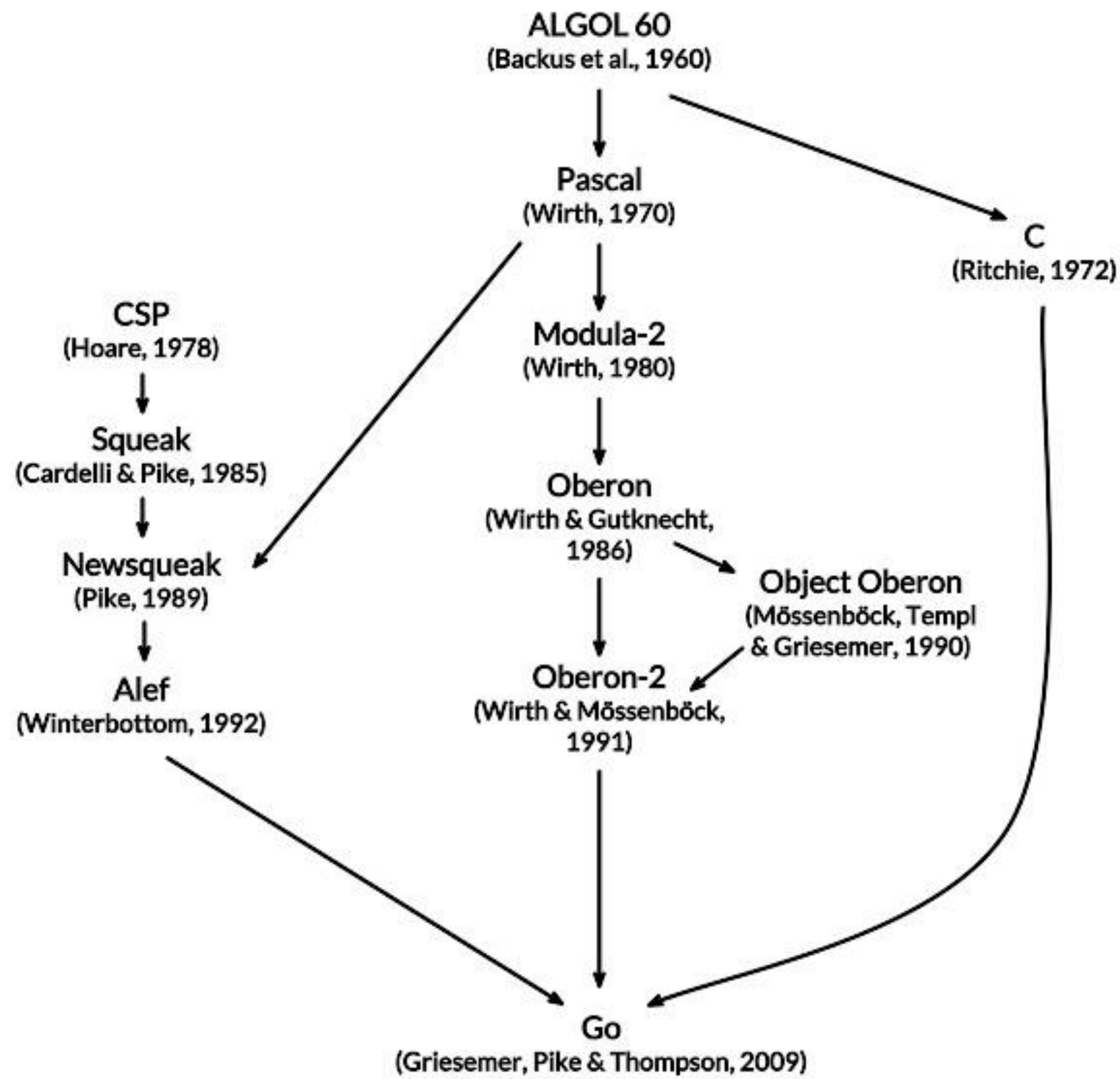
The Go Programming Language

Influences



ALGOL 60
(Backus et al., 1960)
Pascal
(Wirth, 1970)
C
(Ritchie, 1972)
CSP
(Hoare, 1978)
Modula-2
(Wirth, 1980)
Squeak
(Cardelli & Pike, 1985)
Oberon
(Wirth & Gutknecht, 1986)
Newsqueak
(Pike, 1989)
Object Oberon
(Mössenböck, Templ & Griesemer, 1990)
Alef
(Winterbottom, 1992)
Oberon-2
(Wirth & Mössenböck, 1991)
Go
(Griesemer, Pike & Thompson, 2009)

Go

- **developed ~2007 at Google by**

  Robert Griesemer, Rob Pike, Ken Thompson

- **open source**
- **compiled, statically typed**
  - very fast compilation
- **C-like syntax**
- **garbage collection**
- **built-in concurrency**
- **no classes or type inheritance or overloading or generics**
  - unusual interface mechanism instead of inheritance

Why GoLang?

