Hypercode

Learn Go from Scratch

From “Hello, World!” to concurrency —
seventeen lessons covering the fundamentals of Go.

This is the PDF version of Hypercode.
For the interactive experience with a built-in code editor, visit:
hypercode.alexisbouchez.com

Alexis Bouchez

Contents

Introduction

Welcome to **Hypercode**, an interactive course for learning the Go programming language from the ground up.

Why Go?

Go is a statically typed, compiled language designed for simplicity and efficiency. It compiles fast, runs fast, and is easy to read. Here is what makes it stand out:

- **Fast compilation** — Go compiles entire projects in seconds, even large codebases.
- **Simple syntax** — The language has only 25 keywords. If you know any C-style language, you can read Go code immediately.
- **Built-in concurrency** — Goroutines and channels make concurrent programming straightforward.
- **Rich standard library** — HTTP servers, JSON encoding, cryptography, testing — all built in, no third-party dependencies required.
- **Single binary deployment** — Go compiles to a single static binary. No runtime, no VM, no dependencies to install on the target machine.

The Story

Go was created at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. The three designers were frustrated with the complexity of existing systems languages, particularly C++: slow builds, complicated dependency management, and a language specification that had grown unwieldy over decades.

Their goal was a language that combined the performance and safety of a compiled language with the ease of use of a dynamic one. Go was open-sourced in November 2009 and reached its first stable release, Go 1.0, in March 2012 with a strong backward-compatibility promise.

Rob Pike had co-created the Plan 9 operating system and the UTF-8 encoding. Ken Thompson had co-created Unix and the C programming language. Robert Griesemer had worked on the V8 JavaScript engine and the Java HotSpot VM.

Who Uses Go

Go has become the backbone of modern cloud infrastructure. Some of the most influential open-source projects are written in Go:

- **Docker** — the containerization platform that changed how software is deployed.

- **Kubernetes** — the container orchestration system that powers cloud-native applications.
- **Terraform** — the infrastructure-as-code tool used to manage cloud resources.

Major companies use Go extensively: Google, Uber, Dropbox, Twitch, Cloudflare, and many others. It is especially popular for building microservices, CLI tools, DevOps tooling, and network services.

About This Book

This book contains seventeen lessons organized into eight chapters. Each lesson explains a concept, demonstrates it with code examples, and ends with an exercise for you to practice. Solutions are provided at the end of each lesson. The lessons are designed to be worked through in order, as each builds on the concepts introduced before it.

- **Foundations** — How Go programs are structured: packages, imports, and the `main` function. Variables, types, and constants.
- **Control Flow** — Conditionals with `if` / `else` and `switch`. Loops with `for`.
- **Functions** — Declaring functions, multiple return values, named returns, and `defer`.
- **Data Structures** — Slices (dynamic arrays), maps (hash tables), and strings in depth.
- **Custom Types** — Pointers, structs with methods, interfaces, type assertions, and structural typing.
- **Error Handling** — Go's explicit approach to errors using values instead of exceptions.
- **Generics** — Type parameters for functions and data structures.
- **Concurrency** — Goroutines, channels, and the `select` statement.

1

Foundations

1.1 Hello, World!

The Anatomy of a Go Program

Every Go source file starts with a package declaration. A package is how Go organizes code: it serves as a namespace, a unit of compilation, and a mechanism for controlling visibility.

The `main` package is special. It tells the Go compiler that this is an executable program, not a library. Without it, you have a package that other code can import, but nothing you can actually run.

```
package main
```

Imports

The `import` keyword brings other packages into scope. The `fmt` package (short for "format") handles formatted I/O: printing to the terminal, formatting strings, reading input.

```
import "fmt"
```

When you need multiple packages, Go uses a grouped syntax:

```
import (
    "fmt"
    "math"
    "strings"
)
```

The Entry Point

Every executable needs a starting point. In Go, that is `func main()`, with no parameters and no return value. When you run a Go program, execution begins here and here only.

```
func main() {
    fmt.Println("Hello, World!")
}
```

`fmt.Println` writes its arguments to standard output, followed by a newline character.

Exported Names

Notice the capital `P` in `Println`. In Go, any name that starts with an uppercase letter is *exported*, visible outside its package. A lowercase name is *unexported*, private to the package.

No `public` or `private` keywords. The casing **is** the access control. This is a deliberate design choice that makes visibility immediately obvious when reading code.

▷ Your Task

Write a program that prints exactly `Hello, World!` to standard output.
Starter code:

```
package main

import "fmt"

func main() {
    // Write your first Go program here
}
```

✓ Solution

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

1.2 Variables

Declaring Variables

Go is statically typed, but it does not force you to spell out every type. You have two primary ways to declare variables.

The `var` Keyword

The explicit form. You state the name, the type, and optionally an initial value:

```
var name string = "Go"
var year int = 2009
var ratio float64 = 3.14
```

If you provide an initial value, the type can be omitted. The compiler infers it:

```
var name = "Go"     // inferred as string
var year = 2009     // inferred as int
```

Short Declaration

Inside functions, the `:=` operator declares and initializes in one step. This is the form you will use most often:

```
name := "Go"
year := 2009
awesome := true
```

The `:=` operator is only available inside functions. Package-level variables must use `var`.

Zero Values

Every type in Go has a *zero value*: the value a variable holds if you declare it without initializing it. This is a guarantee, not an accident. There are no uninitialized variables in Go.

Type	Zero Value
`int`, `float64`	`0`
`string`	`""` (empty string)
`bool`	`false`
pointers, slices, maps	`nil`

```
var count int     // 0
var label string  // ""
var ready bool    // false
```

Constants

Values that never change are declared with `const`. Constants must be known at compile time. You cannot assign the result of a function call to a constant.

```
const pi = 3.14159
const maxRetries = 3
```

▷ Your Task

Declare three variables using the short declaration operator:

- `name` with value `"Go"`
- `year` with value `2009`
- `awesome` with value `true`

Then print them using the format string provided in the starter code.

