Gorgo Go for Java Programmers

June 11, 2026

Contents

2 Types and Variables **1**
Basic Types . . . 1
`var` Declarations and `:=` . . . 2
Zero Values . . . 3
Integer Literal Prefixes and the Digit Separator . . . 4
Constants . . . 4
Type Casts . . . 6
Type Definitions . . . 6
Type Aliases . . . 7
`new` and `make` . . . 7
Built-in `min`, `max`, and `clear` (Go 1.21) . . . 8
The Blank Identifier . . . 8
Structs . . . 8
Pointers . . . 10
Try It . . . 11
Key Points . . . 12
Exercises . . . 12

Chapter 2

Types and Variables

Go's type system will feel familiar to a Java programmer but has several sharp edges: every variable has a meaningful zero value from birth, numeric types never coerce implicitly, and the capitalization of one letter can change a type from private to public. This chapter covers the basic types, how to declare variables, constants, the handful of built-in functions that round out the type system, and structs — Go's primary mechanism for grouping data, attaching behavior, and composing types.

Basic Types

Go's primitive types map roughly to Java's, with a few differences worth noting.

Integer Types

Go type	Width	Java equivalent
`int`	Platform-native (32 or 64 bit)	`int` (always 32 bit in Java)
`int8`	8 bit	`byte`
`int16`	16 bit	`short`
`int32`	32 bit	`int`
`int64`	64 bit	`long`
`uint`, `uint8`, `uint16`, `uint32`, `uint64`	Unsigned variants	No direct equivalent

`int` is the idiomatic integer type for general use. Its width matches the platform's native word size — 64 bits on every modern system you will care about.

Wut: In Java, `int` is always 32 bits. In Go, `int` can be 32 or 64 bits depending on the platform. If you need a guaranteed 32-bit integer, use `int32`.

Floating-Point Types

`float32` and `float64` correspond to Java's `float` and `double`. The same as Java: prefer `float64` for most uses.

Boolean and String

`bool` and `string` are the same concept as Java. In Go, strings are **immutable byte sequences** (not character sequences — that distinction matters a lot and gets its own chapter). `true` and `false` are the only bool literals, same as Java.

`byte` and `rune`

`byte` is an alias for `uint8`. `rune` is an alias for `int32` and represents a Unicode code point. These two types come up constantly when working with text.

```
var b byte = 'A'   // b == 65
var r rune = '🎵'  // r == 127925
```

A full treatment of strings, bytes, and runes is in Chapter 3.

`var` Declarations and `:=`

Go has two ways to declare a variable.

`var`

```
var name string
var count int = 0
var ratio float64 = 1.5
```

`var` works anywhere — inside or outside a function. When you provide an initializer, the type is optional:

```
var title = "Flaming June"  // type inferred as string
```

`:=` Short Declaration

Inside a function body you can use the short declaration operator:

```
artist := "BT"
streams := 4_200_000
```

The compiler infers the type from the right-hand side. `:=` is the idiomatic choice inside functions; `var` is preferred for package-level variables and when you want to declare a variable without an initial value.

Multiple variables can appear on the left side, which is how Go handles functions that return more than one value:

```
title, artist := "Flaming June", "BT"  // two new string variables
n, err := fmt.Println("hello")         // int and error from one call
```

Each variable on the left gets the corresponding value from the right. You will see this form constantly when calling functions that return a result alongside an error.

Trap: `:=` is only valid **inside a function**. Using it at package level is a syntax error.

When to Use Each

Use `:=` when you have an initializer and you are inside a function — it is shorter and more readable. Use `var` when you want to declare a variable at its zero value, when you need a package-level variable, or when the explicit type improves clarity.

Redeclaration with :=

`:=` can appear on the left side with variables that already exist in the current scope, **as long as at least one variable on the left is new**. The following example shows how you can reuse `err` across a chain of calls:

```
n, err := fmt.Println("first")
m, err := fmt.Println("second")  // err already declared; m is new --- OK
n, err := fmt.Println("third")   // error because n and err are already declared
```

If every variable on the left already exists in scope, `:=` is a compile error.

Zero Values

In Java, reading a local variable before you initialize it is a compile error. In Go, every type has a **zero value** that variables are initialized to automatically.

