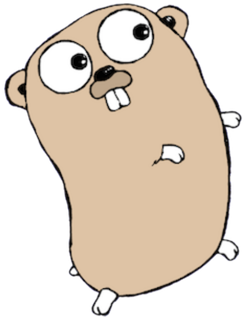
Go



- **developed ~2007 by**
  Robert Griesemer, Rob Pike, Ken Thompson
- **open source**

- **compiled, statically typed**
- **syntax looks sort of like C**
- **garbage collection**
- **built-in concurrency**
- **no classes or type inheritance or overloading or generics**
  - unusual interface mechanism instead of inheritance

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

Hello world in Go

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```

```
$ go run hello.go    # to compile and run

$ go help            # for more
```

Go constructs

- **constants, variables, types**
- **operators and expressions**
- **statements, control flow**
- **data: structs, pointers, arrays, slices, maps**
- **functions**
- **libraries and packages**
- **interfaces**
- **concurrency: *goroutines*, channels**
- **etc.**

Constants, variables, operators

- **constants**
  - bool, string;   int, float, complex (all of various sizes)
  - quotes: `‘char’`, `“string”`

```
const pi = 3.14
const World = "世界"     // strings are unicode
```

- **variables**

```
var x, y, z = 0, 1.23, false    // global variables
x := 0; y := 1.23; z := false   // inside a function
```

  Go infers the type from 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
  - string concatenation uses +

Statements, control flow: if-else

- **statements**
  - assignment, control flow, function call, ...
  - scope indicated by mandatory braces; no ; terminator needed
- **control flow: if-else, for, switch**

```
if opt-stmt; boolean {
    statements
} else if opt-stmt; boolean {
    statements
} else {
    statements
}

if c := getchar(); c != EOF { // scope of c is whole if-else
    ...
}
```

More control flow: for

- **Looping with for**

```
for opt-stmt; boolean; opt-stmt { // can drop stmts and ;'s
    statements     // break, continue    (with optional labels)
}

for {
    // runs for a long time
}

for index := range something {
    ...
}
```

More control flow: switch

- **Switch**

```
switch opt-stmt; opt-expr {
case exprlist: statements    // no fallthrough
case exprlist: statements
default: statements
}
```

```
switch Suffix(file) {
case ".gz":  return GzipList(file)
case ".tar": return TarList(file)
case ".zip": return ZipList(file)
}
```

  - can also switch on types

Structs and pointers (adapted from Go Tour)

```
type Vertex struct {
    X, Y int
}
var (
    p = Vertex{1, 2}  // has type Vertex
    q = &Vertex{1, 2} // has type *Vertex
    r = Vertex{X: 1}  // Y:0 is implicit
    s = Vertex{}      // X:0 and Y:0
    t = new(Vertex)   // pointer to a 0,0 vertex
)
func main() {
    fmt.Println(p, q, r, s, t)
}
```

Arrays and slices

- **an array is a fixed-length sequence of same-type items**
- **a slice is a variable-length but fixed capacity**

```
food := []string {"beer", "pizza", "coffee"}
```

- **use `make` to create new slices**

```
food := make([]string, 3, 10) // initial len, [capacity]
```

- **elements accessed as `slice[index]`**
  - indices from 0 to `len(slice)-1` inclusive
  - slicing: `food[start:end]` is elements `start..end-1`
- **slices are very efficient (passed as small structures)**
  - arrays are passed by value
- **most library functions work on slices**
- **slices are mutable: if the slice changes, that's visible to all variables that refer to it**

Maps (== associative arrays)

- **unordered collection of key-value pairs**
  - keys are any type that supports == and != operators (e.g., built-ins)
  - values are any type

```
m := map[string] int {"pizza":200, "beer":100}
m["coke"] = 50
wine := m["wine"]            // 0 if not there
coffee, found := m["coffee"] // 0, false if not present
delete(m, "chips")        // ok if not present
```

Functions

```
func name(arg, arg, arg) (ret, ret) {
    statements of function
}

func div(num, denom int) (q, r int) {
    // computes quotient & remainder. denom should be > 0
    q = num / denom
    r = num % denom
    return     // returns two named values, q and r
}
```

- **functions are objects**
  - can assign them, pass them to functions, return them from functions
- **parameters are passed call by value (including arays!)**
- **functions can return any number of results**

- **defer statement queues operation until function returns**

```
    defer f.close()
```

Methods & pointers

- can define methods 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())
}
```

Methods, pointers and interfaces

- **can attach methods to any type**
- **fmt package uses %s to print anything that has a String() method**
  - %v uses reflection to print any type at all
  - type information and some basic operations available at run time

```
type World int  // defines a new type.  could be any type here

func (w *World) String() string {  // receiver w unused here
    return "world"
}

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

Interfaces

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

- 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

  `interface{}` is empty set of methods

  so every value satisfies `interface{}`

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

Sort interface

- Sort interface defines three methods
- any type that implements those three methods can sort

```
// Package sort provides primitives for sorting slices
// and user-defined collections.
package sort

// A type, typically a collection, that satisfies sort.Interface
// can be sorted by the routines in this package.  The methods
// require that the elements of the collection be enumerated by
// an integer index.
type Interface interface {
  // Len is the number of elements in the collection.
  Len() int
  // Less reports whether the element with
  // index i should sort before the element with index j.
  Less(i, j int) bool
  // Swap swaps the elements with indexes i and j.
  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)
}
```

Concurrency: goroutines & channels

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

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

- **channels coordinate computations by explicit communication**
  - no locks, semaphores, mutexes, etc

Example: web crawler (with thanks to Russ Cox's video)

- **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**

Declarations

```
package main
import "fmt"
import "io"
import "io/ioutil"
import "net/http"
import "time"
type Site struct {
  Name string
  URL string
}
var sites = []Site {
  {"Go", "http://golang.org"},
  {"Python", "http://python.org"},
  {"Scala", "http://scala-lang.org"},
  {"Ruby", "http://ruby-lang.org"},
  {"Perl", "http://perl.org"},
}
```

Version 1: no parallelism

```
func main() {
  start := time.Now()
  for _, site := range sites {
    count(site.Name, site.URL)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

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

Version 2: parallelism with goroutines

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

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

Version 2: main() for parallelism with goroutines

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

Review: Formatter in AWK

```
/./  { for (i = 1; i <= NF; i++)
          addword($i)
     }
/^$/ { println(); print "" }
END  { println() }

function addword(w) {
    if (length(line) + length(w) > 60)
        println()
    line = line space w
    space = " "
}

function println() {
    if (length(line) > 0)
        print line
   line = space = ""
}
```

Formatter in Go

```
var line, space = "",  ""

func main() {
  scanner := bufio.NewScanner(os.Stdin)
  for scanner.Scan() {
    if line := scanner.Text(); len(line) == 0 {
      printline()
      fmt.Println()
    } else {
      for _, wds := range strings.Fields(line) {
        addword(wds)
      }
    }
  }
  printline()
}
func addword(word string) {
  if len(line) + len(word) > 60 {
    printline()
  }
  line = line + space + word
  space = " "
}
func printline() {
  if len(line) > 0 {
     fmt.Println(line)
  }
  line = ""; space = ""
}
```