- Why to create a new language at all?
- Three key languages exist: Java, Python, C/C++
- There were some limitations that Google was running into, that might not be able to fix, given the history and designs of the existing languages.
- Python is very easy to use, but it is interpreted languages. So it can be a little bit of difficult to run applications at Google scale.
- Java is very quick but its type system is becoming increasingly complex over the time as additional features are layered into the language.
- C/C++ are quick as well, but it suffers from a complex type system. Also its compile times are notoriously slow.
- All these three languages were created at the time when multithread applications were rare. So working with highly parallel and concurrent applications like Google is challenging.

Key Features

- **Strong and Statically Typed**
- **Excellent Community**
- **Simplicity in Features Supported**
- **Fast Compilation**
- **Garbage Collector Support**
- **Built-in Concurrency**
- **Compile to standalone binary (with all Go dependency)**

Useful Resources:

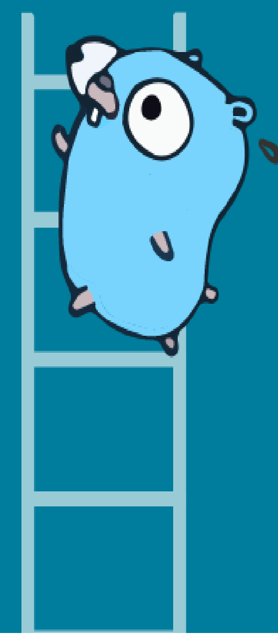
- **https://go.dev/**
- **https://go.dev/doc/effective_go**
- **https://go.dev/doc/**
- **https://go.dev/play/**

GO

Why Go · Get Started · Docs · Packages · Play · Blog

Build fast, reliable, and efficient software at scale

- ✓ Go is an open source programming language supported by Google
- ✓ Easy to learn and get started with
- ✓ Built-in concurrency and a robust standard library
- ✓ Growing ecosystem of partners, communities, and tools

