L1 MODULE 2

PRESENTED BY: GOPAL RANJAN

CONTENT

Golang

Go Style Guide

Version control with Git

Task Link

ABOUT GOLANG

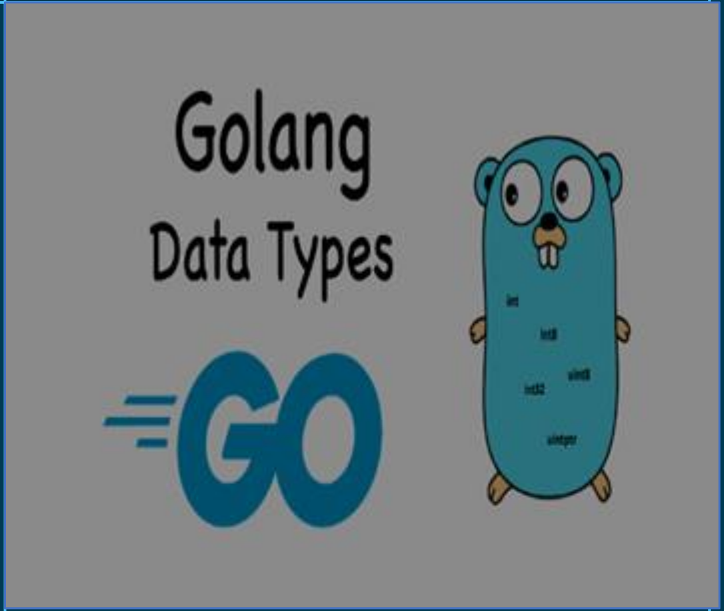
- Go is an open-source programming language developed by Google.
- Widely used in cloud computing, microservices, and DevOps tools.
- Every Go program is made up of packages.
- Programs start running in a package `main`.
- "fmt", package provided for formatting.
- A name is exported if it begins with a capital letter.

DATA TYPES IN GO

Go is statically typed, meaning that once a variable type is defined, it can only store data of that type.

Some basic data types include:

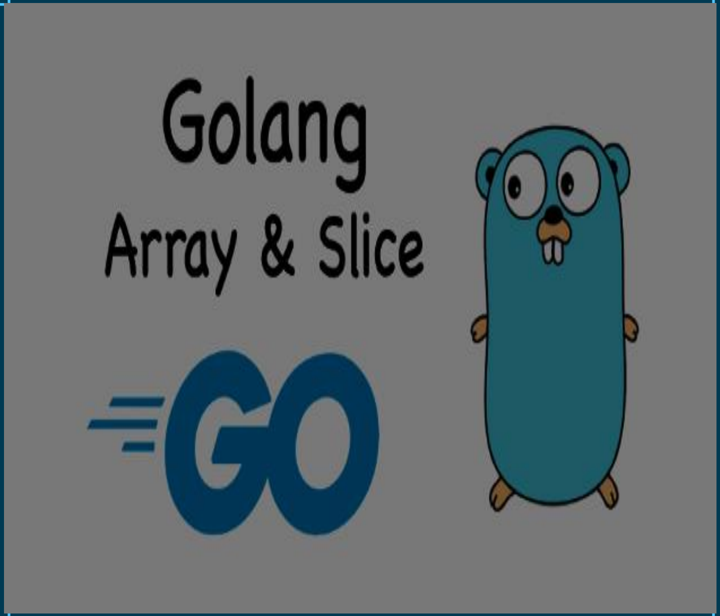
- `Bool`
- `String`
- `int  int8  int16  int32  int64`
- `uint uint8 uint16 uint32 uint64 uintptr`
- `byte // alias for uint8`
- `rune // alias for int32`
- `float32 float64`
- `complex64 complex128`



