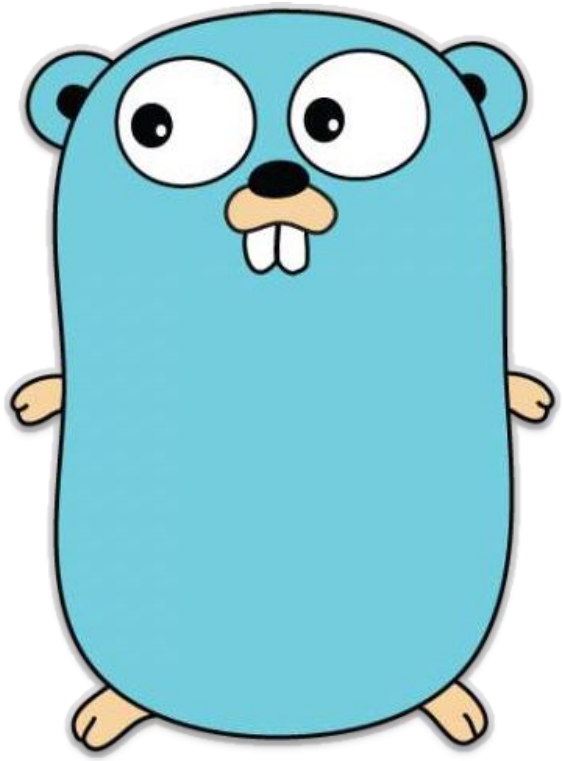

Overview of Go Language

Less is More

Agenda – Lab 1

- What is Go?
- Who uses Go?
- First Go program – imports and exports
- Concurrency in Go
- "Less is More" – Go design philosophy
- Flow control statements

Go Resources – Go learn Go!

- A Tour of Go: https://go.dev/tour/list
- Go by Example: https://gobyexample.com
- Effective Go: https://go.dev/doc/effective_go

What is Go (golang)

Go (also called golang) is an open source programming language that makes it easy to build simple, reliable, and efficient software.

- Go is **natively compiled** (Go does not use a VM, Go programs gets compiled directly to machine code like C, C++)
- Uses **static typing** (types can be inferred though)
- Scalable to large systems
- Though it's general purpose programming language, but it's targeted towards **System programming** and **server side programming** (similar to C, C++, Rust, D)
- Clean syntax
- Has excellent **support for concurrency and networking**.
- Go is **garbage collected**
- Comes with a **rich standard library**
- Go compiler is available on Linux, OS X, Windows, various BSD & Unix versions
- Go is open source

Who uses Go

- Many Google web applications and systems including YouTube, Kubernetes containers and download server dl.google.com
- Docker, a set of tools for deploying Linux containers
- Dropbox, migrated some of their critical components from Python to Go
- SoundCloud, for many of their systems
- Cloud Foundry, a platform as a service (PaaS)
- Couchbase, Query and Indexing services within the Couchbase Server
- MongoDB, tools for administering MongoDB instances
- ThoughtWorks, some tools and applications around continuous delivery and instant messages
- SendGrid, a transactional email delivery and management service
- The BBC, in some games and internal projects
- Novartis, for an internal inventory system

https://en.wikipedia.org/wiki/Go_(programming_language)#Notable_users

Domains where Go is being used today

- Large scale distributed systems
- Cloud – Many PaaS Cloud platforms supports hosting Go code including GCP, Cloud Foundry (with build pack) and Heroku. Many cloud platforms supports SDKs in Go including GCP, AWS and Azure
- Web development
- Scripting
- Systems programming

First Go program

- Every executable Go Program should contain a package called **main**. This tells the Go compiler to compile the package into an executable program rather than a shared library.
- Package statement should be the **first line** of any go source file.

- The entry point of a Go program should be the `main` function of main package. When the executable is run, `main()` is automatically called.

```
// name of the source code file is main.go (it could be whatever you like)
// this package is called main
package main

// Entry point of a Go program is main.main i.e. main function of main package
func main() {
    // println built-in function is called
    println("Hello from Go")     // prints Hello from Go in console
}
```

First Go program – imports

- Packages can be imported via the `import` statement.

```
// this package is called main
package main

// fmt package contains methods to interact with console like Print and Scan
import "fmt"

func main() {
    fmt.Println("Hello from Go")
}
```

- The start curly has to be in the same line of method name and parentheses.

```
func main()
{
   fmt.Println("Hello from Go")
}
// this code would not compile
```

First Go program – running

- To compile and execute the go program use the `go run` command.

