The Go Programming Language

Methods and Interfaces

Anuchit Prasertsang
Developer

WordCount

- function name is WordCount

WordCount - example

```
func WordCount(s string) map[string]int {
    words := strings.Fields(s)
    r := map[string]int{}
    for _, w := range words {
        r[w] = r[w] + 1
    }
    return r
}

func main() {
    // raw string use ``
    s := `If it looks like a duck, swims like a duck,
        and quacks like a duck, then it probably is a duck.`
    w := WordCount(s)
    fmt.Printf("% #v\n", w)
}
```

Run

Move WordCount to new package

```
bin/
    ...
pkg/
    ...
src/
    github.com/golang/example/
        .git/
    hello/
        main.go                # main เรียกใช้ ฟังก์ชั่น WordCount
        ...
    words/
        words.go               # WordCount function is here

    ... (many more repositories and packages omitted) ...
```

- add print your name inside function **WordCount**.

Public Library

git basic (https://slides.com/anuchito/basic-git-3#/)

- push to github
- remove **src/words**
- go run hello/main.go

Using Library

```
go get [packages]
```

- example

```
go get github.com/AnuchitO/words
```

1. ใช้ lib ตัวเอง
2. ใช้ lib เพื่อน

Exercise - package I

Step I

```
- ให้แยก Person struct ไปอยู่ใน package person
- ให้สร้าง function New ที่รับค่า name, age และคืนค่า เป็น Person
- สร้างไฟล์ person.go เป็น ไฟล์ main เพื่อเรียกใช้ฟังก์ชั่น New และ Print ค่า Person ออกมาดู
- public package person ให้เพื่อนสามารถใช้และเรียกใช้ของเพื่อนข้างๆได้
```

Step II

```
- สร้าง folder workspace ที่ไม่ได้อยู่ภายใต้ GOPATH
- สร้าง folder people ภายใต้ folder workspace
- สร้างไฟล์ people/people.go มี function main เรียกใช้ New Person
- สั่ง run people/people.go
```

Step III

```
- ลบ package person ใน GOPATH
- สั่ง run people/people.go
```

Go Modules

Modules

- A collection of related Go packages that are versioned together as a single unit

Modules record precise dependency requirements and create reproducible builds.

Most often, a version control repository contains exactly one module defined in the repository root. (Multiple modules are supported in a single repository, but typically that would result in more work on an on-going basis than a single module per repository).

Repositories, Modules, and Packages:

- A repository contains one or more Go modules.
- Each module contains one or more Go packages.
- Each package consists of one or more Go source files in a single directory.
- Modules **must be** semantically versioned v(major).(minor).(patch)
- such as v0.1.0 or v1.2.3. The leading **v is required**.

If using Git, tag released commits with their versions.
Public and private module repositories and proxies are becoming available (see FAQ below).

Go inside GOPATH

- export GO111MODULE=on
- go mod init
- go mode tidy

Update dependency

- upgrade
- downgrade

go.mod

```
require (
  github.com/AnuchitO/say v1.0.0
)
```

Replace dependency

- go mod edit -replace github.com/AnuchitO/say=./say

Multiple version

- say v1
- say v2

```
require (
  github.com/AnuchitO/say v1.0.0
  github.com/AnuchitO/say/v2 v2.0.0
)
```

