Go Part 2

No Classes

- Go is not your classic Object Oriented programming language.
- There are no classes in Go.

- Instead it has <u>interfaces & Structs</u>!

trainologic

Structs

- User defined type (**not a class**)
- Declared by composing a fixed set of fields together
- Introduces common OOP patterns & behaviours
    - Encapsulation / Reusability / Polymorphism / Overriding
- Fields may either protected or public following notation rules
- The Idiom is that you don’t instantiate but create value of type

Structs Composition

- Composed of primitive types & user defined types
- Can a struct be composed of other struct?
  - Yes! it is called promotion (structs are user defined types)
  - Promotion of structs allows us to imitate OOP inheritance
- Example:

```go
type carModel int

type Vehicle struct {
	model carModel
	color string
}
```

trainologic

Struct Promotion

- Allows us to share fields / behaviour between structs
- Allows us to override fields / methods (will discuss later)
- Example:

```go
type Person struct {
	name   string
	gender string
	age    int
}

type DoubleZero struct {
	promoted Person
}
```

trainologic

Struct Field Overriding

- We can override fields of promoted structs
- Example:

```
type Person struct {
	name   string
}

type DoubleZero struct {
	Person
	name		string
}

func main() {
	dz := DoubleZero{
		Person: Person{name:"Alice"},
		name:"Double Zero Eight",
	}
}
```

Interfaces

- An interface is a set of method signatures
- A way to achieve polymorphism / code substitutability
- No particular implementation is enforced - an interface is implemented implicitly
  - No Type implements Interface is needed
  - Implementation is done by “satisfying” the interface signature
- Example:

```
type Vehicle interface {
  NumberOfWheels() int
  HasMotor() bool
}
```

trainologic

Interfaces

- So, an interface provides no implementation.
- A type can implement the interface implicitly.
- For example:

```
type Bicycle struct {
  model string
}


func (b Bicycle) NumberOfWheels() int {
  return 2
}

func (b Bicycle) HasMotor() bool {
  return false
}
```

trainologic

Interfaces

- There is no classic Object-Oriented virtual table or inheritance in go.
- Instead, if there is an implementation for the interface methods for the type, the implementation will be invoked:

```
func ShowVehicle(v Vehicle) {
  println(v.NumberOfWheels())
}

func main()  {

  bi := Bicycle{"BMX"}
  ShowVehicle(bi)
```

Dynamic Types

- In Go, each variable may be a static type and a dynamic type.
- The static type is the type stated at the declaration site.
- For interface types, the dynamic type is the actual type stored in the variable at runtime.

Dynamic Types

- Example:

```go
var xx int = 9
var yy interface{} = 9

// doesn't compile
xx = "hello"
// compiles
yy = "hello"
```

Methods

- Methods are just functions with receivers
- Method receiver should be either value or pointer
- It is considered best practice to have the receiver as a **pointer** even if the method does not modify anything
    - Can you think of a reason why?
- Example:

```go
func (p *Programmer) speak() string  {
	return "42 is The Answer to the Ultimate Question of Life"
}
```

Methods

- Struct example:

```
type Animal struct {
	color string
	family string
}

func (a *Animal) makeNoise() string  {
	return "Rawr!!!!!"
}

func main() {
	a := Animal{color: "Brown", family: "Feline"}

	println(a.speak())
}
```

Methods

- Any type example:

```
type Int int

func (i Int) Add(j Int) Int {
        return i + j
}

func main()  {
        i := Int(20)
        j := Int(20)

        println(i.Add(j) + 2)
}
```

Method Overriding

- Similarly to value overriding, the promoter struct overrides methods of promoted struct
- Lets see an example..

Method Overriding

```go
type Person struct {
	name string
}

func (p *Person) speak() string  {
	return "Hello!"
}

type Programmer struct {
	Person
}

func (p *Programmer) speak() string {
	return "42 is The Answer to the Ultimate Question of Life"
}

func main() {
	p := Programmer{Person{42}}
	println(p.speak())
}
```

trainologic

Stringer

- One of the popular interfaces in Go is *Stringer*.
- Defined in the *fmt* package.
- Comes with a single method:
  - String() string.
- Types implementing this interface can be easily converted into strings.
- Think, should there be a default implementation for this interface (like in Java)?

trainologic

Stringer

- We also see here a convention in Go.
- An interface having a single method is named like the method with the addition or the suffix “-er”.

- For example:
  - Read**er**.

Root

- There is no “root of hierarchy” because there is no hierarchy in Go.
- No classes.
- An empty interface is used when you want to accept any value.

Type Checks

- You can check (type assertion) that an interface is holding a specific type by:
- cv := inter.(T)
- Here *cv* will be of type T.
- If it fails, we get *panic* state.
- You can also use:
- cv,ok := inter.(T)

Type Switches

- You can also use switch-cases on types:

```
var x interface{}
switch v := x.(type) {
case string:
	// v is type string
case int:
case Person:
}
```

Error Handling

Error Handling

- For error-handling, Go provides two keywords: *panic* and *recover*.

trainologic

Panic

- When *panic* is called, normal execution stops and the function returns to the caller.
- Of-course, *defer* statements are still executed.
- At the caller site, the function that returned is behaving like a direct invocation of *panic*.
- So, it will continue until the stack of the goroutine rolls all the way back and the program crashes.

- Unless *recover* is invoked!

trainologic

Recover

- recover is a built-in function that regains control of a panicking goroutine*.*
- Is only usable in *defer* statement.
- Returns the value provided to the *panic* function or *nil* if not panicking.

Concurrency

Goroutines

- A goroutine is a function that is executed in a concurrent fashion.
- Created with the ***go*** keyword.
- go f(a,b,c) will invoke the *f* function in a new goroutine.
- Note that goroutine is a lightweight thread and not a physical thread.

Goroutines

- All goroutines run in the same address space and have access to shared values.
- This can lead to nasty race-conditions.
- Go provides **channels** to help with that.

Channels

- A channel allows for passing values between goroutines.
- Created with the *make* function.
- Use the arrows operators to read/write from/to a channel.
- Reading a value: v := <- ch.
- Writing a value: ch <- v.
- By default, the goroutine will block on the channel.
- Note that channels are thread-safe.

trainologic

Channels

- The second parameter to *make* on a channel is the buffer size.
- Reads will block only when the buffer is empty.
- Writes will block only when the buffer is full.

Channels

- The sender can *close* a channel.
- This is not mandatory like with other resources.
- The receiver can get a second boolean parameter stating if the channel was closed.
- Using *for range* with a channel will repeatedly receive from the channel until it is closed.

select

- The ***select*** keyword has syntax similar to *switch*.
- However the cases should receive from channels.
- Whenever one of the cases is ready, it will execute.
- If more than one is ready, one will be executed in random.
- The *default* case, if exists, will be executed if nothing is ready (if blocking would occur).

Mutexes

- The *sync.Mutex* type provides critical-section mutex for handling race-conditions.
- Provides *Lock()* and *Unlock()* methods.