Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 3

Strings, Bytes, and Runes

Go strings look familiar — you create them with double quotes, concatenate with `+`, and pass them to `fmt.Println`. But the model underneath is different enough from Java's that it causes real bugs, especially with non-ASCII text. This chapter covers what Go strings actually are, why `len(s)` might surprise you, and how to work with text correctly.

What a String Really Is

In Java, a `String` is a sequence of `char` values, where each `char` is a UTF-16 code unit. In Go, a string is an **immutable sequence of bytes**. That is the entire definition. Go makes no promises about encoding; by convention almost all Go source code and string data is UTF-8, but the type itself is just bytes.

Under the hood, a string value is a small struct: a pointer to read-only memory and a length. It is essentially a read-only `[]byte` without the capacity field. Because a string is just a pointer and a length, passing a string to a function copies only those two words — the underlying bytes are never duplicated. You can pass a ten-megabyte string to a function and the call is as cheap as passing an `int`.

Tip: Pass strings by value and store them as values in structs — that is the normal Go style. You may occasionally see `*string` when `nil` is needed to distinguish "not set" from an empty string `""`, such as an optional field in a config struct or a JSON payload where omitting a field has a different meaning than sending an empty one. Outside that specific pattern, a `*string` is a code smell.

Tip: `==` compares string *contents* in Go, byte for byte — there is no `.equals()` and no reference-identity trap. `a == b` is true exactly when the two strings hold the same bytes, so the Java habit of reaching for `.equals()` to avoid comparing references simply does not apply here. `<`, `>`, and friends work too, ordering strings lexicographically by byte.

`len(s)` returns the number of **bytes** in the string, not the number of characters.

```
s := "café"
fmt.Println(len(s)) // 5, not 4 --- é is two bytes (0xC3 0xA9)
```

Wut: If you are used to Java where `"café".length()` returns 4, note that Go's `len("café")` returns 5 because `é` is encoded as two bytes in UTF-8. While that might make Java seem enlightened, `"🍕🔥".length()` in Java returns 4!

byte and rune

Go has no `char` type. Instead it has two types for working with individual pieces of text:

- `byte` is an alias for `uint8`. It holds a single ASCII character or one byte of a multibyte sequence.
- `rune` is an alias for `int32`. It holds a full Unicode code point.

Java's `char` is a UTF-16 **code unit**, which means characters outside the Basic Multilingual Plane (anything above U+FFFF) require two Java `char` values. Go's `rune` is a full code point, so one `rune` always represents one character, no matter how far up the Unicode table it lives.

Rune literals use single quotes, just like character literals in Java:

```
var b byte = 'A'    // 65
var r rune = '⌘'    // 8984 (U+2318 PLACE OF INTEREST SIGN)
var r2 rune = 'é'   // 233 (U+00E9)
```

A rune literal has default type `rune` (an alias for `int32`), so `r := 'é'` gives `r` the type `rune`. Until it acquires a type, though, it is an untyped rune constant and participates in constant arithmetic like any other untyped integer — so `'A' + 1` is the perfectly legal constant `66`.

Wut: Printing a `rune` with `%v` (or `fmt.Println`) shows you a number, not a character. Because `rune` is just an alias for `int32`, the default format is the integer:

```
r := 'é'
fmt.Println(r)          // 233
fmt.Printf("%v\n", r)   // 233
fmt.Printf("%c\n", r)   // é
```

Use `%c` (or convert with `string(r)`) when you want to see the character. Java hides this from you because `char` has its own printable identity; in Go a `rune` is an integer wearing a costume.

Indexing and Iteration

Indexing with `s[i]`

Indexing a string with `s[i]` gives you a **byte**, not a character.

```
s := "café"
fmt.Println(s[3]) // 195 (0xC3 --- the first byte of é)
```

Java programmers often expect `s[3]` to yield `'é'`. In Go it yields `195`, the numeric value of the first byte of the two-byte UTF-8 encoding of `é`. Formatting it with `%c` would print `Ã`, which is the Latin character whose code point is 195 — not what you wanted.

Trap: `s[i]` gives you a `byte`. If the string contains any non-ASCII characters, walking it with a plain index loop and printing or storing individual indexed values will corrupt multibyte characters.

Iterating Bytes with a Plain `for` Loop

A plain index loop iterates bytes:

```
s := "café"
for i := 0; i < len(s); i++ {
    fmt.Printf("s[%d] = %d\n", i, s[i])
}
// s[0] = 99   (c)
```

```
// s[1] = 97   (a)
// s[2] = 102  (f)
// s[3] = 195  (first byte of é)
// s[4] = 169  (second byte of é, 0xA9)
```

Iterating Runes with `for range`

`for range` over a string decodes UTF-8 automatically. The loop variable receives a `rune` (the Unicode code point), and the index is the **byte position** of the start of that rune.

```
s := "café!"
for i, r := range s {
    fmt.Printf("s[%d] = %c (%d)\n", i, r, r)
}
// s[0] = c (99)
// s[1] = a (97)
// s[2] = f (102)
// s[3] = é (233)
// s[5] = ! (33)
```

Notice that the index jumps from 3 directly to 5 for the `!` — index 4 never appears because `é` occupies bytes 3 and 4, and byte 4 is not the start of a rune.

