Go Basics

Introduction

- Whoami
- Training style/structure
- Breaks & lunch
- Slides will be available

Whoami

- Eduvision
- brainhive
- Training style
- Questions

Training overview

1. **Go basics**

   Syntax, data structures, interfaces, ...

2. **Go basics**

   Best practices, profiling, docker, rest APIs, ...

3. **Microservices**

   Monoliths, containers, Kubernetes, packaging, ...

4. **Microservices**

   CI/CD, skaffold, logging, monitoring, troubleshooting, ...

5. **Workshop**

   Building microservices with Go

Part 1

1. History
2. Syntax
3. Basic Types
4. Control Flow
5. Exercise 1

Part 2

1. Data Structures
2. Exercise 2
3. Pointers
4. Error Handling
5. Code Organization
6. Testing
7. Homework

History

- Development started in 2007
- Announced at 2009
- First release in 2012

Why?

- Frustration with existing languages/tools at Google
- Alternative to C/C++/Python/Java
- No new system language in a decade
- First of "next-gen" languages (Rust, Elixir, Swift)

Goals

- Safety: type-safe and memory-safe
- Good support for concurrency and communication
- Efficient, latency-free garbage collection
- High-speed compilation

Syntax

- Clean
- Simple
- Readable

Entrypoint

In short:

- `main.main` for executables

Example

```
package main

func main() {
    // <-- Code goes here
}
```

Package declaration

In short:

- Convention: `package name = directory name`
- Multiple packages per directory is not possible

Example

```
package main
//      ^^^^
//      Package name
```

Imports

```
import "fmt"
//     ^^^^^
//     Import `fmt` package from the standard library
```

```
import (
    "os"
    "fmt"
//  ^^^^^
//  Standard library

    "github.com/tmthrgd/go-bitset"
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//  Remote repository (GitHub, GitLab, etc.)
)
```

Function declaration

```
func test(val1 int, val2 int) {
//        ^^^^ ^^^
//        Name Type
}
```

```
func test(val1, val2 int) {
//        ^^^^       ^^^
//        Name       Type for `val` and `val2`
}
```

Return types

```
func test(val1 int) bool {
//                  ^^^^
//                  Return type
    return true
}
```

```
func test(val1 int) (int, bool) {
//                  ^^^^  ^^^^
//                  Return type (tuple)
    return 10, true
}
```

Named return types

```
func test(val1 int) (result bool) {
//                   ^^^^
//                   Name

    result = true
//  ^^^^^^^^^^^^^^
//  Assign as if it was a variable

    return
//  ^^^^^^
//  Implicit return
}
```

Variable declaration - 1/3

```
var val1 int
//  ^^^^ ^^^
//  Name Type

func main() {
    fmt.Println(val1)
}
```

```
var (
    val2 int
    val3 bool
)
// Group variable declarations
```

Variable declaration - 2/3

```
var val1 int = 100
//  ^^^^ ^^^    ^^^
//  Name Type   Value
```

```
var val1 = 100
//  ^^^^    ^^^
//  Name    Value (type inferred)
```

```
var x, y, z = 100, 50, 25
//  ^^^^^^^
//  Multiple assignment
```

Variable declaration - 3/3

Short variable declaration

```
val1 := 100
//      ^^^
//      Value (type inferred)
```

Constants - 1/2

```
const TIMEOUT = 30
//    ^^^^^^^   ^^
//    Name      Value

const (
    HTTP_TIMEOUT = 30
    GRPC_TIMEOUT = 60
)
// Multiple constants
```

Constants - 2/2

```
type Direction int

const (
    DirectionLeft  Direction = iota // 0
    DirectionRight                  // 1
    DirectionUp                     // 2
    DirectionDown                   // 3
    DirectionExit                   // 4
)
```

Basic Types

Nil

Summary:

- Not an actual type
- Represents `null` or `undefined`
- Name: `nil`

Example

```
var err error

if err != nil {
    // Do something
}
```

Boolean

Summary:

- Type name: `bool`
- Either `true` / `false`
- Default value: `false`

Integers - 1/2

Summary:

- Type names: `int` / `uint`
- Sizes: `8` / `16` / `32` / `64`
- Target specific: `int`, `uint` (32/64-bit)
- Use `int` unless hard requirements
- Default value: `0`

Integers - 2/2

Summary:

- Which size to use for integer literals?
- Languages deal with this differently
- Small integers can overflow

Example

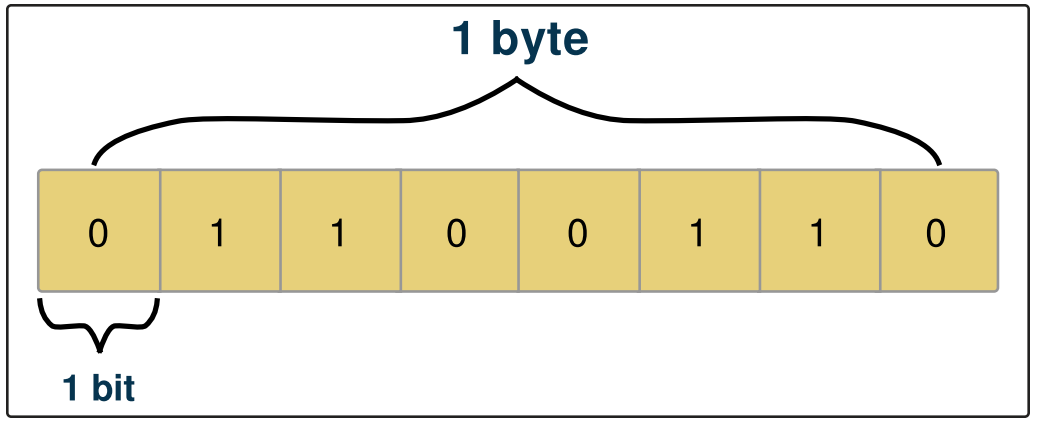
```
var x int8 = 100

x = x * 2

// Output: -56
```

Floats

- Type names:
  - `float32`
  - `float64`
- Example: `1.145`
- Default value: `0`

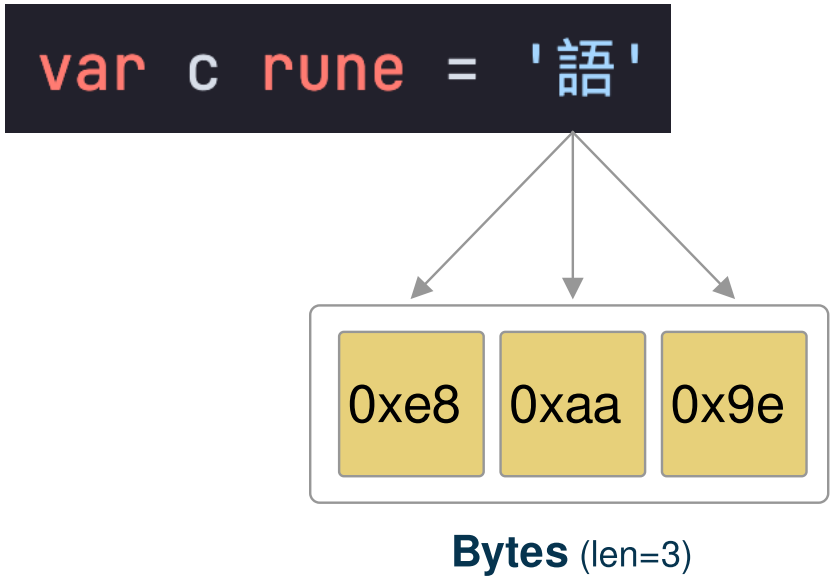
Byte

- Type name: `byte`
- Example: `0x66`
- Alias of: `uint8`
- Default value: `0`



Rune - 1/2

- Type name: `rune`
- Example: `'語'`
- Alias of: `int32`
- Unicode code point
- Default value: `0`



Rune - 2/2

Why `int32` and not `uint32`?