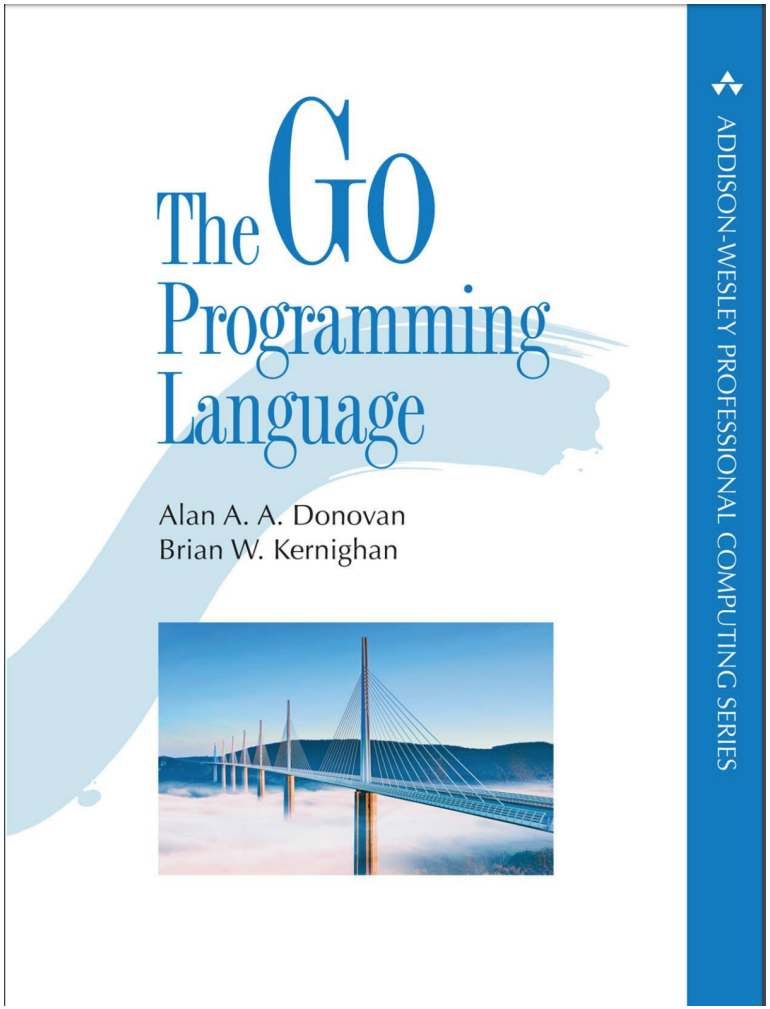
Clean cache

```
go clean -modcache
```

Module compatibility and semantic versioning

research.swtch.com/vgo-module (https://research.swtch.com/vgo-module)