Starter code:

```
package main

import "fmt"

func main() {
    // Declare your variables here using :=

    fmt.Printf("name: %s, year: %d, awesome: %t\n",
        name, year, awesome)
}
```

✓ Solution

```
package main

import "fmt"

func main() {
    name := "Go"
    year := 2009
    awesome := true

    fmt.Printf("name: %s, year: %d, awesome: %t\n",
        name, year, awesome)
}
```

2

Control Flow

2.1 Conditionals

Making Decisions

If / Else

Go's `if` statement looks like most languages, minus the parentheses around the condition:

```
if x > 10 {
    fmt.Println("big")
} else if x > 5 {
    fmt.Println("medium")
} else {
    fmt.Println("small")
}
```

The braces are mandatory, even for single-line bodies. This eliminates an entire class of bugs that other languages suffer from.

If with Init Statement

Go has a unique feature: you can run a short statement before the condition. The variable you declare is scoped to the `if` block:

```
if length := len(name); length > 10 {
    fmt.Println("long name")
} else {
    fmt.Println("short name")
}
// length is not accessible here
```

This pattern keeps variables tightly scoped. You will see it everywhere in Go, especially with error handling.

Switch

Go's `switch` is cleaner than most languages. Cases do not fall through by default, so no `break` statements are needed:

```
switch day {
case "Monday":
    fmt.Println("start of the week")
case "Friday":
    fmt.Println("almost weekend")
default:
    fmt.Println("regular day")
}
```

Switch Without a Condition

A `switch` with no value acts as a clean alternative to long `if-else` chains:

```
switch {
case temp <= 0:
    fmt.Println("freezing")
case temp <= 15:
    fmt.Println("cold")
case temp <= 30:
    fmt.Println("warm")
default:
    fmt.Println("hot")
}
```

This reads naturally and scales better than nested `if-else` blocks.

▷ Your Task

Write a function `classifyTemp` that takes an integer temperature in Celsius and returns a string:

- `"freezing"` if temp ≤ 0
- `"cold"` if temp ≤ 15
- `"warm"` if temp ≤ 30
- `"hot"` if temp > 30

Starter code:

```
package main

import "fmt"

func classifyTemp(temp int) string {
    // Your code here
    return ""
}

func main() {
    fmt.Println(classifyTemp(-5))
    fmt.Println(classifyTemp(10))
    fmt.Println(classifyTemp(25))
    fmt.Println(classifyTemp(35))
}
```

✓ Solution

```go
package main

import "fmt"

func classifyTemp(temp int) string {
    switch {
    case temp <= 0:
        return "freezing"
    case temp <= 15:
        return "cold"
    case temp <= 30:
        return "warm"
    default:
        return "hot"
    }
}

func main() {
    fmt.Println(classifyTemp(-5))
    fmt.Println(classifyTemp(10))
    fmt.Println(classifyTemp(25))
    fmt.Println(classifyTemp(35))
}
```

2.2 Loops

The Only Loop You Need

Go has exactly one loop keyword: `for`. It does the work of `for`, `while`, and `do-while` from other languages.

Classic For Loop

The three-component form, similar to C or Java:

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

While-Style Loop

Drop the init and post statements. You get a while loop:

```
n := 1
for n < 100 {
    n *= 2
}
```

Infinite Loop

Drop everything. Use `break` to exit:

```
for {
    line := readInput()
    if line == "quit" {
        break
    }
}
```

Continue

`continue` skips to the next iteration:

```
for i := 0; i < 10; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i) // only odd numbers
}
```

For Range

The `range` keyword iterates over slices, arrays, strings, maps, and channels. It gives you both the index and the value:

```
names := []string{"Alice", "Bob", "Charlie"}
for i, name := range names {
```

```
    fmt.Printf("%d: %s\n", i, name)
}
```

Use `_` to discard the index when you do not need it:

```
for _, name := range names {
    fmt.Println(name)
}
```

▷ Your Task

Write a function `fizzBuzz` that takes an integer `n` and prints the numbers from 1 to `n` (inclusive), one per line, with these substitutions:

- `"FizzBuzz"` if the number is divisible by both 3 and 5
- `"Fizz"` if the number is divisible by 3
- `"Buzz"` if the number is divisible by 5
- The number otherwise

Starter code:

```
package main

import "fmt"

func fizzBuzz(n int) {
    // Your code here
}

func main() {
    fizzBuzz(15)
}
```

✓ Solution

```
package main

import "fmt"

func fizzBuzz(n int) {
    for i := 1; i <= n; i++ {
        switch {
        case i%15 == 0:
            fmt.Println("FizzBuzz")
        case i%3 == 0:
            fmt.Println("Fizz")
        case i%5 == 0:
            fmt.Println("Buzz")
        default:
            fmt.Println(i)
        }
    }
}

func main() {
    fizzBuzz(15)
```

```
}
```

3

Functions

3.1 Functions in Go

Functions are declared with `func`, followed by the name, parameters, and return type:

```
func greet(name string) string {
    return "Hello, " + name
}
```

Parameter types come *after* the name, not before. This was a deliberate choice. The Go team believes it reads more naturally, especially as declarations get complex.

When consecutive parameters share a type, you can group them:

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

Multiple Return Values

This is one of Go's most distinctive features. A function can return more than one value:

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}
```

The caller unpacks the results:

```
result, err := divide(10, 3)
```

This pattern is the foundation of Go's error handling. Instead of exceptions, functions return errors as values.

Named Return Values

You can name your return values, which documents what the function returns and allows "naked" returns:

```
func swap(a, b string) (first, second string) {
    first = b
    second = a
    return // returns first and second
}
```

Use named returns sparingly. They are most useful for documenting return values in short functions. In longer functions, explicit returns are clearer.

Functions as Values

Functions are first-class values. You can assign them to variables, pass them as arguments, and return them from other functions:

```
double := func(x int) int {
    return x * 2
}
fmt.Println(double(5)) // 10
```

▷ Your Task

Write a function `minMax` that takes a `[]int` and returns two `int` values: the minimum and the maximum values from the slice.

Starter code:

