Getting Started

Overview of Go Programming Language

Go is a statically typed, systems programming language.

```
// Dynamic Typing
var x = 10; // integer
console.log(10);

x = "messing up data types"; // string
console.log(x);
```

```
// Static Typing
var x = 10; // integer
fmt.Println(x);

x = "messing up data types"; // string, incompatible assignment
```

Setting up Go Development Environment

- Installing Go
- Do `go version` to verify go installation.
- Go environment variables:
    - `GOROOT`: Path to go binary and standard library, usually something like `/usr/local/go` on Linux
    - `GOPATH`: Developer workspace that contains go application source code, dependencies & go application binaries
    - `GOBIN`: Default path to Go application binaries, usually `$GOPATH/bin`
- Modules: Top level parent collection of packages constituting application code
- Packages: Collection of functions providing one functionality, e.g. strings package allows working with strings
    - Package name required as the first line of a go file
    - `package main` and `func main` are required for any go program to run
    - Package name should be unique
    - There cannot be more than one package in a single directory, except package names suffixed with `_test`

Working with Packages and Modules

Basics of Go Syntax and Conventions

- Package: Done
- Functions and the main() function: Done
- Importing and using packages: Done
- Variables: Done
- Struct: Done
- Members: Done
- Anonymous members: Done
- Interfaces

```
type Animal interface {
Get(property Property)
Speak()
}

// Concrete Class
type Dog struct {
Name, Breed string
Age, Weight int8
}

func (d Dog) Get(property Property) {
switch property {
case Name:
fmt.Println(d.Name)
case Breed:
fmt.Println(d.Breed)
}
}
func (d Dog) Speak() {
fmt.Println(Bark)
}

type Cat struct {
Name, Breed string
Age, Weight int8
}

func (c Cat) Get(property Property) {
switch property {
case Name:
fmt.Println(c.Name)
case Breed:
fmt.Println(c.Breed)
}
}
func (c Cat) Speak() {
fmt.Println(Meow)
fmt.Println("I can also bark, woof woof")
}
```

```
type Property string

const (
Name   Property = "name"
Breed  Property = "breed"
Age    Property = "age"
Weight Property = "weight"
)

type Action string

const (
Bark Action = "bark"
Meow Action = "meow"
)

func X() {
d := Dog{}
d = Cat{} // Invalid
fmt.Println(d)

// Polymorphism: poly=many, morph=shape shift
var a Animal
a = Dog{}
a.Speak()

a = Cat{} // Valid as both of them satisfy the interface{}
a.Speak()
}
```

- Defining interfaces
- Implicitness of Interfaces

```
// `implements` keyword is available to signal that Interface contract has
been fulfilled
// In case of Golang, once receiver functions are implemented with the same
method name & signatures as the ones in the interface,
// contract is said to be fulfilled
```

- panic and recover

```
func main() {
a, b := 1, 1
v := X(a, b)
fmt.Println(v)

b = 0
v = X(a, b)
```

```
fmt.Println(v)

a, b = 22, 11
v = X(a, b)
fmt.Println(v)

y, err := Y(a, 0)
if err != nil {
fmt.Println(err)
}

y, err = Y(a, b)
if err != nil {
fmt.Println(err)
}
fmt.Println(y)
}

func Y(a, b int) (int, error) {
if b <= 0 {
return 0, fmt.Errorf("denominator cannot be less than zero")
}

return a / b, nil
}

func X(a, b int) int {
defer func() {
// recover block should be inside a deferred anonymous function
// recover should always be done in the goroutine that is supposed to panic
if r := recover(); r != nil {
fmt.Println(r)
}
}()

return a / b
}
```

- defer

```
func main() {
a, b := 1, 0
v := X(a, b)
fmt.Println(v)

a, b = 22, 11
v = X(a, b)
fmt.Println(v)
}

func X(a, b int) int {
defer func() {
```

```
if r := recover(); r != nil {
fmt.Println(r)
}
}()

defer func() { // Executed in Last In First Out Order
fmt.Println("called in defer")
}()

fmt.Println("something")
fmt.Println("something else")
time.Sleep(5 * time.Second)
fmt.Println("woke up")

return a / b
}
```

- Concurrency

  Programs are sequential by default, which means each line executed in the order that it is written. In certain scenarios, it makes sense to start execution of certain processes at the same time.

  - Concurrency: Started at the same time, under execution, but not always necessarily being executed at the same time.
  - Parallelism: Necessarily run at the same time.
  - Processes
    - Any independent application runs as a separate process on the CPU
    - Different processes do not share resources by default
  - Threads
    - Unit of a process
    - Different threads within the same process share resources
  - Goroutines
    - When writing parallel or concurrent application, developer has to consider OS Processes & Threads and their properties
    - To abstract, Go language runtime takes cares of this by introducing goroutines
    - Goroutines are lightweight tasks performed on the thread/threads and scheduled by the Go Scheduler(part of Go Runtime)
    - Two types of goroutines:
      - Run by the Go Runtime(default),
      - User goroutines.
    - Main function, GC, Go Sched, etc are default goroutines
    - If main goroutine of an application exits, the application does no wait for other goroutines to finish

```go
package main

import (
  "encoding/json"
  "net/http"
)

type Data struct{
  ID string `json:"id"`
  Name string `json:"name"`
}

type Private struct {
  Address struct{} `json:"address"`
  Phone string `json:"phone"`
}

/*
[{
"id":"1",
"name": "Shashank"
},{
"id":"2",
"name": "Priyadarshi"
}]
*/

func main(){
  const API_URL = "https://example.com"

  var (
    req *http.Request
    client *http.Client
  )

  resp, err := client.Do(req)
  if err != nil{}

  body, err := io.ReadAll(resp.Body)
  if err != nil{}

  var data []Data
  err = json.Unmarshal(body, &data)
  if err != nil{}

  for _, d := range data {
    go func(){ // Anonymous function as a concurrent goroutine
      url := fmt.Sprintf("%s/%d", API_URL, d.ID)

      resp, err := client.Do(req)
      if err != nil{}

      body, err := io.ReadAll(resp.Body)
```

```
      if err != nil{}

      var private Private
      err = json.Unmarshal(body, &private)
      if err != nil{}
    }()
  }
}
```

```
package main

import (
"fmt"
"sync"
)

func main() {
var wg = &sync.WaitGroup{}
wg.Add(2)

// ch := make(chan bool) // Buffered channel, synchronous
ch := make(chan bool, 2) // Unbuffered channel, asynchrounous

go func() {
defer wg.Done()
fmt.Println(1)
ch <- true
}()

go func() {
defer wg.Done()
<-ch
fmt.Println(2)
}()

wg.Wait()
close(ch)

// ch <- true // panic: send on closed channel

fmt.Println(3)

// ch <- false // in case of buffered channels: panic: all goroutines are
asleep - deadlock! reason: no reader present before write!
// ch <- false // in case of unbuffered channels of size 1: panic: all
goroutines are asleep - deadlock! reason: channel is full!
}
```