# Go-lang Crash Course

Larry Dewey

# Changelog:

| Date | Version | Changes |
|---|---|---|
| March 26, 2024 | 0.1 | Initial Release based off Go 1.22.1 |
| | | |

# Contributors:

| User | Version |
|---|---|
| larrydewey | 0.1 |
| | |

# Contents


**1 Directory Structure** **6**

**2 Packages** **7**

**3 Data Types** **7**
- 3.1 Basic Data Types . . . 7
- 3.2 Composit Data Types . . . 8
- 3.3 Special Data Types and Constructs . . . 9

**4 Variables and Constants** **12**
- 4.1 Variables . . . 12
- 4.2 Constants . . . 12

**5 Control Flow** **13**
- 5.1 Conditional Statements . . . 13
- 5.2 Loops . . . 13
- 5.3 Switch Statements . . . 14

**6 Functions** **15**

**7 Error Handling** **15**
- 7.1 Error Interface . . . 15
- 7.2 Returning Errors from Functions . . . 16

**8 Pointers** **16**

**9 Methods** **17**
- 9.1 Defining Methods . . . 18
- 9.2 Calling Methods . . . 18
- 9.3 Receiver Types . . . 18

**10 Generic Types** **19**
- 10.1 Type Constraints . . . 20
- 10.2 Embedding . . . 20

**11 Interfaces** **21**
- 11.1 Defining Interfaces . . . 21
- 11.2 Implementing Interfaces . . . 21

11.3 Using Interfaces . . . . . 22

**12 Concurrency** **23**
12.1 Goroutines . . . . . 23
12.2 Channels . . . . . 24
12.3 Synchronization (Mutexes and Semaphors) . . . . . 24
12.4 Select . . . . . 26
12.5 Challenges of Concurrency . . . . . 26

**13 Reflection** **26**

**14 Testing** **28**
14.1 Functions . . . . . 28
14.2 Subtests . . . . . 29
14.3 Table-driven Testing . . . . . 29
14.4 Running Tests . . . . . 30
14.5 Benchmarking . . . . . 30

# 1 Directory Structure

There isn't a widely recognized or enforced directory structure layout called "bless" in Go. There are, however, some popular conventions for structuring Go projects, and the most common one is the "Standard Go Project Layout" championed by golang-standards.

```
<Project Name>/
├── cmd/
├── internal/
├── pkg/
├── tests/
├── assets/
├── configs/
└── docs/
```

Following is a breakdown of all the directories:

- **cmd/** - This directory contains applciation-specific entry points; usually one per application or service. This is where your main executable resides (`main.go`).
- **internal/** - This directory holds private application and package code that is not meant to be used by other projects.
- **pkg/** - This directory contains public, reusable packages that can be imported and used by other projects.
- **test/** - This directory holds unit and integration tests for your code.
- **assets/** (Optional) - This directory can store static assets like images, CSS, or JavaScript files used by your application.
- **configs/** (Optional) - This directory can hold configuration files for different environments (e.g., development, production).
- **docs/** (Optional) - This directory can store project documentation, such as design documents or API documentation.

# 2 Packages

In Go-lang, packages provide the fundamental construct for name-spacing and modularization. Utilizing the keyword `package`, you can specify the designated namespace of a particular module within your project. Like in C/C++ style programming, Go-lang utilizes `main` as the default package and function for an executable.

## Executable

```
1 | // The package for an executable would be
2 | package main
```

## Non-executable

```
1 | // The package for an library or module would be
2 | package MyLib
```

# 3 Data Types

Go-lang provides a significant number of static data types. Each of these types may be defined when using a variable with `var`, but often may also be inferred when using the `:=` operator.