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

- Try running a simple hello program from a terminal and from VS Code.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("Welcome to Go!")
    fmt.Println("The time is", time.Now())
}
```

See Go_setup slides for detailed instructions on how to install Go on your computer.

Exported Names

- In Go, a name is exported if it begins with a capital letter. Lowercase names are unexported or private, and can only be accessed within its own package.

```
package main

// preferred shorthand – factored import statements
import (
    "fmt"
    "math"
)

func main() {
    // Pi is an exported name
    fmt.Println(math.Pi)
}
```

- This applies to fields of structs when using RPC, JSON encoding, and other serialization methods – Go RPC will only send struct fields whose names start with capital letters.

Concurrency in Go

- Why do we need concurrency?
  - **Data parallelism** – the same task is performed concurrently on subsets of some data (speed, working on massive datasets)
  - **Task parallelism** – different tasks are performed concurrently
- Concurrency vs. parallelism
  - Concurrency is the illusion of multiple tasks running in parallel via interleaving of instructions (i.e. the CPU rapidly switching between different tasks – **only one** task is being executed at any given moment)
  - Concurrency is **single-core**, parallelism is **multi-core** (or multi-machine)
- Goroutines are **lightweight threads** – having tens or even hundreds of thousands is the norm.

```
go func() {
    fmt.Println("I am a Goroutine")
}()
```

Less is More – Go design philosophy

- Go deliberately omits many features common in other languages, such as inheritance, pointer arithmetic, assertions and exceptions.
    - For example, Go's design encourages you to explicitly check for errors when they occur as opposed to the convention in other languages of throwing exceptions and sometimes catching them.
- Go enforces rules that are recommendations in other languages, such as banning cyclic dependencies, unused variables or imports, and implicit type conversions.
- The `gofmt` tool automatically standardizes indentation, spacing, etc. Other tools like `godoc` and `go test` suggest standard approaches to API documentation and testing.

Less is exponentially More: https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html

Simplifications in Go over C/C++

- regular syntax (don't need a symbol table to parse)
- garbage collection (only)
- no header files
- explicit dependencies
- no circular dependencies
- constants are just numbers
- int and int32 are distinct types
- letter case sets visibility
- methods for any type (no classes)
- no subtype inheritance (no subclasses)
- package-level initialization and well-defined order of initialization
- files compiled together in a package
- package-level globals presented in any order
- no arithmetic conversions (constants help)
- interfaces are implicit (no "implements" declaration)
- embedding (no promotion to superclass)
- methods are declared as functions (no special location)
- methods are just functions
- interfaces are just methods (no data)
- methods match by name only (not by type)
- no constructors or destructors
- post increment and post decrement are statements, not expressions
- no pre increment or pre decrement
- assignment is not an expression
- evaluation order defined in assignment, function call (no "sequence point")
- no pointer arithmetic
- memory is always zeroed
- legal to take address of local variable
- no "this" in methods
- segmented stacks
- no const or other type annotations
- no templates
- no exceptions
- built-in string, slice, map
- array bounds checking

Simplifications in Go over C/C++

- regular syntax (don't need a symbol table to parse)
- garbage collection (only)
- no header files
- explicit dependencies
- no circular dependencies
- constants are just numbers
- int and int32 are distinct types
- letter case sets visibility
- methods for any type (no classes)
- no subtype inheritance (no subclasses)
- package-level initialization and well-defined order of initialization
- files compiled together in a package
- package-level globals presented in any order
- no arithmetic conversions (constants help)
- interfaces are implicit (no "implements" declaration)
- embedding (no promotion to superclass)
- methods are declared as functions (no special location)
- methods are just functions
- interfaces are just methods (no data)
- methods match by name only (not by type)
- no constructors or destructors
- post increment and post decrement are statements, not expressions
- no pre increment or pre decrement
- assignment is not an expression
- evaluation order defined in assignment, function call (no "sequence point")
- no pointer arithmetic
- memory is always zeroed
- legal to take address of local variable
- no "this" in methods
- segmented stacks
- no const or other type annotations
- no templates
- no exceptions
- built-in string, slice, map
- array bounds checking

It's just a lot friendlier Go really doesn't want you to shoot yourself in the foot

For loops – Go's while and for each

- Another simplification: Go has only one looping construct, the `for` loop!
  - Go does not have `while`, `do while`, `for each`, or `for in` keywords.

