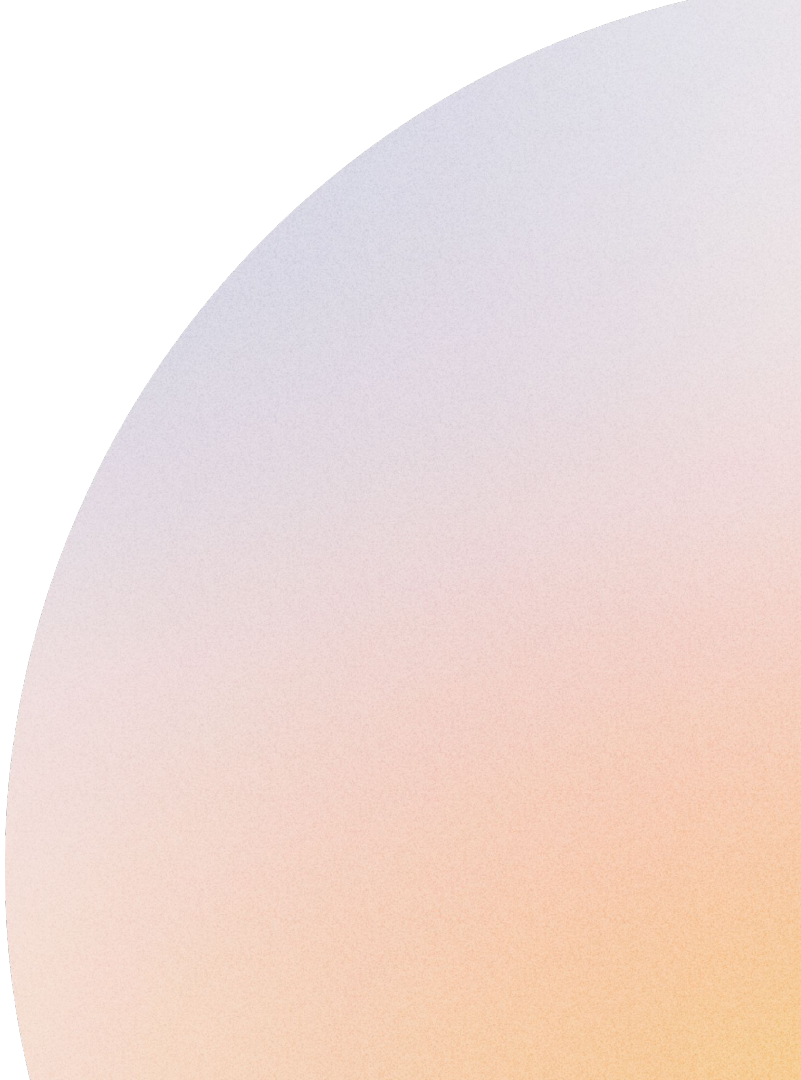
Methods and Interfaces in Go

Ambush Journey Program

1. Methods
2. Interfaces
3. Type Assertions
4. Factories



Methods

Methods

- Go does not have classes
- But it does allow for defining functions that work on top of a specific type or struct
    - We call these functions **methods**

Methods

```
 8  type Vertex struct {
 9      X, Y float64
10  }
11
12  func abs(v Vertex) float64 {
13      return math.Sqrt(v.X*v.X + v.Y*v.Y)
14  }
15
16  func main() {
17      v := Vertex{3, 4}
18      fmt.Println(abs(v))
19  }
```

Normal function declaration

```
 8  type Vertex struct {
 9      X, Y float64
10  }
11
12  func (v Vertex) Abs() float64 {
13      return math.Sqrt(v.X*v.X + v.Y*v.Y)
14  }
15
16  func main() {
17      v := Vertex{3, 4}
18      fmt.Println(v.Abs())
19  }
```

Method declaration

Methods

- It's possible to declare methods for non-struct types as well
  - In this case, a new type needs to be made (alias)

```
 8  type MyFloat float64
 9
10  func (f MyFloat) Abs() float64 {
11      if f < 0 {
12          return float64(-f)
13      }
14      return float64(f)
15  }
16
17  func main() {
18      f := MyFloat(-math.Sqrt2)
19      fmt.Println(f.Abs())
20  }
```

Methods

- The argument over which the method is declared is called **receiver**

Methods

- The argument over which the method is declared is called **receiver**
- It's also possible to have the receiver be a pointer
  - We call this receiver a **pointer receiver**

Methods

- The argument over which the method is declared is called **receiver**
- It's also possible to have the receiver be a pointer
  - We call this receiver a **pointer receiver**

```
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
```

Methods

- The argument over which the method is declared is called **receiver**
- It's also possible to have the receiver be a pointer
  - We call this receiver a **pointer receiver**
  - What's the advantage of this?

Practice

1. Modify the banking application made in the data types and pointers presentation to use methods for deposits and withdrawals

Interfaces

Interfaces

- Interfaces let us define a set of **method signatures**, and it can hold any type that **implements** these methods
- It can define multiple methods, but unlike some programming languages, it *can't* define properties

Interfaces

```
7   type Polygon interface {
8       Area() int
9   }
10
11  type Square struct {
12      l int
13  }
14
15  type Rectangle struct {
16      h int
17      w int
18  }
19
20  func (s *Square) Area() int {
21      return s.l * s.l
22  }
23
24  func (r *Rectangle) Area() int {
25      return r.h * r.w
26  }
```

```
38  func main() {
39      var p Polygon
40      p = &Square{5}
41      fmt.Println(p.Area())
42      p = &Rectangle{3, 4}
43      fmt.Println(p.Area())
44  }
```

Both `*Rectangle` and `*Square` structs implement interface `Polygon`

Note: pointer syntax is important!