## 3.1 Basic Data Types

### bool

Represents boolean values (`true` or `false`).

```
1 | var isTrue bool = true
```

### Numeric Types

Go-lang supports various numeric types, including:

- `int`, `int8`, `int16`, `int32`, `int64`: Signed integers of various sizes.
- `uint`, `uint8`, `uint16`, `uint32`, `uint64`: Unsigned integers of various sizes.
- `float32`, `float64`: Floating-point numbers of various precision.
- `complex64`, `complex128`: Complex numbers containing imaginary data.

```
var numInt int = 10
var numFloat float64 = 3.14
```

### string

Represents a sequence of characters, much like in C/C++.

```
var message string = "Hello, world!"
```

## 3.2 Composit Data Types

### Array

- Represents a fixed-size collection of items of the same type.
- Globally defined instances live on the heap, locally scoped live on the stack.

```
var numbers [string] = [3]string{"one", "two", "three"}
fmt.Println("Numbers:", numbers)  // Output: Numbers: [one two
↪  three]
```

### Slice

- Represents a dynamically-sized sequence. Slices are built on top of arrays.
- Starting pointer, length, and capacity may live on the stack, but the elements live on the heap.

```
var numbers []int = []int{1, 2, 3, 4, 5}
```

### Map

Represents a collection of key-value pairs.

```
var ages map[string]int = map[string]int{"Alice": 30, "Bob": 35}
```

### Struct

Used to create user-defined data types representing a collection of related fields.

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

var p Person = Person{Name: "Alice", Age: 30}
```

## 3.3 Special Data Types and Constructs

### Pointer

Represents a memory address.

```
var ptr *int = &numInt
```

### Function

Represents a function

```
func add(a, b int) int {
  return a + b
}
```

### Interface

Represents a set of method signatures.

```
type Shape interface {
  Area() float64
}
```

### Channel

Represents a communication mechanism between `goroutines`.

```
ch := make(chan int)
```

### Error

Represents an error condition.

```
err := errors.New("Something went wrong")
```

### Constants

Represents a fixed value.

```
const pi = 3.14159
```

### Nil

Represents the absence of a value.

```
var emptySlice []int
if emptySlice == nil {
  fmt.Println("Slice is nil")
}
```

### Type Alias

Represents an alternative name for an existing type.

```
type MyInt int
var num MyInt = 10
```

### Struct Tag

Metadata attached to struct fields for encoding or other purposes.

```
type Person struct {
  Name string `json:"name"`
  Age  int  `json:"age"`
}
```

### Context

Carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

```
ctx, cancel := context.WithCancel(context.Background())
```

### Mutex

Provides a locking mechanism to synchronize access to shared resources.

```
var mu sync.Mutex
```

### WaitGroup

Coordinates multiple `goroutines` waiting for a collection of tasks to finish.

```
var wg sync.WaitGroup
```

# 4 Variables and Constants

## 4.1 Variables

In Go-lang all variables are considered mutable. The underlying data type may prevent manipulation, but often Go-lang will perform the modification through copying the original data, applying the mutation, and assigning the new memory to the old pointer.

Variables are using the `var` keyword followed by the variable name and its data type. Assignment is optional, but all default values will be `0` , `nil` , or `""`.

```
var name string = "Alice"
var age int
```

There is also a short-hand variable declaration syntax which signifies to the compiler that it should infer the data type to the best of its ability. This is done using the `:=` operator.

```
name := "Charlie"
// Is equivalent to:
var name string = "Charlie"
```

## 4.2 Constants

Often times when programming in Go-lang, you will find that you would like to define a variable which may never be changed. This is done by utilizing a constant value or `const` . These may be un-typed constants or typed constants. However, you may **NOT** use the shorthand operator `:=` with constants.

# 5 Control Flow

## 5.1 Conditional Statements

Conditional statements in Go-lang behave similar to other C-style programming languages, but does not require parenthesis.

```
if age > 18 {
fmt.Println("You are an adult.")
} else if age > 21 {
fmt.Println("You are of drinking Age.")
} else {
fmt.Println("You are not an adult.")
}
```

## 5.2 Loops

Go-lang does not provide anything other than `for` for looping. While this is a limited option, this construct may be used to emulate behaviors like:

- **Traditional C-style for-loops**

```
for i := 0; i < 5; i++ {
  fmt.Println(i)
}
```

- **For-each-style loops** - In Go-lang for-each style loops will always provide index values when looping. They may be ignored, as seen below, but they will always be provided. It is possible to ignore the values, and only use the index by capturing only one value after `for`.