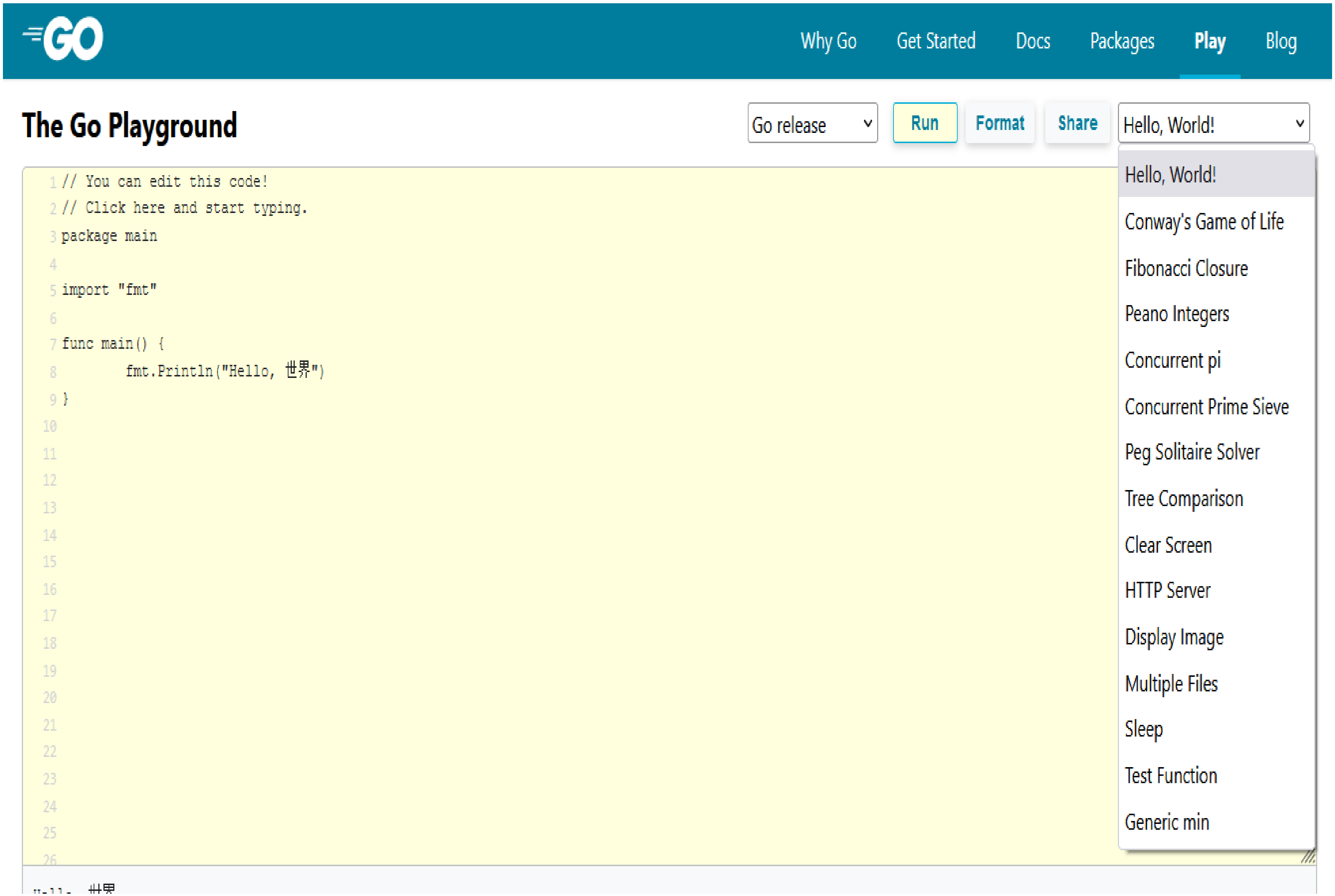


Get Started | Download

Download packages for Windows 64-bit, macOS, Linux, and more

The go command by default downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. Learn more.

Companies using Go


GO
Why Go
Get Started
Docs
Packages
Play
Blog
The Go Playground
Go release
Run
Format
Share
Hello, World!
// You can edit this code!
// Click here and start typing.
package main
import "fmt"
func main() {
fmt.Println("Hello, 世界")
}
Hello, World!
Conway's Game of Life
Fibonacci Closure
Peano Integers
Concurrent pi
Concurrent Prime Sieve
Peg Solitaire Solver
Tree Comparison
Clear Screen
HTTP Server
Display Image
Multiple Files
Sleep
Test Function
Generic min

- Managing Go installations -- How to install multiple versions and uninstall.
- Installing Go from source -- How to check out the sources, build them on your own machine, and run them.

1. Go download.

Click the button below to download the Go installer.

Download Go for Linux
go1.17.6.linux-amd64.tar.gz (129 MB)

Don't see your operating system here? Try one of the other downloads.

Note: By default, the go command downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. Learn more.

2. Go install.

Select the tab for your computer's operating system below, then follow its installation instructions.

Linux Mac Windows

1. Extract the archive you downloaded into /usr/local, creating a Go tree in /usr/local/go.

   Important: This step will remove a previous installation at /usr/local/go, if any, prior to extracting. Please

Getting Library Packages: go get github.com/nsf/gocode

Hello world example

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, 世界")
}
```

fmt: Format Strings Library Package

Some Notes

- No semicolon at the end of statements or declarations
- Go natively handles Unicode
- Every Go program is made up of packages (similar to C libraries or Python packages)
  - Package: one or more .go source files in a single directory
- Source file begins with package declaration (which package the file belongs to), followed by list of other imported packages
  - Programs start running in main
  - fmt package contains functions for printing formatted output and scanning input

Go Tool

- Go is a compiled language

- To compile & run the program, use go run
  - $ go run helloworld.go
  - hello, 世界

- To build the program into binary, use go build
  - $ go build helloworld.go

- $ ls helloworld*
  - helloworld   helloworld.go

- $ ./helloworld
  - hello, 世界

Packages

- Go codes live in packages
- Programs start running in package main
- Packages contain type, function, variable, and constant declarations
- Packages can even be very small or very large
- Case determines visibility: a name is exported if it begins with a capital letter
  - Foo is exported, foo is not

Imports

- `Import` statement: groups the imports into a parenthesized, “factored” statement

```
package main
import (
    "fmt"
    "math")

func main() {
    fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```

Functions

- Function can take zero or more arguments

```
func add(x int, y int) int {
    return x + y
}
```

  - `add` takes as input two arguments of type `int`
- Type comes *after* variable name
- Shorter version for input arguments:

```
func add(x, y int) int {
```

- Function can return any number of results

```
func swap(x, y string) (string, string) {
    return y, x
}
```

  - Useful to to return both result and error values

Functions

```
package main

import "fmt"

func swap(x, y string) (string, string) {
        return y, x
}

func main() {
        a, b := swap("hello", "world")
        fmt.Println(a, b)
}
```

Functions

- Return values can be named

```
package main

import "fmt"

func split(sum int) (x, y int) {
	x = sum * 4 / 9
	y = sum - x
	return // same as return x, y
}

func main() {
	fmt.Println(split(17))
}
```

Variables

- **var** statement: declares a list of variables
  - Type is last
- Can be at package or function level

```
package main
import "fmt"

var c, python, java bool

func main() {
    var i int
    fmt.Println(i, c, python, java)
}
```

- Can include initializers, one per variable
  - If initializer is present, type can be omitted
- Variables declared without an explicit initial value are given their *zero value*
- Short variable declaration using `:=`

Types

- Usual basic types
  - `bool, string, int, uint, float32, float64, …`
- Type conversion

```
var i int = 42
var f float64 = float64(i)
```

  - Unlike in C, in Go assignment between items of different type requires an explicit conversion
- Type inference
  - Variable's type inferred from value on right hand side

```
var i int
j := i // j is an int
```

Flow control statements

- for, if (and else), switch
- defer

Looping construct

- Go has only one looping construct: **for** loop
- Three components
  - Init statement
  - Condition expression
  - Post statement

```
sum := 0
    for i := 0; i < 10; i++ {
            sum += i
    }
```

