

Lesson 4: Type System, Structs & Syntax

Variables, Functions, Control Flow, Structs, Interfaces & Methods

Variable Declarations

Go has several ways to declare variables. Coming from TypeScript where you have `let`, `const`, and `var`, Go's system will feel both familiar and different.

The Four Ways to Declare Variables

TypeScript

```
let name: string = "Alice"      // explicit type
let name = "Alice"              // inferred type
const age: number = 30         // constant
```

Go equivalents

```
// 1. Full declaration (rarely used)
var name string = "Alice"

// 2. Type inferred by var (used for package-level vars)
var name = "Alice"

// 3. Short declaration := (MOST COMMON, inside functions only)
name := "Alice"

// 4. Constant
const age = 30
```

The **`:=` operator** is what you'll use 90% of the time. It declares AND assigns in one step, with the type inferred from the right side. It only works inside functions — at the package level you must use `var`.

Multiple declarations:

```
// Declare multiple variables at once
x, y := 10, 20
name, age := "Alice", 30

// Var block (common at package level)
var (
    host = "localhost"
    port = 8080
    debug bool // defaults to false (zero value)
)
```

Zero Values (Go's Default Values)

In TypeScript, uninitialized variables are `undefined`. In Go, every type has a **zero value** — there is no `undefined` or `null` for basic types:

Type	Zero Value	TS Equivalent
<code>int, float64</code>	0	<code>undefined</code> (or 0 if initialized)
<code>string</code>	<code>""</code> (empty string)	<code>undefined</code> (or "")
<code>bool</code>	<code>false</code>	<code>undefined</code> (or <code>false</code>)
<code>pointer, slice, map</code>	<code>nil</code>	<code>null / undefined</code>

struct	all fields zero-valued	N/A
--------	------------------------	-----

This means: You can declare `var count int` and immediately use it — it's already 0. No null pointer exceptions for basic types. This is a major safety advantage over TS/JS.

Functions

TypeScript

```
function greet(name: string): string {
    return `Hello, ${name}`
}

const add = (a: number, b: number): number => a + b
```

Go

```
func greet(name string) string {
    return fmt.Sprintf("Hello, %s", name)
}

func add(a, b int) int { // same-type params can be grouped
    return a + b
}
```

Notice: types come **after** the parameter name (`name string` not `string name`). No arrow functions — everything uses `func`. No semicolons (the compiler inserts them).

Multiple Return Values

This is one of Go's most distinctive features. Functions can return multiple values, and this is how **error handling works** idiomatically — no try/catch:

```
// TS: would throw or return Result<T, Error>
// Go: returns (value, error) tuple

func findUser(id int) (User, error) {
    user, err := db.Query("SELECT * FROM users WHERE id = $1", id)
    if err != nil {
        return User{}, fmt.Errorf("finding user %d: %w", id, err)
    }
    return user, nil
}

// Calling it:
user, err := findUser(42)
if err != nil {
    log.Fatal(err) // handle the error
}
fmt.Println(user.Name) // safe to use
```

The if err != nil pattern: You will write this hundreds of times. Yes, it's verbose. But it forces you to handle every error at the call site. No silent swallowed exceptions, no unhandled promise rejections. Every error is explicit.

The blank identifier _ (underscore)

If you don't need one of the return values, use `_` to discard it:

```
user, _ := findUser(42) // ignore the error (usually bad practice)
_, err := findUser(42) // ignore the user, just check for error
```

Control Flow

If / Else

```
// No parentheses around the condition! Braces are mandatory.
if age >= 18 {
    fmt.Println("adult")
} else {
    fmt.Println("minor")
}

// If with init statement (very idiomatic Go):
if user, err := findUser(42); err != nil {
    log.Fatal(err)
} else {
    fmt.Println(user.Name) // user is scoped to this if/else
}
```

For Loops (the only loop keyword in Go)

Go has **no while, no do-while, no forEach**. Just `for`, which does everything:

```
// Classic for loop (like C / TS)
for i := 0; i < 10; i++ {
    fmt.Println(i)
}

// While loop (just for + condition)
for count < 100 {
```

```

        count++
}

// Infinite loop
for {
    // runs forever until break or return
}

// Range loop (like TS for...of / forEach)
names := []string{"Alice", "Bob", "Charlie"}
for index, name := range names {
    fmt.Printf("%d: %s\n", index, name)
}

// Range over map (like TS Object.entries)
for key, value := range myMap {
    fmt.Printf("%s = %v\n", key, value)
}