```
package main

import "fmt"

func minMax(numbers []int) (int, int) {
    // Your code here
    return 0, 0
}

func main() {
    min, max := minMax([]int{3, 1, 4, 1, 5, 9, 2, 6})
    fmt.Printf("min: %d, max: %d\n", min, max)
}
```

✓ Solution

```
package main

import "fmt"

func minMax(numbers []int) (int, int) {
    min, max := numbers[0], numbers[0]
    for _, n := range numbers {
        if n < min {
```

```
            min = n
        }
        if n > max {
            max = n
        }
    }
    return min, max
}

func main() {
    min, max := minMax([]int{3, 1, 4, 1, 5, 9, 2, 6})
    fmt.Printf("min: %d, max: %d\n", min, max)
}
```

3.2 Defer

Cleaning Up After Yourself

The `defer` keyword schedules a function call to run when the surrounding function returns. It is Go's way of ensuring cleanup happens no matter how the function exits.

```
func main() {
    fmt.Println("start")
    defer fmt.Println("deferred")
    fmt.Println("end")
}
// Output:
// start
// end
// deferred
```

The deferred call runs *after* the function body completes but *before* it returns to the caller.

LIFO Order

When you defer multiple calls, they execute in **last-in, first-out** order — like a stack:

```
func main() {
    defer fmt.Println("first")
    defer fmt.Println("second")
    defer fmt.Println("third")
}
// Output:
// third
// second
// first
```

Common Pattern: Open Then Defer Close

The most common use of `defer` is pairing resource acquisition with cleanup on the very next line:

```
f, err := os.Open("data.txt")
if err != nil {
    return err
}
defer f.Close()
// work with f...
```

This makes it impossible to forget to close the file, regardless of how many return paths the function has.

Arguments Are Evaluated Immediately

The arguments to a deferred call are evaluated when the `defer` statement executes, not when the deferred function runs:

```
x := 10
defer fmt.Println(x) // captures 10
```

```
x = 20
// prints 10, not 20
```

Defer in Loops

Be careful with `defer` inside loops. Deferred calls accumulate and only run when the function returns, not when the loop iteration ends:

```
for _, name := range files {
    f, _ := os.Open(name)
    defer f.Close() // these all pile up!
}
```

If you need per-iteration cleanup, extract the body into a separate function so each deferred call runs promptly.

▷ Your Task

Write a function `countdown(n int)` that uses `defer` inside a loop to print numbers in reverse order, followed by `"Go!"`.
For example, `countdown(3)` should print `3`, `2`, `1`, `Go!` (each on its own line).
Hint: defer each number inside the loop. Since deferred calls execute in LIFO order, deferring 1, 2, 3 will print 3, 2, 1. Print `"Go!"` using a defer *before* the loop.
Starter code:

```
package main

import "fmt"

func countdown(n int) {
    // Your code here
}

func main() {
    countdown(3)
}
```

✓ Solution

```
package main

import "fmt"

func countdown(n int) {
    defer fmt.Println("Go!")
    for i := 1; i <= n; i++ {
        defer fmt.Println(i)
    }
}

func main() {
    countdown(3)
}
```

4

Data Structures

4.1 Slices

Dynamic Arrays, Done Right

Go has arrays, but you will rarely use them directly. Arrays have a fixed size baked into their type: `[5]int` and `[10]int` are different types entirely. Instead, Go gives you **slices**: a flexible, dynamic view over an underlying array.

Creating Slices

```
// Slice literal
numbers := []int{1, 2, 3, 4, 5}

// Make a slice with initial length and capacity
data := make([]int, 5)       // len=5, cap=5, filled with zeros
buffer := make([]int, 0, 10) // len=0, cap=10
```

The difference between `[5]int` (array) and `[]int` (slice) is that single missing number. Slices are what you want almost every time.

Length and Capacity

Every slice has two properties: **length** (how many elements it contains) and **capacity** (how many elements the underlying array can hold before reallocation).

```
s := make([]int, 3, 10)
fmt.Println(len(s)) // 3
fmt.Println(cap(s)) // 10
```

Append

`append` adds elements to a slice and returns a new slice. If the underlying array is full, Go allocates a bigger one and copies the data:

```
s := []int{1, 2, 3}
s = append(s, 4, 5)
```

```
// s is now [1, 2, 3, 4, 5]
```

Always reassign the result of `append` back to the slice variable. This is not optional. `append` may return a slice pointing to a completely new array.

Slicing

You can create a new slice from an existing one using the slice operator:

```
s := []int{0, 1, 2, 3, 4}
a := s[1:3]  // [1, 2]     (from index 1, up to but not including 3)
b := s[:3]   // [0, 1, 2]  (from the start)
c := s[2:]   // [2, 3, 4]  (to the end)
```

A sub-slice shares the same underlying array. Modifying one affects the other. If you need an independent copy, use `copy` or `append` to a new slice.

▷ Your Task

Write two functions:

- `sum` — takes a `[]int` and returns the sum of all elements.
- `filterEvens` — takes a `[]int` and returns a new `[]int` containing only the even numbers, in order.

Starter code:

```
package main

import "fmt"

func sum(numbers []int) int {
    // Your code here
    return 0
}

func filterEvens(numbers []int) []int {
    // Return only even numbers
    return nil
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Println(sum(nums))
    fmt.Println(filterEvens(nums))
}
```

✓ Solution

```
package main

import "fmt"

func sum(numbers []int) int {
    total := 0
    for _, n := range numbers {
```

```
        total += n
    }
    return total
}

func filterEvens(numbers []int) []int {
    var result []int
    for _, n := range numbers {
        if n%2 == 0 {
            result = append(result, n)
        }
    }
    return result
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Println(sum(nums))
    fmt.Println(filterEvens(nums))
}
```

4.2 Maps

Key-Value Storage

Maps are Go's built-in hash table. They store key-value pairs and provide constant-time lookups.

Creating Maps

```
// Map literal
ages := map[string]int{
    "Alice": 30,
    "Bob":   25,
}

// Make an empty map
scores := make(map[string]int)
```

The type `map[string]int` reads as "a map from strings to ints." Keys can be any comparable type (strings, ints, booleans, structs without slice/map fields). Values can be anything.

Operations

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

// Insert or update
m["alice"] = 95

// Lookup
score := m["alice"] // 95

// Delete
delete(m, "alice")

// Length
fmt.Println(len(m))
```

The Comma-Ok Idiom

When you look up a key that does not exist, Go returns the zero value for the value type. To distinguish between "key not found" and "key exists with zero value", use the two-value form:

```
value, ok := m["key"]
if ok {
    fmt.Println("found:", value)
} else {
    fmt.Println("not found")
}
```

This is called the "comma-ok" idiom. The second value is a boolean that indicates whether the key was present.

