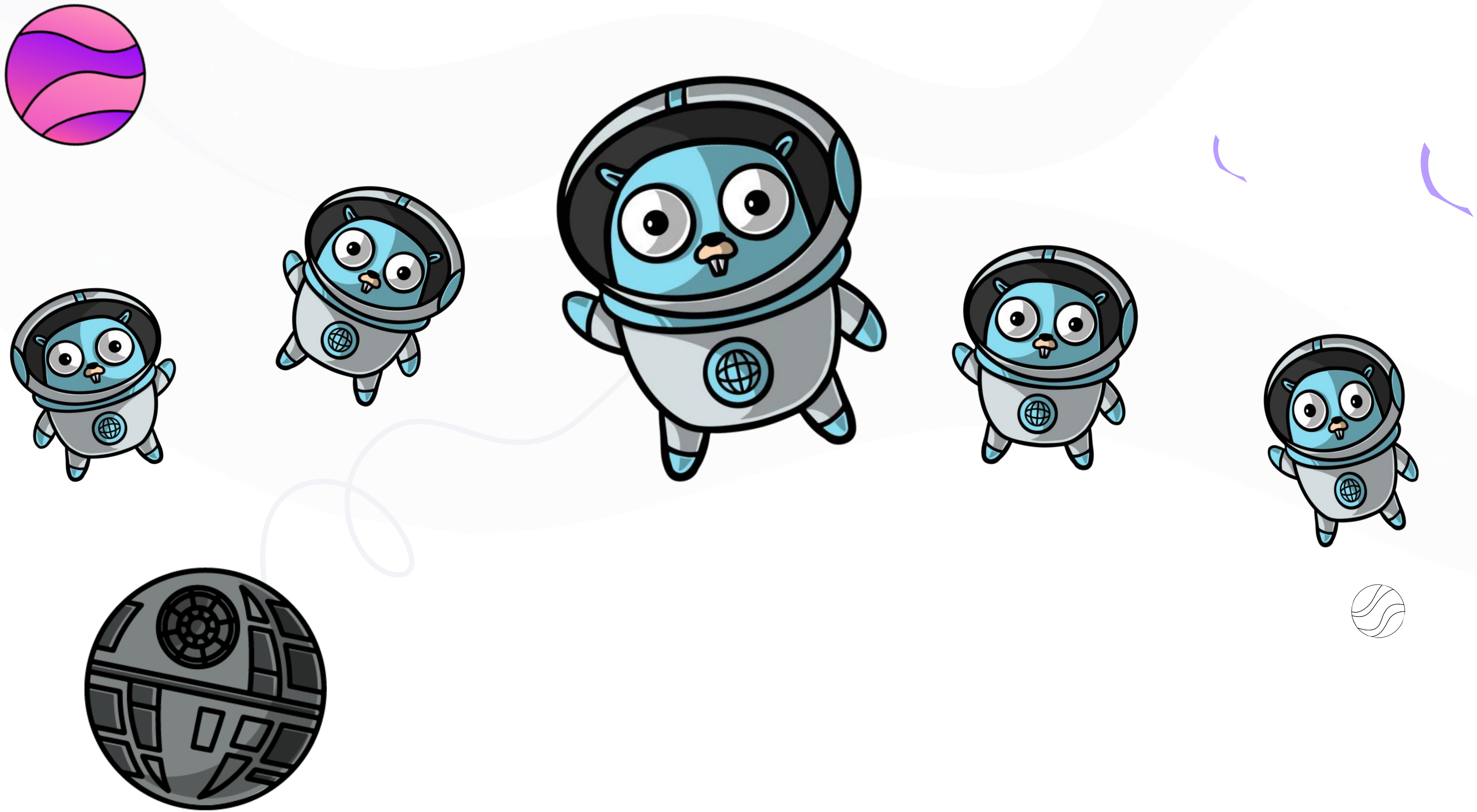


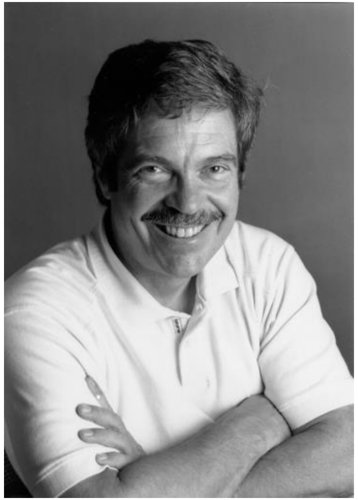
Agenda

- **Is Go an object-oriented language?**
  - What is OOP?
  - Composition
  - Encapsulation
  - Polymorphism
  - Inheritance?
- **Interfaces**
  - Overview
  - Implicit interfaces and Duck typing
  - How can I guarantee my type satisfies an interface?
  - Embedding and Interfaces
  - Interfaces and nil
  - The Empty Interface
  - Dependency Injection

Object-oriented programming (OOP) and GO

What OOP is?