Tip: Use `for range` when you care about characters (runes). Use a plain `for i` loop when you need raw byte access.

String Literals

Go has two forms of string literal.

Interpreted string literals use double quotes and process backslash escapes — `\n`, `\t`, `\uXXXX`, `\UXXXXXXXX`, etc. This is the same as Java.

Raw string literals use backticks and suppress all escape processing. Everything between the backticks is literal, including newlines and backslashes.

```
// interpreted --- \n is a newline
msg := "Bad Apple!!\nfor you\n"

// raw --- \n is two characters: backslash and n
re := `\d+\.\d+`

// raw --- multi-line JSON template
tmpl := `{
  "artist": "BT",
  "album":  "ESCM"
}`
```

Raw string literals are especially useful for regular expressions (where backslashes are abundant) and for embedding multi-line text without escaping.

Converting Between Strings, Bytes, and Runes

You can convert freely among `string`, `[]byte`, and `[]rune`:

```go
s := "hola"

b := []byte(s)  // copy to a mutable byte slice
s2 := string(b) // copy back to a string

r := []rune(s)  // copy to a rune slice
s3 := string(r) // copy back to a string

fmt.Println(s2) // hola
fmt.Println(s3) // hola
```

Wut: These conversions **copy** the data. In Java you can wrap a shared array in a `CharBuffer`, but a `String` always owns its own copy too; in Go the compiler likewise enforces the copy to maintain string immutability.

Converting a single integer to `string` gives you the UTF-8 encoding of that code point, not the decimal representation:

```go
fmt.Println(string(rune(65)))  // A
fmt.Println(string(rune(233))) // é
```

To convert a number to its decimal string representation, use `strconv.Itoa` (covered below).

The `strings` Package

The `strings` package provides the functions you reach for every day. Import it with `import "strings"`.

Searching and Testing

```go
func Contains(s, substr string) bool   // true if substr appears anywhere in s
func HasPrefix(s, prefix string) bool  // true if s starts with prefix
func HasSuffix(s, suffix string) bool  // true if s ends with suffix
func Count(s, substr string) int       // number of non-overlapping occurrences of substr
func Index(s, substr string) int       // byte index of first occurrence, or -1
func EqualFold(s, t string) bool       // true if s and t are equal under Unicode case-folding
```

```go
s := "Bad Apple!!"
fmt.Println(strings.Contains(s, "Apple"))  // true
fmt.Println(strings.HasPrefix(s, "Bad"))   // true
fmt.Println(strings.HasSuffix(s, "!!"))    // true
fmt.Println(strings.Count(s, "p"))         // 2
fmt.Println(strings.Index(s, "Apple"))     // 4
```

`strings.EqualFold` is the case-insensitive equality test, Go's answer to Java's `equalsIgnoreCase`:

```go
fmt.Println(strings.EqualFold("Darude", "DARUDE")) // true
```

Splitting and Joining

```go
func Split(s, sep string) []string         // slice of substrings separated by sep
func Join(elems []string, sep string) string // concatenate elems with sep between each
func Fields(s string) []string             // split around runs of whitespace, dropping empties
```

```go
parts := strings.Split("un,dos,tres", ",")
fmt.Println(parts)                       // [un dos tres]
fmt.Println(strings.Join(parts, " - ")) // un - dos - tres
```

`strings.Fields` splits on runs of whitespace (any amount), which is handy for tokenising messy input where `Split(s, " ")` would leave empty strings:

```
fmt.Println(strings.Fields("  un   dos  tres ")) // [un dos tres]
```

Trimming and Case

```
func TrimSpace(s string) string              // strip leading and trailing whitespace
func Trim(s, cutset string) string           // strip any chars in cutset from both ends
func ToUpper(s string) string                // return s in upper case
func ToLower(s string) string                // return s in lower case
func ReplaceAll(s, old, new string) string   // replace every occurrence of old with new
```

`TrimSpace` strips leading and trailing whitespace. `Trim` strips any characters in the cutset from both ends:

```
fmt.Println(strings.TrimSpace("  flores  "))                 // flores
fmt.Println(strings.Trim("***Sandstorm***", "*"))             // Sandstorm
fmt.Println(strings.Trim("...Better Off Alone...", "/."))     // Better Off Alone
fmt.Println(strings.ToUpper("flowers"))                       // FLOWERS
fmt.Println(strings.ReplaceAll("la la la", "la", "na"))       // na na na
```

