Moving from Inheritance based language to Go

Hannah

Summary

- Go over fundamentals of Go
- Touch on idosycrosices of Go compared to most other Object Oriented Languages.
- Walk through how Go achieves polymorphism in programs without inheritance

Basic Facts of Go

- Static, complied language
- Based off:
  C (expression, syntax, pointers, basic types)
  CSP (used in Go for concurrency pattern)
  Pascal (can be completely functional)
- Usually considered for web and containerized applications, where size, speed and scalability are a factor.
- Lacks Generics, Inheritance, and some more advanced frameworks (encourages people to create them themselves)

Installation Notes

- Can be install with package manager or by website
- $GOPATH is a variable that needs to set on after installing Golang binary to where code is setup
    - all code usually sits in /src file under the GOPATH
- Go Build builds a executable, go run builds and runs specific file
- Go fmt formats all code to a uniform standard

Structures of Go

```
package main // Package Name, main is default, can import others

import "fmt"

var point1 = [2]int{2, 3} //x, y (array of int (statically assigned to 2))
var point2 = [ ]int{4,5} // x, y (slice of an array of int (not static
amount, can have n amount of data))

//Main function is function that is run when run go run or the built
executable
func main() {
    fmt.Printf("%v", DistanceBetween()) //prints file out into
}

// DistanceBetween calculates the distance between two graph points x and y
func DistanceBetween() (x, y) {
    return (point1[0]-point2[0]),  (point1[1]-point2[1])
      //Can return multiple variables inline
}
```

Types and Functions

```
package main

import "fmt"

type Point struct {x, y int} // Structs are a collection of types, more
accurate description

func main() {
    var p1 = Point{1,2} //initializing a type into a variable
    p2 := Point{ x:2, y:2} //different way of init any variable
    fmt.Printf("%v", distanceBetween(p1, p2))
}

func distanceBetween(p1 Point, p2 Point) (Point) {
    return Point{p1.x-p2.x, p1.y-p2.y}
}
```

Objects with Inheritance

```
<?php

class Animal{
    public function move($speed, $distance){
       echo $speed * $distance
    }

    public function noise(){
        echo ‘’;
    }
}

class Cat extends Animal{
    public function noise(){
      echo ‘meow';
    }
}

$animal = new Animal();
$cat= new Cat();

$animal =>noise();   //Output: “”
$cat=>noise();   //Output:”meow”

$animal->move(1, 4);  // Output: ‘1'
$cat->move(1,4);       // Output: ‘1’

?>
```

Inheritance & Composition in Golang

- Object-Oriented, classes, polymorphism, and inheritance are interconnected in most modern languages.

- Go does have objects but does not have classes, which does not allow for inheritance.

- **Composition,** the building of object by combining of types and other structs into a new struct that can have attached functions achieving polymorphism.

- Structs allow for encapsulation still, which is done with public and private variables and function, which is indicated with capitalization in Go. (lower for private, upper for public)

```go
package main

import “fmt"

type Animal struct {
        speed    int
        distance int
        noise    string
}

type Cat struct {
        purr   string
        Animal
}

func (a Animal) Noise() {
        fmt.Printf(“noise")
}

func (a Animal) Move() {
        fmt.Printf("%d", (a.speed * a.distance))
}

func (c Cat) Purr() {
        fmt.Printf("purr")
}
func main() {
        var animal = Animal{speed: 10, distance:
4, noise: "meow"}
        var cat = Cat{}
        cat.purr = "purr"
        cat.Animal = animal
        cat.Purr()        //Output: purr
        cat.Noise()        //Output: meow
        cat.Animal.Noise() //Output: meow
        animal.Noise()    //Output: noise
}
```

```php
<?php

class Animal{
    public function move($speed, $distance) {
       echo $speed * $distance
    }

    public function noise() {
        echo ‘’;
    }
}

class Cat extends Animal{
    public function noise() {
      echo ‘meow';
    }

    public function purr() {
        echo ‘purr’;
    }
}

$animal = new Animal();
$cat= new Cat();

$animal =>noise();   //Output: “”
$cat=>noise();   //Output:”meow”

$animal->move(1, 4);  // Output: ‘1'
$cat->move(1,4);       // Output: ‘1’

$animal->purr();  //fail
$cat->purr();       // Output: ‘purr’

?>
```

Issues with Composition

- Composition allows for ease of building objects and sharing of structure (though not through inheritance), however, polymorphism is missing.

- Interfaces in go allow for polymorphism, by allowing for an interface to be a **contract of functions**. Any object that fulfills that contract can be passed in, allowing for multiple structs to be passed through

- This allows for grouping of data in different ways, including for mocks for testing.

```
package animals

type Animal struct {
        speed    int
        distance int
        noise    string
}

type Animals interface {
        Noise() string
        Move() int
}

type Cat struct {
        purr     string
        paws     string
        toeCount int
        Animal
}

type Dog struct {
        wag      string
        paws     string
        toeCount int
        Animal
}
```

```
func (a Animal) Move() int {
        return (a.speed * a.distance)
}

func (a Animal) Noise() string {
        return a.noise
}

func InitAllAnimals(distance int) (Animals, Animals) {
        var animal = Animal{speed: 10, distance:
distance, noise: "meow"}
        var cat = Cat{purr: "purr", paws: "furry
small", toeCount: 5, Animal: animal}

          animal.noise = "bark"
          animal.speed = 20
          var dog = Dog{wag: "wag the tail", paws: "furry
big", toeCount: 4, Animal: animal}

          return cat, dog

      }
```

```
package main

import (
        "fmt"
        "testingWithGo/Part1-
ObjectsConstructionInGo/interfaces/animals"
)

func main() {
        cats, dogs:= animals.InitAllAnimals(5)
        catMove, DogMove :=
AnimalsMovement(cats, dogs)
        fmt.Printf("CatMove: %d, DogMove %d",
catMove, DogMove)
                }

func AnimalsMovement(animals.Animals,
animals.Animals)(int, int){
        return cats.Move(), dogs.Move()
}
```

```
package main

import "testing"

type AnimalMock struct {
        move     int
        noise    string
}

type AnimalsMock interface {
        Noise() string
        Move() int
}

func (a AnimalMock) Move() int {
        return a.move
}

func (a AnimalMock) Noise() string {
        return a.noise
}
func TestAnimalsMovement(t *testing.T){
        animalMock := AnimalMock{move:10,
noise:"test"}
        test1, test2 :=
AnimalsMovement(animalMock, animalMock)
        if test1 != 10 {
                t.Error("test 1 didn't return 5")
        }
}
```