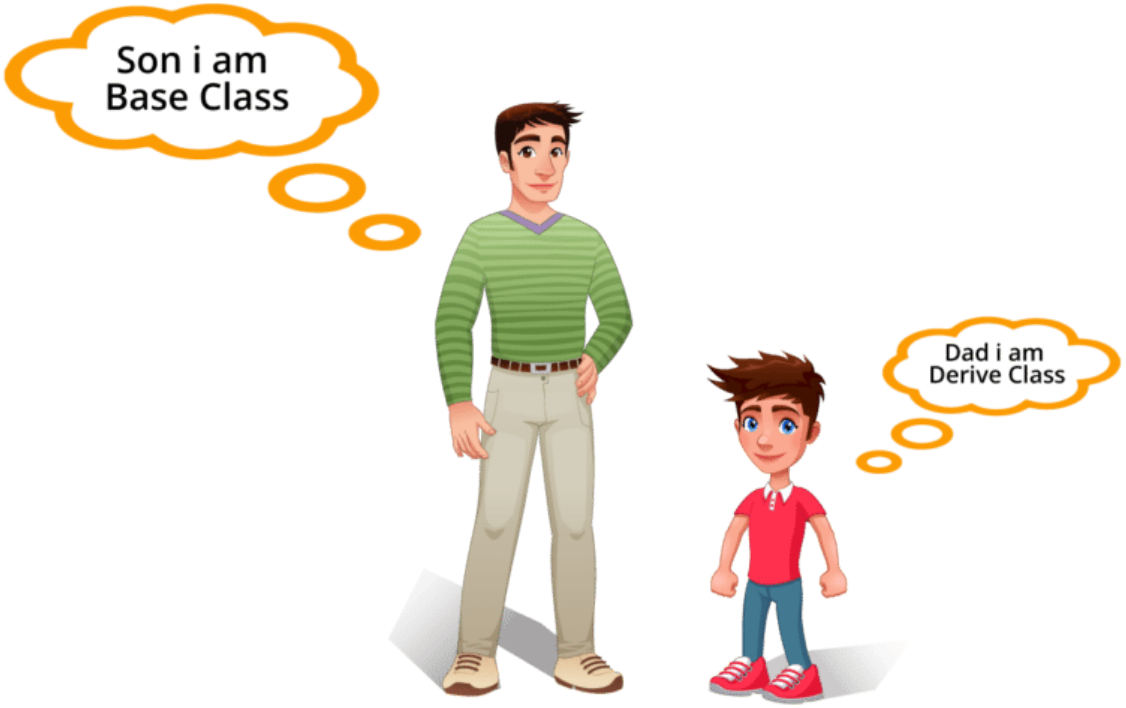
Books


The Go Programming Language
Alan A. A. Donovan
Brian W. Kernighan
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Object Oriented Programming (OOP)

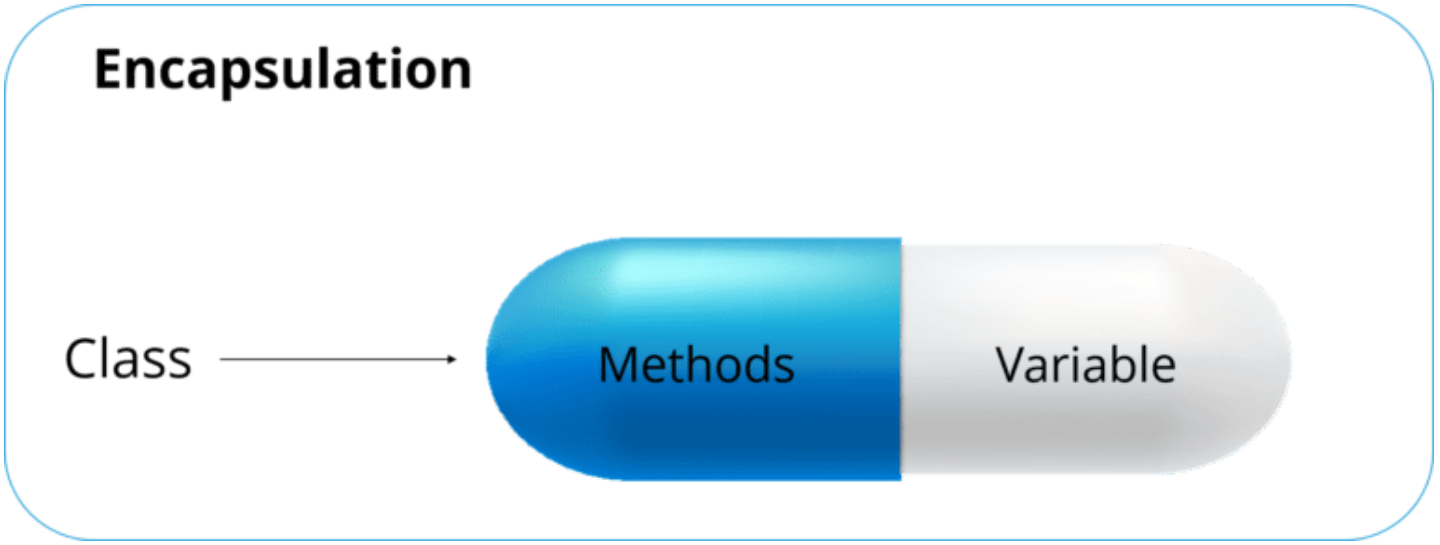
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