Building Strings: `strings.Builder`

`strings.Builder` is the idiomatic way to build up a string incrementally. It is Go's answer to Java's `StringBuilder`. Unlike concatenating with `+` in a loop (which allocates a new string on every iteration), `Builder` maintains a growing buffer and materialises the final string only when you call `String()`.

```
var b strings.Builder
b.WriteString("Sandstorm --- ")
b.WriteString("Darude")
b.WriteByte('\n')
b.WriteRune('🎵')
fmt.Println(b.String())
// Sandstorm --- Darude
// 🎵
```

The relevant methods are:

```
func (b *Builder) WriteString(s string) (int, error)  // append a string
func (b *Builder) WriteByte(c byte) error             // append a single byte
func (b *Builder) WriteRune(r rune) (int, error)      // append a Unicode code point
func (b *Builder) String() string                     // return the accumulated string
func (b *Builder) Reset()                             // clear the buffer for reuse
func (b *Builder) Len() int                           // current length in bytes
```

Tip: Always use `strings.Builder` when you are building a string in a loop. Repeated `s += piece` is $O(n^2)$ in the total length because each `+` copies the entire accumulated string. `Builder` amortises this to $O(n)$.

The `strconv` Package

`strconv` handles conversions between strings and numeric types.

```
func Itoa(i int) string                                  // int → decimal string
func Atoi(s string) (int, error)                         // decimal string → int
func FormatInt(i int64, base int) string                 // int64 → string in base
```

```
func ParseInt(s string, base int, bitSize int) (int64, error)     // base 0 auto-detects prefix
func FormatUint(i uint64, base int) string                        // uint64 → string in base
func ParseUint(s string, base int, bitSize int) (uint64, error)   // string → uint64
func FormatFloat(f float64, fmt byte, prec, bitSize int) string   // float64 → string
func ParseFloat(s string, bitSize int) (float64, error)           // string → float64
func FormatBool(b bool) string                                    // bool → "true"/"false"
func ParseBool(str string) (bool, error)                          // "true"/"false" → bool
```

`Itoa` and `Atoi` are convenient wrappers for base-10 `int`. When you need a specific base or size, reach for `ParseInt`/`ParseUint` and `FormatInt`/`FormatUint` directly:

```
strconv.FormatInt(255, 16)          // "ff"  --- hex
strconv.FormatInt(255, 2)           // "11111111" --- binary
strconv.ParseInt("ff", 16, 64)      // 255, nil
strconv.ParseInt("0xFF", 0, 64)     // 255, nil --- base 0 detects "0x" prefix
strconv.ParseUint("4294967295", 10, 32) // 4294967295, nil --- fits in uint32
```

The `bitSize` parameter (8, 16, 32, or 64) constrains the result range without changing the return type — `ParseInt` always returns `int64`, but passing `bitSize=32` guarantees the value fits in `int32`.

```
s := strconv.Itoa(42)    // "42"
n, err := strconv.Atoi("99") // 99, nil
bad, err := strconv.Atoi("nope") // 0, *strconv.NumError
```

`strconv.Atoi` returns two values: the converted integer and an `error`. If the string is not a valid integer, `err` is non-nil. You should always check the error. [***no-discard-error***] Error handling is covered fully in Chapter 9; for now, the pattern is:

```
n, err := strconv.Atoi(s)
if err != nil {
    // handle the error
}
```

Trap: `fmt.Sprintf("%d", n)` converts an integer to a string and works, but it is significantly slower than `strconv.Itoa` because `fmt` must parse the format string and box the argument into an `interface{}`. Prefer `strconv` when performance matters.

The `unicode/utf8` Package

When you need to work at the rune level without converting the whole string to `[]rune`, the `unicode/utf8` package gives you the tools.

```
func RuneCountInString(s string) int                  // number of runes in s (not bytes)
func DecodeRuneInString(s string) (r rune, size int)  // first rune and its byte width
func ValidString(s string) bool                       // true if s is valid UTF-8
```

```
s := "café"
fmt.Println(len(s))                    // 5 (bytes)
fmt.Println(utf8.RuneCountInString(s)) // 4 (runes)

r, size := utf8.DecodeRuneInString(s)
fmt.Printf("first rune: %c, size: %d\n", r, size) // first rune: c, size: 1

fmt.Println(utf8.ValidString(s))           // true
fmt.Println(utf8.ValidString("\xff\xfe")) // false
```

`DecodeRuneInString` is useful when you want to peel off one rune at a time from the front of a string without a full `for range` loop.

The bytes Package

The `bytes` package mirrors the `strings` package but operates on `[]byte` instead of `string`. Every function in `strings` that takes a `string` has a counterpart in `bytes` that takes `[]byte`. For example, `bytes.Contains`, `bytes.Split`, `bytes.Join`, `bytes.TrimSpace`.

