv)
                if err != nil {
                        return ConfigData{}, err
                }
                return
        }

        func IndentJSON(v interface{}) string {
                s, err := json.MarshalIndent(v, "", "\t")
                if err != nil {
                        return fmt.Sprintf("Error:%s", err)
                } else {
                        return string(s)
                }
        }
```

Create a 2nd file - call it cfg.json with:

```
  {
        "Name": "Corwin",
```

```
        "Year": "100",
        "Value": "200"
}
```

Read Code

Go through this program and read and analyze the following things:

1. `type ConfigData struct` read and understand that this will declare a struct.
2. `var Cfg =...` Declare a variable. It will be a pointer to a string because `flag.Stgring` returns a pointer to a string.
3. `flag.Parse()` this uses the `flag` package to parse the command line arguments.
4. `fns := flag.Args()` this will pick off the remaining arguments from the command line. If you have `echo2 --cfg file.json A B C` this will be a slice, 3 long, with the strings `A`, `B`, `C` in it.
5. `ReadConfig` call the `ReadConfig` function. The function is declared later in the file.
6. `fmt.Print.... %+v` print out a structure with the field names. Very useful for debugging your program.
7. `func ReadConfig` declares a function. Note functions can return a set of values, not just one.
8. `return ConfigData{}, err` declare an empty `ConfigData{}` and return an error.
9. `func IndentJSON.... v interface{}` create a data type that can receive any type that is passed.
10. `json.MarshalIndent` marshal data into a string.

Why did `100` print out. What happened to `200`. Create a new JSON file under a new name and set the year field to 2018.

Submit:

1. Your code. Modify the comment in the code to have your name and assignment number.
2. Your JSON file with the year modified to 2018.

References

1. JSON in Go

(1.5) Read and write JSON file (read-json1.go) (10pts)

Use the following structure to read a JSON file. Create a program that will read JSON in, print it, write it out in a new JSON file. Get the file name for the input and output from the command line.

Add a map/dictionary field in the JSON input that is

```
{
        "TxHash": "Your actual Name",
        "TxIn": 22,
        "TxOut": 44
}
```

Example Run:

```
./read-json1 --input in.json --output out.json
```

The struct to include in your code.

```
type TransactionType struct {
        TxHash  string
        TxIn    int
        TxOut   int
}
```

You will need to use:

1. `ioutil.WriteFile` ioutil package
2. `IndentJSON` from above.

Submit:

1. Your program.
2. Your JSON input file.
3. Your JSON output file.

References

1. ioutil package

(1.6) Test a simple library package (15pts)

Goal: create a go package and a test for that package.

Create a new directory inside your github.com/username. This example will use `pschlump` as the username. You need to substitute your github.com user name.

```
cd
mkdir -p ~/go/src/github.com/pschlump/mkPkg/test1
cd ~/go/src/github.com/pschlump/mkPkg/test1
vi test1.go
```

In this case I created 2 directories, `mkPkg` and inside it `test1` .

A simple example package in the `mkPkg/test1` directory edit a file, lets call it `test1.go` .

```
package test1

// Your Name

// DoubleValue returns twice the value passed.
func DoubleValue ( n int ) int {                                    // note the cap
        return n * 2
}

// add function TrippleValue
```

It will make life simpler if you make the package name the same as the directory name. Don't put blanks in your directory names.

The capital letter at the beginning of `DoubleValue` tells Go that you are exporting the name.

The test code is placed into `mkPkg/test1` in the file `test1_test.go` .

```
package test1

import "testing"

// Your Name

func TestDouble(t *testing.T) {

        tests := []struct {
                in       int
                expected int
        }{
                {
                        in:       23,
                        expected: 46,
```

```
                        },
                        {
                                in:       1,
                                expected: 2,
                        },
                }

                for ii, test := range tests {
                        rr := DoubleValue(test.in)
                        if rr != test.expected {
                                t.Errorf("Test %d, expected %d got %d\n", ii, test.expe
                        }
                }

        }

        // add test for TripleValue at this point
```

to run the test in the directory `mkPkg/test1` :