Inheritance



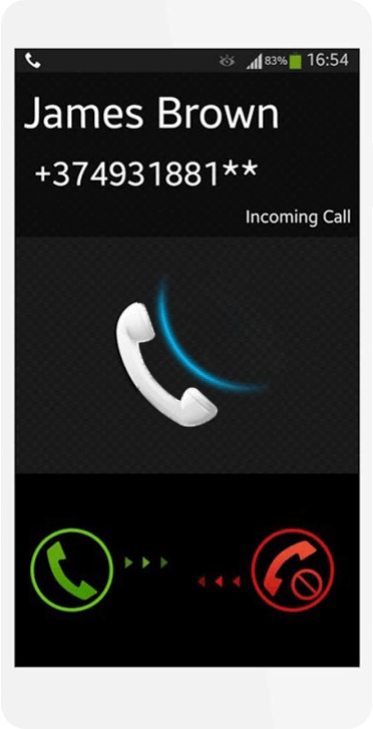
one such concept where the properties of one class can be inherited by the other

Encapsulation



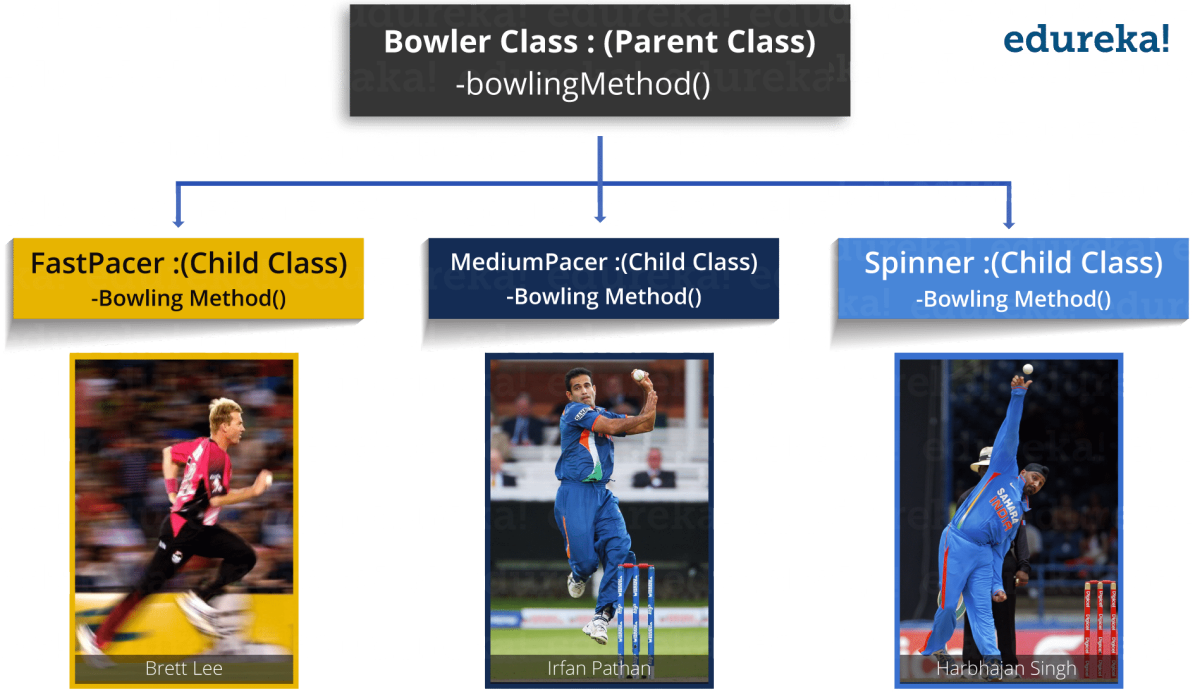
A mechanism where you bind your data and code together as a single unit.
It also means to hide your data in order to make it safe from any modification.

Abstraction



Abstraction refers to the quality of dealing with **ideas** rather than events.
It basically deals with hiding the details and showing the essential things to the user.

