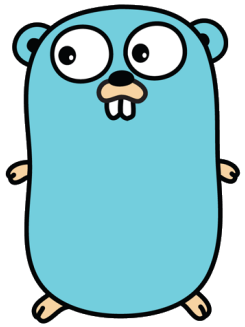
A Whirlwind Tour of Go

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..."
  - $\therefore$ 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..."
  - $\therefore$ 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..."
  - $\therefore$ 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` (single-character Unicode `'A'`, `'\n'`, `'Σ'`).

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` (single-character Unicode `'A'`, `'\n'`, `'Σ'`).
- 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` (single-character Unicode `'A'`, `'\n'`, `'Σ'`).
- Arrays: `[10]int`, `[100]rune`.
- Slices: `[]int`, `[]byte`, `[]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`.
- Structures: `struct{…}`.
- What about `char`? Nope. Instead, we have `rune` (single-character Unicode `'A'`, `'\n'`, `'Σ'`).
- Arrays: `[10]int`, `[100]rune`.
- Slices: `[]int`, `[]byte`, `[]string`.
- Maps: `map[string]int`.

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` (single-character Unicode `'A'`, `'\n'`, `'Σ'`).
- Arrays: `[10]int`, `[100]rune`.
- Slices: `[]int`, `[]byte`, `[]string`.
- Maps: `map[string]int`.
- Channels: `chan 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

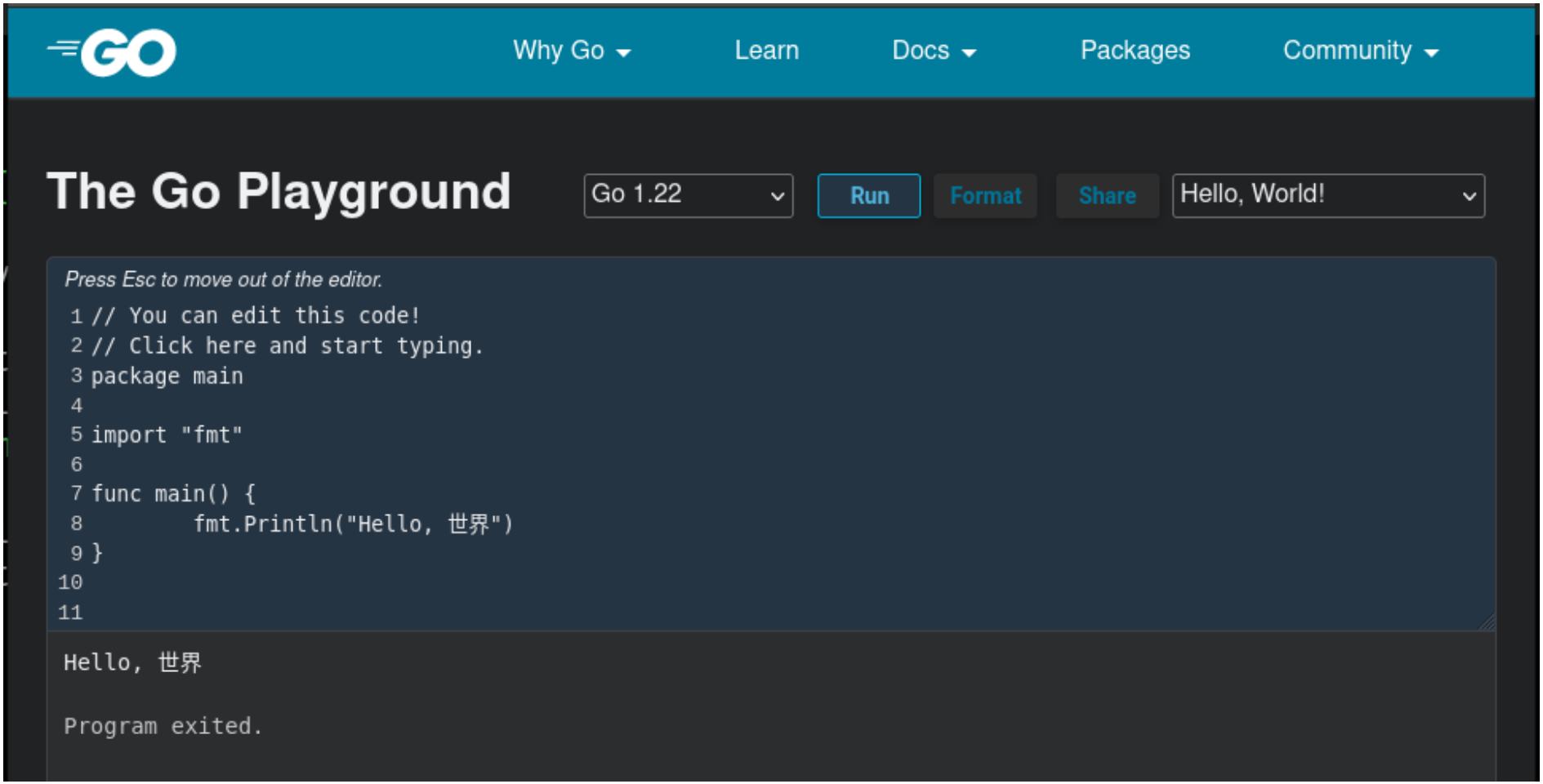
- Basic unit is a *package* (namespace boundary).
- Multiple source files in a package, in the same directory tree.
- Every program must have a `main` package.
- The `main` package has a `main` function.
- Import packages into the program using the `import` statement.
- Always prefix identifiers from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