- No parentheses surrounding the three components of the for statement
- Braces { } are always required

Looping construct

- Init and post statements are optional: for is Go's "while”

```
sum := 1
    for sum < 1000 {
          sum += sum
    }
```

- If we omit condition, infinite loop

```
for {
    }
```

Example: echo

```
// Echo prints its command-line arguments.
package main
import (
        "fmt"
        "os"
)
func main() {
        var s, sep string
        for i := 1; i < len(os.Args); i++ {
                s += sep + os.Args[i]
                sep = " "
        }
        fmt.Println(s)
}
```

"os": provides a platform-independent interface to operating system functionality

s and sep initialized to empty strings

`os.Args` is a slice of strings (see next slides)

Conditional statements: if

- Go's **`if`** (and **`else`**) statements are like for loops:
  - expression is not surrounded by parentheses ( )
  - but braces { } are required

```
if v := math.Pow(x, n); v < limit {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, limit)
    }
```

  - Remember that } `else` must be on the same line
  - Variable v is in scope only within the `if` statement
- `if...else if...else` statement to combine multiple `if...else` statements

Conditional statements: switch

- `switch` statement selects one of many cases to be executed
  - Cases evaluated from top to bottom, stopping when a case succeeds
- Differences from C
  - Go only runs the selected case, not all the cases that follow (i.e., C's break is provided automatically in Go)
  - Switch cases need not be constants, and the values involved need not be integers

Defer statement

- New mechanism to defer the execution of a function *until* the surrounding function returns
  - The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function that contains `defer` has terminated

```
package main
import "fmt"

func main() {
  defer fmt.Println("world")
  fmt.Println("hello")
}
```

```
hello
world
```

- Deferred function calls pushed onto a stack
  - Deferred calls executed in LIFO order
- Great for cleanup things, like closing files or connections!

Pointers

- Pointer: value that contain the address of a variable
  - Usual operators * and &: & operator yields the address of a variable, and * operator retrieves the variable that the pointer refers to

```
var p *int
i := 1
p = &i    // p, of type *int, points to i
fmt.Println(*p) // "1"
*p = 2 // equivalent to i = 2
fmt.Println(i) // "2"
```

- Unlike C, Go has no pointer arithmetic
- Zero value for a pointer is `nil`
- Perfectly safe for a function to return the address of a local variable

Composite data types: structs and array

- Aggregate data types: structs and arrays
- **Struct**: a collection of fields
  - Syntax similar to C, fixed size

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

  - Struct fields are accessed using a dot; can also be accessed through a struct pointer

- **Array**: `[n]T` is an array of n values of type T
  - Fixed size (cannot be resized)

```
var a [2]string
a[0] = "Hello"
```

Composite data types: slices

- `[ ]T` is a **slice** with elements of type `T`: dynamically-sized, flexible view into the elements of an array
  - Specifies two indices, a low and high bound, separated by a colon: `s[i : j]`
  - Includes first element, but excludes last one

```
primes := [6]int{2, 3, 5, 7, 11, 13}
var s []int = primes[1:4]
```

```
[3 5 7]
```

- Slice is a section of an *underlying array*: modifies the elements of the corresponding array
- Length of slice `s`: number of elements it contains, use `len(s)`
- Capacity of slice `s`: number of elements in the underlying array, counting from the first element in the slice, use `cap(s)`

Composite data types: slices

- It is a compile or run-time error to exceed the length (bounds-checked)
- Can be created using **make**
  - Length and capacity can be specified

```
package main
import "fmt"
func main() {
  a := make([]int, 0, 5)      // len(a)=0, cap(a)=5
  printSlice("a", a)
}
func printSlice(s string, x []int) {
  fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x)
}
```

```
b len=0 cap=5 []
```

Composite data types: slices

- New items can be appended to a slice using **append**

```
func append(slice []T, elems ...T) []T
```

  – When append a slice, slice may be enlarged if necessary

```
func main() {
    var s []int
    printSlice(s)

    s = append(s, 0) // works on nil slices
    printSlice(s)

    s = append(s, 1) // slice grows as needed
    printSlice(s)

    s = append(s, 2, 3, 4) // more than one element
    printSlice(s)
}
```

Composite data types: maps

- map: maps keys to values
  - Map type `map[K]V` is a reference to a hash table where K and V are the types of its keys and values
  - Use `make` to create a map

```
m = make(map[string]Vertex)
m["Bell Labs"] = Vertex{
   40.68433, -74.39967,
}
```

- You can insert or update an element in a map, retrieve an element, delete an element, test if a key is present

```
m[key] = element        // insert or update
elem = m[key]           // retrieve
delete(m, key)          // delete
elem, ok = m[key]       // test
```

Range

- range iterates over elements in a variety of data structures
  - range on arrays and slices provides both index and value for each entry
  - range on map iterates over key/value pairs

```
package main
import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

Range: example