Polymorphism



poly means *many* and **morph** means *forms*.

It is the ability of a variable, function or object to take on multiple forms.

In other words, polymorphism allows you define one interface or method and have multiple implementations.

OOP in Go

Normal function

```
package main

import "fmt"

type rectangle struct {
    width  int
    length int
}

func area(rec rectangle) int {
    return rec.width * rec.length
}

func main() {
    rec := rectangle{3, 4}
    a := area(rec)
    fmt.Println()
}
```

Run

Methods

```
package main

import "fmt"

type rectangle struct {
    width  int
    length int
}

func (rec rectangle) area() int {
    return rec.width * rec.length
}

func main() {
    rec := rectangle{3, 4}
    a := rec.area()
    fmt.Println(a)
}
```

Run

Exercise - Person

- Add methods

```
- Walk -> print name + "walking"
- Eat  -> print name + "eating"
- Geeting -> print "hello" + name
- getter for field name
- setter for field name
```

Methods

- Go does not have classes.
- Go define methods on types.

```
type Point struct{ X, Y float64 }

// traditional function
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// same thing, but as a method of the Point type
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

- A method is just a function with a receiver argument.
- the extra parameter **p** is celled the method's **receiver**.
- declare a method as the same package only.

Calling the Method

```
p := Point{1, 2}
q := Point{4, 6}
```

- How to call Distance **function**?
- How to call Distance **method**?
- What happen when declare method in another package?

Calling the Method - continue

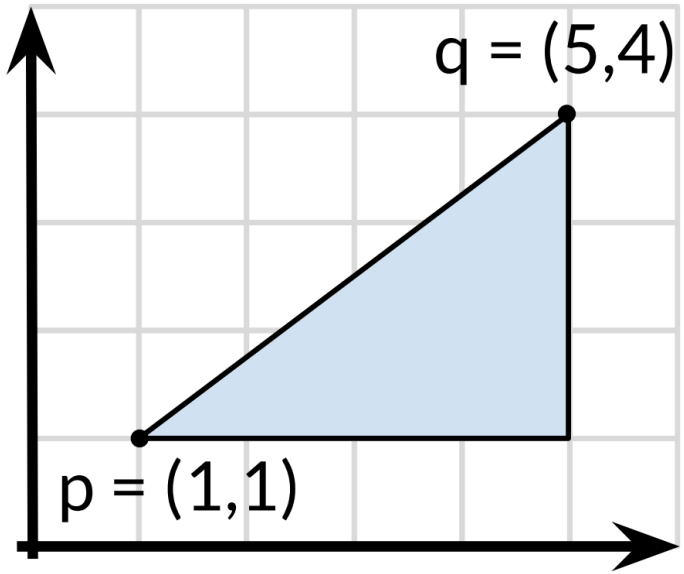
```
fc := Distance(p, q) // function call
mc := p.Distance(q) // method call
fmt.Println(fc)
fmt.Println(mc)
```

- The expression **p.Distance** is called a *selector*, because it selects the appropriate **Distance** method for the receiver **p** of type **Point**.
- Selector are also used to select fields of struct as in **p.X**

How to define methods on another types.

```
type Path []Point
```

- Declare method **Distance()**, return type is float64
- just hard code return is **0.0**
- How to Initialize **Path** and call method **Distance**?



Distance implement

```
type Path []Point

func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}

func main() {
    p := Path{
        {1, 1},
        {5, 1},
        {5, 4},
        {1, 1},
    }

    sum := p.Distance()

    fmt.Println("sum of distance", sum)
}
```

Run

Exercise - Point

- move point to package **geometry**
- package **geometry** อยู่ใน module "github.com/<account>/geometry"
- public package **geometry** to github
- ไฟล์ main ที่เรียกใช้ geometry ต้องยังใช้งานได้อยู่

Methods with a Pointer Receiver

- Add method **ScaleBy** for *Point*

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

- In a realistic program, convention dictates that if any method of **Point** has a poiter receiver, then **all** methods of **Point** should have a pointer receiver
- How to call methods **ScaleBy**?

Calling the method - ScaleBy