Iterating

Use `for range` to iterate over a map. The iteration order is **not guaranteed**. It is intentionally randomized by the runtime:

```
for key, value := range m {
    fmt.Printf("%s: %d\n", key, value)
}
```

▷ Your Task

Write a function `wordCount` that takes a string and returns a `map[string]int` where each key is a word and each value is how many times that word appears.
Use `strings.Fields` to split the string into words (it splits on whitespace).
Starter code:

```
package main

import (
    "fmt"
    "strings"
)

func wordCount(s string) map[string]int {
    // Your code here
    return nil
}

func main() {
    result := wordCount("the cat sat on the mat the cat")
    fmt.Println(result["the"])
    fmt.Println(result["cat"])
    fmt.Println(result["sat"])
    fmt.Println(result["on"])
    fmt.Println(result["mat"])
}
```

✓ Solution

```
package main

import (
    "fmt"
    "strings"
)

func wordCount(s string) map[string]int {
    counts := make(map[string]int)
    for _, word := range strings.Fields(s) {
        counts[word]++
    }
    return counts
}

func main() {
```

```
    result := wordCount("the cat sat on the mat the cat")
    fmt.Println(result["the"])
    fmt.Println(result["cat"])
    fmt.Println(result["sat"])
    fmt.Println(result["on"])
    fmt.Println(result["mat"])
}
```

4.3 Strings

More Than Meets the Eye

Go strings are immutable sequences of bytes. This seems simple until you encounter non-ASCII text and discover the difference between bytes and characters.

Bytes vs Runes

A `string` in Go is a read-only slice of bytes. The `len` function returns the number of **bytes**, not the number of characters:

```
fmt.Println(len("hello"))  // 5
fmt.Println(len("héllo"))  // 6 (é is 2 bytes in UTF-8)
```

Go uses the term **rune** for what most languages call a character. A rune is an alias for `int32` and represents a single Unicode code point.

To get the actual character count, use `utf8.RuneCountInString`:

```
import "unicode/utf8"

fmt.Println(utf8.RuneCountInString("héllo")) // 5
```

Iterating Over Strings

A `for range` loop over a string yields runes, not bytes:

```
for i, r := range "héllo" {
    fmt.Printf("%d: %c\n", i, r)
}
// 0: h
// 1: é    (byte index 1, but é is 2 bytes wide)
// 3: l    (byte index jumps to 3)
// 4: l
// 5: o
```

The index is the byte position, but `r` is a full rune. This is the safe way to process strings character by character.

The strings Package

The `strings` package provides essential string operations:

```
import "strings"

strings.Contains("seafood", "foo")     // true
strings.HasPrefix("seafood", "sea")    // true
strings.HasSuffix("seafood", "food")   // true

strings.ToUpper("hello")               // "HELLO"
strings.ToLower("HELLO")               // "hello"

strings.Split("a,b,c", ",")            // ["a", "b", "c"]
strings.Join([]string{"a", "b"}, "-")  // "a-b"
```

```
strings.Fields("  foo  bar  baz  ")    // ["foo", "bar", "baz"]
```

`strings.Fields` splits on any whitespace and ignores leading/trailing spaces. It is often more useful than `strings.Split`.

Conversions

You can convert between strings, byte slices, and rune slices:

```
s := "hello"
b := []byte(s)   // string to bytes
r := []rune(s)   // string to runes
s2 := string(b)  // bytes back to string
s3 := string(r)  // runes back to string
```

For number conversions, use the `strconv` package:

```
import "strconv"

s := strconv.Itoa(42)        // int to string: "42"
n, err := strconv.Atoi("42") // string to int: 42, nil
```

▷ Your Task

Write a function `acronym(s string) string` that takes a phrase and returns its acronym. Take the first letter of each word and uppercase it.
For example, `"Portable Network Graphics"` becomes `"PNG"`.
Use `strings.Fields` to split into words, and `strings.ToUpper` to uppercase.
Starter code:

```
package main

import (
    "fmt"
    "strings"
)

func acronym(s string) string {
    // Your code here
    return ""
}

func main() {
    fmt.Println(acronym("Portable Network Graphics"))
    fmt.Println(acronym("as soon as possible"))
}
```

✓ Solution

```
package main

import (
```

```
    "fmt"
    "strings"
)

func acronym(s string) string {
    words := strings.Fields(s)
    result := ""
    for _, w := range words {
        for _, r := range w {
            result += strings.ToUpper(string(r))
            break
        }
    }
    return result
}

func main() {
    fmt.Println(acronym("Portable Network Graphics"))
    fmt.Println(acronym("as soon as possible"))
}
```

5

Custom Types

5.1 Pointers

Understanding Pointers

A pointer holds the memory address of a value. Instead of passing data around by copying it, you can pass a pointer to the original data.

Pointer Types and Operators

The type `*T` is a pointer to a value of type `T`. The `&` operator takes the address of a variable. The `*` operator dereferences a pointer, giving you the value it points to:

```
x := 42
p := &x         // p is *int, points to x
fmt.Println(*p)  // 42 (read through the pointer)
*p = 100         // modify x through the pointer
fmt.Println(x)   // 100
```

Zero Value

The zero value of a pointer is `nil`. A nil pointer does not point to anything. Dereferencing a nil pointer causes a runtime panic:

```
var p *int       // p is nil
fmt.Println(p)   // <nil>
```

Passing by Value vs Pointer

Go is pass-by-value. When you pass a variable to a function, the function gets a copy. To let a function modify the original, pass a pointer:

```
func increment(x *int) {
    *x++
}

n := 5
```

```
increment(&n)
fmt.Println(n) // 6
```

Without the pointer, `increment` would modify a copy and `n` would stay 5.