Type	Zero value
`int`, `int8` ... `int64`	`0`
`uint`, `uint8` ... `uint64`	`0`
`float32`, `float64`	`0.0`
`bool`	`false`
`string`	`""` (empty string)
Pointer, slice, map, channel, function, interface	`nil`

```
var count int       // 0
var name string     // ""
var active bool     // false
```

Zero values make it safe to use a variable before assigning to it. Structs are zero-valued field by field.

nil

`nil` is the zero value for the six reference-like type categories listed above. It is Go's counterpart to Java's `null`, but with one important difference: `nil` is often safe and useful rather than a source of panic. A nil slice has length zero and works correctly with `len`, `range`, and `append`. [***nil-slice-preferred***] A nil map can be read from (the result is always the zero value for the value type). These behaviors are covered in detail when slices and maps (both Chapter 7) are introduced.

Dereferencing a nil pointer, though, always panics — the Go counterpart to Java's `NullPointerException` (writing to a nil map panics too — see Chapter 7):

```
var p *int
fmt.Println(p)  // <nil>
fmt.Println(*p) // panic: runtime error: invalid memory address or nil pointer dereference
```

Tip: Design your types so the zero value is useful. A `strings.Builder` at its zero value is an empty, ready-to-use builder — no constructor call needed (covered in Chapter 3).

Wut: Java's `null` can be assigned to any reference type and causes a `NullPointerException` on any access. Go's `nil` is type-specific: a nil slice or map is genuinely usable, whereas a nil pointer panics on dereference.

Integer Literal Prefixes and the Digit Separator

Go supports nearly the same literal prefix syntax you may know from modern Java — plus the `0o` octal prefix, which Java lacks:

```
bin  := 0b1010_1100   // binary, underscores allowed
oct  := 0o755         // octal; 0o prefix is clearer, but leading-zero (0755) still works
hex  := 0xDEAD_BEEF   // hexadecimal
big  := 1_000_000     // one million, easier to read
```

The underscore _ is a digit separator — it is ignored by the compiler and exists only for readability. You can place it anywhere inside a numeric literal, but not at the start or end.

> **Tip:** Go 1.13 added the `0o` prefix as an unambiguous way to write octal literals. The old leading-zero form (`010` = 8) still works, exactly as in Java. Prefer `0o` in new code — a reader can't misread `0o10` as decimal ten.

Constants

Constants in Go are declared with `const`. Unlike Java, Go constants are limited to booleans, numbers (integers, floats, complex, and runes), and strings — no arrays, slices, structs, or any other composite type. They can be typed or untyped:

```
const Pi float64 = 3.14159   // typed constant
const E  = 2.71828           // untyped floating-point constant
```

Untyped constants are precise and flexible — the compiler treats them as arbitrary-precision numbers and assigns a concrete type at the point of use.

```
const Streams = 1_200_000_000  // untyped integer constant

var plays32 int32   = Streams  // concrete type: int32
var plays64 int64   = Streams  // same constant, now int64
var ratio   float64 = Streams  // same constant, now float64
```

A typed constant like `Pi float64` can only be used where `float64` is expected. `Streams` works as any numeric type that can represent its value.

> **Wut:** Java's `static final` fields let you name a constant of any type — arrays, lists, custom objects, whatever. Go's `const` is strictly for primitive values: booleans, numbers, and strings. `const Primes = []int{2, 3, 5, 7}` is a compile error. For a named, package-level slice or array you'd reach for `var`, accepting that it is technically mutable.

`iota` and const blocks

Constants can also be declared in a block. For example:

```
const (
    Rock = 1
    Pop = 2
    HipHop = 3
    Reggaeton = 4
)
```

but what about this code?

```go
const (
    Rock = 1
    Pop
    HipHop
    Reggaeton
)
```

You would think this would be a syntax error or the constants would be set to the same values as the previous example. It turns out that there is a third option. The Go Language spec states that if the initialization expression is missing for a constant in a const block, the expression used will be "equivalent textually to the substitution of the first preceding non-empty expression list and its type if any" (The Go Authors 2025, sec. "Constant declarations"). That means the following code will print all ones.

```go
fmt.Printf("%v %v %v %v", Rock, Pop, HipHop, Reggaeton)
```

So, why the `textual substitution`? Why give one iota about that rule? Read on!

`iota` is an untyped integer constant whose value is the position of a constant specification within a `const` block, starting at zero.