```go
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
```

```
$ go run range2.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
```

Methods

- Go does not have classes, but supports **methods** defined on struct types
- A method is a function with a special *receiver* argument (extra parameter before the function name)
  - The receiver appears in its own argument list between the `func` keyword and the method name

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Interfaces

- An *interface type* is defined as a named collection of method signatures
- Any type (struct) that implements the required methods, implements that interface
  - Instead of designing the abstraction in terms of what kind of data our type can hold, we design the abstraction in terms of *what actions* our type can execute
- A type is not explicitly declared to be of a certain interface, it is implicit
  - Just implement the required methods
- Let's code a basic interface for geometric shapes

Interface: example

```
package main

import "fmt"
import "math"

// Here's a basic interface for geometric shapes.
type geometry interface {
    area() float64
    perim() float64
}

// For our example we'll implement this interface on
// rect and circle types.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```

Interface: example

```
// To implement an interface in Go, we just need to
// implement all the methods in the interface. Here we
// implement `geometry` on `rect`s.
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// The implementation for `circle`s.
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Interface: example

```go
// If a variable has an interface type, then we can call
// methods that are in the named interface. Here's a
// generic `measure` function taking advantage of this
// to work on any `geometry`.
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The `circle` and `rect` struct types both
    // implement the `geometry` interface so we can use
    // instances of these structs as arguments to `measure`.
    measure(r)
    measure(c)
}
```

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

Concurrency in Go

- Go provides concurrency features as part of the core language
- Goroutines and channels
  - Support CSP concurrency model
- Can be used to implement different concurrency patterns

Goroutines

- A **goroutine** is a lightweight thread managed by the Go runtime

```
go f(x, y, z)  // starts a new goroutine running
               // f(x, y, z)
```

- Goroutines run in the same address space, so access to shared memory must be synchronized

Channels

- Communication mechanism that lets one goroutine sends values to another goroutine
  - A channel is a thread-safe queue managed by Go and its runtime
  - It blocks threads that read on it, etc.
- Hides a lot of pain of inter-thread communication
  - Internally, it uses mutexes and semaphores just as one might expect
- Multiple senders can write to the same channel
  - Really useful for notifications, multiplexing, etc.
- And it's totally thread-safe!
- But be careful: only one can `close` the channel, and can't send after close!

Channels

- A typed conduit through which you can send and receive values using the channel operator `<-`

```
ch <- v      // Send v to channel ch
v := <- ch // Receive from ch, and
             // assign value to v
```

Data flows in the direction of the arrow

- Channels must be created before use

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

- Sends and receives block until the other side is ready
  - Goroutines can synchronize without explicit locks or condition variables

Channels: example

Distributed sum: sum is distributed between two Goroutines

```
import "fmt"
func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // send sum to c
}
func main() {
        s := []int{7, 2, 8, -9, 4, 0}
        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // receive from c

        fmt.Println(x, y, x+y)
}
```

Channels: example

Fibonacci sequence: iterative using channel

```
package main
import "fmt"
func fib(c chan int) {
        x, y := 0, 1
        for {
                c <- x
                x, y = y, x+y
        }
}
func main() {
        c := make(chan int)
        go fib(c)
        for i := 0; i < 10; i++ {
                fmt.Println(<-c)
        }
}
```

More on channels

- By default, channel operations block
- Buffered channels do not block if they are not full
  - Buffer length as `make` second argument to initialize a buffered channel

    ```
    ch := make(chan int, 100)
    ```

  - Sends to a buffered channel block only when buffer is full
  - Receives block when buffer is empty
- Close and range on buffers
  - Sender can `close` a channel
  - Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression

    ```
    v, ok := <-ch
    ```

    - ok is false if there are no more values to receive and the channel is closed
  - Use `for i := range ch` to receive values from the channel repeatedly until it is closed

More on channels

- `select` can be used to wait for messages on one of several channels

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...use x...
case ch3 <- y:
    // ...
default:
    // ...
}
```

- You can implement timeouts by using a timer channel

```
//to wait 2 seconds
timer := time.NewTimer(time.Second * 2)
    <- timer.C
```