```
        go test
```

It should print out 'PASS' when the test work. If the test fails you should get errors.

It is really important that you understand how Go testing works. This will be 1 of 2 ways in which your code will be graded. You will be expected to develop your own unit tests. I will supply my grader with additional tests and those will be run with your code.

We will create a simple main program that will use the package that `test1` . Go up 1 level to `~/go/src/github.com/pschlump/mkPkg` .

Edit a main program:

```
        package main

        // Your Name

        import (
                "fmt"

                "github.com/pschlump/myPkg/test1" // import package you created
        )

        func main() {
                out := test1.DoubleValue(8)     // Call function in your package
                fmt.Printf("out = %d\n", out)   // should print "out = 16"
                // add call to TripleValue at this point
```

```
        // add call to TripleValue at this point
}
```

Submit:

1. Your package.
2. Your test code.
3. Your main program.

References

1. Intro to packages
2. Intro to testing is really good and has way more on testing.

(1.7) Printing (print1.go) (10pts)

The `fmt` package provides lots of useful output options. Go and read about fmt.

In the following program

```
package main

import (
        "encoding/json"
        "fmt"
)

var IVar int
var SVar string
var I64Var int64
var UIVar uint64

type Example17 struct {
        A int
        B string
}

var E17 Example17

// Add int64 and uint64 types
var SliceOfString []string
var MapOfString map[string]string
var MapOfBool map[string]bool

// init will initilize data before main() runs.  You can have more than one ini
func init() {
```

```
        SliceOfString = make([]string, 0, 10)
        MapOfString = make(map[string]string)
        MapOfBool = make(map[string]bool)
}

func main() {
        SliceOfString = append(SliceOfString, "AAA", "BBB")
        MapOfString["mark"] = "first"
        MapOfString["twain"] = "last"
        MapOfBool["mark"] = true
        MapOfBool["twain"] = false

        fmt.Printf("IVar = %d, type of IVar %T\n", IVar, IVar)
        fmt.Printf("IVar = %v, type of IVar %T\n", IVar, IVar)

        // TODO: add prints for your int64 and uint64 types

        fmt.Printf("SVar = %s, type of SVar %T\n", SVar, SVar)
        fmt.Printf("Address of SVar = %s, type of SVar %T\n", &SVar, &SVar)
        fmt.Printf("E17 = %s, type of E17 %T\n", &E17, &E17)
        fmt.Printf("    E17 = %+v, E17 as JSON: %s\n", &E17, IndentJSON(E17))

        // TODO: add prints for the other types above - so you can see them pri
        // TODO: use a %s and a %T for SliceOfString
        // TODO: use a %s and a %T for MapOfString
        // TODO: use a %#v and a %T for MapOfBool
        // TODO: Print out each of them with the IndentJSON function.
}

func IndentJSON(v interface{}) string {
        s, err := json.MarshalIndent(v, "", "\t")
        if err != nil {
                return fmt.Sprintf("Error:%s", err)
        } else {
                return string(s)
        }
}
```

Add a print statement to show the type of the variables. See the comments with TODO in the code.

Submit:

1. Your program with the extra print statements.
2. The output from running the program with the extra print statements.

(1.8) Copy a File (copy1.go, copy2.go) (20pts)

Implement a simple program to copy a file. See copy a file in go and modify the code to take the input and output file names from the command line. Go and read about `defer` and see how it is used.

```
copy-file input-file-name output-file-name
```

There is a line in the example code, `_, err = io.Copy...` . What is the `_` for? Replace it with a statement like this: (note the := that will declare test1)