It is Go's idiomatic way to define enumerations.

```go
const (
    Free     = iota  // 0
    Standard = iota  // 1
    Premium  = iota  // 2
    Lossless = iota  // 3
)
```

We can take advantage of the textual substitution rule to simplify the preceding example:

```go
const (
    Free     = iota  // 0
    Standard         // 1
    Premium          // 2
    Lossless         // 3
)
```

The `iota` will be implicitly copied to `Standard`, `Premium`, and `Lossless`.

You can use `iota` in expressions to offset the starting value:

```go
const (
    Debut     = iota + 1  // 1
    Sophomore             // 2
    Certified             // 3
)
```

`iota` resets to zero at each new `const` block.

Tip: Skip value zero with `iota + 1` when zero should represent "unset" or "unknown." This lets you detect a variable that was declared but never assigned a meaningful position.

Wut: You can also have comma-separated const identifiers with corresponding comma-separated initializers. They are not common because they mess with formatting, and `iota` doesn't increment between comma-separated identifiers.

Type Casts

Go has no implicit numeric widening. In Java, assigning an `int` to a `long` variable just works; in Go, every conversion between different types must use a cast. Java casts look like `int i = (int) myLong`, but Go uses a functional form of the cast using the `T(value)` syntax.

```
var i int     = 42
var f float64 = float64(i)   // required: int → float64
var u uint    = uint(f)      // required: float64 → uint
```

This applies to all numeric types — `int32` to `int64`, `float32` to `float64`, `int` to `byte`, and so on. The compiler rejects any assignment that silently changes the representation.

These conversions are fully type-checked at compile time, with no runtime type test involved: unlike Java's cast, `T(value)` can never fail at run time (though converting a non-constant value still executes an instruction). Java overloads the word *cast* for two very different operations — the compile-time `(int) myLong` and the runtime `(String) obj`, where the JVM checks the actual type and may throw `ClassCastException`. Go splits these apart. Compile-time conversions use `T(value)` and are called *casts* or *conversions*; the runtime operation of checking whether an interface value holds a particular concrete type or interface is a separate construct called a *type assertion* (covered in the interfaces chapter), written `v.(T)`.

Wut: Java widens automatically: `long x = someInt` compiles without complaint. Go requires `int64(someInt)` — every time, with no exceptions. The upside: you can always tell from reading the code exactly what conversions are happening.

When you need a cast:

- Any numeric type to any other numeric type: `float64(n)`, `int32(x)`, `byte(r)`
- `string` to `[]byte` or `[]rune`, and back: `[]byte(s)`, `string(b)`
- Between a named type and its underlying type (covered in Type Definitions below): `float64(c)`

When you do not need a cast:

- Assigning an untyped constant to any compatible type — `var x float64 = 42` works because `42` is an untyped constant that fits `float64`
- Passing a value to an interface parameter — any type that satisfies the interface is accepted automatically, no cast required
- Between a type and its alias (covered in Type Aliases below)

Trap: Narrowing conversions of *non-constant* values are silent. Given `f := 3.9`, `int(f)` truncates to `3`, and given `n := 300`, `byte(n)` wraps to `44` — no panic, no error. (Converting the literals directly, as in `int(3.9)` or `byte(300)`, is instead a compile error: the compiler checks constant conversions and rejects ones that lose information or overflow.) Go trusts you to know what you are doing when you write an explicit conversion on a variable.

Type Definitions

```
type Celsius float64
type Fahrenheit float64
```

`type Celsius float64` creates a **new, distinct type**. Even though both `Celsius` and `float64` have the same underlying representation, they are different types. Assignment between them requires an explicit conversion:

```
var c Celsius    = 100.0
var f float64    = float64(c)   // explicit conversion required
var g float64    = c            // compile error: cannot use c (Celsius) as float64
```

This is intentional. You cannot accidentally pass a temperature in Fahrenheit where the function expects Celsius.

Type Aliases

```go
type Seconds = float64  // alias, not a new type
```

An alias introduces a new name for the same type. `Seconds` and `float64` are interchangeable — there is no conversion needed. Aliases are most useful in large-scale refactoring or when bridging packages.

> **Wut:** `type Celsius float64` and `type Celsius = float64` look almost identical but mean opposite things. Without `=` you get a new type with conversion rules. With `=` you get a synonym.

`new` and `make`

Go has two allocation built-ins. They serve different purposes.

