

A Whirlwind Tour of Go

Just the Cool Parts

Steve Willoughby

10-Apr-2024

v0.0.1



The Point

- “What *is* Go?”
- “What is it actually good for?”
- “Why should I care?”

30 Seconds of History

- Designed by Rob Pike, Ken Thompson, and Robert Griesemer.
- Includes direct experience with C from day 1 to now.

30 Seconds of History

- Designed by Rob Pike, Ken Thompson, and Robert Griesemer.
- Includes direct experience with C from day 1 to now.
- “If we were to design C today, knowing what we know now, on today’s technology...”

30 Seconds of History

- Designed by Rob Pike, Ken Thompson, and Robert Griesemer.
- Includes direct experience with C from day 1 to now.
- “If we were to design C today, knowing what we know now, on today’s technology...”
 - ∴ Go’s syntax is very much like C’s
 - ... but cleaned up and streamlined a bit.

30 Seconds of History

- Designed by Rob Pike, Ken Thompson, and Robert Griesemer.
- Includes direct experience with C from day 1 to now.
- “If we were to design C today, knowing what we know now, on today’s technology...”
 - ∴ Go’s syntax is very much like C’s
 - ... but cleaned up and streamlined a bit.
- Dreamed up while waiting on a 45-minute C⁺⁺ compile

30 Seconds of History

- Designed by Rob Pike, Ken Thompson, and Robert Griesemer.
- Includes direct experience with C from day 1 to now.
- “If we were to design C today, knowing what we know now, on today’s technology...”
 - ∴ Go’s syntax is very much like C’s
 - ... but cleaned up and streamlined a bit.
- Dreamed up while waiting on a 45-minute C⁺⁺ compile
 - Fast compilation
 - Native binary compiler with low overhead
 - Strong static typing
 - Extraordinarily spartan

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.
- Special things: `complex64`, `complex128`.

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.
- Special things: `complex64`, `complex128`.
- Structures: `struct{...}`.

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.
- Special things: `complex64`, `complex128`.
- Structures: `struct{...}`.
- What about `char`? Nope. Instead, we have `rune`.

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.
- Special things: `complex64`, `complex128`.
- Structures: `struct{...}`.
- What about `char`? Nope. Instead, we have `rune`.
- Arrays: `[10]int`, `[100]rune`.

Intrinsic Data Types

- The usual suspects: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `bool`, `byte`, `float32`, `float64`, `string`.
- Special things: `complex64`, `complex128`.
- Structures: `struct{...}`.
- What about `char`? Nope. Instead, we have `rune`.
- Arrays: `[10]int`, `[100]rune`.
- Slices: `[]int`, `[]byte`, `[]string`.
- Maps: `map[string]int`.

Expressions and Operators

- Arithmetic: +, -, *, /, %.
- Relational: ==, !=, >, <, >=, <=.
- Logical: &&, ||, !.
- Bitwise: &, |, ^, <<, >>, &^.

```
// x &^ y == x & (^y)
```

Expressions and Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`.
- Relational: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Logical: `&&`, `||`, `!`.
- Bitwise: `&`, `|`, `^`, `<<`, `>>`, `&^`. `// x &^ y == x & (^y)`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `:=`.

Expressions and Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`.
- Relational: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Logical: `&&`, `||`, `!`.
- Bitwise: `&`, `|`, `^`, `<<`, `>>`, `&^`. `// x &^ y == x & (^y)`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `:=`.
- Reference/Dereference: `&`, `*`.
- Unary: `+`, `-`, `^`. `// ^x`
- Increment/Decrement: `++`, `--`. `// x++ or x--`

Expressions and Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`.
- Relational: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Logical: `&&`, `||`, `!`.
- Bitwise: `&`, `|`, `^`, `<<`, `>>`, `&^`. `// x &^ y == x & (^y)`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `:=`.
- Reference/Dereference: `&`, `*`.
- Unary: `+`, `-`, `^`. `// ^x`
- Increment/Decrement: `++`, `--`. `// x++ or x--`
- Channel I/O: `<-`. `// channel<-x or <-channel`

Expressions and Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`.
- Relational: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Logical: `&&`, `||`, `!`.
- Bitwise: `&`, `|`, `^`, `<<`, `>>`, `&^`. `// x &^ y == x & (^y)`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `:=`.
- Reference/Dereference: `&`, `*`.
- Unary: `+`, `-`, `^`. `// ^x`
- Increment/Decrement: `++`, `--`. `// x++ or x--`
- Channel I/O: `<-`. `// channel<-x or <-channel`
- Blank identifier: `_`.