```
numbers := []int{1, 2, 3, 4, 5}
for _, num := range numbers {  // _ discards the index value
  fmt.Println(num)
}

// Accessing both index and value
for i, num := range numbers {
  fmt.Println("Index:", i, "Value:", num)
}
```

- **While-style loops**

```
end := 5

for end > 0 {
  // Some code to do exactly five times.
  end -= 1
}
```

- **Forever loops**

```
for {
  // Code to be executed repeatedly
  break  // Example of breaking out of the loop
}
```

## 5.3 Switch Statements

Executes code based upon a specifically matched value

```
switch language := "Go"; language {
  case "Go":
    fmt.Println("You are learning the Go language!")
  default:
    fmt.Println("Good luck with your programming journey!")
}
```

# 6 Functions

Functions are fundamental building blocks that allow you to organize your code into reusable blocks. Return types may contain more than one value. Go-lang does this by utilizing a structure similar to a Tuple. This is also how the error system works.

```
// Function with a single return type.
func <function_name>([parameter parameterType],..) [return type] {
  // Does something here with parameters. Possibly returning
  // something.
}

// Function with multiple return types.
func <function_name>([parameter parameterType],..) ([return
↪  type],..) {
  // Does something here with parameters. Possibly returning
  // multiple things.
}

// Example:
func getFullName(firstName, lastName string) (string, string) {
  return firstName, lastName
}

fname, lname := getFullName("Charlie", "Brown")
fmt.Println(fname, lname)  // Prints "Charlie Brown"
```

# 7 Error Handling

In Go-lang, error handling differs from some other programming languages. It utilizes a built-in `error` interface instead of exceptions.

## 7.1 Error Interface

Any type that implements the `error` interface can be used to represent an error. The `error` interface has a single method: `Error() string` which returns a human-readable error message.

```go
type MyError struct {
  message string
}

// Implementing the error interface for MyError.
func (err MyError) Error() string {
  return err.message
}
```

## 7.2 Returning Errors from Functions

Functions can indicate errors by returning a value of a type that implements the `error` interface. The calling code can then check for the returned error and handle it appropriately.

```go
// Although this example uses a function, this is true for in-line
↪  checking, as well.
func openFile(filePath string) (*os.File, error) {
  file, err := os.Open(filePath)
  if err != nil {
  return nil, err  // Return nil for file and the error
  }
  return file, nil  // Return opened file and nil for no error
}
```

# 8 Pointers

Pointers are a powerful tool that allows you to work with memory addresses directly. They play a crucial role in various aspects of Go-lang programming, such as:

- **Manipulating data indirectly:** Pointers let you modify the value of a variable stored elsewhere in memory.
- **Passing arguments by reference:** When you pass a pointer to a function, you're essentially passing the memory address of the original variable, allowing the function to modify it directly.

- **Working with dynamic data structures:** Pointers are fundamental for building and managing data structures like slices, maps, and linked lists. When working with pointers, if you are updating or changing a value, it must be de-referenced by using the `*` operator.

```
var age int = 30
agePtr := &age    // agePtr is a pointer to an integer (int *)
fmt.Println(*agePtr)  // Prints 30 (dereferencing agePtr to access
↪  the value at its address)
```

**CAUTION:**

- Pointers can introduce complexity and potential errors if not used carefully.
- Be mindful of `nil` pointer exceptions and ensure you're de-referencing valid pointers.
- Use clear variable names and comments to improve code readability when working with pointers.

# 9 Methods

In Go-lang, methods provide a way to attach functionality (functions) to user-defined types (structs). They allow you to create objects that encapsulate data (attributes) and behavior (methods that operate on that data). Here's a breakdown of how methods work:

## 9.1 Defining Methods

Methods are declared similar to regular functions, but they have a receiver argument placed before the function name within the parentheses. The receiver argument specifies the type (struct) that the method belongs to. It allows the method to access and potentially modify the fields of the struct.

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

func (p Person) Greet() {
  fmt.Println("Hello, my name is", p.Name)
}
```

**In this example:**

- `Person` is the struct type.
- `Greet` is a method that takes a receiver argument `p` of type `Person`.
- Inside the method, `p.Name` accesses the `Name` field of the `Person` object.