`new`

`new(T)` allocates memory for a value of type `T`, initializes it to the zero value, and returns a `*T`. Conceptually, the memory is allocated on the heap — a region of memory that lives on even after the local scope (block of code) finishes. In reality, Go does *escape analysis* to see if there is a chance the allocation will be used outside of the local scope. If not, the memory will be allocated on the stack and will be deallocated once the scope — usually the function or block of code — finishes. If the reference will be used outside of the local scope — it was passed to another function or added to a collection object — the compiler will allocate the memory on the heap where it will live until there are no more references to it.

```go
p := new(int)    // *int pointing to 0
*p = 42
fmt.Println(*p)  // 42
```

You will see `new` occasionally, but struct literals with `&` are more common in practice:

```go
type Point struct{ X, Y int }
pt := &Point{X: 3, Y: 4}  // same idea; more idiomatic for structs
```

If you've worked in other languages like C or C++, you will see that as a potential for a dangling pointer.

```go
func GenPoint() *Point {
    p := Point{X: 3, Y: 4}
    return &p
}
```

However, with escape analysis, Go is able to see that p needs to survive past the return of the function and allocates p on the heap.

You cannot use `new` to create a ready-to-use map or channel — `new(map[string]int)` gives you a pointer to a nil map.

> **Tip:** `new` is for scalar types when you need a pointer. `make` is for slices, maps, and channels.

`make` initializes slices, maps, and channels — the three built-in reference types that need internal setup before use. The details are covered alongside each type: maps and slices in Chapter 7, channels in Chapter 10.

Built-in `min`, `max`, and `clear` (Go 1.21)

Go 1.21 added three built-ins that Java programmers typically import from utility libraries.

`min` and `max`

`min` and `max` are type-safe and variadic — they work on any ordered type and accept any number of arguments:

```
smallest := min(3, 1, 4, 1, 5)   // 1
largest  := max(3, 1, 4, 1, 5)   // 5
lo       := min(a, b)            // works for float64, string, etc.
```

No `Math.min(a, b)` gymnastics.

`clear`

`clear` zeroes the elements of a slice or deletes all entries from a map:

```
nums := []int{1, 2, 3}
clear(nums)        // nums is now [0, 0, 0], len unchanged

scores := map[string]int{"DJ Analyzer": 99}
clear(scores)      // scores is now an empty map
```

For slices, `clear` zeroes elements but does not change the length. For maps, it removes all keys.

The Blank Identifier

The blank identifier _ discards any value you do not need.

```
n, _ := fmt.Println("Hola")  // keep the byte count, discard the error
```

Go requires you to use every declared variable, so _ is your escape hatch when a function returns multiple values and you only need some of them. You can also use it in a `for range` loop to discard the index or value.

Tip: If you find yourself using _ for every return value from a function, that is a signal you should reconsider whether you need to call the function at all.

Structs

A struct is a composite type that groups named fields together. In Java, you use a class; in Go, you use a struct. The critical difference is that structs are **value types** in Go — assigning a struct copies all its fields — whereas in Java every object variable is a reference.

Struct Definition

Define a struct type with `type T struct { ... }`:

```
type Track struct {
    Title    string
    Artist   string
    BPM      int
    Duration float64 // seconds
}
```

The zero value of a `Track` has `Title = ""`, `Artist = ""`, `BPM = 0`, and `Duration = 0.0`. You can use a zero-value struct immediately; there is no constructor required.

Fields can have the same type listed once, separated by commas:

```
type Point struct {
    X, Y float64 // two fields, both float64
}
```

Struct Literals

There are two forms of struct literal.

Named form (preferred):

```
t := Track{
    Title:    "Gouryella",
    Artist:   "Gouryella",
    BPM:      138,
    Duration: 208.0,
}
```

Positional form (fragile):

```
t := Track{"Gouryella", "Gouryella", 138, 208.0}
```

> **Trap:** The positional form breaks silently if you ever add, remove, or reorder fields. The named form is explicit and resilient to future changes. Use positional form only for tiny, stable types like `Point{1.0, 2.0}` where the meaning is immediately obvious.

Anonymous structs are useful for one-off data shapes, especially in tests and temporary groupings:

```
entry := struct {
    Title  string
    Plays  int
}{"Out Of The Blue", 1_800_000_000}

fmt.Println(entry.Title, entry.Plays)
```

Struct as a Value Type

Assigning a struct creates an independent copy:

```
a := Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 126}
b := a           // b is a copy; a and b are independent
b.BPM = 130
fmt.Println(a.BPM) // 126 --- a is unchanged
fmt.Println(b.BPM) // 130
```

In Java, `b = a` for an object would make both variables point to the same object. In Go, you get two distinct structs.