Let's ask the man who made up the term 'object-oriented'



- **"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP“**

(Alan Kay)

OOP Original Conception

- Messaging
- Local retention, protection, and hiding of state-process
- Extreme late-binding of all things

Common OOP mechanics

- Encapsulation
- Composition
- Polymorphism
- Inheritance

Inheritance

- **Inheritance** means that child class is able use fields and methods of parent class.

- **Go doesn't support inheritance.**

“Object-oriented programming, at least in the best-known languages, involves too much discussion of the relationships between types, relationships that often could be derived automatically. Go takes a different approach.”

(Go Developers)

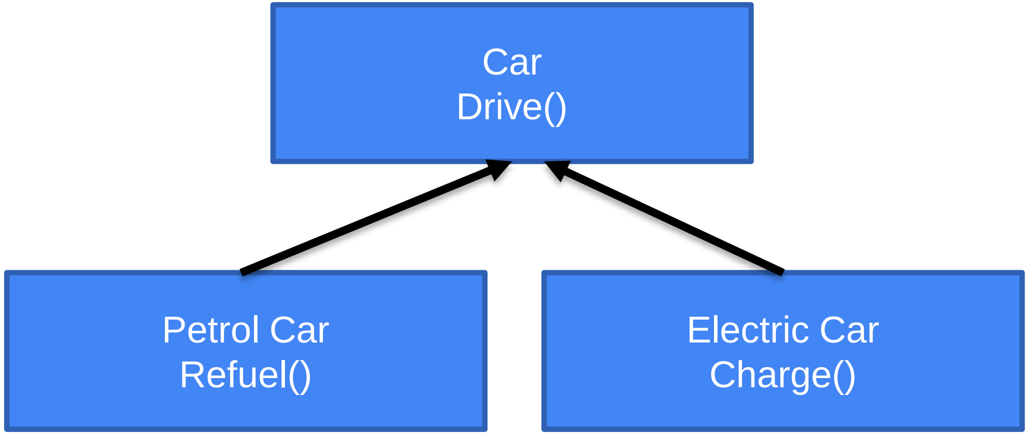
Composition over inheritance

- **Golang doesn't support inheritance. But Go supports composition.**

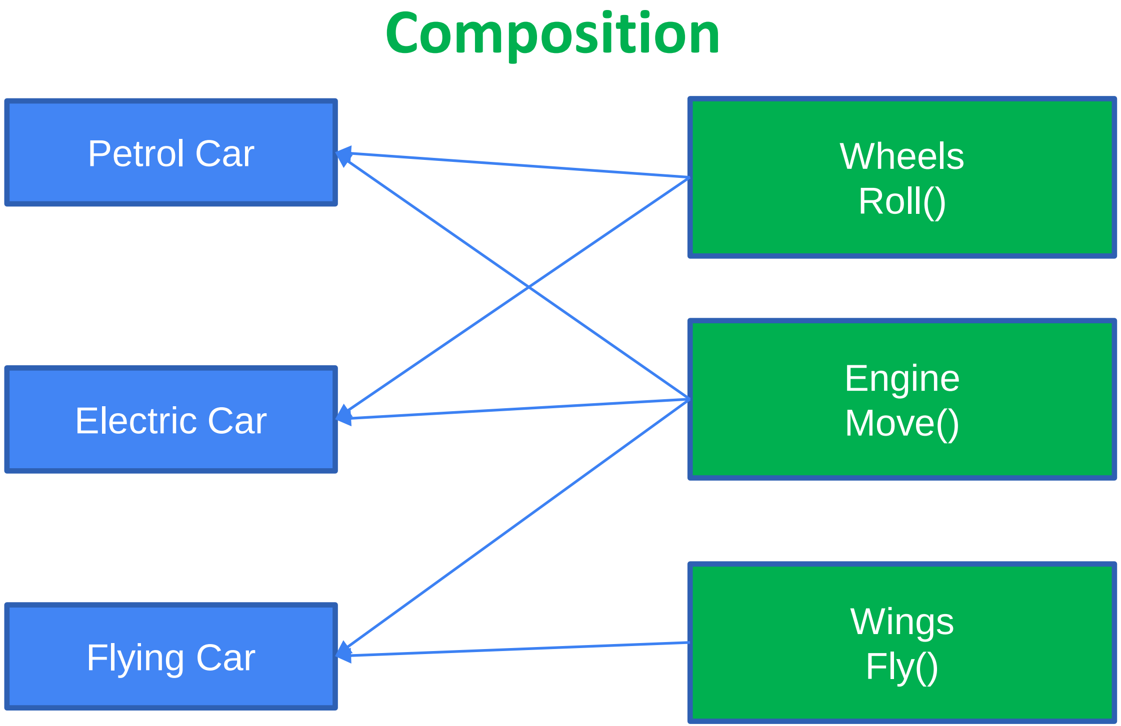
- **Composition** can be achieved in Go is by embedding one struct type into another.