- Again, integers can overflow
- Enough space for all unicode code points
- Similar to array indices

String

- Type name: `string`
- Not nullable
- Read-only slice of bytes (e.g. `[]byte`)
- No utf8 requirement
- Default value: `""`

Type conversions - 1/2

In short:

- No implicit type conversion
- Syntax: `newType(value)`

Implicit type conversion in Javascript

```
var value = "10";
var output = value + 20;

console.log(output);
// Output: "1020"
```

Type conversions - 2/2

```
var x int = 10
var y float32 = 1.5

z := x * y
// Error: invalid operation: x * y (mismatched types int and float32)
```

```
var x int = 10
var y float32 = 1.5

z := float32(x) * y
//   ^^^^^^^
//   Convert `int` to `float32`
```

Control Flow

For loop

Example

```
for i := 0; i < 10; i++ {
//  ^^^^^^  ^^^^^^  ^^^
//  Init    Cond    Post

    println(i)
}
```

Breakdown

- Start with `i := 0`
- Check if `i < 10` is `true` befor each iteration
- Increment `i` by `1` after each iteration

While loop

```
for ; i < 10;  {
}
```

```
for i < 10 {
//  ^^^^^^
//  Only a condition
}
```

Forever

```
for {
}
```

If statement - 1/2

```
if i < 10 {
// ^^^^^^
// Condition without parentheses
}
```

With variable declaration

```
if i := rand.Intn(100); i < 10 {
// ^^^^^^^^^^^^^^^^^^^  ^^^^^^
// Declaration          Condition
}
```

If statement - 2/2

```
i := 100
// i = 100

if i := rand.Intn(100); i < 50 {
// ^
// Shadows the outer `i`
//
// i = random number
}

// i = 100
println(i)
```

Switch - 1/2

- Not fallthrough, so `break` is optional
- Evaluated from top to bottom
- No `default` case required

Example

```
switch flag.Arg(0) {
    case "help":
        fmt.Println("Help!")
    default:
        fmt.Println("Unknown command")
}
```

Switch - 2/2

```
switch flag.Arg(0) {
    case "remove", "rm":
        fmt.Println("Remove!")
    case "list", "ls":
        fmt.Println("List!")
    case "help", "h":
        fmt.Println("Help!")
        fallthrough
//      ^^^^^^^^^^^
//      Explicitly enable fallthrough
    default:
        fmt.Println("Show help!")
}
```

Exercise 1

Prerequisites

- Install Go
- Setup your IDE
  - Visual Studio Code
  - GoLand
  - Vim/Neovim

Exercise 1

Summary

- Iterate over each character using a for loop
- Check and count vowels
- Print the total number of vowels

Go to

training.brainhive.nl

Review

Recap

- What did we discuss?
- How are we doing?
- What's next?
    - Data Structures
    - Pointers
    - Error Handling
    - Code Organization
    - Testing

Data Structures

Structs - 1/6

Summary:

- Collection of fields
- Composition > inheritance

Example

```
type Member struct {
    Id int
    Username, Email string
}
```

Structs - 2/6

Fields

```
type Member struct {
    Id int
//  ^^
//  Exported

    name string
//  ^^^^
//  Not exported
}
```

Structs - 3/6

Inheritance (in Java)

```
class Animal {
    String name;
}

class Dog extends Animal {
    public void display() {
        System.out.println(name);
    }
}

Dog poodle = new Dog();
poodle.name = "Cheerio";
poodle.display();
```

Structs - 4/6

Composition (in Go)

```
type Animal struct {
    Name string
}

type Dog struct {
    Animal
}

func (d Dog) Display() {
    fmt.Println(d.Name)
}

dog := Dog{Animal: Animal{Name: "Cheerio"}}
dog.Display()
```

Structs - 5/6

Constructor

```
type Member struct {
    Id int
    Username, Email string
}

func NewMember(id int, username, email string) Member {
    return Member{
        Id: id,
        Username: username,
        Email: email,
    }
}

member := NewMember(1, "john", "john@test.com")
```

Structs - 6/6

Methods

```
type Member struct {
    Id int
    Username, Email string
}

func (m Member) Signature() string {
//    ^^^^^^^^  ^^^^^^^^^
//     Receiver  Method
    return m.Username + " <" + m.Email + ">"
}

signature := member.Signature()

fmt.Println(signature) // Output: "john <john@test.com>"
```

Arrays

Summary:

- Fixed length, length is part of it's type
- Arrays are values, stack allocated

Example

```
var values [5]int
//         ^^^^^^
//         Length and type

values := [...]int{1, 2, 3}
//         ^^^
//         Infer length
```

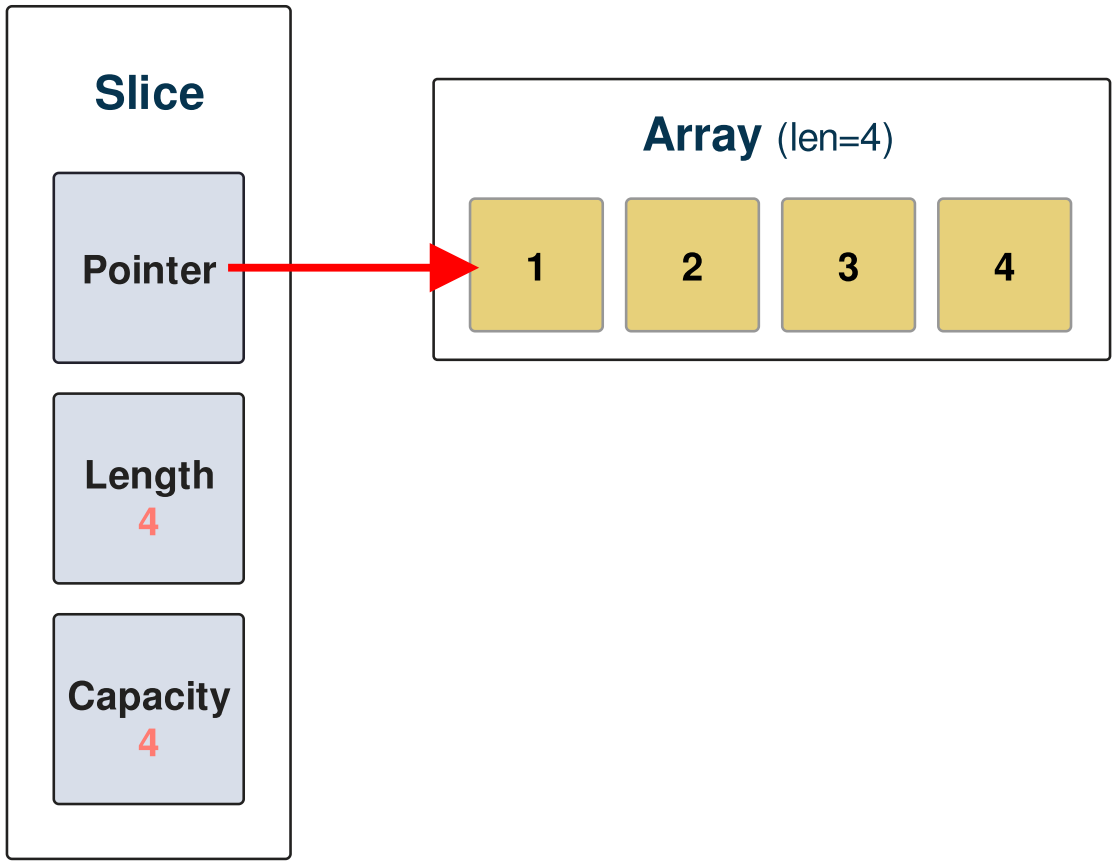
Slices - 1/8

Summary:

- Dynamically sized
- Points to an array
- Can grow or shrink
- Is nullable, defaults to: `nil`