GO ARRAYS & SLICES

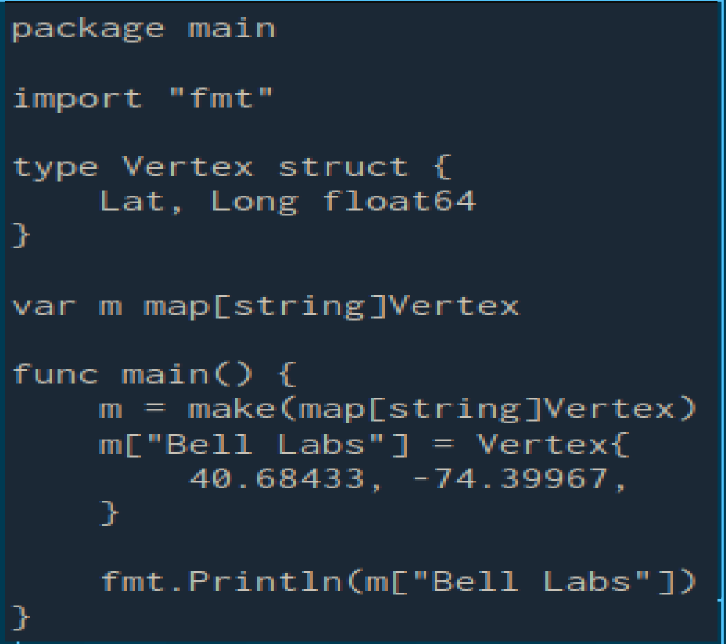
- The type `[n]T` is an array of `n` values of type `T`.
- `var a [10]int` //declares a variable `a` as an array of ten integers.
- Slice is a dynamically-sized, flexible view into the elements of an array.
- The type `[]T` is a slice with elements of type `T`.
- A slice is formed by specifying two indices, a low and high bound, separated by a colon:
- `a[low : high]` - Selects a half-open range which includes the first element but excludes the last one.
- A slice does not store any data, it just describes a section of an underlying array.
- The `make` function allocates a zeroed array and returns a slice that refers to that array:

  ```
  func make([]T, len, cap) []T
  ```
- **`Range: returns two values,`** first is the index, and the second copy of the element at that index.



MAPS

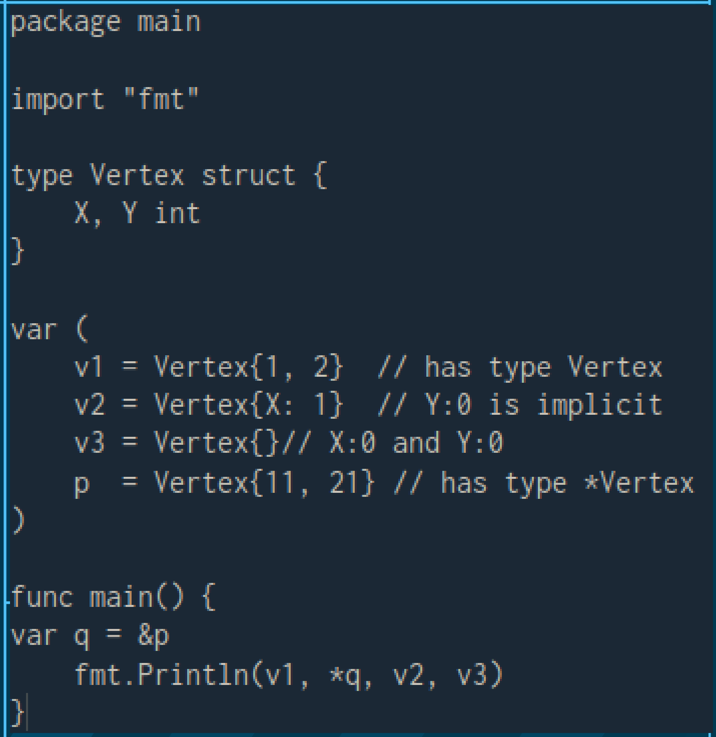
- A map maps keys to values.
- The zero value of a map is `nil`. A `nil` map has no keys, nor can keys be added.
- The `make` function returns a map of the given type, initialized and ready for use.
- Mutating in **MAPS:**
  - **INSERT:** `m[key] = elem`
  - **RETRIEVE:** `elem = m[key]`
  - **DELETE:** `delete(m, key)`
  - **TEST:** `elem, ok = m[key]`
    - **If Key is present, Ok is true else false.**
    - **If Key is not in map, then elem is zero**



```
package main

import "fmt"

type Vertex struct {
	Lat, Long float64
}

var m map[string]Vertex

func main() {
	m = make(map[string]Vertex)
	m["Bell Labs"] = Vertex{
		40.68433, -74.39967,
	}

	fmt.Println(m["Bell Labs"])
}
```