Go Syntax

- Type declarations *follow* identifier names

```
var x int
```

```
var UserName string
```

```
func AddNumbers(x, y int) int { ... }
```

```
func DivideNumbers(x, y int) (int, error) { ... }
```

```
type Shape struct {
```

```
    X      int
```

```
    Y      int
```

```
    Color  ColorCode
```

```
}
```

Program Structure

- Basic unit is a *package* (namespace boundary).

Program Structure

- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.

Program Structure

- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.
- Every program must have a `main` package.
- The `main` package has a `main` function.

Program Structure

- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.
- Every program must have a `main` package.
- The `main` package has a `main` function.
- Import packages into the program using the `import` statement.
- Always prefix identifiers from imported packages with their package name.

Program Structure

- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.
- Every program must have a `main` package.
- The `main` package has a `main` function.
- Import packages into the program using the `import` statement.
- Always prefix identifiers from imported packages with their package name.
- Identifiers can be *public* or *private* w/r/t package boundaries.

Program Structure

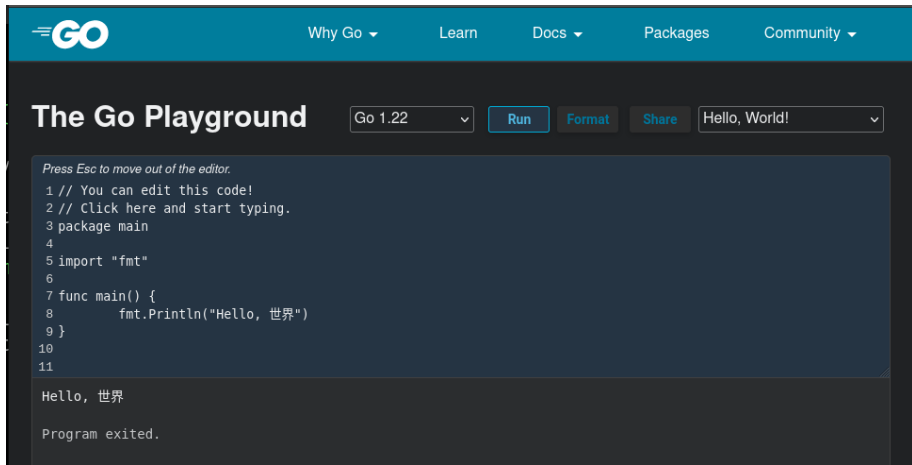
- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.
- Every program must have a `main` package.
- The `main` package has a `main` function.
- Import packages into the program using the `import` statement.
- Always prefix identifiers from imported packages with their package name.
- Identifiers can be *public* or *private* w/r/t package boundaries.
 - Identifier names starting with an uppercase letter are public.
 - All others are private.

Hello, World

```
/* Standard-issue "Hello, World" program in Go */  
  
package main  
  
import "fmt"  
  
func main() {  
    fmt.Println("Hello, 世界")  
}
```

The Playground

- Interactive playground to immediately try something in Go.
- <https://go.dev/play/>



The screenshot shows the Go Playground web interface. At the top is a teal navigation bar with the Go logo and links for 'Why Go', 'Learn', 'Docs', 'Packages', and 'Community'. Below this is a dark grey header area with the title 'The Go Playground', a version selector set to 'Go 1.22', and buttons for 'Run', 'Format', and 'Share'. To the right of these buttons is a text input field containing 'Hello, World!'. The main area is a code editor with a dark background. It contains the following Go code:

```
1 // You can edit this code!
2 // Click here and start typing.
3 package main
4
5 import "fmt"
6
7 func main() {
8     fmt.Println("Hello, 世界")
9 }
10
11
```

Below the code editor, the output of the program is displayed: 'Hello, 世界' followed by 'Program exited.' on a new line.

Importing Third-Party Packages

- Standard library package names are simple names:

```
import "fmt"  
import "encoding/json"  
import "flag"  
import "math"
```

Importing Third-Party Packages

- Standard library package names are simple names:

```
import "fmt"  
import "encoding/json"  
import "flag"  
import "math"
```

- Getting packages from public repositories:

```
import "github.com/MadScienceZone/go-gma/v5/dice"
```

