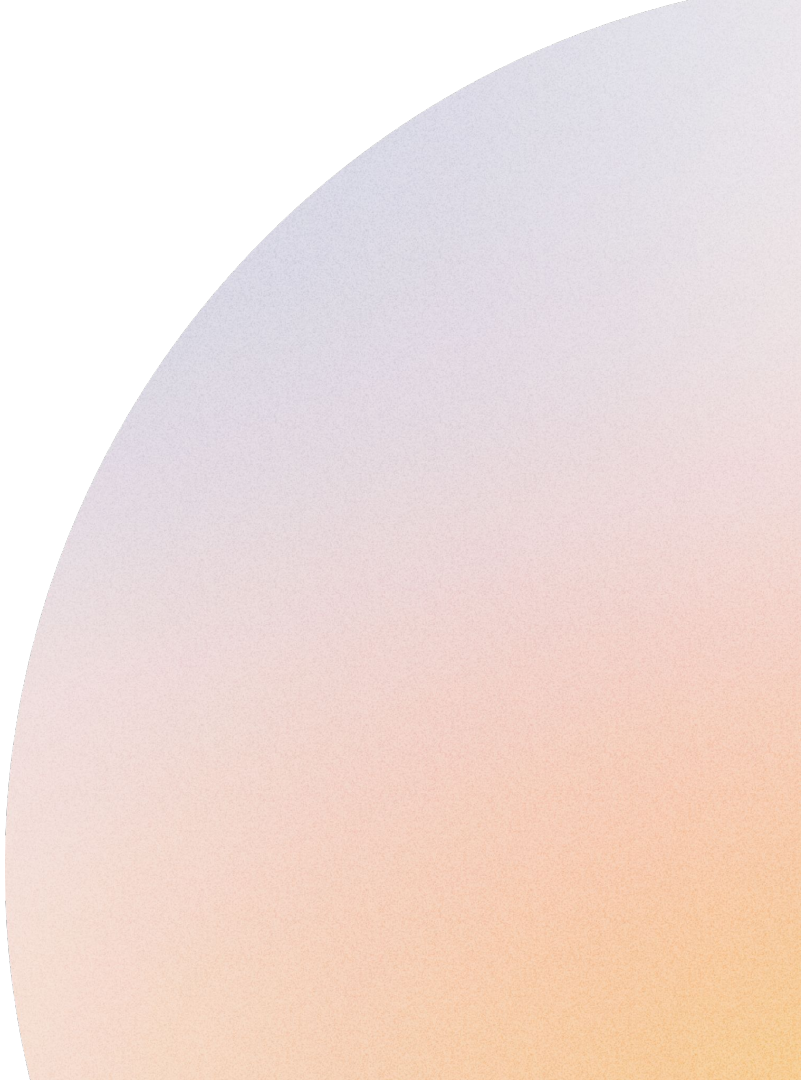
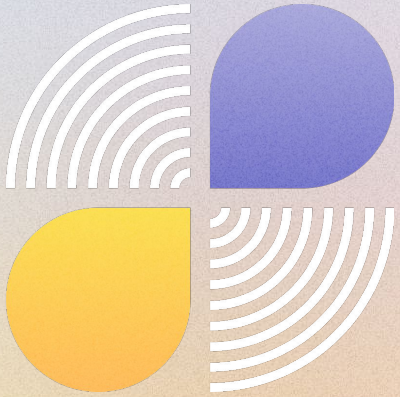
Pattern Matching and Error Handling

Ambush Journey Program

1. Pattern Matching
2. Errors
3. Defer, Panic and Recover





Pattern Matching

Pattern Matching

One of Go's most powerful features is called **pattern matching**

- It's a tool used in assignments, in such manner that Go will try to match the elements on the **right** side of the assignment to the ones on the **left**
- Let's analyse this in an example

Pattern Matching

```go
func main() {
    x, y := 1, 2
    fmt.Println(x, y)
}
```

`1` and `2` are matched to `x` and `y`

```go
func foo() (int, int) {
    return 1, 2
}

func main() {
    x, y := foo()
    fmt.Println(x, y)
}
```

The return of function `foo` (two integers) is matched to `x` and `y`

```go
func foo() (string, int) {
    return "foo", 2
}

func main() {
    x, y := foo()
    fmt.Println(x, y)
}
```

The return of function `foo` (one integer and one string) is matched to `x` and `y`

Pattern Matching

- If the amount of values on one side doesn't match the amount of values on the other side, an error will happen

```go
func foo() string {
    return "foo"
}

func main() {
    x, y := foo()
    fmt.Println(x, y)
}
```

Mismatch: 2 values on the left hand side, 1 value on the right hand side

```go
func foo() (string, int) {
    return "foo", 2
}

func main() {
    x := foo()
    fmt.Println(x)
}
```

Mismatch: 2 values on the right hand side, 1 value on the left hand side

Pattern Matching

- Pattern matching is very useful when dealing with errors, specially for testing if a specific action has failed or not

```
func main() {
    src, err := os.Open("products.txt")
    if err != nil {
        return
    }

    fmt.Println(src)
}
```