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*

GO | Search packages or symbols | Why Go | Learn | Docs | Packages | Community

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
)
```

Conditionals

```
var x int

if x > 10 {
    fmt.Println("X␣exceeds␣10.")
} else {
    fmt.Println("X␣is␣tiny.")
}

if x *= 2; x > 10 {
    fmt.Println("Now␣X␣is␣big.")
} else {
    fmt.Println("X␣is␣still␣small.")
}
```

Switches

```
var x int

switch x {
case 0:
    fmt.Println("X is nothing.")
case 1, 3, 5:
    fmt.Println("X is odd.")
case 2, 4, 6:
    fmt.Println("X is even.")
default:
    fmt.Println("X is bigger than I can count.")
}
```

Loops

```
// infinite loop
for {
}

// while loop
for thing.IsReady() {
}

// traditional 3-part for loop
for i := 0; i < 10; i++ {
}
```

Loops

```go
// loop over interval [0,10)
for i := range 10 {
}

// loop over elements of a collection
for i, v := range []int{1, 4, -3, 153} {
}

// loop over data received from channel
for item := range channel {
}
```

Arrays

- The number of elements *is part of the type* (`[10]int` vs. `[15]int`).

Arrays

- The number of elements *is part of the type* ([10]int vs. [15]int).
- Variables declared are initialized empty but ready for use

```
var things [5]string

things[0] = "raindrops on roses"
things[1] = "whiskers on kittens"
things[2] = "copper kettles"
things[3] = "woolen mittens"
things[4] = "doorbells"

fmt.Println("I like", things[2])
fmt.Println("I also like", things)
fmt.Println("I know", len(things), "things.")
```

Arrays

- Or you can specify an array literal value to use in an expression or assign to a variable

```
things := [5]string{
    "raindrops␣on␣roses",
    "whiskers␣on␣kittens",
    "copper␣kettles",
    "woolen␣mittens",
    "doorbells",
}

fmt.Println("I␣like", things[2])
fmt.Println("I␣also␣like", things)
fmt.Println("I␣know", len(things), "things.")
```

Slices

- Specify a range [*n*:*m*] as the index into an array to get a subset of the array values with indices from *n* to $m - 1$.
- The value is a *slice*, not an *array*. It's a different type.
  - For [5]`string`, the value is []`string`.

```
fmt.Println("Some things:", things[1:3])
fmt.Println("Some things:", things[:3])
fmt.Println("Some things:", things[1:])
fmt.Println("Some things:", things[:])
```

Slices

- Dimensionless "arrays": `[]int`.
- Actually a "view" into an underlying array.
  - Go creates and manages the underlying array automatically for you.

```
var things []string

things = append(things, "doorbells")
things = append(things, "sleighbells", "schnitzel")
fmt.Println(len(things), things)
// prints:  3 [doorbells sleighbells schnitzel]
```

Slices

- Can also specify a slice of values as a literal.

```
things := []string{
    "doorbells",
    "sleighbells",
    "schnitzel",
}
primes := []int{2, 3, 5, 7, 11, 13}
lowPrimes := slices.Delete(primes, 3, len(primes))
fmt.Println(lowPrimes)
// prints:  [2 3 5]
```

Maps

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

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 may vote.\n", name)
    } else {
        fmt.Printf("%s is not eligible.\n", name)
    }
}
```

Maps

```
Ages := map[string]int{
    "Alice": 14,
    "Bob": 22,
    "Charlie": 27,
    "Daria": 42,
}
fmt.Println(Ages)

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

Maps

```
aliceAge := Ages["Alice"]        // 14
eveAge := Ages["Eve"]            // 0

aliceAge, exists := Ages["Alice"]        // 14, true
eveAge, exists := Ages["Eve"]            // 0, false

