JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#04

WE DON'T HAVE ENUM, WE GOT IÓTA

BUILDING TYPE-SAFE ENUMS AND STATE MACHINES IN GO

We don't have enum, we got iota

Go's Sequential Identifier - Automatic Counter, Zero Magic

```go
const (
    A = iota    //0
    B           //1
    C           //2
)
```

```go
type OrderStatus int

const (
    Pending PackageStatus = iota
    Processing
    Shipped
    Delivered
)
```

```go
type LogLevel int

const (
    Debug LogLevel = iota    //0
    Info                     //1
    Warn                     //2
    Error                    //3
    Fatal                    //4
)
```

```go
const (
    A1 = iota + 1
    B1
    C1
)
```

+

What is iota?

The Auto-Incrementer

iota is a special identifier used in const blocks to generate successive integer constants.

```
const (
    A = iota    //0
    B           //1
    C           //2
)
```

*You only need to write iota once. Go will continue the incrementing pattern for every line in the block automatically.

Creating Custom Types

Type Safety First

To build a real enum, don't just use int. Define a custom type to prevent logic bugs.

```
type OrderStatus int

const (
    Pending OrderStatus = iota
    Processing
    Shipped
    Delivered
)
```

Now your functions can strictly require an OrderStatus type instead of a generic integer.

The "Unknown" State Pattern

Avoid the Default Zero

In Go, the zero-value of an int is 0. If your first state is 0, an uninitialized variable might look like a valid state.

```
type OrderStatus int

const (
    Unknown OrderStatus = iota //0
    Pending
    Processing
    Shipped
    Delivered
)
```

Pro tip: Start your iota enums with an Unknown or Invalid state at index 0 to catch uninitialized data

Skipping Values and Offsets

Advanced iota Math

You can manipulate the counter.

Use the blank identifier _ to skip values or add offsets for specific starting points

```
const (
    Small = iota + 10 // Starts at 10
    Medium            // 11
    Large             // 12
)
```

```
const (
    _ = iota // Skip 0
    A
    B
    C
)
```

Implementing a Stringer

Making Enums Human Readable

Printing 1 in a production log is a nightmare for debugging

```go
type OrderStatus int

const (
    Unknown OrderStatus = iota
    Pending
    Shipped
)

func (s OrderStatus) String() string {
    return [...]string{"Unknown", "Pending", "Shipped"}[s]
}
```

Implementing the .String() method ensures your logs show "Shipped" instead of "2". Your SRE team will thank you

Bitmasking with iota

Complex State (Bitmasks)

Need a variable to hold multiple flags? Use bit-shifting with iota

```go
type Roles int

const (
    Admin  Roles = 1 << iota // 1 (0001)
    Editor                   // 2 (0010)
    Viewer                   // 4 (0100)
)
```

```
go run main.go
Admin =  1
Editor =  2
Viewer =  4
```

*This is the standard Go pattern for permission systems and file modes

The "State Machine" Pattern

Real-World Logic

Enums aren't just for storage; they drive behavior.

```go
func HandleOrder(status OrderStatus) {
    switch status {
    case Unknown:
        // trigger unknown status handling (something wrong?)
    case Pending:
        // trigger pending handling function
    case Shipped:
        // trigger shipped handling function
    }
}
```

Combining custom types with switch statements (will expand on those in a few days 🙂) creates the foundation for robust, predictable backend logic

GOLANG /// BA

To summarize:

- Use custom types for safety.
- Start at 0 with an "Unknown" state.
- Implement .String() for observability.

Do you prefer using the stringer tool to auto-generate your enum names, or do you write the .String() method by hand? Let’s hear your opinions 🙂