When you want shared mutation — so that changes through one variable are visible through another — use a pointer:

```
pa := &Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 126}
pb := pa           // both point to the same Track
pb.BPM = 130
fmt.Println(pa.BPM) // 130 --- shared!
```

Tip: Passing large structs to functions by value copies every field. For structs with many fields or large fields, pass a pointer (`*Track`) to avoid the copy cost and to allow the function to mutate the original. The rule of thumb from Chapter 6 applies: use pointer receivers when the struct is large or when mutation is required. [***no-mixed-receivers***] and [***default-pointer-receiver***]

Pointers

Java hides pointers — every object variable is secretly a reference to heap memory, but the language never lets you see or manipulate the raw address. Go makes pointers explicit: `&` takes an address and `*` follows it. Understanding Go pointers helps you predict when a function can modify its caller's data, and why some methods need a `*T` receiver rather than a plain `T` receiver.

& and * — Address-Of and Dereference

Two operators are all you need to work with Go pointers.

```
&expr   // address-of: produces a pointer to expr
*pExpr  // dereference: follows the pointer to reach the value
```

A **pointer type** names what the pointer points to, with a `*` prefix:

```
var p *int      // p is a pointer to int; its zero value is nil
var s *string   // s is a pointer to string; also nil
```

Here is the whole idea in one small program:

```
package main

import "fmt"

func main() {
    x := 42
    p := &x          // p holds the address of x
    fmt.Println(p)   // something like 0xc0000b4008
    fmt.Println(*p)  // 42 --- dereference to get the value
    *p = 99          // write through the pointer
    fmt.Println(x)   // 99 --- x changed because *p and x are the same memory
}
```

The zero value of any pointer type is `nil`. Dereferencing a `nil` pointer causes a runtime panic — the same kind of `NullPointerException` you know from Java, just with a different name.

```
var p *int
fmt.Println(p)  // <nil>
fmt.Println(*p) // panic: runtime error: invalid memory address or nil pointer dereference
```

Wut: In Java, every object variable is secretly a reference (pointer) to heap memory, but the language provides no syntax to get or store the raw address. In Go, pointers are explicit: you use `&` to take an address and `*` to follow it. The Java "hidden pointer" and the Go pointer are the same idea — Go just shows you the machinery.

Pointers to Basic Types

Java programmers sometimes wonder why you would ever have a pointer to an `int`. In Java, primitive `int` is always a value, and you box it into `Integer` when you need reference semantics. In Go you use `*int` directly

— no boxing, no wrapper class.

```go
func newInt(n int) *int {
    v := n
    return &v // safe: compiler moves v to the heap if needed (escape analysis, see new)
}

p := newInt(7)
fmt.Println(*p) // 7
```

No Pointer Arithmetic

In C and C++, you can write `ptr + 1` to advance a pointer to the next element in an array. Go forbids this entirely.

```go
x := 42
p := &x
p++      // compile error: invalid operation: p++ (non-numeric type *int)
p + 1    // compile error: invalid operation: p + 1 (mismatched types *int and untyped int)
```

Tip: No pointer arithmetic means the compiler and garbage collector always know exactly what a pointer points to. It also eliminates an entire class of security vulnerabilities (buffer overruns, out-of-bounds reads) that plague C codebases. If you need to walk through elements, use a slice — slices carry length and capacity and support range iteration safely.

When you genuinely need unsafe pointer arithmetic (for C interop or performance-critical operations), the `unsafe` package exists, but it earns its name. That is an advanced topic far outside normal day-to-day Go.