Composition over inheritance

Inheritance



Composition over inheritance



Composition

```go
type Employee struct {
    Name string
    ID string
}
func (e Employee) Description() string {
    return fmt.Sprintf("%s (%s)", e.Name, e.ID)
}
type Manager struct {
    Employee
    Reports []Employee
}
func (m Manager) FindNewEmployees() []Employee {
    // do business logic
}
```

Composition

```go
m := Manager{
    Employee: Employee{
        Name: "Tom Tompson",
        ID: "112233",
    },
    Reports: []Employee{},
}
fmt.Println(m.ID) // prints 112233
fmt.Println(m.Description()) // prints Tom Tompson (112233)
```

Encapsulation

- **Encapsulation** is the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse.

- But Go language does not support classes and objects - in the Go encapsulation is achieved by using packages.

Encapsulation

```
type Person struct {
   Name string
   Age  int
   creditCard creditCard
}
```

Go provides two different types of identifiers

Exported – start with capital letter – Name, Age

Unexported – identifiers – creditCard

Encapsulation

```go
type creditCard struct {
    Bank   string
    Number string
    pin    string
    cvc    string
}

func NewCreditCard() creditCard {
    return creditCard{
        Bank:   "Big Bank",
        Number: "XXXX-XXXX-XXXX-XXXX",
        pin:    "0000",
        cvc:    "111",
    }
}
```

https://go.dev/play/p/qvzNkklNBCD

Encapsulation

```go
func NewPerson(name string, age int) Person {
    return Person{
        Name:       name,
        Age:        age,
        creditCard: NewCreditCard(),
    }
}
```

Polymorphism

- **Polymorphism** (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a several classes and structs and each of them will have own implementation.

- In Go, polymorphism is achieved by implementing interfaces.

Is Go OOP?

Let's ask GO developers

"Yes and no”

(Go Developers)

https://go.dev/doc/faq#Is_Go_an_object-oriented_language

Interfaces

Interfaces

- **Interface** describes the expected behavior to which type must satisfy.
- it's the only one abstract type that we have in Golang.

Interfaces

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

Method String() must be implemented in order to satisfy Stringer interface

Interfaces

```go
type Community struct {
    Name string
    Interest string
}

func main() {
    community := Community{
        Name: "Golang United",
        Interest: "Go",
    }
    fmt.Println(community) // {Golang United Go}
}
```

https://go.dev/play/p/Jmicu2vRchS

Interfaces

```go
type Community struct {
    Name string
    Interest string
}

func (c Community) String() string {
    return fmt.Sprintf(
        "Name: %s, Interest: %s",
        c.Name,
        c.Interest,
    )
}
func main() {
    community := Community{
        Name: "Golang United",
        Interest: "Go",
    }
    fmt.Println(community) // Name: Golang United, Interest: Go
}
```

https://go.dev/play/p/Jmicu2vRchS

Implicit interfaces and duck typing

- Real star of Go’s design it's **implicit interfaces.**
- Implicit interfaces called duck typing in Functional programing.

- **Duck Typing** means “If it walks like a duck and quacks like a duck, it’s a duck.”