Example

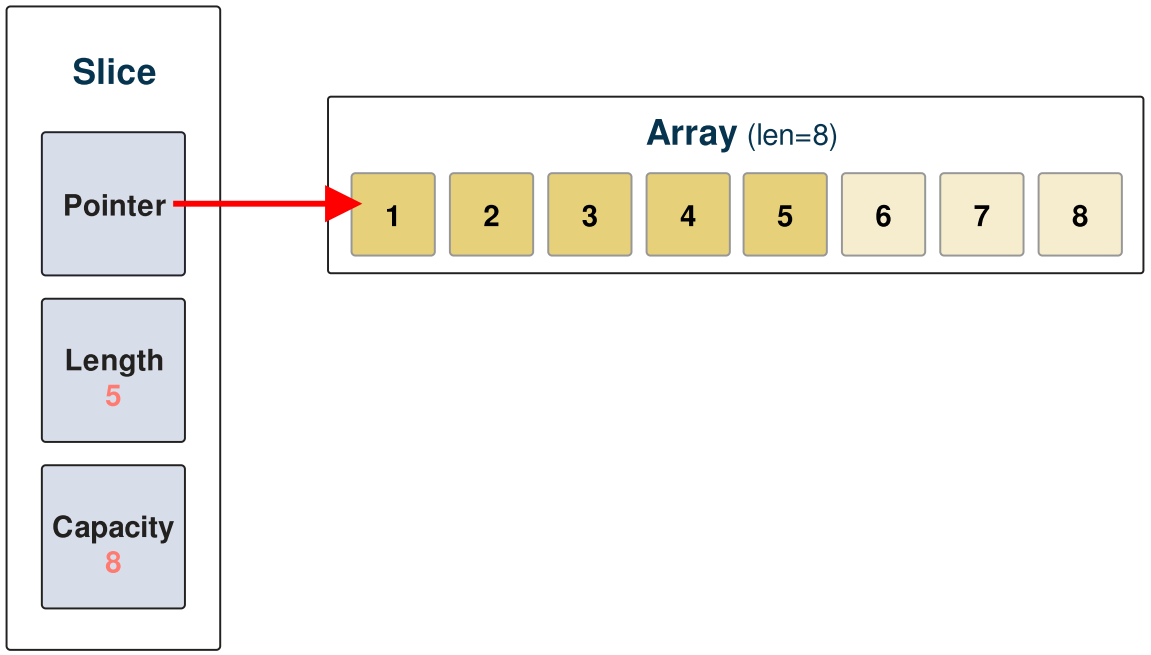
```
slice := []int{1, 2, 3, 4}
```



Slices - 2/8

Summary:

- Use `append` to add elements to a slice
- Underlying array will be copied
- Length `!=` capacity



Example

```
slice := []int{1, 2, 3, 4}
slice = append(slice, 5)
//      ^^^^^^^^^^^^^^^^
//      Creates a new slice
```

Slices - 3/8

Create slices

```
array := [...]int{1, 2, 3}
slice := array[:]
//       ^^^^^^^^
//       Create slice from array


sub := slice[1:2]
// Output: [2]

sub2 := slice[1:]
// Output: [2, 3]
```