## 9.2 Calling Methods

You call a method on a struct instance using the dot notation (`.`). The struct instance (object) is provided before the dot, followed by the method name.

```
var alice Person = Person{Name: "Alice", Age: 30}
alice.Greet() // Calls the Greet method on the alice object
```

## 9.3 Receiver Types

The receiver argument of a method can be of two types:

- **Value receiver:** The method receives a copy of the struct value. Any modifications within the method won't affect the original struct.
- **Pointer receiver:** The method receives a pointer to the struct. This allows the method to modify the original struct's fields.

```go
func (p *Person) SetAge(newAge int) {
  p.Age = newAge  // Modifies the Age field of the struct pointed
  ↪  to by p
}
```

**In this Example:**

- `SetAge` is a pointer receiver method.
- It takes a pointer to Person (`*Person`) and modifies the `Age` field of the struct it points to.

# 10 Generic Types

Generics, introduced in Go-lang version 1.18, provide a way to write functions and data structures that can work with various data types without sacrificing type safety. Generics use placeholders called type parameters to represent the actual data type that will be used when the generic function or type is instantiated.

```go
func PrintSlice[T any](slice []T) {
  for _, element := range slice {
    fmt.Println(element)
  }
}
```

**In this Example:**

- `PrintSlice` is a generic function.
- `T` is a type parameter that can be any type.

## 10.1 Type Constraints

You can optionally specify constraints on the type parameters using interfaces. This ensures that only types that implement the specified interface can be used as the actual type when instantiating the generic function or type.

```
type Ordered interface {
  < (other Ordered) bool
}

func Min[T Ordered](values []T) T {
  // Implementation to find the minimum value based on the Ordered
  ↪  interface
}
```

**In this Example:**

- `Ordered` is an interface that defines a `<` comparison method.
- `Min` is a generic function with a type constraint.
- It only accepts types that implement the `Ordered` interface, ensuring they can be compoared for finding the minimum value.

## 10.2 Embedding

In Go-lang, embedding — also known as aggregation — allows you to reuse the fields and methods of a type (struct or interface) within another type. It's a powerful mechanism for code organization and promoting code re-usability. In Go-lang, fields are semi-inherited into the child structure.

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

type Employee struct {
  Person  // Embeds the Person struct fields
  JobTitle string
}
```

**In this Example:**

- `Person` is a struct with fields `Name` and `Age`.
- `Employee` embeds the `Person` struct.
- An `Employee` object has all the fields of `Person` (`Name`, `Age`) along with its own field `JobTitle`.

# 11 Interfaces

Interfaces are a powerful mechanism for defining contracts between different parts of your code. They act like blueprints that specify the behavior a type must implement without dictating the underlying implementation details.

## 11.1 Defining Interfaces

Interfaces are declared using the `interface` keyword followed by a name. They can contain method signatures, which define the names and parameter types of methods the implementing type must provide.

```
1 interface Printer {
2   Print() string
3 }
```

**In this Example:**

- `Printer` is an interface
- It defines a single method `Print()` that takes no arguments and returns a string.

## 11.2 Implementing Interfaces

Any type (struct, another interface) can implement an interface by declaring that it "implements" the interface name. The type must then provide implementations for all the methods defined in the interface.

```go
type Person struct {
  Name string
}

func (p Person) Print() string {
  return fmt.Sprintf("Hello, my name is %s", p.Name)
}
```

**In this Example:**

- The `Person` struct implements the `Printer` interface.
- It provides a `Print()` method that satisfies the interface signature.

## 11.3 Using Interfaces

You can declare variables of the interface type. These variables can then hold references to objects of any type that implements the interface.

```go
func PrintInfo(p Printer) {
  fmt.Println(p.Print())
}
```

**In this Example:**

- The `PrintInfo` function takes a parameter of type `Printer`.
- It can be called with any object that implements the `Print()` method (like a `Person` object in this case).

Just like with generics, it is possible to embed interfaces within each other.

```
interface Shape { Area() float64 }

interface Colored { Color() string }

// New interface combining functionalities from both Shape and
↪  Colored
type ColoredShape interface {
  Shape
  Colored
}

type ColoredSquare struct {
  side float64
  color string
}

func (s ColoredSquare) Area() float64 {
  return s.side * s.side
}

func (s ColoredSquare) Color() string {
  return s.color
}

// ColoredSquare implements both Shape and Colored interfaces
↪  (inherited through ColoredShape)
var _ ColoredShape = ColoredSquare{}
```

# 12 Concurrency

Go-lang excels at handling concurrent tasks, allowing your program to make progress on multiple operations seemingly simultaneously. This improves responsiveness and potentially speeds up execution by utilizing multiple cores or processors. Here's a breakdown of the key components that enable concurrency:

## 12.1 Goroutines

Goroutines are lightweight threads of execution managed by the Go runtime. You launch a goroutine using the go keyword followed by a function call. Unlike

threads in some other languages, goroutines are much cheaper to create and manage. Multiple goroutines can run concurrently within the same address space, sharing the same memory.