STRUCTS

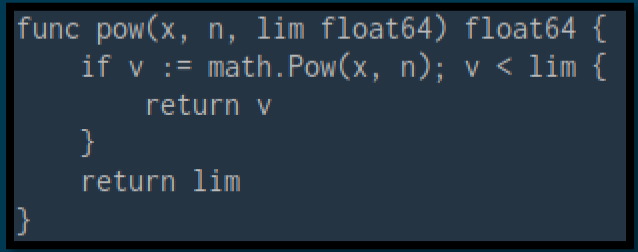
- A `struct` is a collection of fields.
- Struct fields are accessed using a dot.
- A struct literal denotes a newly allocated struct value by listing the values of its fields.
- The special prefix & returns a pointer to the struct value.



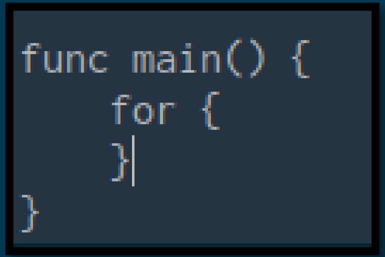
```go
package main

import "fmt"

type Vertex struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2}  // has type Vertex
    v2 = Vertex{X: 1}  // Y:0 is implicit
    v3 = Vertex{}// X:0 and Y:0
    p  = Vertex{11, 21} // has type *Vertex
)

func main() {
var q = &p
    fmt.Println(v1, *q, v2, v3)
}
```

CONTROL STATEMENTS

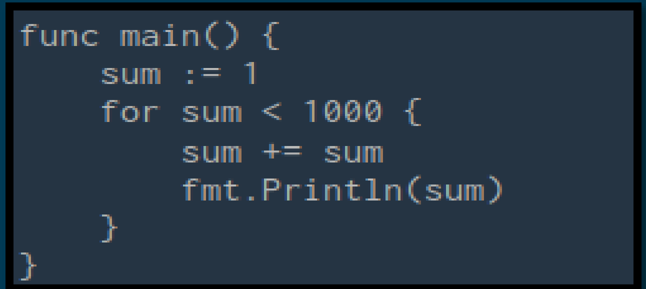
- Go has only one looping construct, the `for` loop.
- There are no parentheses surrounding the three components of the `for` statement and the braces { } are always required.
- The init and post statements are optional.
- `if` statements: the expression need not be surrounded by parentheses ( ) but the braces { } are required.



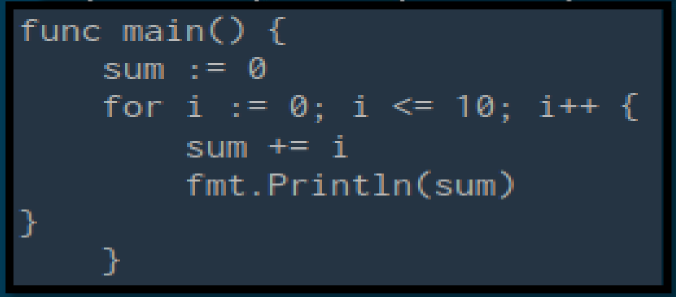
```go
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}
```