Implicit interfaces and duck typing

```go
type Crew struct {
    Driver
}

type Driver interface {
    Drive()
}

type Truck struct{}

func (t Truck) Drive() {
    fmt.Println("I drive Truck")
}

type Tesla struct{}

func (t Tesla) Drive() {
    fmt.Println("I drive Tesla")
}
```

```go
type Bike struct{}

func (b Bike) Drive() {
    fmt.Println("I drive Bike")
}

func main() {
    driver1 := Truck{}
    driver2 := Tesla{}
    driver3 := Bike{}

    team1 := Crew{
        driver1,
    }
    team2 := Crew{
        driver2,
    }
    team3 := Crew{
        driver3,
    }

    team1.Drive()
    team2.Drive()
    team3.Drive()
}
```

https://go.dev/play/p/mE8g3IBLlKl

Pros Cons Implicit interfaces

- Pros
  - With Implicit interfaces helps to avoid rewriting your code to depend on a new interface.
  - You can create interfaces for yours structs later, when it will be required.
  - Ease of mocks implementation.

- Cons
  - Harder to realize which interfaces struct implements

Embedding and interfaces

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}

type ReadCloser interface {
    Reader
    Closer
}
```

Does type satisfy an interface?

```
type T struct{}
var _ I = T{}
var _ I = (*T)(nil)
```

Verify that T implements I.
Verify that *T implements I.

Does type satisfy an interface?

```go
type Hellower interface {
    Hello()
}
type Bayer interface {
    Bay()
}
type HelloBay struct{}

func (HelloBay) Hello() {
    fmt.Println("hello!")
}
func (HelloBay) Bay() {
    fmt.Println("bay!")
}
func main() {
    var _ Hellower = (*HelloBay)(nil)
    var _ Bayer = (*HelloBay)(nil)
}
```

https://go.dev/play/p/bkuoHfTDLQB

Interfaces and nil

```
var s *string
fmt.Println(s == nil)
var i interface{}
fmt.Println(i == nil)
i = s
fmt.Println(i == nil)
```

Use nil to represent the zero value for an interface instance

prints true
prints true
prints false

Interfaces and nil

```
var s *string                    //(type=*string,value=nil)
fmt.Println(s == nil)            //true
var i interface{}                              //(type=nil,value=nil)
fmt.Println(i == nil)            // true
i = s                                          //(type=*string,value=nil)
fmt.Println(i == nil)            // prints false
fmt.Println(i==(*string)(nil))// prints true
```

The Empty Interface Says Nothing

```go
var i interface{}
i = 20
i = "hello"
i = struct {
    Language string
    Type string
} {"Go", "Backend"}
```

Use cases for empty interfaces #1

Unmarshalling JSONs

```go
// one set of braces for the interface{} type,
// the other to instantiate an instance of the map
data := map[string]interface{}{}
contents, err := ioutil.ReadFile("testdata/sample.json")
   if err != nil {
   return err
}
defer contents.Close()
json.Unmarshal(contents, &data)
// the contents are now in the data map
```

Use cases for empty interfaces #2

Data structures

```go
type LinkedList struct {
    Value interface{}
    Next *LinkedList
}

func (ll *LinkedList) Insert(pos int, val interface{}) *LinkedList {
    if ll == nil || pos == 0 {
        return &LinkedList{
            Value: val,
            Next: ll,
        }
    }
    ll.Next = ll.Next.Insert(pos-1, val)
    return ll
}
```

Use cases for empty interfaces #3

```go
done := make(chan struct{})
go func() {
    doLongRunningThing()
    close(done)
}()
// do some other bits
// wait for that long running thing to finish
<-done
// do more things
```

Close channels

General advice for empty interfaces

- **Try to avoid using empty interface**. As we've seen, Go is designed as a strongly typed language and attempts to work around this are unidiomatic.

Type Assertion

```go
var n int
var i interface{}
n = 10
i = n
intValue, ok := i.(int)
if !ok {
    fmt.Println("could not read the value, incorrect type")
} else {
    fmt.Println(intValue)
}
```

https://go.dev/play/p/5G5r1xVueAv

Type Assertion

```
var n int
var i interface{}
n = 10
i = n
intValue := i.(string)//panic:interface conversion: interface {} is int, not string
fmt.Println(intValue) // not executed
```

https://go.dev/play/p/99L8NCacNec

Switch type assertion