Using select: example

Fibonacci sequence: iterative using two channels, the latter being used to quit

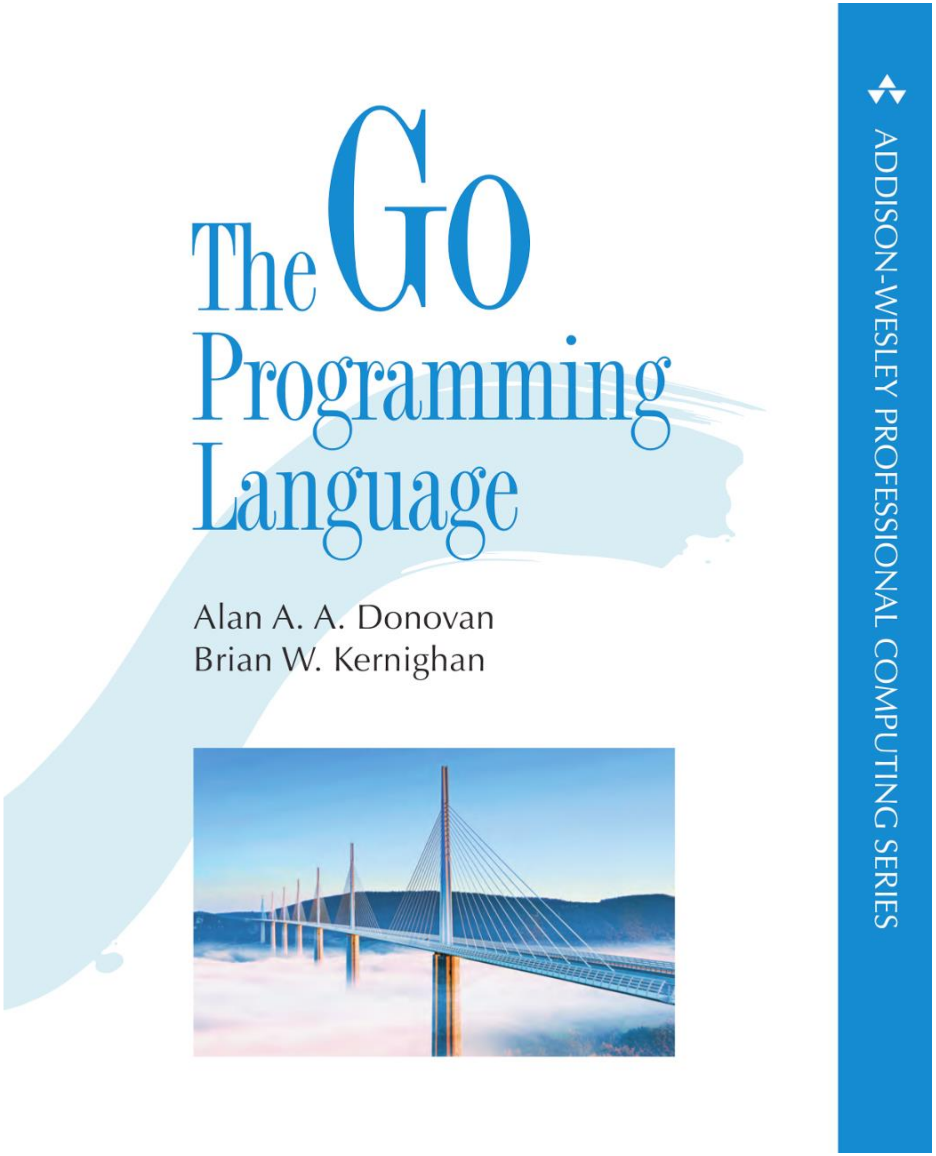
```
package main
import "fmt"

func fibonacci(c, quit chan int) {
        x, y := 0, 1
        for {
                select {
                case c <- x:
                        x, y = y, x+y
                case <-quit:
                        fmt.Println("quit")
                        return
                }
        }
}
```

Using select: example

```
func main() {
        c := make(chan int)
        quit := make(chan int)
        go func() {
                for i := 0; i < 10; i++ {
                        fmt.Println(<-c)
                }
                quit <- 0
        }()
        fibonacci(c, quit)
}
```

Reference


The Go Programming Language
Alan A. A. Donovan
Brian W. Kernighan
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Types, constants, variables

- **basic types**

  `bool string  int8 int16 int32 int64  uint8 …  int  uint`

  `float32 float64 complex64 complex128`

  quotes: `'世'`, `"UTF-8 string"`, `` `raw string` ``

- **variables**

```
var c1, c2 rune
var x, y, z = 0, 1.23, false    // variable decls

x := 0; y := 1.23; z := false  // short variable decl
```

  Go infers the type from the type of the initializer

  assignment between items of different type requires an explicit conversion, e.g., int(float_expression)

- **operators**
  - mostly like C, but `++` and `--` are postfix only and not expressions
  - assignment is not an expression
  - no `?:` operator

Echo command:

```
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Echo command (version 2):

```go
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

Arrays and slices

- an array is a fixed-length sequence of same-type items

```
months := [...]string {1:"Jan", 2:"Feb", /*...,*/ 12:"Dec"}
```

- a slice is a subsequence of an array

```
summer := months[6:9]; Q2 := months[4:7]
```

- elements accessed as `slice[index]`
  - indices from 0 to `len(slice)-1` inclusive