When you are working with data that is already in a `[]byte` (reading from a file or network, for instance), using `bytes` functions avoids the copy that `string(b)` would require.

`bytes.Buffer` is the older alternative to `strings.Builder` and supports both reading and writing — it implements `io.Reader` and `io.Writer`, making it useful for testing code that writes to an `io.Writer`.

```
func Contains(b, subslice []byte) bool // true if subslice is within b
func ToUpper(s []byte) []byte          // a copy of s with all Unicode letters uppercased
```

```
data := []byte("good days")
fmt.Println(bytes.Contains(data, []byte("days"))) // true
fmt.Printf("%s\n", bytes.ToUpper(data))           // GOOD DAYS

var buf bytes.Buffer
fmt.Fprintf(&buf, "%d good days", 365) // buf implements io.Writer
fmt.Println(buf.String())              // 365 good days
```

Try It

Type this in and run it. It exercises the byte/rune distinction, `for range`, `strings.Builder`, `strconv`, and a couple of `strings` helpers all at once. Watch how the byte index from `range` lines up with the bytes the builder records.

```
package main

import (
    "fmt"
    "strconv"
    "strings"
    "unicode/utf8"
)

func main() {
    line := "Ojitos Lindos --- Bad Bunny"

    fmt.Println("bytes:", len(line))                    // byte count
    fmt.Println("runes:", utf8.RuneCountInString(line)) // rune count

    var b strings.Builder
    for i, r := range line {
        if r == 'o' || r == 'O' {
            b.WriteString("[" + strconv.Itoa(i) + "]") // tag each o with its byte index
        } else {
            b.WriteRune(r)
        }
    }
```

```
    fmt.Println(b.String())

    fmt.Println(strings.ToUpper(line))
    fmt.Println("contains Bunny:", strings.Contains(line, "Bunny"))
}
```

Try these modifications:

- Replace the ASCII title with one containing accented characters (say a Spanish phrase with `ñ` or `é`) and watch `len` and `RuneCountInString` diverge.
- Switch the `for range` loop to a plain `for i := 0; i < len(line); i++` loop and observe how it now visits bytes instead of runes.
- Use `strconv.Atoi` to parse a track number out of a string like `"track 7"` (after trimming) and handle the error.

Key Points

- Go strings are immutable byte sequences; `len(s)` counts bytes, not characters.
- `byte` is `uint8`; `rune` is `int32` (a full Unicode code point).
- `s[i]` yields a `byte`; use `for range` to iterate runes.
- Java `char` is a UTF-16 code unit; Go `rune` is a full code point — they are different concepts.
- Raw string literals (backticks) pass through all characters literally, including backslashes and newlines.
- Converting between `string`, `[]byte`, and `[]rune` always copies data.
- `strings.Builder` is the idiomatic way to build strings in a loop; avoid `+=` in loops.
- `strconv.Atoi` returns an `error`; always check it.
- `utf8.RuneCountInString` gives the true character count; `len` gives the byte count.
- The `bytes` package mirrors `strings` for `[]byte` data.

Exercises

1. **Think about it:** Go strings are described as "immutable sequences of bytes." Java strings are also immutable. Given that both languages have immutable strings, why does Go's `for range` behave differently from Java's enhanced `for` loop over `s.toCharArray()`? What would have to be true about the loop for both languages to give the same result?

2. **What does this print?**

```
package main

import "fmt"

func main() {
    s := "Alizée"
    fmt.Println(s[4])
}
```

3. **Calculation:** `len("Alizée")` returns how many bytes? And `utf8.RuneCountInString("Alizée")` returns how many runes? (Hint: `Alizée` is spelled `A`, `l`, `i`, `z`, `é`, `e` — the accented `é` (U+00E9) comes before the final plain `e`. The five plain ASCII letters are one byte each, and `é` is two bytes in UTF-8.)

4. **Where is the bug?** The following function tries to reverse each byte's case by operating on raw bytes:

```
func swapCase(s string) string {
    b := []byte(s)
    for i := range b {
        switch {
```

```
        case b[i] >= 'A' && b[i] <= 'Z':
            b[i] += 32
        case b[i] >= 'a' && b[i] <= 'z':
            b[i] -= 32
        }
    }
    return string(b)
}

func main() {
    fmt.Println(swapCase("Héroe"))
}
```

`é` is encoded as bytes 0xC3 (195) and 0xA9 (169). `H` is 72 and `e` is 101. Trace through the loop byte by byte. What does the function actually print, and what is fundamentally wrong with this approach for non-ASCII strings?

5. **Write a program:** Write a function `reverseString(s string) string` that returns the string with its runes in reverse order. For example, `reverseString("café")` should return `"éfac"`. Test it with at least one string that contains a multibyte character to confirm it handles Unicode correctly. (Hint: convert to `[]rune` first.)