When to Use Pointers

Use pointers when you need to:

- **Modify the caller's data** — the most common reason
- **Avoid copying large structs** — passing a pointer is cheaper than copying a large value
- **Signal absence** — a nil pointer can mean "no value"

The `new` function allocates memory and returns a pointer to the zero value:

```
p := new(int)    // *int pointing to 0
*p = 42
```

▷ Your Task

Write two functions:

- `swap(a, b *int)` — swaps the values that `a` and `b` point to
- `double(x *int)` — doubles the value that `x` points to

Starter code:

```
package main

import "fmt"

func swap(a, b *int) {
    // Your code here
}

func double(x *int) {
    // Your code here
}

func main() {
    a, b := 3, 5
    swap(&a, &b)
    fmt.Println(a, b)

    x := 7
    double(&x)
    fmt.Println(x)
}
```

✓ Solution

```
package main

import "fmt"
```

```
func swap(a, b *int) {
    *a, *b = *b, *a
}

func double(x *int) {
    *x *= 2
}

func main() {
    a, b := 3, 5
    swap(&a, &b)
    fmt.Println(a, b)

    x := 7
    double(&x)
    fmt.Println(x)
}
```

5.2 Structs and Methods

Custom Types

Structs are Go's way of grouping related data. If you are coming from an object-oriented language, structs are the closest thing to classes, but without inheritance.

Defining a Struct

```
type Point struct {
    X float64
    Y float64
}
```

Creating Instances

```
// Named fields (preferred for clarity)
p1 := Point{X: 1.0, Y: 2.0}

// Positional (fragile, avoid unless struct is tiny)
p2 := Point{1.0, 2.0}

// Zero value (all fields are zero-valued)
var p3 Point // {0, 0}
```

Methods

Methods are functions attached to a type. They are declared with a *receiver* between the `func` keyword and the method name:

```
func (p Point) Distance() float64 {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}
```

Call methods with dot notation:

```
p := Point{X: 3, Y: 4}
fmt.Println(p.Distance()) // 5
```

Pointer Receivers

A value receiver gets a copy of the struct. A pointer receiver gets a reference and can modify the original:

```
func (p *Point) Scale(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

Use a pointer receiver when:

- The method needs to modify the struct

- The struct is large and copying would be expensive
- You want consistency (if any method uses a pointer receiver, all should)

Go automatically handles the conversion: you can call a pointer-receiver method on a value, and vice versa.

▷ Your Task

Define a `Rect` struct with fields `Width` and `Height` (both `float64`).
Add two methods:

- `Area()` returns the area (`Width * Height`)
- `Perimeter()` returns the perimeter (`2 * (Width + Height)`)

Starter code:

```
package main

import "fmt"

// Define your Rect struct here

// Add Area() method

// Add Perimeter() method

func main() {
    r := Rect{Width: 5, Height: 3}
    fmt.Printf("%.1f\n", r.Area())
    fmt.Printf("%.1f\n", r.Perimeter())
}
```

✓ Solution

```
package main

import "fmt"

type Rect struct {
    Width  float64
    Height float64
}

func (r Rect) Area() float64 {
    return r.Width * r.Height
}

func (r Rect) Perimeter() float64 {
    return 2 * (r.Width + r.Height)
}

func main() {
    r := Rect{Width: 5, Height: 3}
    fmt.Printf("%.1f\n", r.Area())
    fmt.Printf("%.1f\n", r.Perimeter())
}
```

5.3 Interfaces

Implicit Contracts

Interfaces in Go define behavior. An interface is a set of method signatures. Any type that implements all the methods of an interface automatically satisfies it. No `implements` keyword needed.

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

Any type with an `Area() float64` method satisfies `Shape`. The type does not even need to know the interface exists.

Why This Matters

This design means you can define interfaces *after* the concrete types are written. You can define an interface in your package that is satisfied by types from a third-party library, without modifying that library.

This is fundamentally different from Java or C# where implementing an interface is an explicit declaration. Go's approach is called *structural typing*.

Using Interfaces

Interfaces let you write functions that accept any type with the right behavior:

```
func printArea(s Shape) {
    fmt.Printf("Area: %.2f\n", s.Area())
}
```

This function works with circles, rectangles, triangles, or anything else that has an `Area()` method.

The Stringer Interface

The `fmt` package defines a commonly used interface:

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

If your type implements `String()`, the `fmt` functions will use it automatically:

```
type Point struct { X, Y int }

func (p Point) String() string {
    return fmt.Sprintf("(%d, %d)", p.X, p.Y)
}

fmt.Println(Point{1, 2}) // prints "(1, 2)"
```

The Empty Interface

The type `interface{}` (or its alias `any` since Go 1.18) has no methods, so every type satisfies it. It is Go's version of "accept anything":

```
func printAny(i any) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

Use it sparingly. Overusing `any` throws away the type safety that makes Go reliable.

Type Assertions

When you have a value of type `any` (or any interface), you can extract the underlying concrete value using a **type assertion**:

```
var i any = "hello"

s, ok := i.(string)  // s = "hello", ok = true
n, ok := i.(int)     // n = 0, ok = false
```

Always use the comma-ok form. Without it, a failed type assertion panics:

```
s := i.(string) // works
n := i.(int)    // panic: interface conversion
```

Type Switches

A **type switch** lets you branch based on the concrete type of an interface value:

```
func classify(i any) string {
    switch v := i.(type) {
    case string:
        return "string: " + v
    case int:
        return fmt.Sprintf("int: %d", v)
    case bool:
        return fmt.Sprintf("bool: %v", v)
    default:
        return "unknown"
    }
}
```

Inside each case, `v` is already the correct type — no further assertion needed. Type switches are cleaner than chains of type assertions when you need to handle multiple types.