Automatic API Documentation

• <https://pkg.go.dev/repository-url>

The screenshot shows the Go Package Documentation page for the 'dice' package. The page has a dark theme with teal accents. At the top, there's a navigation bar with the Go logo, a search bar, and links for 'Why Go', 'Learn', 'Docs', 'Packages', and 'Community'. Below the navigation bar, the breadcrumb trail reads 'Discover Packages > github.com/MadScienceZone/go-gma/v5 > dice'. The package name 'dice' is prominently displayed with a 'package' tag. Below this, metadata is shown: 'Version: v5.17.0' (marked as 'Latest'), 'Published: Feb 28, 2024', 'License: BSD-3-Clause', 'Imports: 16', and 'Imported by: 0'. A section with tabs for 'Details', 'Repository', and 'Links' follows. 'Details' shows checkmarks for 'Valid go.mod file', 'Redistributable license', 'Tagged version', and 'Stable version', along with a link to 'Learn more about best practices'. 'Repository' shows the GitHub link 'github.com/MadScienceZone/go-gma'. 'Links' shows 'Open Source Insights'. Below this is a 'Jump to ...' dropdown and a sidebar with navigation links: 'Documentation' (selected), 'Overview', 'Index', 'Constants', 'Variables', 'Functions', 'Types', and 'Source Files'. The main content area is titled '<> Documentation' and 'Overview'. It describes the package's purpose: 'Package dice provides a general facility for generating random numbers in fantasy role-playing games.' It then explains the usage model, mentioning 'DieRoller' and providing an example code snippet. Below the text, there are two code blocks. The first block shows a usage example with 'Roll' and 'RollOnce' functions. The second block shows how to create a 'NewDieRoller' instance and use its 'DoRoll' method. A refresh icon is visible in the bottom right corner of the code area.

Discover Packages > github.com/MadScienceZone/go-gma/v5 > dice

dice package

Version: v5.17.0 **Latest** | Published: Feb 28, 2024 | License: BSD-3-Clause | Imports: 16 | Imported by: 0

Details [Valid go.mod file](#) [Redistributable license](#) [Tagged version](#) [Stable version](#) [Learn more about best practices](#)

Repository github.com/MadScienceZone/go-gma

Links [Open Source Insights](#)

Jump to ...

Documentation

Overview

Index

Constants

Variables

Functions

Types

Source Files

<> Documentation

Overview

Package dice provides a general facility for generating random numbers in fantasy role-playing games.

The preferred usage model is to use the higher-level abstraction provided by DieRoller, which rolls dice as described by strings. For example:

```
label, results, err := Roll("d20+16 | c")
label, result, err := RollOnce("15d6 + 15 fire + 1 acid")
```

If you need to keep the die roller itself around after the dice are rolled, to query its status, or to produce a repeatable string of die rolls given a custom seed or number generator, create a new DieRoller value and reuse that as needed:

```
dr, err := NewDieRoller()
label, results, err := dr.DoRoll("d20+16 | c")
```

“Factored” Notation

```
import "fmt"  
import "encoding/json"  
import "flag"  
import "math"
```

“Factored” Notation

```
import "fmt"
import "encoding/json"
import "flag"
import "math"

import (
    "fmt"
    "encoding/json"
    "flag"
    "math"
)
```


“Factored” Notation

```
var initialized bool
var usernames    []string
var Greeting     string    = "Hello"
var TheAnswer    = 42

var (
    initialized bool
    usernames    []string
    Greeting     string    = "Hello"
    TheAnswer    = 42
)
```

“Factored” Notation

```
const initialized = false
const Greeting    = "Hello"
const TheAnswer   byte = 42

const (
    initialized      = false
    Greeting         = "Hello"
    TheAnswer        byte = 42
)
```

“Factored” Notation and iota

```
type MessageType byte
const (
    ServerCommand MessageType = 0
    ServerReply    MessageType = 1
    ServerError    MessageType = 2
    UrgentMessage  MessageType = 3
)
```

“Factored” Notation and iota

```
type MessageType byte
const (
    ServerCommand MessageType = 0
    ServerReply    MessageType = 1
    ServerError    MessageType = 2
    UrgentMessage  MessageType = 3
)
```

```
type MessageType byte
const (
    ServerCommand MessageType = iota
    ServerReply    MessageType = iota
    ServerError    MessageType = iota
    UrgentMessage  MessageType = iota
)
```

“Factored” Notation and iota

```
type MessageType byte
const (
    ServerCommand MessageType = 0
    ServerReply      MessageType = 1
    ServerError      MessageType = 2
    UrgentMessage    MessageType = 3
)
```

```
type MessageType byte
const (
    ServerCommand MessageType = iota
    ServerReply
    ServerError
    UrgentMessage
)
```

“Factored” Notation and iota Expressions

```
type MessageType byte
const (
    ServerCommand MessageType = 0x01
    ServerReply      MessageType = 0x02
    ServerError      MessageType = 0x04
    UrgentMessage    MessageType = 0x08
)
```

```
type MessageType byte
const (
    ServerCommand MessageType = 1 << iota
    ServerReply
    ServerError
    UrgentMessage
)
```

Arrays

- No, never mind. Let's talk about slices instead.