```

Switch (much more powerful than TS/JS switch)

```

// No break needed! Cases don't fall through by default.
switch role {
case "admin":
    fmt.Println("full access")
case "editor", "writer": // multiple values in one case
    fmt.Println("write access")
default:
    fmt.Println("read only")
}

// Switch with no expression (replaces if/else chains):
switch {
case age < 13:
    fmt.Println("child")
case age < 18:
    fmt.Println("teenager")
default:
    fmt.Println("adult")
}

```

No fallthrough by default! In TS/JS you forget a `break` and get bugs. Go cases automatically break. If you explicitly want fallthrough, you use the `fallthrough` keyword (rare).

Structs — Go's Core Data Structure

Structs are Go's answer to TypeScript interfaces, classes, and types — all rolled into one. There are **no classes** in Go. Structs hold data, and you attach methods to them separately.

Defining a Struct

TypeScript

```
interface User {  
    id: number  
    name: string  
    email: string  
    isActive: boolean  
    createdAt: Date  
}
```

Go

```
type User struct {  
    ID      int      `json:"id" db:"id" `  
    Name    string   `json:"name" db:"name" `  
    Email   string   `json:"email" db:"email" `  
    IsActive bool    `json:"is_active" db:"is_active" `  
    CreatedAt time.Time `json:"created_at" db:"created_at" `  
}
```

Notice the **struct tags** (backtick strings after each field). These tell JSON marshaling, database libraries, and validators how to map struct fields to external names. This replaces what decorators or Prisma schema do in TS. Fields are **Capitalized** because they need to be exported (public) for JSON encoding to work.

Creating Struct Instances

```
// Named fields (preferred, like TS object literal)  
user := User{  
    ID:      1,  
    Name:    "Alice",  
    Email:   "alice@example.com",  
    IsActive: true,  
}  
  
// Zero-value struct (all fields get their zero values)  
var emptyUser User // ID=0, Name="", Email="", IsActive=false  
  
// Pointer to struct (very common)  
userPtr := &User{Name: "Bob", Email: "bob@example.com"}
```

Accessing and Modifying Fields

```
fmt.Println(user.Name)      // "Alice"  
user.Email = "new@email.com" // mutate directly  
  
// Pointer access uses the SAME dot syntax (no -> like in C)  
userPtr.Name = "Bobby"      // Go auto-dereferences
```

Methods on Structs

Go doesn't put methods inside structs (no class body). Instead, you attach methods to types using a **receiver**:

TypeScript class

```
class User {  
    constructor(public name: string, public email: string) {}
```

```

        greet(): string {
            return `Hi, I'm ${this.name}`
        }
    }

Go methods

// Value receiver (gets a COPY, cannot mutate original)
func (u User) Greet() string {
    return fmt.Sprintf("Hi, I'm %s", u.Name)
}

// Pointer receiver (can MUTATE the original struct)
func (u *User) Deactivate() {
    u.IsActive = false // modifies the original
}

// Usage:
user := User{Name: "Alice", IsActive: true}
fmt.Println(user.Greet()) // "Hi, I'm Alice"
user.Deactivate() // user.IsActive is now false

```

Value vs Pointer receiver: Use a **pointer receiver** (`*User`) when the method needs to modify the struct or when the struct is large (avoids copying). Use a **value receiver** (`User`) for read-only methods on small structs. Rule of thumb: if any method uses a pointer receiver, make all methods use pointer receivers for consistency.

Interfaces — Go's Polymorphism

This is where Go gets really interesting compared to TS. Go interfaces are **satisfied implicitly** — you never write `implements`. If a type has all the methods an interface requires, it automatically satisfies that interface. This is called **structural typing** (TS has this too, but Go's version is more powerful because it works at the method level).

Defining and Implementing an Interface

TypeScript

```
interface Notifier {
    send(to: string, msg: string): void
}

class EmailNotifier implements Notifier { // explicit
    send(to: string, msg: string): void { ... }
}
```

Go

```
type Notifier interface {
    Send(to string, msg string) error
}

type EmailNotifier struct {
    SMTPHost string
}

// No "implements" keyword! Just define the method:
func (e *EmailNotifier) Send(to string, msg string) error {
    // send email logic...
    return nil
}

type SlackNotifier struct {
    WebhookURL string
}

func (s *SlackNotifier) Send(to string, msg string) error {
    // post to Slack...
    return nil
}
```

Using the interface:

```
func alertUser(n Notifier, userEmail string) {
    err := n.Send(userEmail, "Your order shipped!")
    if err != nil {
        log.Printf("notification failed: %v", err)
    }
}

// Both work because both satisfy Notifier:
alertUser(&EmailNotifier{SMTPHost: "smtp.gmail.com"}, "a@b.com")
alertUser(&SlackNotifier{WebhookURL: "https://..."}, "#general")
```

Why implicit interfaces are powerful: You can define an interface in *your* package that a type from a *third-party* library satisfies — without that library even knowing about your interface. This makes testing and dependency injection trivial. No mocking libraries needed.