```
var ctr int = 0

// same as while
for ctr < 5 {
    fmt.Println(ctr)
    ctr++
}
```

- The basic for loop looks as it does in C or Java, except that the `( )` are gone and the `{ }` are required.

```
for ctr := 0; ctr < 10; ctr++ {
    fmt.Println(ctr)
}
```

For (cont'd)

- As in C or Java, you can leave the pre and post statements empty.

```
ctr := 0
for ; ctr < 10; {
    ctr += 1
    fmt.Println(ctr)
}
```

- Semicolons can also be dropped: C's while is spelled "for" in Go.

```
ctr := 0
for ctr < 10 {          // behaves in the same way as while ctr < 100 in C or Java
    ctr += 1
    fmt.Println(ctr)
}
```

For (cont'd)

- Endless or infinite loop

```
for {
    // do something – this loop would never end
}
```

- Range – iterating through a slice, array, map, channel, or string (for-each)

```
cities := []string {"Kolkata", "Bangalore", "Mumbai"}

// you can also skip the index or value by assigning to '_'
for index, value := range cities {
    fmt.Println(index, value)
}

for index := range cities {
    fmt.Println(index)
}
```

If, else if, else

- The `if` statement looks as it does in C or Java, except that the `( )` are gone and the `{ }` are required.

```
var salary int = 100

if salary < 50 {
    fmt.Println("you are underpaid")
} else {
    fmt.Println("you are sufficiently paid")
}
```

- `if` and `switch` accept an initialization statement (a variable declared in the if short statement is only in scope until the end of the if else).

```
if err := file.Chmod(0664); err != nil {
    log.Fatal(err)
    return err
}
```

Switch

- Go’s switch statement is more general than C's – expressions need not be constants or even integers.
- Unlike C or Java, Go's switch statement only runs the selected case. In effect, the break statement needed by C is provided automatically in Go.
- (Go actually has an explicit fallthrough statement.)

```
rating := 2
switch rating {
case 2:
    fmt.Println("You are rated Consistent")
    fallthrough
case 1:
    fmt.Println("You need to improve a bit")
}
```

Output:

```
You are rated Consistent
You need to improve a bit
```

```
rating := 2
switch rating {
case 4:
    fmt.Println("You are rated Excellent")
case 3:
    fmt.Println("You are rated Good")
case 2:
    fmt.Println("You are rated Consistent")
case 1:
    fmt.Println("You need to improve a bit")
default:
    fmt.Println("You have no rating")
}
```

Switch (cont'd)

- A switch with no condition is the same as `switch true`, which the Go tutorial points out as a clean way to write long if-then-else chains.

```
switch {
    t := time.Now()
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    case t.Hour() < 22:
        fmt.Println("Good evening.")
    default:
        fmt.Println("Good night.")
}
```

- Switch cases evaluate cases from top to bottom.

Defer

- A defer statement defers the execution of a function until the surrounding function returns.
- The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.
- Defer is commonly used to simplify functions that perform various clean-up actions.

```
package main
func main() {
    defer println("World")
    println("Hello")
}

// prints Hello World
```

```
package main
func main() {
    // you can also defer anonymous functions
    defer func() {
        println("World")
    }()
    println("Hello")
}
```

- A very common and convenient use for defers, as we'll see later, is in deferring unlocks of mutexes.

Defer - rules

1. *A deferred function's arguments are evaluated when the defer statement is evaluated.*
2. *Deferred function calls are executed in Last In First Out order after the surrounding function returns*
3. *Deferred functions may read and assign to the returning function's named return values.*

Slides will be posted some time on Weds– for now, have this

Project Euler Problem 11:
https://projecteuler.net/problem=11

Implement in Go

https://pastebin.com/ndQDw36i