Slices

```
var Ages map[string]int
Ages = make(map[string]int, 10)

Ages["Alice"] = 14
Ages["Bob"] = 22
Ages["Charlie"] = 27
Ages["Daria"] = 42
fmt.Println(Ages)

for name, age := range Ages {
    if age >= 18 {
        fmt.Printf("%s is allowed to vote.\n", name)
    } else {
        fmt.Printf("%s is not eligible to vote.\n", name)
    }
}
```


Maps

Conditionals

Loops

Structures

Method Functions

Composition

Polymorphism

Goroutines—Calling a Function in the “Background”

```
func countdown() {  
    for i := 10; i >= 0; i-- {  
        fmt.Printf(">>> %d <<<\n", i)  
        time.Sleep(1 * time.Second)  
    }  
}
```


Goroutines—Calling a Function in the “Background”

```
func countdown() {  
    for i := 10; i >= 0; i-- {  
        fmt.Printf(">>> %d <<<\n", i)  
        time.Sleep(1 * time.Second)  
    }  
}  
  
func main() {  
    countdown()  
    fmt.Println("Starting a long-running task...")  
    time.Sleep(15 * time.Second)  
    fmt.Println("Done. Exiting.")  
}
```

Goroutines—Calling a Function in the “Background”

```
func countdown() {  
    for i := 10; i >= 0; i-- {  
        fmt.Printf(">>> %d <<<\n", i)  
        time.Sleep(1 * time.Second)  
    }  
}  
  
func main() {  
    go countdown()  
    fmt.Println("Starting a long-running task...")  
    time.Sleep(15 * time.Second)  
    fmt.Println("Done. Exiting.")  
}
```

Global ID Generation (Naïve)

```
type GameState struct {  
    NextMessageID int  
}
```

Global ID Generation (Naïve)

```
type GameState struct {  
    NextMessageID int  
}  
  
var gameServer GameState  
  
gameServer.NextMessageID++  
client.ID = gameServer.NextMessageID
```

Global ID Generation (Mutex)

```
type GameState struct {  
    NextMessageID int  
    Lock          sync.Mutex  
}
```

Global ID Generation (Mutex)

```
type GameState struct {  
    NextMessageID int  
    Lock          sync.RWMutex  
}
```

Global ID Generation (Mutex)

```
type GameState struct {  
    NextMessageID int  
    Lock          sync.RWMutex  
}  
  
func (state *GameState) GetNextID() int {  
    state.Lock.Lock()  
    state.NextMessageID++  
    nextID := state.MessageID  
    state.Lock.Unlock()  
    return nextID  
}
```

Global ID Generation (Mutex)

```
type GameState struct {  
    NextMessageID int  
    Lock          sync.RWMutex  
}  
  
func (state *GameState) GetNextID() int {  
    state.Lock.Lock()  
    state.NextMessageID++  
    nextID := state.MessageID  
    state.Lock.Unlock()  
    return nextID  
}  
  
client.ID = gameServer.GetNextID()
```


Global ID Generation (Mutex)

```
type GameState struct {  
    NextMessageID int  
    Lock          sync.RWMutex  
}  
  
func (state *GameState) GetNextID() int {  
    state.Lock.Lock()  
    defer state.Lock.Unlock()  
  
    state.NextMessageID++  
    return state.NextMessageID  
}  
  
client.ID = gameServer.GetNextID()
```

Global ID Generation (Channel)

```
func serveMessageIDs(c chan int) int {  
    var id int  
    for {  
        c <- id  
        c++  
    }  
}
```

Global ID Generation (Channel)

```
func serveMessageIDs(c chan int) int {  
    var id int  
    for {  
        c <- id  
        c++  
    }  
}
```

```
IDSource := make(chan int)  
go serveMessageIDs(IDSource)
```

Global ID Generation (Channel)

```
func serveMessageIDs(c chan int) int {  
    var id int  
    for {  
        c <- id  
        c++  
    }  
}
```

```
IDSource := make(chan int)  
go serveMessageIDs(IDSource)  
  
client.ID = <-IDSource
```

Channels

```
ch := make(chan byte)
```

Channels

```
ch := make(chan byte)

fmt.Println("Writing to channel")

ch <- 42
```

Channels

```
ch := make(chan byte)

fmt.Println("Writing to channel")

ch <- 42

fmt.Println("Reading from channel")
x := <-ch
fmt.Println("Read", x, "from channel")
```

Channels

```
ch := make(chan byte)

fmt.Println("Writing to channel")

ch <- 42          // DEADLOCKED!

fmt.Println("Reading from channel")
x := <-ch
fmt.Println("Read", x, "from channel")
```


Channels

```
ch := make(chan byte)
go func(c chan byte) {
    x := <-c
    fmt.Println("Read", x, "from channel")
}(ch)

fmt.Println("Writing to channel")
ch <- 42
```

Buffered Channels

```
ch := make(chan byte, 1)
```

Buffered Channels

```
ch := make(chan byte, 1)

fmt.Println("Writing to channel")
ch <- 42

fmt.Println("Reading from channel")
x := <-ch
fmt.Println("Read", x, "from channel")
```