Ages["Eve"] = 20
delete(Ages, "Bob")

if _, exists := Ages[name] {
    fmt.Prinln("We␣do␣know␣about", name)
}

if age, exists := Ages[name]; exists {
    fmt.Printf("We␣know␣%s's␣age␣is␣%d.\n", name, age)
} else {
```

Structures

```
type Triangle struct {
    Base int
    Height int
    X int
    Y int
}

var t1 Triangle
var t2 Triange = Triangle{Base: 3, Height: 1}
t3 := Triangle{
    Base:   100,
    Height: 42,
    X:      -3,
    Y:      14,
}
```

Method Functions

```
func Area(t Triangle) float64 {
    return (t.Base * t.Height) / 2.0
}

func Translate(t Triangle, dx, dy int) Triangle {
    t.X += dx
    t.Y += dy
    return t
}

fmt.Println("t1 area =", Area(t1))
t2 = Translate(t2, +3, -2)
```

Method Functions

```
func Area(t Triangle) float64 {
    return (t.Base * t.Height) / 2.0
}

func Translate(t *Triangle, dx, dy int) {
    t.X += dx
    t.Y += dy
}

fmt.Println("t1 area =", Area(t1))
Translate(&t2, +3, -2)
```

Method Functions

```
func (t Triangle) Area() float64 {
    return (t.Base * t.Height) / 2.0
}

func (t *Triangle) Translate(dx, dy int) {
    t.X += dx
    t.Y += dy
}

fmt.Println("t1 area =", t1.Area())
t2.Translate(+3, -2)
```

Composition

```
type baseShape struct {
    X int
    Y int
}

func (s *baseShape) Translate(dx, dy int) {
    s.X += dx
    s.Y += dy
}
```

Composition

```
type Triangle struct {
    baseShape
    Base   int
    Height int
}

func (t Triangle) Area() float64 {
    return (t.Base * t.Height) / 2.0
}
```

Composition

```
type Rectangle struct {
    baseShape
    Width  int
    Height int
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Composition

```
type Polygon struct {
    baseShape
    Sides  int
    Length float64
    Radius float64
}

func (p Polygon) Area() float64 {
    return float64(p.Sides) / 2.0 * p.Length * p.Radiu
}
```

Composition

```
type Circle struct {
    baseShape
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * math.Pow(c.Radius, 2)
}
```

Polymorphism

```
shapes := []Shape{
    Triangle{X: 3, Y: 12, Base: 3, Height: 2},
    Circle{X: 0, Y: 22, Radius: 1.5},
    Rectangle{Height: 100, width: 50},
}

for i, shape := range shapes {
    fmt.Println("#%d␣area=%f\n", i, shape.Area())
    shape.Translate(-1, -1)
}
```

Polymorphism

```
type Shape interface {
    Area() float64
    Translate(int, int)
}

func reportArea(s Shape) {
    fmt.Printf("The area is %f\n", s.Area()
}
```

Type Assertions

```
f(42)
f(-2)

func f(mystery any) {     // any == interface{}
    var v int

    // we know it's an int, just treat it as one
    v = mystery + 15

    fmt.Println("int␣mystery␣is", v)
}
```

Type Assertions

```
f(42)
f(-2)

func f(mystery any) {     // any == interface{}
    var v int

    x := mystery.(int)
    v = x + 15

    fmt.Println("int␣mystery␣is", v)
}
```

Type Assertions

```
f(42)
f("hello")

func f(mystery any) {     // any == interface{}
    var v int

    x := mystery.(int)
    v = x + 15

    fmt.Println("int␣mystery␣is", v)
}
```

Type Assertions

```
f(42)
f("hello")

func f(mystery any) {     // any == interface{}
    var v int

    x, ok := mystery.(int)
    v = x + 15

    fmt.Println("int␣mystery␣is", v)
}
```

Type Switch

```
f(42)
f("hello")

func f(mystery any) {     // any == interface{}
    var v int

    switch x := mystery.(type) {
    case int:
        v = x + 15
    case string:
        fmt.Println("string", x)
    default:
        // handle the unknown type
    }
}
```

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")
```

Error Values

```
func main() {
    var intval int
    var err    error

    for i, arg := range os.Args {
        intval, err := strconv.Atoi(arg)
        if err != nil {
            fmt.Printf("Arg␣#%d␣(\"%s\"):␣%v.\n",
                        i, arg, err)
        } else {
            fmt.Printf("Arg␣#%d␣==␣%d\n", i, intval)
        }
    }
}
```

Error Values

```
func main() {
    var intval int
    var err    error

    for i, arg := range os.Args {
        intval, err = strconv.Atoi(arg); err != nil {
            fmt.Printf("Arg␣#%d␣(\"%s\"):␣%v.\n",
                        i, arg, err)
        } else {
            fmt.Printf("Arg␣#%d␣==␣%d\n", i, intval)
        }
    }
}
```