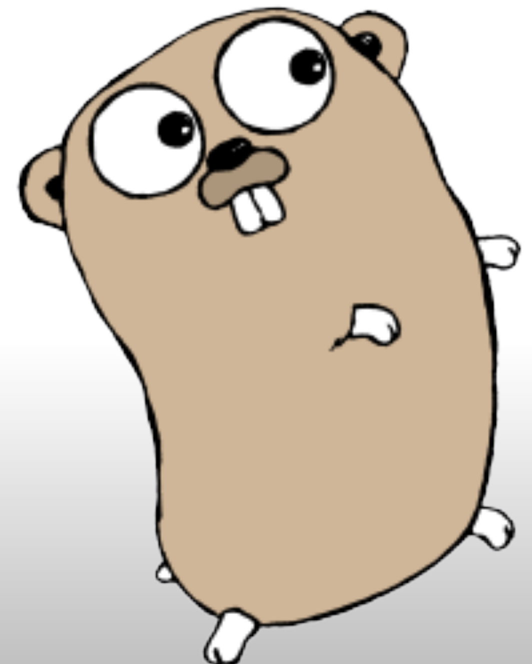
Google

Go

David Ellis
Matt Coble

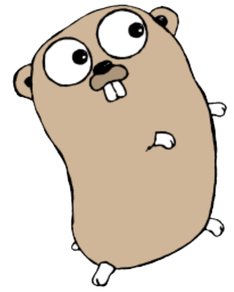


Gordon the Gopher

"The Go programming language is an open source project to make programmers more productive. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language."

- golang.org

Why **Go**?



"We realized that the kind of software we build at Google is not always served well by the languages we had available. Robert Griesemer, Ken Thompson, and myself decided to make a language that would be very good for writing the kinds of programs we write at Google." - **Rob Pike**

Google needed a language that would:

- Support efficient, controllable, easy-to-write concurrency
- Be easy to learn
- Run fast
- Be "fun to work in"

Who started **Go**?

The **Go** Core Design Team

- **Ken Thompson** - a "Unix Founding Father", creator of the B programming language, and the joint winner (along with Dennis Ritchie) for the 1983 Turing Award for Operating Systems Theory

- **Rob Pike** - Unix developer and creator of the first windowing system for Unix

- **Robert Griesemer** - part of the team that worked on Java HotSpot (the primary JVM)

Go Timeline

- September 2007 - Development Begins
- November 2009 - Official Announcement of Existence
- May 2010 - Rob Pike announces Go is being used for "real stuff" at Google
- Early 2012 - Go version 1 scheduled for release

Where is **Go** now?

- Fully Open Source
- Compilers available for Unix, Linux, and MacOS X using x86-32 and x86-64 processors, also available for ARM processors using Linux
- Windows compiler in process
- Supported by Google's App Engine cloud computing service
- Primary commercial use in Google's back-end systems

A Simple **Go** Program in Two Forms

```
package main;

import fmt "fmt";

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

```
package main
import (
    "fmt"
)

func main() {
    hello := "Hello"
    var world string
    world = "世界”
  fmt.Printf(”%s, %s¥n”,
        hello, world)
}
```

Go Primitive Types

- Machine-specific sizing: int, uint, bool(?), uintptr
- Explicitly sized: int8, int16, int32, int64, uint8 (aka byte), uint16, uint32, uint64, float32, float64, complex64, complex128

Each type is distinct from the others; explicit type-casting is required for any conversion.

Declaring Variables:

- Explicitly typed: `var i int = 0`
- Implicitly typed*: `i := 0`

All variables are statically typed.

*Unlike some languages with the := operator, Go uses := only for simultaneous declaration and assignment

Strings and Arrays in **Go**

Arrays

- Treated as values, not as references/pointers
- Can be referenced by pointers, or by **slices**
- Size is stored as part of type, and retrieved via **len()** function
- Declared by: `var intArray [10]int`

Strings

- Treated as variables which hold a string as an indexed value
- Can be compared using equality operators and combined using '+'
- Legal: `str := "hi"; str = "bye"; c := str[2]; ptr := &str`
- Illegal: `str[1] = 'o'; (*ptr)[1] = 'f'`

Reference types in **Go**

Slices

- Provide a bounded reference to a subarray
- Can reference an anonymous array
- Can be extended or combined using the append() function
- Example: `slice := myArr[1:4]; sort(myArr[2:5])`

Maps

- Store a keyed array
- Example: `m := map[string]int{"one":1 , "two":2}`

Functions in **Go**

- Declared via the **func** keyword
- Pass-by-value by default
- Can also accept pointers or reference types
- Can return multiple values

Examples:

```
func funk() int {
    ...
}
```

```
func funk(i *int, j int){
    ...
}
```

```
func funk()(i int, b bool){
    ...
}
```

Packages and Scope in **Go**

- A package is defined by a single source file (unlike package designations in Java), and can include multiple type definitions, functions, etc.
- Execution begins with a function named main() within a package named main
- The programmer can specify a custom name for a package when importing, otherwise the package is accessible through the same name it was imported as (not including file path)
- Functions, global variables, and types given a name beginning with a capital letter are "exported" and accessible to other packages by using package.member syntax, such as: `packageName.Function()`

User-Defined Types in **Go**

- The programmer can use the **type** keyword to define a custom type based around a pre-defined type, most commonly **struct**
- Methods for the custom type can then be defined within the same package by specifying a "receiver variable" for a function

```
type MyType struct {
    i int
    ...
}

func (m *MyType) Funk(i int) int {
    m.i = i
    ...
}
```

Polymorphism: **Go** Interfaces

- An **interface** consists of a collection of methods
- Any type that provides method(s) with the same name(s) and signature(s) is considered to implement the interface
- Functions can then specify parameters using the interface name, allowing them to accept any type that implements the interface

```
type MyInterface interface {
    MyFunction(...)
    ...
}

func Funk(m *MyInterface) int {
    m.MyFunction(...)
    ...
}
```

Concurrency in **Go**: goroutines and channels

goroutines provide a lightweight and simple way of running functions concurrently within the same namespace

- go runtime manages goroutines automatically
- lighter weight than threads
- start a goroutine by prefixing the function call with the **go** keyword

channels provide a reference type that can be used to communicate between goroutines

- can be buffered or unbuffered
- receivers block until data is received; senders block until data is written to buffer
- can send/receive complex types (including user-defined)

Sample code: **Concurrent Pi**

```
/*This demonstrates Go's ability to
handle large numbers of concurrent
processes. It is an unreasonable way to
calculate pi.*/
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(pi(5000))
}

/* pi launches n goroutines to compute
an approximation of pi. */
```

```
func pi(n int) float64 {
    ch := make(chan float64)
    for k := 0; k <= n; k++ {
        go term(ch, float64(k))
    }
    f := 0.0
    for k := 0; k <= n; k++ {
        f += <-ch
    }
    return f
}

func term(ch chan float64, k float64) {
    ch <- 4 * math.Pow(-1, k) / (2*k + 1)
}
```

Evaluating **Go**: Readability

Pros

- "clean-looking, semicolon-free code"
- similar to C-family languages
- post-fix typing makes function pointers clearer (than C)

Cons

- post-fix typing takes some getting used to
- implicit typing and semi-colon insertion

Evaluating **Go**: Writability

Pros:

- implicit typing, case-based scoping, and semi-colon insertion make writing quicker
- built-in reference types (slices, etc.) provide good checking
- custom type definitions allow great flexibility
- goroutines and channels make concurrency easy
- open-sourced language + large built-in library
- similar to other C-family languages
- concise package names
- interfaces provide polymorphism

Cons

- post-fix typing and conventions differing from other C-family languages take some getting used to
- no inheritance

Evaluating **Go**: Reliability

Pros:

- strongly typed and strongly type-checked, index-checking
- simple scoping
- exception-handling system

Cons:

- aliasing through slices and pointers
- limited scope options (exported or hidden, no others)
- implicit typing and semi-colon insertion can produce unexpected errors

Evaluating **Go**: Cost

Pros:

- free and open
- quick to write
- similar to C-family languages
- almost as fast as C++ (in some tests)
- profiling and formatting tools provided

Cons:

- new language: no experienced programmers, compiler still in progress, limited documentation
- slower than Java (in some tests)
- no IDE support, yet

Sources

- www.golang.org
- http://www.theregister.co.uk/2011/05/05/google_go/
- http://en.wikipedia.org/wiki/Go_%28programming_language%29
- http://news.cnet.com/8301-30685_3-20063164-264.html
- Google I/O 2011: Writing Web Apps in Go: http://www.youtube.com/watch?v=-i0hat7pdpk&noredirect=1
- http://www.theregister.co.uk/2011/06/03/google_paper_on_cplusplus_java_scala_go/