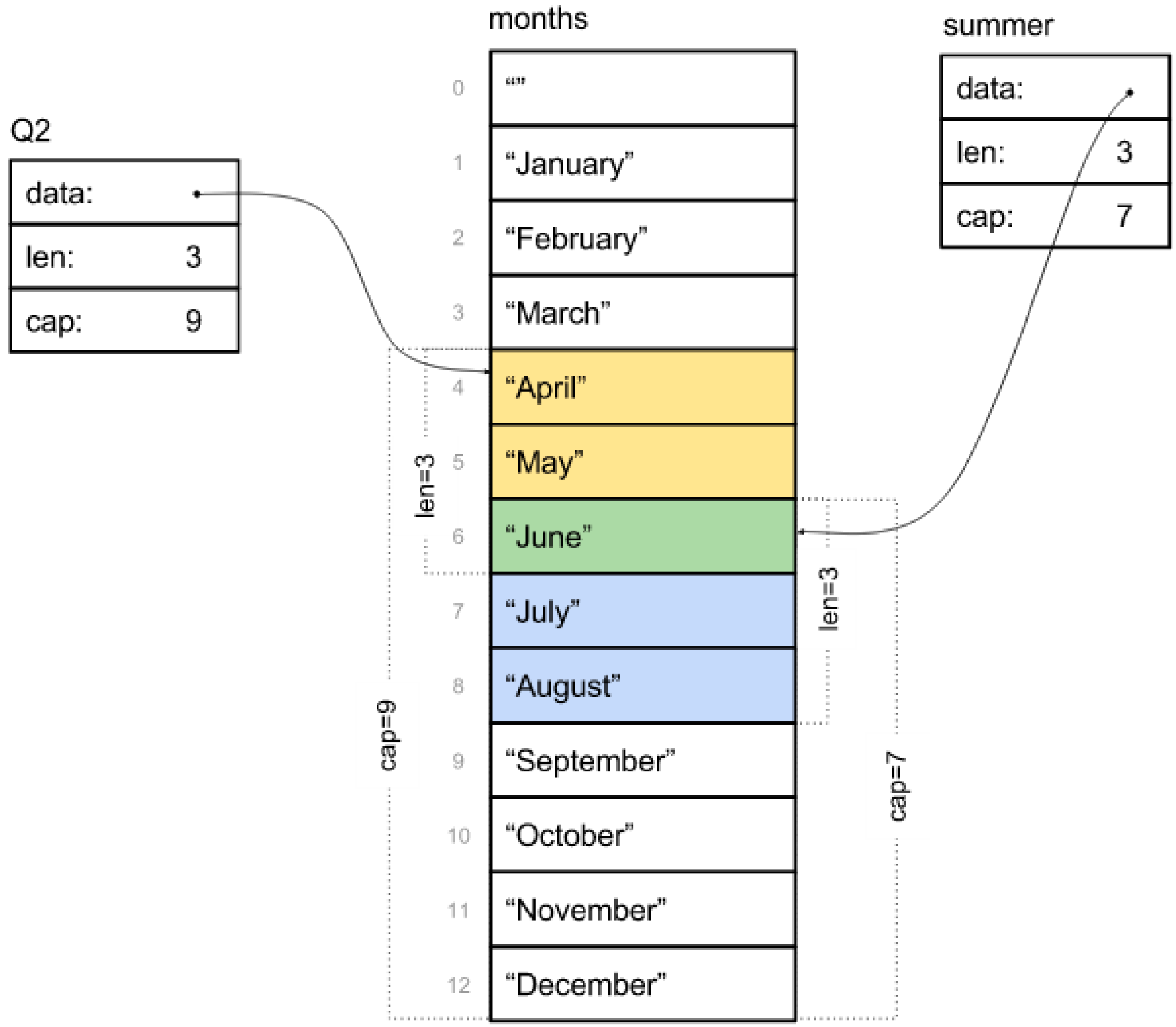
`summer[0:3]` is elements `months[6:9]`

```
summer[0] = "Juin"
```

- loop over a slice with for range

```
for i, v := range summer {
   fmt.Println(i, v)
}
```

- slices are very efficient (represented as small structures)
- most library functions work on slices



Q2
data:
len: 3
cap: 9
months
0 “”
1 “January”
2 “February”
3 “March”
4 “April”
5 “May”
6 “June”
7 “July”
8 “August”
9 “September”
10 “October”
11 “November”
12 “December”
len=3
cap=9
len=3
cap=7
summer
data:
len: 3
cap: 7

Maps (== associative arrays)

- **unordered collection of key-value pairs**
  - keys are any type that supports == and != operators
  - values are any type

```
// Find duplicated lines in stdin.
func main() {
  counts := make(map[string]int)
  in := bufio.NewScanner(os.Stdin)
  for in.Scan() {
    counts[in.Text()]++
  }
  for line, n := range counts {
    if n > 1 {
        fmt.Printf("%d\t%s\n", n, line)
    }
  }
}
```

Methods and pointers

- **can define methods that work on any type, including your own:**

```
type Vertex struct {
    X, Y float64
}
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    fmt.Println(v, v.Abs())
}
```

Interfaces

- **an interface is satisfied by any type that implements all the methods of the interface**
- **completely abstract: can't instantiate one**
- **can have a variable with an interface type**
- **then assign to it a value of any type that has the methods the interface requires**

- **a type implements an interface merely by defining the required methods**
  - it doesn't declare that it implements them

- **Writer: the most common interface**

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Sort interface

- **sort interface defines three methods**
- **any type that implements those three methods can sort**
- **algorithms are inside the soft package, invisible outside**