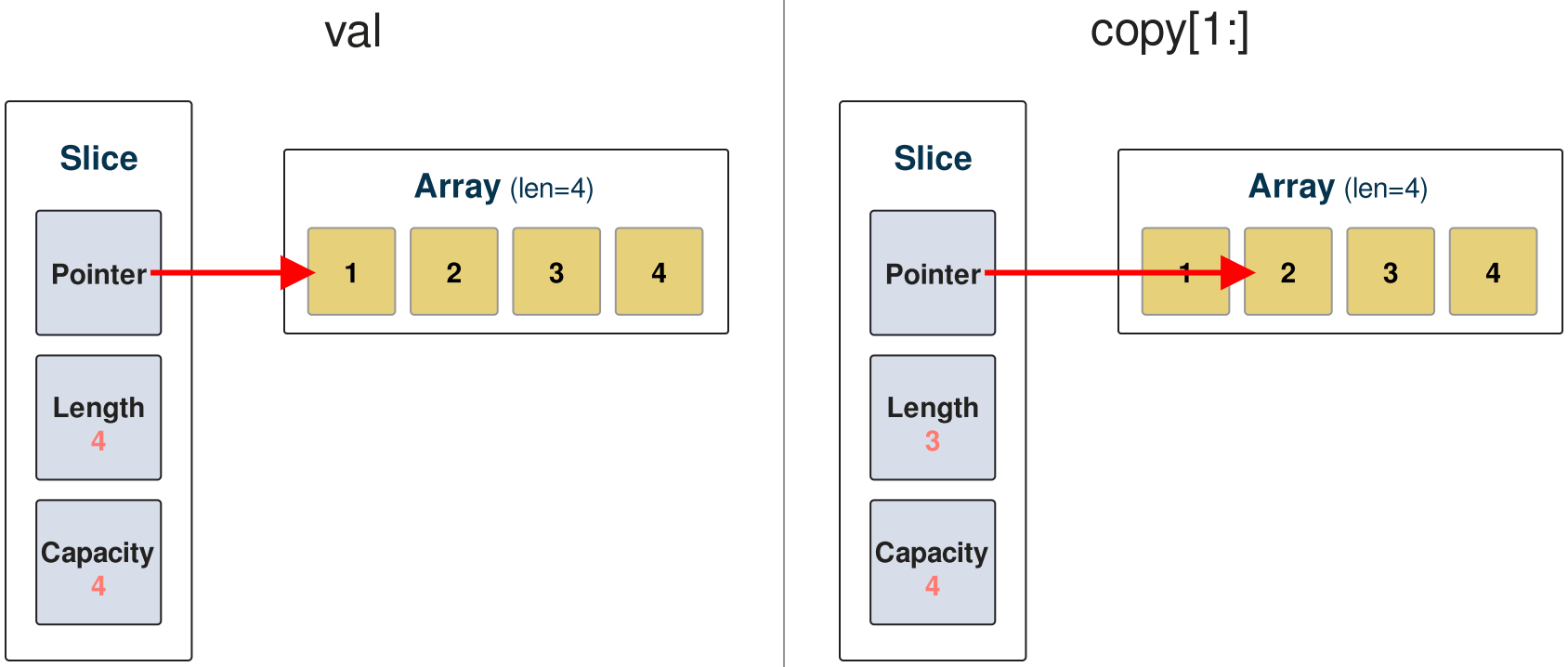
Slices - 4/8

What will happen if we change `copy`?

Example

```
val := []int{1, 2, 3, 4}
copy := val[1:]
//      ^^^^^^^
//      Create slice, from 2nd element

copy[0] = 10
```

Slices - 5/8


val
Slice
Pointer
Length
4
Capacity
4
Array (len=4)
1
2
3
4
copy[1:]
Slice
Pointer
Length
3
Capacity
4
Array (len=4)
1
2
3
4

Slices - 6/8

Result

```
val := []int{1, 2, 3, 4}
copy := val[1:]

copy[0] = 10
//      ^^^^
//      Changes the underlying array

fmt.Println(val)  // Output: [1 10 3 4]
fmt.Println(copy) // Output: [10 3 4]
```

Slices - 7/8

Make

- Creates a zeroed array, returns slice with reference to array
- Better for performance as it will allocate once

Example

```
x := make([]int, 10)
//         ^^^^^  ^^
//         Type   Size

x[1] = 25
```

Slices - 8/8

Most common operations

```
x := []int{1, 2, 3}

x = append(x, 4)
//  ^^^^^^
//  Append data to slice

x = x[:len(x)-1]
//  ^^^^^^^^^^^^
//  Remove the last element

y := make([]int, len(x))
copy(x, y)
// Copy `x` to `y`
```

Range

```
val := []int{1, 2, 3}

for i, elem := range val {
//   ^   ^^^^
// Key Value
}


for i := range val {
//   ^
//  Key
}
```

Maps 1/2

```
data := make(map[string]int)
//            ^^^^^^ ^^^
//            Key    Value

data["key"] = 100


for key, value := range data {
    fmt.Println(key, value)
}
```

Maps 2/2

```
data := make(map[string]int)

value := data["key"]
//       ^^^^^^^^^^^^
//       Returns the default value if key doesn't exist


if value, ok := data["key"]; ok {
// ^^^^^  ^^
// Value  Key exists

    fmt.Println(value)
}
```

Interfaces - 1/4

Summary:

- Defines a set of methods
- Implicitly implemented
- Typically defined by the consumer

Example

```
type Notifier interface {
    SendNotification(content string) error
}
```

Interfaces - 2/4

Implementation 1

```
type EmailService struct {
    to string
}

func (e EmailService) SendNotification(content string) error {
    fmt.Printf("Sent email to %s: %s\n", e.to, content)
    return nil
}
```

Interfaces - 3/4

Implementation 2

```
type SMSService struct {
    phoneNumber string
}

func (s SMSService) SendNotification(content string) error {
    fmt.Printf("Sent SMS to %s: %s\n", s.phoneNumber, content)
    return nil
}
```