Interfaces

```
7   type Polygon interface {
8       Area() int
9   }
10
11  type Square struct {
12      l int
13  }
14
15  type Rectangle struct {
16      h int
17      w int
18  }
19
20  func (s *Square) Area() int {
21      return s.l * s.l
22  }
23
24  func (r *Rectangle) Area() int {
25      return r.h * r.w
26  }
```

Pointer syntax is important! `Rectangle` and `Square` do not implement `Polygon` (method is defined on top of pointers)

```
func main() {
    var p Polygon
    p = Square{5}
    fmt.Println(p.Area())
    p = Rectangle{3, 4}
    fmt.Println(p.Area())
}
```

Interfaces

- Interfaces can define multiple methods
  - Remember that types are important as well

```
type Polygon interface {
    Area() int
    Perimeter() int
}
```

Interfaces

- Interfaces can define multiple methods
  - Remember that types are important as well

```
type RightTriangle struct {
    c1 int
    c2 int
    h  int
}

func (t *RightTriangle) Area() float64 {
    return float64(t.c1*t.c2) / 2
}
```

Interfaces

- Interfaces can define multiple methods
- Implementation is made implicitly (no `implements` keyword)

Interfaces

- Interfaces can define multiple methods
- Implementation is made implicitly (no `implements` keyword)
- It's possible to define an **empty interface**, which does not define any methods to be implemented
    - Can be used to receive values of any type

```
var i interface{}
i = 5
fmt.Println(i)
i = "foo"
fmt.Println(i)
i = &Square{2}
fmt.Println(i)
```

Stringer

- A common interface is the **Stringer** interface defined by the `fmt` package, which defines elements that can be printed by its **print** functions
  - Similar to `toString()` in Java

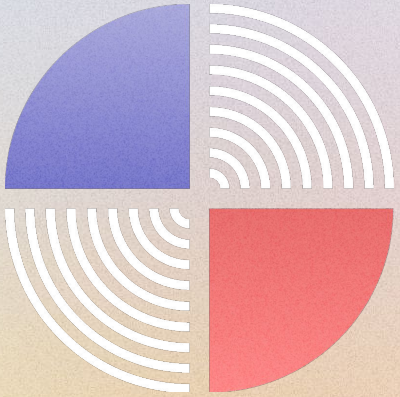
```
type Stringer interface {
    String() string
}
```

Stringer

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

func (p Person) String() string {
    return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
    a := Person{"Arthur Dent", 42}
    z := Person{"Zaphod Beeblebrox", 9001}
    fmt.Println(a, z)
}
```



Type Assertions

Type Assertions

- We can use type assertions to guarantee that a specific variable implementing an **interface** contains a value of the type in question
  - If performing an assignment and the assertion proceeds, the value is stored in the assigned variable

```
var i interface{} = "hello"

s := i.(string)
fmt.Println(s)
```

Type Assertions

- In order to **test** for a type before using it, we can use pattern matching

```
var i interface{} = "hello"

s := i.(string)
fmt.Println(s)

s, ok := i.(string)
fmt.Println(s, ok)

f, ok := i.(float64)
fmt.Println(f, ok)
```

Note: if the assertion fails, a *zero value* will be stored in the assigned variable

Type Assertions

- In order to **test** for a type before using it, we can use pattern matching
  - If we don't use pattern matching and the assertion fails, a runtime error happens

```
var i interface{} = "hello"

f := i.(float64) // panic
fmt.Println(f)
```

Type Switches

- It's possible to use a switch block to make multiple assertions
  - This special switch block is called a **type switch**

Type Switches

```
5   func do(i interface{}) {
6       switch v := i.(type) {
7       case int:
8           fmt.Printf("Twice %v is %v\n", v, v*2)
9       case string:
10          fmt.Printf("%q is %v bytes long\n", v, len(v))
11      default:
12          fmt.Printf("I don't know about type %T!\n", v)
13      }
14  }
15
16  func main() {
17      do(21)
18      do("hello")
19      do(true)
20  }
```

Practice

2. Create an interface for `User` containing the withdraw and deposit methods you created before
3. Create a special type of user called `LoanerUser`, whose balance can be negative up to $ 500 negative credits

Factories

Factories

- Given an interface `I` and structs `A, B` that implement this interface, a factory function is a function that conditionally returns an instance of the structs `A` and `B` (which implement the interface `I`)
  - It's a way of simplifying the creation of instances from implementations of an interface

Factories

```go
type Polygon interface {
    Area() int
}

type Square struct {
    l int
}

type Rectangle struct {
    h int
    w int
}

func (s Square) Area() int {
    return s.l * s.l
}

func (r Rectangle) Area() int {
    return r.h * r.w
}
```

```go
func makePolygon(p string) Polygon {
    switch p {
    case "square":
        return Square{4}
    case "rectangle":
        return Rectangle{3, 4}
    default:
        return nil
    }
}

func main() {
    rect := makePolygon("rectangle")
    sq := makePolygon("square")
    fmt.Println(sq.Area())
    fmt.Println(rect.Area())
}
```

Methods and Interfaces in Go

Ambush Journey Program