▷ Your Task

Define a `Shape` interface with a single method: `Area() float64`.
Define two types:

- `Circle` with a `Radius float64` field
- `Square` with a `Side float64` field

Implement `Area()` on both types. The area of a circle is `math.Pi * r * r`.
Write a function `totalArea` that takes a `[]Shape` and returns the sum of all areas.

Write a function `describeShape` that takes a `Shape` and returns a string using a type switch:

- For a `Circle`, return `"circle with radius X.XX"`
- For a `Square`, return `"square with side X.XX"`
- For anything else, return `"unknown shape"`

Starter code:

```
package main

import (
    "fmt"
    "math"
)

// Define Shape interface

// Define Circle struct and Area method

// Define Square struct and Area method

// Write totalArea function

// Write describeShape function using a type switch

func main() {
    shapes := []Shape{
        Circle{Radius: 5},
        Square{Side: 3},
    }
    fmt.Printf("%.2f\n", totalArea(shapes))
    for _, s := range shapes {
        fmt.Println(describeShape(s))
    }
}
```

✓ Solution

```
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    Area() float64
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
```

```
}

type Square struct {
    Side float64
}

func (s Square) Area() float64 {
    return s.Side * s.Side
}

func totalArea(shapes []Shape) float64 {
    total := 0.0
    for _, s := range shapes {
        total += s.Area()
    }
    return total
}

func describeShape(s Shape) string {
    switch v := s.(type) {
    case Circle:
        return fmt.Sprintf("circle with radius %.2f", v.Radius)
    case Square:
        return fmt.Sprintf("square with side %.2f", v.Side)
    default:
        return "unknown shape"
    }
}

func main() {
    shapes := []Shape{
        Circle{Radius: 5},
        Square{Side: 3},
    }
    fmt.Printf("%.2f\n", totalArea(shapes))
    for _, s := range shapes {
        fmt.Println(describeShape(s))
    }
}
```

6

Error Handling

6.1 Errors as Values

Go does not have exceptions. Instead, functions that can fail return an error value alongside their result. This is arguably Go's most important design decision.

The error Interface

The built-in `error` type is an interface with a single method:

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

Any type that has an `Error() string` method is an error. This is the simplest possible contract.

Creating Errors

The standard library provides two ways to create simple errors:

```
import "errors"

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

```
import "fmt"

err := fmt.Errorf("user %s not found", username)
```

`fmt.Errorf` works like `fmt.Sprintf` but returns an error. Use it when you need formatted messages.

The Error-Checking Pattern

The canonical Go pattern: call a function, check the error immediately, handle it or return it:

```
result, err := doSomething()
if err != nil {
    return fmt.Errorf("doSomething failed: %w", err)
}
// use result
```

The `%w` verb wraps the original error, preserving the chain for debugging. This pattern appears hundreds of times in any real Go codebase.

Custom Error Types

For richer error information, define your own error type:

```
type ValidationError struct {
    Field   string
    Message string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", e.Field, e.Message)
}
```

When to Return Errors

A function should return an error when:

- An operation can fail (file I/O, network, parsing)
- The failure is recoverable (the caller can do something about it)
- The failure is expected in normal operation

Do not return errors for programming mistakes (like passing a nil pointer where one is never expected). Use panics for those.

▷ Your Task

Write a function `validateAge` that takes an `int` and returns an `error`:
- If age is negative, return an error with the message `"age cannot be negative"`
- If age is greater than 150, return an error with the message `"age is unrealistic"`
- Otherwise, return `nil` (no error)

Starter code:

```
package main

import (
    "errors"
    "fmt"
)

func validateAge(age int) error {
    // Your code here
    return nil
}
```

```
func main() {
    for _, age := range []int{25, -1, 200} {
        if err := validateAge(age); err != nil {
            fmt.Println(err)
        } else {
            fmt.Println("valid")
        }
    }
}
```

✓ Solution

```
package main

import (
    "errors"
    "fmt"
)

func validateAge(age int) error {
    if age < 0 {
        return errors.New("age cannot be negative")
    }
    if age > 150 {
        return errors.New("age is unrealistic")
    }
    return nil
}

func main() {
    for _, age := range []int{25, -1, 200} {
        if err := validateAge(age); err != nil {
            fmt.Println(err)
        } else {
            fmt.Println("valid")
        }
    }
}
```

7

Generics

7.1 Generic Functions

Type Parameters

Go 1.18 introduced generics, allowing you to write functions and types that work with any type. Before generics, you had to write separate functions for each type or use `interface{}` and lose type safety.

Syntax

A generic function declares one or more *type parameters* in square brackets before the regular parameters:

```
func Print[T any](val T) {
    fmt.Println(val)
}
```

`T` is a type parameter. `any` is a *constraint* that means "any type at all". The function can be called with any type:

```
Print[int](42)
Print[string]("hello")
```

Type Inference

In most cases the compiler can figure out the type argument from the regular arguments, so you can omit it:

```
Print(42)       // T inferred as int
Print("hello")  // T inferred as string
```

Constraints

Constraints restrict which types a type parameter can accept. The `any` constraint allows everything. The `comparable` constraint allows types that support `==` and `!=`:

```go
func Contains[T comparable](slice []T, target T) bool {
    for _, v := range slice {
        if v == target {
            return true
        }
    }
    return false
}
```

You can also define your own constraint interfaces:

```go
type Number interface {
    int | float64
}

func Sum[T Number](nums []T) T {
    var total T
    for _, n := range nums {
        total += n
    }
    return total
}
```

Multiple Type Parameters

A function can have more than one type parameter:

```go
func Map[T any, U any](slice []T, f func(T) U) []U {
    result := make([]U, len(slice))
    for i, v := range slice {
        result[i] = f(v)
    }
    return result
}
```

▷ Your Task

Write a generic function `Filter[T any]` that takes a `[]T` and a `func(T) bool` predicate. It should return a new `[]T` containing only the elements for which the predicate returns `true`.

Starter code:

```go
package main

import "fmt"

func Filter[T any](slice []T, predicate func(T) bool) []T {
    // Your code here
    return nil
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6}
    evens := Filter(nums, func(n int) bool {
        return n%2 == 0
    })
```

```
    fmt.Println(evens)

    words := []string{"hi", "hello", "hey", "greetings"}
    short := Filter(words, func(s string) bool {
        return len(s) <= 3
    })
    fmt.Println(short)
}
```

✓ Solution

```
package main

import "fmt"

func Filter[T any](slice []T, predicate func(T) bool) []T {
    var result []T
    for _, v := range slice {
        if predicate(v) {
            result = append(result, v)
        }
    }
    return result
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6}
    evens := Filter(nums, func(n int) bool {
        return n%2 == 0
    })
    fmt.Println(evens)

    words := []string{"hi", "hello", "hey", "greetings"}
    short := Filter(words, func(s string) bool {
        return len(s) <= 3
    })
    fmt.Println(short)
}
```

7.2 Generic Types

Generic Structs

Just like functions, struct types can have type parameters. This lets you build reusable data structures that work with any type.

```
type Pair[T any, U any] struct {
    First  T
    Second U
}
```

You create instances by specifying the type arguments:

```
p := Pair[string, int]{First: "age", Second: 30}
```

Methods on Generic Types

Methods on a generic type must redeclare the type parameters in the receiver, but they cannot introduce new ones:

```
func (p Pair[T, U]) Swap() Pair[U, T] {
    return Pair[U, T]{First: p.Second, Second: p.First}
}
```

A Generic Container

Here is a practical example: a simple linked list:

```
type Node[T any] struct {
    Value T
    Next  *Node[T]
}

func (n *Node[T]) Append(val T) {
    current := n
    for current.Next != nil {
        current = current.Next
    }
    current.Next = &Node[T]{Value: val}
}
```

Type Constraint Interfaces

You can define interfaces that constrain type parameters to types supporting specific operations:

```
type Number interface {
    int | int8 | int16 | int32 | int64 | float32 | float64
}

type Stats[T Number] struct {
    Values []T
}
```

```
func (s Stats[T]) Sum() T {
    var total T
    for _, v := range s.Values {
        total += v
    }
    return total
}
```

▷ Your Task

Implement a generic `Stack[T any]` struct backed by a slice. It should have three methods:

- `Push(val T)` — adds a value to the top of the stack
- `Pop() (T, bool)` — removes and returns the top value; returns the zero value of `T` and `false` if the stack is empty
- `Peek() (T, bool)` — returns the top value without removing it; returns the zero value of `T` and `false` if the stack is empty

Starter code:

```
package main

import "fmt"

type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(val T) {
    // Your code here
}

func (s *Stack[T]) Pop() (T, bool) {
    // Your code here
    var zero T
    return zero, false
}

func (s *Stack[T]) Peek() (T, bool) {
    // Your code here
    var zero T
    return zero, false
}

func main() {
    s := &Stack[int]{}
    s.Push(10)
    s.Push(20)
    s.Push(30)

    val, ok := s.Peek()
    fmt.Printf("Peek: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)
```

```
    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)
}
```

✓ Solution

```
package main

import "fmt"

type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(val T) {
    s.items = append(s.items, val)
}

func (s *Stack[T]) Pop() (T, bool) {
    if len(s.items) == 0 {
        var zero T
        return zero, false
    }
    val := s.items[len(s.items)-1]
    s.items = s.items[:len(s.items)-1]
    return val, true
}

func (s *Stack[T]) Peek() (T, bool) {
    if len(s.items) == 0 {
        var zero T
        return zero, false
    }
    return s.items[len(s.items)-1], true
}

func main() {
    s := &Stack[int]{}
    s.Push(10)
    s.Push(20)
    s.Push(30)

    val, ok := s.Peek()
    fmt.Printf("Peek: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)
```

```
    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)

    val, ok = s.Pop()
    fmt.Printf("Pop: %d (%v)\n", val, ok)
}
```

8

Concurrency

8.1 Goroutines and Channels

Concurrency in Go

Concurrency is one of Go's defining features. Go makes it easy to run functions concurrently and communicate between them safely.

Goroutines

A goroutine is a lightweight thread managed by the Go runtime. You start one by putting the `go` keyword before a function call:

```
go doSomething()
```

That is it. The function runs concurrently with the rest of your program. Goroutines are extremely cheap: you can launch thousands without concern.

```
func printNumbers() {
    for i := 1; i <= 5; i++ {
        fmt.Println(i)
    }
}

func main() {
    go printNumbers() // runs concurrently
    fmt.Println("started")
    time.Sleep(time.Second) // wait for goroutine
}
```

Channels

Channels are Go's way of communicating between goroutines. A channel is a typed conduit through which you send and receive values:

```
ch := make(chan int)  // create a channel of ints

go func() {
```

```
    ch <- 42  // send a value into the channel
}()

val := <-ch  // receive a value from the channel
fmt.Println(val) // 42
```

The `<-` operator is used for both sending and receiving. Sends and receives block until the other side is ready, which naturally synchronizes goroutines.

Channel Direction

Function signatures can restrict a channel to send-only or receive-only:

```
func producer(ch chan<- int) {  // can only send to ch
    ch <- 42
}

func consumer(ch <-chan int) {  // can only receive from ch
    val := <-ch
    fmt.Println(val)
}
```

This makes your intent clear and catches mistakes at compile time.

Closing Channels and Range

A sender can close a channel to signal that no more values will be sent. The receiver can detect this:

```
ch := make(chan int)

go func() {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch)
}()

for val := range ch {
    fmt.Println(val)
}
```

The `range` loop receives values from the channel until it is closed. This is the cleanest way to consume all values from a channel.

Buffered Channels

By default, channels are *unbuffered*: a send blocks until another goroutine receives. A **buffered channel** has a capacity, allowing sends to proceed without a receiver until the buffer is full:

```
ch := make(chan int, 3)  // buffer holds up to 3 values

ch <- 1  // does not block
ch <- 2  // does not block
ch <- 3  // does not block
// ch <- 4 would block here (buffer full)
```

```
fmt.Println(<-ch) // 1
```

Buffered channels are useful when the sender and receiver run at different speeds, or when you know the exact number of values that will be sent.

Synchronization with Channels

When you need to wait for a goroutine to finish, you can use a channel as a signal:

```
done := make(chan bool)

go func() {
    fmt.Println("working...")
    done <- true  // signal completion
}()

<-done  // wait for the goroutine to finish
```

For waiting on multiple goroutines, use `sync.WaitGroup`. Call `Add` before launching, `Done` when each goroutine finishes, and `Wait` to block until all are done:

```
var wg sync.WaitGroup
for i := 0; i < 3; i++ {
    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        fmt.Println(n)
    }(i)
}
wg.Wait() // blocks until all goroutines call Done
```

▷ Your Task

Write a function `squares(n int) <-chan int` that returns a receive-only channel. It should start a goroutine that sends the squares of 1 through `n` into the channel, then closes it.

Starter code:

```
package main

import "fmt"

func squares(n int) <-chan int {
    // Your code here
    return nil
}

func main() {
    for val := range squares(5) {
        fmt.Println(val)
    }
}
```

✓ **Solution**

```go
package main

import "fmt"

func squares(n int) <-chan int {
    ch := make(chan int)
    go func() {
        for i := 1; i <= n; i++ {
            ch <- i * i
        }
        close(ch)
    }()
    return ch
}

func main() {
    for val := range squares(5) {
        fmt.Println(val)
    }
}
```

8.2 Select

Multiplexing Channels

The `select` statement lets a goroutine wait on multiple channel operations simultaneously. It is like a `switch` statement, but each case is a channel send or receive.

```
select {
case msg := <-ch1:
    fmt.Println("from ch1:", msg)
case msg := <-ch2:
    fmt.Println("from ch2:", msg)
}
```

If multiple channels are ready, `select` picks one at random. If none are ready, it blocks until one becomes ready.

Non-Blocking Operations

Add a `default` case to make channel operations non-blocking:

```
select {
case msg := <-ch:
    fmt.Println("received:", msg)
default:
    fmt.Println("no message available")
}
```

Without `default`, the `select` blocks. With `default`, it executes the default case immediately if no channel is ready.

Pattern: Merging Channels

A common use of `select` is merging multiple channels into one. This lets a consumer read from a single channel regardless of how many producers exist:

```
func merge(ch1, ch2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        defer close(out)
        for ch1 != nil || ch2 != nil {
            select {
            case v, ok := <-ch1:
                if !ok { ch1 = nil; continue }
                out <- v
            case v, ok := <-ch2:
                if !ok { ch2 = nil; continue }
                out <- v
            }
        }
    }()
    return out
}
```

Setting a channel to `nil` after it closes prevents `select` from receiving zero values from it, because receives on a nil channel block forever.

▷ Your Task

Write a function `merge(ch1, ch2 <-chan int) <-chan int` that merges two integer channels into one. The output channel should receive all values from both input channels and close when both inputs are exhausted.

Key steps:

1. Create an output channel and launch a goroutine
2. Loop while at least one input channel is still open
3. Use `select` to receive from whichever channel is ready
4. When a channel is closed (`ok` is false), set it to `nil` so `select` ignores it
5. Close the output channel when both inputs are exhausted

Starter code:

```
package main

import "fmt"

func merge(ch1, ch2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        // Loop while ch1 or ch2 is still open.
        // Use select to receive from whichever is ready.
        // When ok is false, set the channel to nil and continue.
    }()
    return out
}

func main() {
    a := make(chan int)
    b := make(chan int)

    go func() {
        for _, v := range []int{1, 3, 5} {
            a <- v
        }
        close(a)
    }()

    go func() {
        for _, v := range []int{2, 4, 6} {
            b <- v
        }
        close(b)
    }()

    for v := range merge(a, b) {
        fmt.Println(v)
    }
}
```

✓ Solution

```go
package main

import "fmt"

func merge(ch1, ch2 <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for ch1 != nil || ch2 != nil {
            select {
            case v, ok := <-ch1:
                if !ok {
                    ch1 = nil
                    continue
                }
                out <- v
            case v, ok := <-ch2:
                if !ok {
                    ch2 = nil
                    continue
                }
                out <- v
            }
        }
    }()
    return out
}

func main() {
    a := make(chan int)
    b := make(chan int)

    go func() {
        for _, v := range []int{1, 3, 5} {
            a <- v
        }
        close(a)
    }()

    go func() {
        for _, v := range []int{2, 4, 6} {
            b <- v
        }
        close(b)
    }()

    for v := range merge(a, b) {
        fmt.Println(v)
    }
}
```

What's Next?

Congratulations on completing all seventeen lessons! You now have a solid foundation in Go's core language features. Here are some directions to explore next:

The Standard Library

Go ships with a comprehensive standard library. Key packages to explore:

- `net/http` — build web servers and HTTP clients
- `encoding/json` — encode and decode JSON
- `os` and `io` — file system and I/O operations
- `testing` — write and run tests
- `context` — manage cancellation and timeouts

Build Something

The best way to learn is to build. Some project ideas:

- A command-line tool (file organizer, task tracker, URL shortener)
- A REST API with `net/http`
- A concurrent web scraper
- A chat server using WebSockets

References

Here are the best resources for continuing your Go journey:

- **The Go Programming Language Specification** — `https://go.dev/ref/spec`
- **Effective Go** — `https://go.dev/doc/effective_go`
- **Go by Example** — `https://gobyexample.com`
- **The Go Blog** — `https://go.dev/blog`
- **The Go Playground** — `https://go.dev/play`
- ***The Go Programming Language*** by Alan Donovan and Brian Kernighan (Addison-Wesley, 2015)
- **Go Standard Library Documentation** — `https://pkg.go.dev/std`

Built with Hypercode
https://hypercode.alexisbouchez.com