Try It

Type this in and run it. It exercises most of the chapter at once: a typed `iota` enum, the digit separator, a numeric conversion, a struct copied by value, a pointer that mutates through `*`, and the built-in `max`. Watch how the boosted copy diverges from the original while the original stays put.

```go
package main

import "fmt"

type Genre int

const (
    House Genre = iota + 1 // 1
    Trance                 // 2
    Techno                 // 3
)

type Track struct {
    Title  string
    Artist string
    BPM    int
    Genre  Genre
}

func boost(t *Track, by int) {
    t.BPM += by // mutate through the pointer
```

```go
}

func main() {
    var plays int // zero value: 0
    t := Track{   // named struct literal
        Title:  "Eternity",
        Artist: "Anyma & Chris Avantgarde",
        BPM:    124,
        Genre:  Trance,
    }

    copyOfT := t // structs are value types --- this copies
    boost(&copyOfT, 4)

    plays = 2_500_000 // digit separator for readability
    rate := float64(plays) / 60.0
    fmt.Printf("%s by %s [genre %d]\n", t.Title, t.Artist, t.Genre)
    fmt.Println("original BPM:", t.BPM, "boosted copy BPM:", copyOfT.BPM)
    fmt.Printf("plays/min: %.1f, faster of (124,128): %d\n", rate, max(124, 128))
}
```

Try these modifications:

- Pass `t` to `boost` by value instead of by pointer (`boost(t Track, by int)`) and watch the change vanish.
- Add a `Lossless` quality enum as a second `iota` block and print where it resets to zero.
- Drop the `float64(plays)` conversion and see the compiler reject the implicit `int`-to-`float64` mix.

Key Points

- Go has sized integer types (`int8` through `int64`, unsigned variants); `int` is the idiomatic general-purpose integer and is platform-native width.
- `byte` is an alias for `uint8`; `rune` is an alias for `int32`.
- Declare variables with `var` (anywhere) or `:=` (inside functions only).
- Every type has a zero value; Go never leaves a variable uninitialized.
- Integer literals support `0b` (binary), `0o` (octal), `0x` (hex), and `_` as a digit separator.
- `const` and `iota` are Go's enumeration mechanism; `iota` counts from zero per `const` block.
- `type Celsius float64` creates a new distinct type; `type Celsius = float64` is a synonym.
- `new(T)` returns a zeroed `*T`; `make` initializes slices, maps, and channels.
- `min`, `max`, and `clear` are built-in since Go 1.21.
- `_` discards values you do not need; every declared variable must otherwise be used.
- Structs are value types; assigning a struct copies it; use pointers for shared mutation.
- The `cmp` and `maps` packages (Go 1.21) provide comparison and map utilities; `cmp` is introduced in Chapter 7, and `maps` in Chapter 14.

Exercises

1. **Think about it:** Go's zero values mean a declared-but-unassigned variable is always valid. Java requires local variables to be assigned before use. What are the practical benefits of Go's approach? Can you think of a case where Go's zero values might mask a bug instead of preventing one?

2. **What does this print?**

   ```go
   package main
   ```

```go
import "fmt"

type StreamingTier int

const (
    Free     StreamingTier = iota
    Standard
    Premium
    Lossless
)

func main() {
    fmt.Println(Free, Standard, Premium, Lossless)
    tier := Premium
    fmt.Printf("tier type: %T, value: %d\n", tier, tier)
}
```

3. **Calculation:** Given the following declarations, which assignments compile and which produce errors? Identify each line.

```go
type Bpm float64

var tempo Bpm    = 120.0
var raw  float64 = tempo          // line A
var cvt  float64 = float64(tempo) // line B
var same Bpm     = cvt            // line C
```

4. **Where is the bug?**

```go
package main

import "fmt"

func main() {
    x := 10
    y := 20
    x, y := x + y, x  // reassign both
    fmt.Println(x, y)
}
```

5. **Write a program:** Declare a `const` block using `iota` that represents five chart positions for your favorite genre: `Debut`, `Rising`, `Peak`, `Declining`, and `Legacy`, numbered 1 through 5 (use `iota + 1`). Print each constant's name and value using `fmt.Printf` with `%d`. Then declare a variable of that type, assign it the `Peak` value, and print its Go type using `%T`.

6. **What does this print?**

```go
package main

import "fmt"

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

func main() {
    a := 5
```

```
    b := &a
    double(b)
    fmt.Println(a)
    fmt.Println(*b)
}
```

7. **Where is the bug?** The following code tries to append a suffix to a string through a pointer.

```go
package main

import "fmt"

func addExcitement(s *string) {
    s += "!"
}

func main() {
    msg := "Out Of The Blue"
    addExcitement(&msg)
    fmt.Println(msg)
}
```

(This code does not compile — what is the type error?)

The Go Authors. 2025. "The Go Programming Language Specification." https://go.dev/ref/spec. https://go.dev/ref/spec.