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



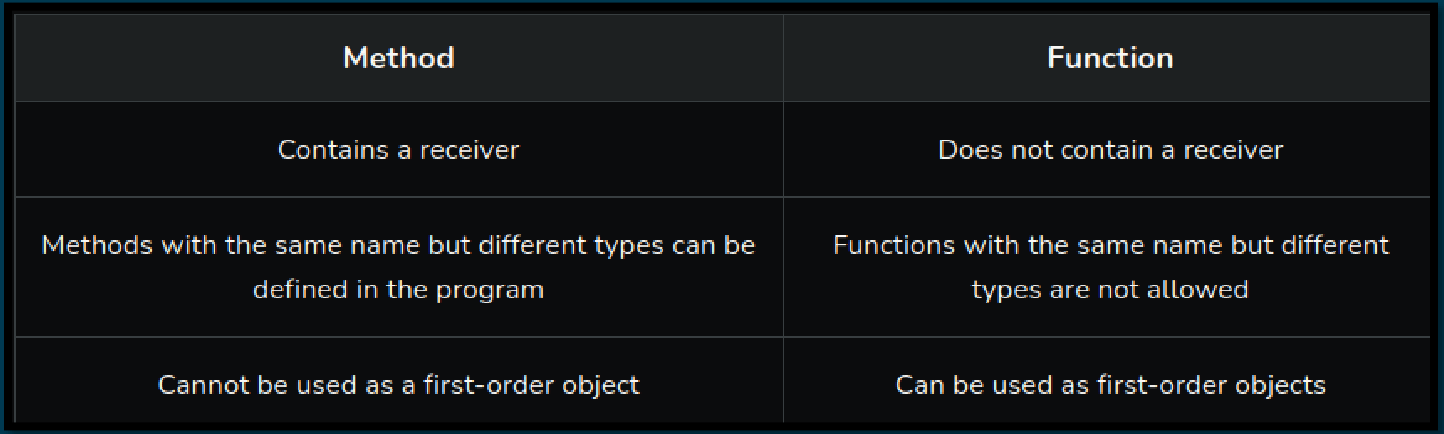
```go
func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
        fmt.Println(sum)
    }
}
```



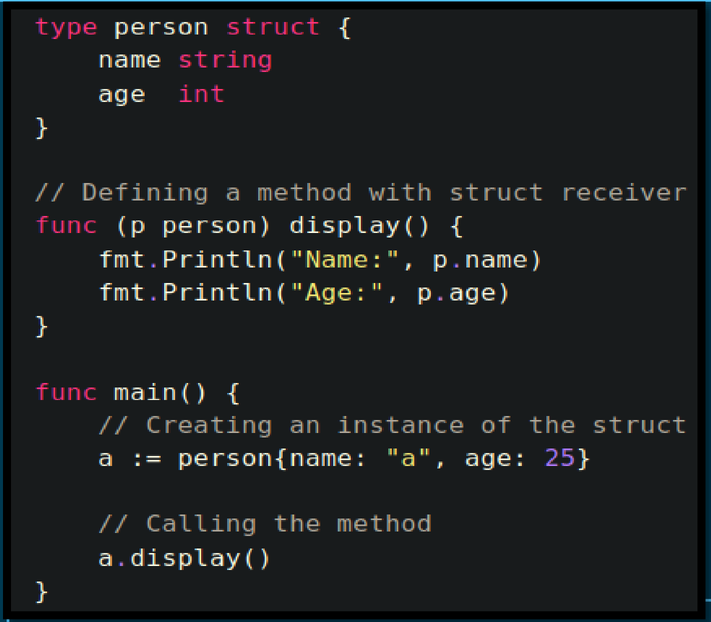
```go
func main() {
    sum := 0
    for i := 0; i <= 10; i++ {
        sum += i
        fmt.Println(sum)
}
    }
```

METHODS IN GO

- Go methods are functions but with a receiver argument, which allows access to the receiver's properties.
- The receiver can be a struct or non-struct type, but both must be in the same package.
- Methods cannot be created for types defined in other packages, including built-in types like `int` or `string`; otherwise, the compiler will raise an error.