Common Standard Library Interfaces

Interface	Method(s)	Used For
fmt.Stringer	String() string	Custom string representation (like <code>toString()</code>)
error	Error() string	Error values (built into the language)
io.Reader	Read(p []byte) (int, error)	Reading from any source (files, HTTP, etc.)
io.Writer	Write(p []byte) (int, error)	Writing to any destination
io.Closer	Close() error	Cleaning up resources
http.Handler	ServeHTTP(w, r)	HTTP request handling
sort.Interface	Len, Less, Swap	Custom sorting

The Empty Interface: `interface{}` and `any`

```
// interface{} matches ANY type (like TS "any")
// Since Go 1.18, you can also write "any" as an alias:

func printAnything(v any) {
    fmt.Println(v)
}

printAnything(42)          // works
printAnything("hello")     // works
printAnything(User{})      // works
```

Use `any` sparingly. Unlike TS, you lose all type safety and need **type assertions** to get it back: `val, ok := v.(string)`

Struct Embedding (Go's "Inheritance")

Go has **no inheritance**. Instead, it has **composition through embedding**. You embed one struct inside another, and the outer struct "inherits" all fields and methods:

TypeScript (*inheritance*)

```
class Animal {
    constructor(public name: string) {}
    speak(): string { return this.name + " speaks" }
}
class Dog extends Animal {
    fetch(): string { return this.name + " fetches" }
}
```

Go (*composition via embedding*)

```
type Animal struct {
    Name string
}
func (a Animal) Speak() string {
    return a.Name + " speaks"
}

type Dog struct {
    Animal           // embedded! No field name.
    Breed string
}
func (d Dog) Fetch() string {
    return d.Name + " fetches" // Name is promoted from Animal
}

dog := Dog{Animal: Animal{Name: "Rex"}, Breed: "Lab"}
dog.Speak() // "Rex speaks" - method promoted from Animal
dog.Fetch() // "Rex fetches"
dog.Name    // "Rex" - field promoted from Animal
```

Embedding is not inheritance. There's no polymorphism through embedding — a Dog is NOT an Animal. It HAS an Animal. For polymorphism, use interfaces.

Custom Types and Type Aliases

```
// Custom type (creates a NEW distinct type)
type UserID int
type Role string
type Celsius float64

// You can attach methods to custom types:
func (c Celsius) ToFahrenheit() float64 {
    return float64(c)*9/5 + 32
}

// Enum-like pattern (Go has no enums):
const (
    RoleAdmin  Role = "admin"
    RoleEditor Role = "editor"
    RoleViewer Role = "viewer"
)
```

Custom types add **type safety**. A `UserID` cannot accidentally be passed where an `int` is expected without explicit conversion. This is stronger than TS's type aliases, which are just aliases (a `type UserID = number` in TS is still just a number).

Quick Syntax Reference: Go vs TypeScript

Concept	TypeScript	Go
String interpolation	`Hello \${name}`	fmt.Sprintf("Hello %s", name)
Print to console	console.log(x)	fmt.Println(x)
Null check	if (x !== null)	if x != nil
Ternary	x ? a : b	No ternary. Use if/else.
Spread	[...arr, 4]	append(arr, 4)
Destructuring	const {a, b} = obj	a, b := obj.A, obj.B
Optional chaining	user?.name	N/A (check nil explicitly)
Async/await	await fetch(url)	No async. Use goroutines.
Lambda / closure	(x) => x * 2	func(x int) int { return x*2 }
Type assertion	val as string	val.(string)
Map/filter/reduce	arr.map(fn)	Manual for loop (no built-in)
Import	import { x } from 'y'	import "path/to/pkg"
Package export	export function X	func X (capitalize)

Next Lesson: Lesson 5 will dive deep into Go's concrete data types — strings, numbers, slices, maps, arrays, and pointers — with practical examples.