The Go Programming Language
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
Influences
Outline

- history
- basic constructs, simple programs
- arrays & slices
- maps
- methods, interfaces
- concurrency, goroutines
Hello world in Go

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

$ go run hello.go      # to compile and run
$ go build hello.go   # to create a binary

$ go help              # for more
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 and declarations
  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):

// 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] = "Jun"
    ```

- 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

```
<table>
<thead>
<tr>
<th>data</th>
<th>len</th>
<th>cap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```
months
```

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>January</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>February</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>March</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>April</td>
<td></td>
<td></td>
<td>April</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>May</td>
<td></td>
<td></td>
<td>May</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>June</td>
<td></td>
<td></td>
<td>June</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>July</td>
<td></td>
<td></td>
<td>July</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>August</td>
<td></td>
<td></td>
<td>August</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>September</td>
<td></td>
<td></td>
<td>September</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>October</td>
<td></td>
<td></td>
<td>October</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>November</td>
<td></td>
<td></td>
<td>November</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>December</td>
<td></td>
<td></td>
<td>December</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
summer
```

```
<table>
<thead>
<tr>
<th>data</th>
<th>len</th>
<th>cap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Maps (== associative arrays)

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

```go
// 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 {
        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:

```go
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 {
}
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**
  
  ```go
  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

```go
define the sort interface
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() {
    fmt.Println(people)
    sort.Sort(ByAge(people))
    fmt.Println(people)
}
Tiny web server

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

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

- http.ResponseWriter implements Writer interface
```
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)
    }
    if err != nil {
        fmt.Fprintf(os.Stderr, "curl: copying %s: %v\n", url, err)
        os.Exit(1)
    }
}
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
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

```go
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)
}
```
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

```go
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)
}
```
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()
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)

```python
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()```
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)
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