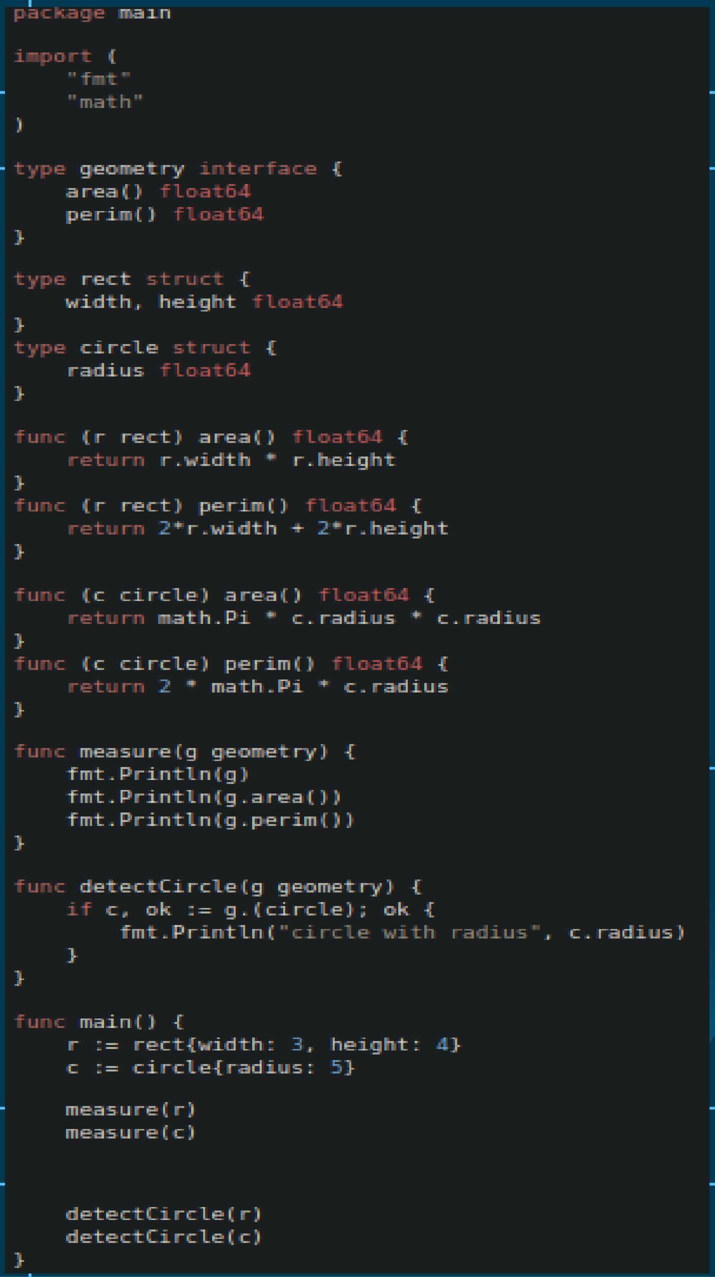
Method	Function
Contains a receiver	Does not contain a receiver
Methods with the same name but different types can be defined in the program	Functions with the same name but different types are not allowed
Cannot be used as a first-order object	Can be used as first-order objects



```go
type person struct {
    name string
    age  int
}

// Defining a method with struct receiver
func (p person) display() {
    fmt.Println("Name:", p.name)
    fmt.Println("Age:", p.age)
}

func main() {
    // Creating an instance of the struct
    a := person{name: "a", age: 25}

    // Calling the method
    a.display()
}
```

INTERFACES

- **Interfaces** are named collections of method signatures.
- No need to use implement keyword Just give the definition of methods defined in an interface.
- Go Interfaces are implemented implicitly.