```
go func() {
  fmt.Println("Hello from a goroutine!")
}
```

## 12.2 Channels

Channels are a communication mechanism between goroutines. They act as synchronized queues that allow goroutines to send and receive data. A channel must be of a specific data type, and only values of that type can be sent or received through the channel. Sending and receiving operations on channels are blocking by default, meaning the sending goroutine will wait until there's a receiver ready, and vice versa.

```
ch := make(chan string)

go func() {
  ch <- "Message from a goroutine!"
}

message := <-ch
fmt.Println(message)
```

## 12.3 Synchronization (Mutexes and Semaphors)

While goroutines share memory, uncontrolled access can lead to race conditions (unexpected behavior due to concurrent access). Synchronization primitives like mutexes (mutual exclusion locks) and semaphores provide mechanisms for goroutines to coordinate access to shared resources.

### Mutexes

A goroutine can acquire a lock on amutex before accessing a shared resource, ensuring only one goroutine can access it at a time.

```go
var mutex sync.Mutex
func updateCounter() {
  mutex.Lock()
  defer mutex.Unlock()  // Ensures unlock even in case of panic or
  ↪  errors
  // Access and update shared counter here
}
```

### Semaphores

Semaphores control access to a limited number of resources. A goroutine trying to acquire a semaphore that's already full will be blocked until another goroutine releases it.

```go
// Simulate a shared resource (e.g., database connection)
var resource sync.Mutex

func accessResource(name string) {
  resource.Lock()
  defer resource.Unlock()
  fmt.Println("Goroutine", name, "accessing resource")
  // Simulate some work on the resource
  fmt.Println("Goroutine", name, "working on resource")
}

func main() {
  // Semaphore using 2 concurrent buffered channel
  sem := make(chan struct{}, 2)

  for i := 0; i < 5; i++ {
    go func(i int) {
      <-sem // Acquire semaphore (block if full)
      defer func() { sem <- struct{}{} }() // Release semaphore
      accessResource(fmt.Sprintf("G%d", i))
    }(i)
  }

  // Wait for all goroutines to finish
  for i := 0; i < 5; i++ {
    <-sem
  }
}
```

## 12.4 Select

The `select` statement allows a goroutine to wait on multiple communication channels or operations simultaneously. It's a powerful tool for managing concurrency and handling multiple potential events. It will unblock when one of the channels or operations becomes ready.