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

Sort interface (adapted from Go Tour)

```
type Person struct {
  Name string
  Age  int
}
func (p Person) String() string {
  return fmt.Sprintf("%s: %d", p.Name, p.Age)
}
type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

func main() {
  people := []Person{{"Bob",31}, {"Sue",42}, {"Ed",17}, {"Jen",26},}
  fmt.Println(people)
  sort.Sort(ByAge(people))
  fmt.Println(people)
}
```

Tiny version of curl

```
func main() {
  url := os.Args[1]
  resp, err := http.Get(url)
  if err != nil {
       fmt.Fprintf(os.Stderr, "curl: %v\n", err)
       os.Exit(1)
  }
  _, err = io.Copy(os.Stdout, resp.Body)
  if err != nil {
       fmt.Fprintf(os.Stderr, "curl: copying %s: %v\n",
                                            url, err)
       os.Exit(1)
  }
}
```

Tiny web server

```
func main() {
  http.HandleFunc("/", handler)
  http.ListenAndServe("localhost:8000", nil)
}

// handler echoes Path component of the request URL r.
func handler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

- `http.ResponseWriter` implements Writer interface

Concurrency: goroutines & channels

- **channel: a type-safe generalization of Unix pipes**
  - inspired by Hoare's Communicating Sequential Processes (1978)

- **goroutine: a function executing concurrently with other goroutines in the same address space**
  - run multiple parallel computations simultaneously
  - loosely like threads but much lighter weight

- **channels coordinate computations by explicit communication**
  - locks, semaphores, mutexes, etc., are much less often used

Example: web crawler

- **want to crawl a bunch of web pages to do something**
  - e.g., figure out how big they are
- **problem: network communication takes relatively long time**
  - program does nothing useful while waiting for a response
- **solution: access pages in parallel**
  - send requests asynchronously
  - display results as they arrive
  - needs some kind of threading or other parallel process mechanism
- **takes less time than doing them sequentially**

Version 1: no parallelism

```
func main() {
  start := time.Now()
  for _, site := range os.Args[1:] {
    count("http://" + site)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

func count(url string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    fmt.Printf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  fmt.Printf("%s %d [%.2fs]\n", url, n, dt)
}
```

Version 2: parallelism with goroutines

```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url, n, dt)
}
```

Version 2: main() for parallelism with goroutines

```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}
```

Version 2: count() for parallelism with goroutines

```
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url, n, dt)
}
```

Python version, no parallelism

```
import urllib2, time, sys

def main():
  start = time.time()
  for url in sys.argv[1:]:
    count("http://" + url)
  dt = time.time() - start
  print "\ntotal: %.2fs" % (dt)

def count(url):
  start = time.time()
  n = len(urllib2.urlopen(url).read())
  dt = time.time() - start
  print "%6d  %6.2fs   %s" % (n, dt, url)

main()
```

Python version, with threads

```
import urllib2, time, sys, threading

global_lock = threading.Lock()

class Counter(threading.Thread):
  def __init__(self, url):
    super(Counter, self).__init__()
    self.url = url

  def count(self, url):
    start = time.time()
    n = len(urllib2.urlopen(url).read())
    dt = time.time() - start
    with global_lock:
      print "%6d  %6.2fs   %s" % (n, dt, url)

  def run(self):
    self.count(self.url)

def main():
  threads = []
  start = time.time()
  for url in sys.argv[1:]:  # one thread each
    w = Counter("http://" + url)
    threads.append(w)
    w.start()

  for w in threads:
    w.join()
  dt = time.time() - start
  print "\ntotal: %.2fs" % (dt)

main()
```

Python version, with threads (main)

```
def main():
  threads = []
  start = time.time()
  for url in sys.argv[1:]:  # one thread each
    w = Counter("http://" + url)
    threads.append(w)
    w.start()

  for w in threads:
    w.join()
  dt = time.time() - start
  print "\ntotal: %.2fs" % (dt)

main()
```

Python version, with threads (count)

```
import urllib2, time, sys, threading

global_lock = threading.Lock()

class Counter(threading.Thread):
  def __init__(self, url):
    super(Counter, self).__init__()
    self.url = url

  def count(self, url):
    start = time.time()
    n = len(urllib2.urlopen(url).read())
    dt = time.time() - start
    with global_lock:
      print "%6d  %6.2fs   %s" % (n, dt, url)

  def run(self):
    self.count(self.url)
```

Where will Go go?

- comparatively small but rich language

- efficient; compilation is very fast
- concurrency model is convenient and efficient
- object model is unusual but seems powerful

- significant use at Google and elsewhere
- mostly for web server applications

    - "C for the 21st century" ?

Go source materials

- **official web site:**
  golang.org
- **Go tutorial, playground**
- **Rob Pike on why it is the way it is:**
  http://www.youtube.com/watch?v=rKnDgT73v8s
- **Russ Cox on interfaces, reflection, concurrency**
  http://research.swtch.com/gotour