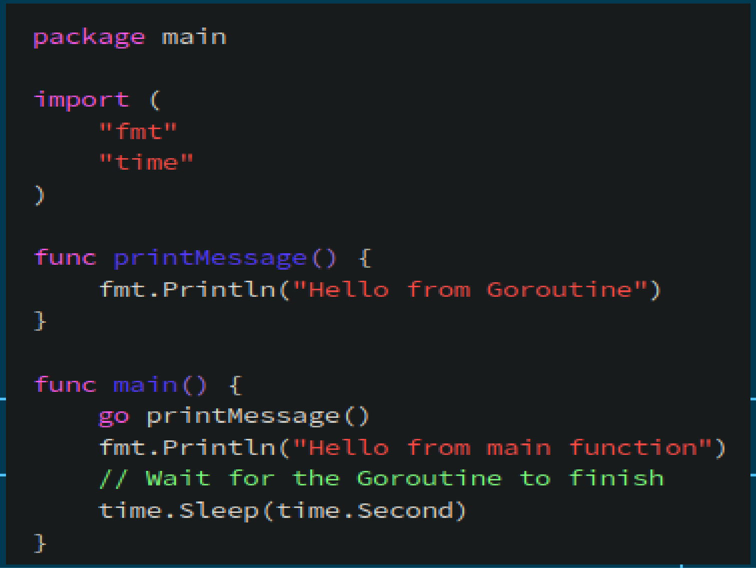
```go
package main

import (
    "fmt"
    "math"
)

type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func detectCircle(g geometry) {
    if c, ok := g.(circle); ok {
        fmt.Println("circle with radius", c.radius)
    }
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    measure(r)
    measure(c)


    detectCircle(r)
    detectCircle(c)
}
```

GENERICS & CONCURRENCY

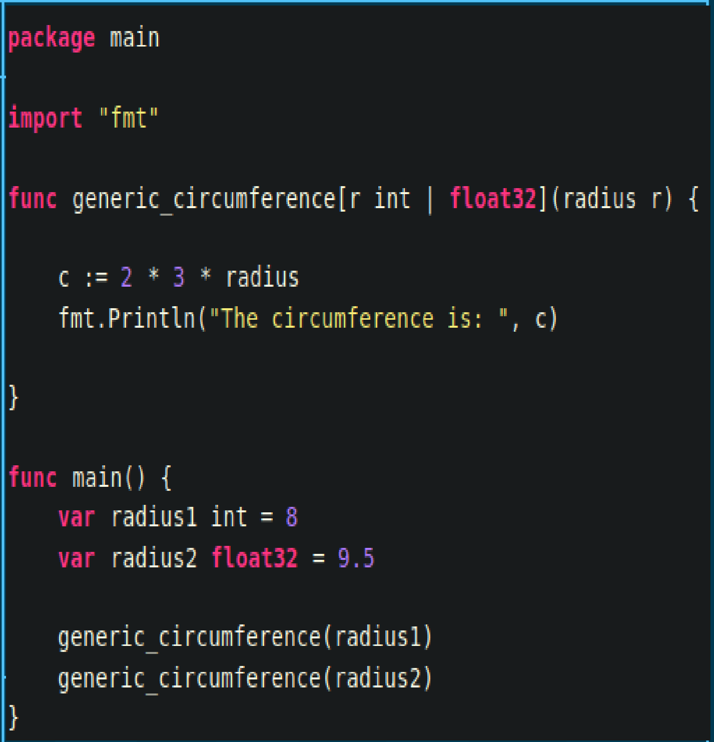
- **Generics** are a way of writing code that is independent of the specific types being used.

- **Concurrency** in computing enables different parts of a program to execute independently, potentially improving performance and allowing better use of system resources.

- A **Goroutine** is a lightweight thread of execution. The term comes from the phrase “Go subroutine”, reflecting the fact that Goroutines are functions or methods that run concurrently with others.



```go
package main

import (
    "fmt"
    "time"
)

func printMessage() {
    fmt.Println("Hello from Goroutine")
}

func main() {
    go printMessage()
    fmt.Println("Hello from main function")
    // Wait for the Goroutine to finish
    time.Sleep(time.Second)
}
```



```go
package main

import "fmt"

func generic_circumference[r int | float32](radius r) {

    c := 2 * 3 * radius
    fmt.Println("The circumference is: ", c)

}

func main() {
    var radius1 int = 8
    var radius2 float32 = 9.5

    generic_circumference(radius1)
    generic_circumference(radius2)
}
```

GO STYLE GUIDE

- Outline the foundation of Go style at Google.
- Serves as a reference for writing Go code, emphasizing readability, consistency and maintainability.
- Incorporates lessons learned from years of Go development at Google.
- Widely respected and followed by many Go developers, even outside of Google.
- Guide may deprecate certain practices or features over time. Developers are expected to stay updated with the latest guidelines.