Errors

The Error interface

- Go has a special type of interface named `error`, which is similar to the `Stringer` interface
- As shown in the previous example, many functions in Go return an error as a second parameter, which can be checked when looking for errors

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

The Error interface

- New errors can be created by defining structs that contain the `Error()` method

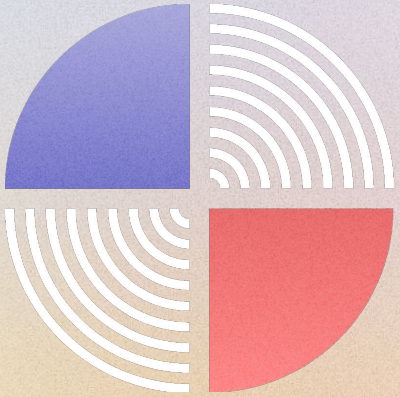
```
type ErrNegativeSqrt float64

func (e ErrNegativeSqrt) Error() string {
    return fmt.Sprintf("Cannot Sqrt negative number: %f", float64(e))
}
```

The Error interface

```
23  func Sqrt(x float64) (float64, error) {
24      if x > 0 {
25          z := x
26          for Abs(x-z*z) > 0.001 {
27              z -= (z*z - x) / (2 * z)
28          }
29          return z, nil
30      } else {
31          return 0, ErrNegativeSqrt(x)
32      }
33  }
34
35  func main() {
36      number := -2.0
37      sqrt, err := Sqrt(number)
38      if err != nil {
39          fmt.Println(err)
40      } else {
41          fmt.Printf("Square root of %.2f is %.6f\n", number, sqrt)
42      }
43  }
```

Practice

1. Based on the previous Sqrt function, create a function that takes two numbers as parameters and divides the first by the second, and if the second one is a zero, it returns a custom error, with a defined Error() implementation



Defer, Panic and Recover

Defer

- On the class about **control flows** we learned about defer

```go
func main() {
    fmt.Println("countdown")

    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}
```

```
countdown
done
9
8
7
6
5
4
3
2
1
0
```

Defer

- On the class about **control flows** we learned about defer
- Deferring a function is specially useful when dealing with files (we are going to do that later)
- Besides defer, there are two extra control flow mechanisms that are useful, mainly for error handling: **panic** and **recover**

Panic

- When the built-in **panic** function is called, Go will suspend the current execution flow of the program and initiate a panicking routine
- Any deferred functions will be executed in the last-in-first-out order
- Panicking routine will stop when a call to **recover** is found

Recover

- The **recover** function can be used to “catch” panicking routines, and resume normal program flow
- After a panicking routine is recovered, the flow continues from the parent function (not the one that called the deferring function, but the parent of that one)

Panic and Recover

```go
func main() {
    fmt.Println("Calling f.")
    f()
    fmt.Println("Returned normally from f.")
}

func handlePanic() {
    if r := recover(); r != nil {
        fmt.Println("Recovered from panic:", r)
    }
}

func f() {
    defer handlePanic()
    i := 5
    if i > 3 {
        panic("Panicking")
    }
    fmt.Println("f finished execution")
}
```

- If `i` is greater than 3, a panicking routine starts and is caught by the deferred `handlePanic` function
- If `i` is smaller than 3, no panic happens and the program flow is executed normally
    - Notice the difference on the printed data when changing the value of `i`

Panic and Recover

```go
func main() {
    fmt.Println("Calling f.")
    f()
    fmt.Println("Returned normally from f.")
}

func handlePanic() {
    if r := recover(); r != nil {
        fmt.Println("Recovered from panic:", r)
    }
}

func f() {
    defer handlePanic()
    i := 5
    if i > 3 {
        panic("Panicking")
    }
    fmt.Println("f finished execution")
}
```

```
$ go run main.go
Calling f.
f finished execution
Returned normally from f.

murilo@DESKTOP-A2CF033 MINGW64 ~
$ go run main.go
Calling f.
Recovered from panic: Panicking
Returned normally from f.
```

- If `i` is greater than 3, a panicking routine starts and is caught by the deferred `handlePanic` function
- If `i` is smaller than 3, no panic happens and the program flow is executed normally
    - Notice the difference on the printed data when changing the value of `i`

Panic and Recover

```go
func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func handlePanic(message string) {
    if r := recover(); r != nil {
        fmt.Println(message, r)
    }
}

func f() {
    defer handlePanic("Recovered in f")
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}
```

```go
func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
        fmt.Println("This won't print")
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

```
$ go run main.go
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

Panic and Recover

- The **panic** and **recover** mechanisms, along with **defer**, can be used for a role similar to that of try/catch blocks in most programming languages (since Go doesn't have a try/catch block)
- It's also useful when working with concurrency and goroutines
    - We will learn more about these later

Practice

1. Take the first example with function `f()` in slide 19 and try playing with different values of `i`. See what happens when you run the program.
2. Take the second example in slide 20, with `g()` and `f()`, and try deferring `handlePanic` from different places (line 6 in `main`; line 24 in `g`), and see what happens. You can also try executing `g()` with different initial values.

Pattern Matching and Error Handling

Ambush Journey Program