```go
func main() {
   var n int
   var i interface{}
   n = 10
   i = n
   i = "test string"
   doCheck(i)
}

func doCheck(i interface{}) {
   switch j := i.(type) {
   case nil:
      fmt.Println("i has nil value")
   case int:
      fmt.Println("i is a int type, value is:", j)
    case string:
      fmt.Println("j is a string type, values is:", j)
    default:
      fmt.Println("doCheck desn't support this type")
   }
}
```

https://go.dev/play/p/9BqXEYJ_n9z

Use type assertion and Type use cases

- **check if one interface behind implements another interface. This allows you to specify optional interfaces**

- Errors use type assertion under the hood as well: `errors.Is` and `errors.As`(used in order to check the exact error) functions
- Context

Dependency Injection

- **dependency injection** is a design pattern in which an object receives other objects that it depends on.
- **Implicit interfaces make dependency injection easier**
- Go allows to implement dependency injection without any external library, framework.

Dependency Injection

```go
func LogOutput(message string) {
  fmt.Println(message)
}

type SimpleDataStore struct {
  userData map[string]string
}

func (sds SimpleDataStore) UserNameForID(userID string) (string, bool) {
  name, ok := sds.userData[userID]
  return name, ok
}

func NewSimpleDataStore() SimpleDataStore {
  return SimpleDataStore{
    userData: map[string]string{
      "1": "Fred",
      "2": "Mary",
      "3": "Pat",
    },
  }
}
```

Dependency Injection

```go
type DataStore interface {
  UserNameForID(userID string) (string, bool)
}

type Logger interface {
  Log(message string)
}

type LoggerAdapter func(message string)

func (lg LoggerAdapter) Log(message string) {
  lg(message)
}
```

Dependency Injection

```go
type SimpleLogic struct {
  l  Logger
  ds DataStore
}

func NewSimpleLogic(l Logger, ds DataStore) SimpleLogic {
  return SimpleLogic{
    l:   l,
    ds: ds,
  }
}

func (sl SimpleLogic) SayHello(userID string) (string, error) {
  sl.l.Log("in SayHello for " + userID)
  name, ok := sl.ds.UserNameForID(userID)
  if !ok {
    return "", errors.New("unknown user")
  }
  return "Hello, " + name, nil
}

func (sl SimpleLogic) SayGoodbye(userID string) (string, error) {
  sl.l.Log("in SayGoodbye for " + userID)
  name, ok := sl.ds.UserNameForID(userID)
  if !ok {
    return "", errors.New("unknown user")
  }
  return "Goodbye, " + name, nil
}
```

Dependency Injection

```go
type Logic interface {
  SayHello(userID string) (string, error)
}
type Controller struct {
  l     Logger
  logic Logic
}
func NewController(l Logger, logic Logic) Controller {
  return Controller{
    l:     l,
    logic: logic,
  }
}
func (c Controller) SayHello(w http.ResponseWriter, r *http.Request) {
  c.l.Log("In SayHello")
  userID := r.URL.Query().Get("user_id")
  message, err := c.logic.SayHello(userID)
  if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(err.Error()))
    return
  }
  w.Write([]byte(message))
}
```

Dependency Injection

```go
func main() {
  l := LoggerAdapter(LogOutput)
  ds := NewSimpleDataStore()
  logic := NewSimpleLogic(l, ds)
  c := NewController(l, logic)
  http.HandleFunc("/hello", c.SayHello)
  http.ListenAndServe(":8080", nil)
}
```

Tips for interfaces

- Program to an interface, not an implementation.
- If there’s an interface in the standard library that describes what your code needs, use it!
- Many client-specific interfaces are better than one general purpose interface. (Interface Segregation Principle)

Questions

Task

Task

- Will be provided boilerplate code with comments that describe expected behavior of methods.

  Prerequisite:
  - Execute commands
    - `go mod init`
    - `go mod tidy`

  Required to implement:
  - Calculate Area and Perimeter for shapes `Circle`, `Triangle`, `Rectangle` in *circle.go*, *triangle.go*, *rectangle.go* files.
  - Circle, Triangle, Rectangle must satisfy to `Shape` interface.
  - Methods for `Box` structure in *box.go* file.

  Final solution must satisfy to provided unit tests *in box_test.go* file.

Thanks