VERSION CONTROL WITH GIT

KEY TERMINOLOGIES

- **VCS/SCM:** A **version control system** (abbreviated as **VCS**) is a tool that manages different versions of source code. A **source code manager** (abbreviated as **SCM**) is another name for a version control system.

- **Commit (snapshot):** Git thinks of its data like a set of snapshots of a mini file system. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that Snapshot.

- **Repository (repo):** A directory that contains your project work, as well as a few files(hidden by default in Mac OS X) which are used to communicate with Git.

- **Working Directory:** The files that you see in your computer's file system. When you open your project files up on a code editor, you're working with files in the Working Directory.

- **Checkout:** When content in the repository has been copied to the Working Directory. It is possible to checkout many things from a repository; a file, a commit, a branch, etc.

- **Checkout:** When content in the repository has been copied to the Working Directory. It is possible to checkout many things from a repository; a file, a commit, a branch, etc.

- **Staging Area or Staging Index or Index:** A file in the Git directory that stores information about what will go into your next commit. You can think of the staging area as a prep table where Git will take the next commit. Files on the Staging Index are poised to be added to the repository.

- **SHA:** A SHA is basically an ID number for each commit. It is a 40-character string composed of characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. "SHA" is shorthand for "SHA hash". A SHA might look like this: e2adf8ae3e2e4ed40add75cc44cf9d0a869afeb6

- **Branch:** A branch is when a new line of development is created that diverges from the main line of development. This alternative line of development can continue without altering the main line.

- GIT Init: Used to create a new, empty repository in the current directory.

```
$ git init
```

- GIT Clone: Used to create an identical copy of an existing repository.

```
$ git clone <path-to-repository-to-clone>
```

- GIT Status: Will display the status of repository.

```
$ git status
```

- GIT Log: Used to display all the commits of a repository by displaying SHA, Author, Date and the message.

```
$ git log
```

- **GIT Log OneLine: lists one commit per line, shows the first 7 characters of the SHA and shows the commit message**

```
$ git log --oneline
```

- **GIT Log Stat: Displays the modified files, number of lines added or removed and summary with total no of modifies files and lines that have been added or removed.**

```
$ git log --stat
```

- **GIT Log Patch: Displays the actual changes made to a file.**

```
$ git log –p or --patch
```

- **GIT Show: Displays only one commit .**

```
$ git show #shows the most recent commit
$ git show <SHA_ID> #shows commit with given SHA ID
```

- GIT Add: Used to move the files from working directory to staging index, take a space separated list of file name .

```
$ git add <file1> <file2> … <fileN>
```

- GIT Commit: Takes file from staging index and saves them in repository.

```
$ git commit

# To pass the commit message directly

$ git commit –m "Commit message"
```

- GIT Diff: Used to see changes that have been made but not committed yet.

```
$ git diff
```

- GIT Ignore: Used to let git know about the files to be not to be tracked.

```
.gitignore
```

- GIT Tag: Used to add a marker on a specific commit.

```
   $ git tag -a beta # -a to create annotated tag because annotated info of the person creating it,
date it was created and message for the tag

   $ git tag -d beta # for deleting the tag
```

- GIT Branch: Used to list all branches in repo, create new and delete branches.

```
   $ git branch # Lists all branches in the repo

   $ git branch <branch_name> # Create a new branch

   $ git branch -d <branch_name> # Delete the branch
```

- GIT Checkout: Used to switch between branches.

```
   $ git checkout <branch_name>

   $ git checkout -b <branch_name> #creates and switch to new branch
```

- GIT Merge: Used to combine branches in git.

```
$ git merge <other_branch>
```

- GIT Commit Amend: Used to update the most recent commit instead of creating a new one.

```
$ git commit --amend
```

- GIT Revert: Used to reverse a previously made commit.

```
$ git revert <SHA-of-commit-to-revert>
```

- GIT Reset: Used to erase commits.

```
$ git revert <reference-to-commit>

# erase commits with hard tag.
# moves committed changes to staging index with –soft tag.
# unstages committed changes –mixed flag
```

TASK LINK : TASK_MANAGER

THANK YOU