Interfaces - 4/4

Usage

```
func SendAlert(n Notifier) {
//              ^^^^^^^^
//              Interface type

    err := n.SendNotification("alert")
    if err != nil {
        panic(err)
    }
}
```

Pointers - 1/3

Summary:

- Points to a value in memory
- Contains a memory address
- Default value: `nil`

Example

```
var x *int
//    ^^^^
//    Pointer type
```

Pointers - 2/3

Syntax

```
var x *int
// `x` is nil

y := 0
x = &y
//  ^^
//  Let `x` point to `y`

fmt.Println(*x)
//          ^^
//          Dereference `x` to get the value
```

Pointers - 3/3

Structs

```
type Member struct {
    Name string
}

member := &Member{Name: "john"}
//        ^^^^^^^^^
//        Pointer to struct

fmt.Println(member.Name)
//           ^^^^^^
//           Implicit dereference (only for structs)
```

Demonstration

- Pointers
- Structs and methods

Errors

Summary:

- Values
- Explicit error handling
- No exceptions or try/catch

Defined as:

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

Handling errors

```
import (
    "os"
    "log"
)

func main() {
    content, err := os.ReadFile("test.txt")
    if err != nil {
        log.Fatal(err)
    }
}
```

Returning errors

```
import "errors"

func ParseCommand(command string) (Command, error) {
    switch command {
        case "left":
            return CommandLeft, nil
        // ...
        default:
            return "", errors.New("invalid command")
    }
}
```

Error flow

```
import "fmt"

func parseAll(commands []string) ([]Command, error) {
    var result []Command

    for _, cmd := range commands {
        parsed, err := ParseCommand(cmd)
        if err != nil {
            return nil, fmt.Errorf("failed to parse command: %w", err)
        }

        result = append(result, parsed)
    }

    return result, nil
}
```

Custom error

Summary:

- Errors are values
- Need to implement `error` interface

Example

```
type ResponseError struct {
    Message      string
}

func (r ResponseError) Error() string {
    return r.Message
}
```

Type assertions

```
var err error

if responseErr, ok := err.(ResponseError); ok {
    fmt.Println(responseErr.StatusCode)
}
```

Type switch

```
switch errData := err.(type) {
    case ResponseError:
        fmt.Println(errData.StatusCode)
    case OtherError:
        fmt.Println(errData.Error())
}
```

Code Organization

Project Layout

Hierarchy

Modules

- Previously
  - `GOPATH` environment variable
  - Projects in `$GOPATH/src`
  - Binaries in `$GOPATH/bin`
- Go 1.11
  - Experimental support for Go modules (with `GO111MODULE`)
- Go 1.13
  - Go modules by default

Go modules

- `go.mod`
  - Module import path
  - Go version
  - Dependencies
  - Replace directives (optional)
- `go.sum`
  - Hashes of direct/indirect dependencies
  - Used for verification/reproducable versions

Demonstration

Exercise 2

Summary

- Understand vCard format and target fields
- Create a struct to represent contact details
- Parse vCards and fill the struct
- Display the extracted contact information

Go to

training.brainhive.nl

Review

Testing

Unittests - 1/3

- Go has a built-in testing framework
- Define test suite in `_test.go`
- Tests are functions with `Test` prefix
- Run tests with `go test`

Unittests - 2/3

Example

```
import "testing"

func TestSignature(t *testing.T) {
    signature := Member{Name: "john", Email: "test@example.com"}.Signature()
    expected := "john <test@example.com>"

    if signature != expected {
        t.Errorf("expected %s, got %s", expected, signature)
    }
}
```

Unittests - 3/3

Summary:

- Use `t.Run` to group tests

Example

```
func TestSignature(t *testing.T) {
    t.Run("with name", func(t *testing.T) {
        // <- test with name
    })

    t.Run("without name", func(t *testing.T) {
        // <- test without name
    })
}
```

Exercise 2.1

Demonstration

- Unittests

Goals

- Write tests for previous assignment
- Make sure it fails on invalid input

Homework

Go to

adventofcode.com/2022/day/7