```
test1, err := io.Copy(to, from)
```

Recompile the code - what happens?

Implement a copy file program that uses `ioutil.ReadFile` and `ioutil.WriteFile` . What are the disadvantages of doing the copy in this way? Write a paragraph explaining the disadvantages of the `ioutil.ReadFile` , `ioutil.WriteFile` disadvantages.

Submit:

1. Both of your programs. The original with the `io.Copy(...)` in it and the one that uses `ioutil.ReadFile` .
   Add a note that this is a copy from the website, copy a file in go and a note with your name and email address.
2. Your written paragraph.

References

1. defer
2. log

(1.9) Hash and a simple test in Go (ksum.go) (30pts)

This is the first set of code that we will directly use in building our blockchain.

A hash is a number, usually large, that maps a set of data into a unique number. A different set of data will result in a different hash. The file `file1` will hash to `ecd67ca5a72802084fcea4883b6877ecfba7f95c0aece07ea504359d54eb4610` . That's a big number. Note that the number is in base 16 when it was printed out (so it has `0..9` and `a..f` for digits). It is possible that two different sets of data will produce the same value. This is called a hash collision. A good hash rarely has collisions.

We will use a bunch of different hash functions. All of the has functions have a similar interface in Go. Today's function is Keccak256. This is the hash that is used in Ethereum. It is a `sha3` derivative.

The Go documentation for sha3 includes keccak256 but we will be using the one in the Go-Ethereum package.

An example of using it is: keccak256 Note that the example is wrong, the output will be in lower case. The example shows it in upper case. Go and read the example.

This Ethereum code includes a function that you will want to copy. Lines 44 to 51. Give credit where credit is due. (See Below - I copied it)

Note that the Keccak256 function takes a slice of byte, `[]byte` , and returns a slice of byte. We will need to type-cast strings into this type to get it to work in the demo.

Copy 1.8's ioutil.ReadFile version of the copy into a new directory called `ksum` . We are goging to modify it to print out the keccak256 sum of a file. This is like md5sum or sha1sum. Go and read the documentation on these 2 command line utilities. We will build the keccak256 sum program. You don't need to implement any of the command line options like `-b/--binary` . Just read more than one file and print out the results.

Sample Output:

```
$ ksum file1 file2.txt file3
file1 ecd67ca5a72802084fcea4883b6877ecfba7f95c0aece07ea504359d54eb4610
file2.txt 0695253b82a83d557392ab196ff309a1fedc6cbab0d7d4186d2664dcec92b5ff
file3 fb15d651aaf994584aa6da109b5dba096de83bf2f44da6a224cf41d8d5e92f14
```

I have supplied the fiels `file1` , `file2.txt` , and `file3` .

Process as many files as are on the command line.

So start out with an exampel - that just calculates the hash of the string `"bob"` :

```
package main

import (
        "fmt"

        "github.com/ethereum/go-ethereum/crypto/sha3" // you can't click on thi
)

func main() {
        fmt.Printf("%x\n", Keccak256([]byte("bob"))) // type cast from string,
```

```
}

// Keccak256 calculates and returns the Keccak256 hash of the input data.
// From: https://github.com/ethereum/go-ethereum/blob/master/crypto/crypto.go l
func Keccak256(data ...[]byte) []byte {
        d := sha3.NewKeccak256()
        for _, b := range data {
                d.Write(b)
        }
        return d.Sum(nil)
}
```

After you create/copy this code into a file, in a directory called ksum, you will need to

```
go get
```

to have Go pull in `github.com/ethereum/go-ethereum` package and all it's sub packages including: `github.com/ethereum/go-ethereum/crypto/sha3` . If you do not do the `go get` you will get an error `cannot find package...` . The `go get` will pull in the dependencies for this from github so it will take will take a little bit.

Add in the ability to process the command line. `ioutil.ReadFile` returns a byte slice and an error. Report the error if it occurs. You will not need a type cast to pass the byte slice to Keccak256.

`fmt.Printf` can print out in hex. Note the `%x` format.

Pseudo-Code for your program.

1. Process command line to get the file names.
2. For each argument (file name from the command line)
   - read in the data in the file.
   - calculate the hash for the data.
   - print out file name and the hex string for the hash.

Submit:

1. Test your code with file1, file2.txt, file3 . Save the output. Submit the output with the hashes. Test your code with some other files of your choice. Submit the output from that with the files that you used.
2. your code for doing this (remember your name in the program)

(1.10) Summary

Go is a simple language that Google developed. It is not an “academic” research language. There are no new cool features. The good things about go are:

1. The go compiler is really fast.
2. The code is statically linked and optimized.
3. The set of tools for the language extensive (go vet, golint, testing, flame graphs etc.)
4. The garbage collector is revolutionary.
5. It matches with modern hardware.
6. The language is industrial scale.
7. Lots of really good documentation.
8. Extensive library.
9. Easy to learn.
10. Fun to program in.
11. Reliable.

Go is missing generics and objects. When I first started to use Go I thought that it would be a big deal to not have objects. Turns out I really don’t need or want objects. Generics would be useful on rare occasions.

Save your code from this assignment. We will be using chunks of it in the next assignment.