```
select {
  case msg := <-ch1:
    fmt.Println("Received on ch1:", msg)
  case msg := <-ch2:
    fmt.Println("Received on ch2:", msg)
  default:
    fmt.Println("No messages received!")
}
```

## 12.5 Challenges of Concurrency

- **Complexity:** Concurrency adds complexity to programs compared to traditional sequential programming.
- **Race Conditions:** It's crucial to carefully design your concurrent code to avoid race conditions and ensure proper synchronization between goroutines.
- **Deadlocks:** Deadlocks can occur if goroutines are waiting on each other indefinitely. Proper resource management and channel usage are essential to prevent deadlocks.

# 13 Reflection

In Go, reflection is a powerful but sometimes complex feature that allows your program to examine and manipulate its own structure at runtime. It provides functionalities to:

- **Inspect Types:** You can discover the type of a variable or value at runtime using the reflect package.
- **Access Values:** Reflection allows you to access the underlying value stored in a variable, even if it's an interface or pointer.

- **Modify Values:** In some cases, you can modify the value stored in a variable using reflection (be cautious due to potential risks).
- **Invoke Methods:** Reflection can be used to call methods on objects dynamically based on the type information obtained at runtime.

## How Reflection Works:

1. **The `reflect` Package:** Provides the core functionalities for reflection; including various functions and types to work with reflected values and types. To utilize reflection, this library must be imported.
2. **Types and Values:** Reflection revolves around two key concepts:
    - `reflect.Type` - represents the type informatin of a variable or value.
    - `reflect.Value` - represents the actual value stored in a variable.
3. **Getting Type Information:**
    - The `reflect.TypeOf(x)` function takes a variable `x` and returns its corresponding `reflect.Type`.
    - You can then use methods on the `reflect.Type` to get details about the type, such as its kind (e.g., `int`, `string`, `struct`), name, and underlying fields for structs.
4. **Getting and Modifying Values:**
    - The `reflect.ValueOf(x)` function takes a variable `x` and returns its corresponding `reflect.Value`.
    - This `reflect.Value` can be used to access the underlying value through its `Interface()` method (if applicable) or by converting it to a specific type using the `Kind()` and other methods.
    - **CAUTION:** Modifying values using reflection should be done cautiously. Improper modifications can lead to unexpected behavior or program crashes. It's generally recommended to avoid modifying values via reflection unless absolutely necessary.

### Use Cases

- **Generic Code:** Reflection can be useful for writing generic functions or data structures that can work with various types at runtime.
- **Encoding/Decoding:** It can be helpful for implementing custom encoders or decoders that work with different data formats based on type information.
- **Validation:** Reflection can be used to introspect a struct's fields and perform validation checks on their values.

### When to Avoid Reflection

- **Performance:** Reflection can incur some overhead compared to direct access. Use it judiciously when the benefits outweigh the performance cost.
- **Complexity:** Reflection can add complexity to your code. If a simpler, non-reflective approach achieves the same result, prefer the simpler solution.

# 14 Testing

Go-lang prioritizes code maintainability and test-ability. The `testing` package provides the foundation for writing unit tests. It offers functionalities for defining test cases, running tests, and reporting results. The primary utility to implement unit testing is through using test functions.

## 14.1 Functions

Test functions are the building blocks of Go-lang unit testing. They are named `Test*` followed by a descriptive name (ex. `TestSum_PositiveNumbers`). Each testing function takes a pointer to a `testing.T` object as its argument. This pointer to an object provides methods for logging information, reporting errors, and marking the test as failed.

```go
func Sum(a, b int) int {
  return a + b
}

// Example of a unit test for positive numbers.
func TestSum_PositiveNumbers(t *testing.T) {
  result := Sum(5, 3)
  if result != 8 {
    t.Errorf("Expected 8, got %d", result) // Report an error if
    ↪  assertion fails
  }
}
```

As you can imagine, this quickly can become disorganized when needing to run multiple unit tests which are closely related. Fortunately, Go-lang provides subtests and table-driven tests to help with this.

## 14.2 Subtests

Subtests are supported within a test function using `t.Run`, This allows you to group related test cases and report them individually within the overall test.

```go
func TestSum(t *testing.T) {
  t.Run("positive numbers", func(t *testing.T) {
    result := Sum(5, 3)
    if result != 8 {
      t.Errorf("Expected 8, got %d", result)
    }
  })
  t.Run("negative numbers", func(t *testing.T) {
    result := Sum(-5, -3)
    if result != -8 {
      t.Errorf("Expected -8, got %d", result)
    }
  })
}
```

## 14.3 Table-driven Testing

Table-driven testing provides even further granularity, where you define a set of test cases as a slice of structs or maps. This promotes code re-usability and

improves test readability.

```
type testCase struct {
  a int
  b int
  expected int
}

func TestSum(t *testing.T) {
  testCases := []testCase {
    {a: 5, b: 3, expected: 8},
    {a: -5, b: -3, expected: -8},
  }

  for _, tc := range testCases {
    result := Sum(tc.a, tc.b)
    if result != tc.expected {
      t.Errorf(
        "Expected %d for (%d, %d), got %d",
        tc.expected,
        tc.a,
        tc.b,
        result
      )
    }
  }
}
```

## 14.4 Running Tests

You use the `go test` command on the command line to run your tests. This command searches for files ending in `_test.go` within the current directory and its subdirectories. It then executes all the test functions it finds and reports the results, including failures and timings.

## 14.5 Benchmarking

Go-lang also supports benchmarks, which are essentially timed tests used to compare the performance of different code implementations. Benchmark functions are typically named `Benchmark*` and follow a similar structure to test functions.