```
func main() {
    r := &Point{1, 2}
    r.ScaleBy(2)
    fmt.Println(*r)   // "{2, 4}"

    // or this:
    p := Point{1, 2}
    pptr := &p
    pptr.ScaleBy(2)
    fmt.Println(p) // "{2, 4}"

    // or this:
    p = Point{1, 2}
    (&p).ScaleBy(2)
    fmt.Println(p)  // "{2, 4}"
}
```

Run

- How about we change **ScaleBy** to be value type not a pointer type?

Composing Types by Struct Embedding

- **ColorerPoint** that contains all the fields of **Point**.

```
import "image/color"

type Point struct{ X, Y float64 }
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

Run

- How about this?

```
    var cp ColoredPoint
    cp.X = 1
    fmt.Println(cp.Point.X)
    cp.Point.Y = 2
    fmt.Println(cp.Y)
```

Run

- How about methods?

Applies methods of Point

- We can call method of the embedded **Point** field using a receiver of type **ColorerPoint**, even though ColorerPoint has no declared methods

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}

d := p.Distance(q.Point)
fmt.Println(d)
p.ScaleBy(2)
q.ScaleBy(2)

d = p.Distance(q.Point)
fmt.Println(d)
```

Run

Method Values

```
p := Point{1, 2}
q := Point{4, 6}
distanceFromP := p.Distance
fmt.Println(distanceFromP(q))

var origin Point
fmt.Println(distanceFromP(origin))
scaleP := p.ScaleBy
scaleP(2)
scaleP(3)
scaleP(10)
```

Run

Method Expressions

```
p := Point{1, 2}
q := Point{4, 6}
distance := Point.Distance
fmt.Println(distance(p, q))
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)
fmt.Printf("%T\n", scale) // "func(*Point, float64)"
```

Run

Encapsulation

Go has only one mechanism to control the visibility of names: **capitalized** identifiers are exported from the package in which they are defined , and **uncapitalized** names are not.

```
type IntSet struct {
    words []uint64
}
```

- What happen if we change **Y** to **y** of **Point**?

Interfaces

empty interface

```
package main

import "fmt"

func main() {
    var v interface{}
    v = 1
    fmt.Printf("%T %#v\n", v, v)
    v = "1"
    fmt.Printf("%T %#v\n", v, v)
}
```

Run

interface: type assertion

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"
    s := i.(string)
    fmt.Println(s)
    if s, ok := i.(string); ok {
        fmt.Println(s)
    }
}
```

Run

interface: type switch

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"
    switch v := i.(type) {
    case int:
        fmt.Printf("%T %d", v, v)
    case string:
        fmt.Printf("%T %s", v, v)
    default:
        fmt.Println("undefined type")
    }
}
```

Run

interface

To define a set of method signatures

```
Interfaces specify behaviors.
An interface type defines a set
of methods:


type areaer interface {
    area() int
}
```

Interfaces are implemented implicitly

Excercise - Interface

```
type triangle struct {
    base  float64
    height float64
}

func (rec triangle) area() float64 {
    return 0.5 * rec.base * rec.height
}
```

- How do we define interface for rectangle and triangle?

interface: Stringer

```
type Stringer interface {
    String() string
}
```

interface: error

```
type error interface {
    Error() string
}
```

Idiom Error Handling in Go

References

Object-Oriented Programming (https://www.edureka.co/blog/object-oriented-programming/)

Thank you

Anuchit Prasertsang
Developer
anuchit.prasertsang@gmail.com (mailto:anuchit.prasertsang@gmail.com)
https://github.com/AnuchitO (https://github.com/AnuchitO)
@twitter_AnuchitO (http://twitter.com/twitter_AnuchitO)