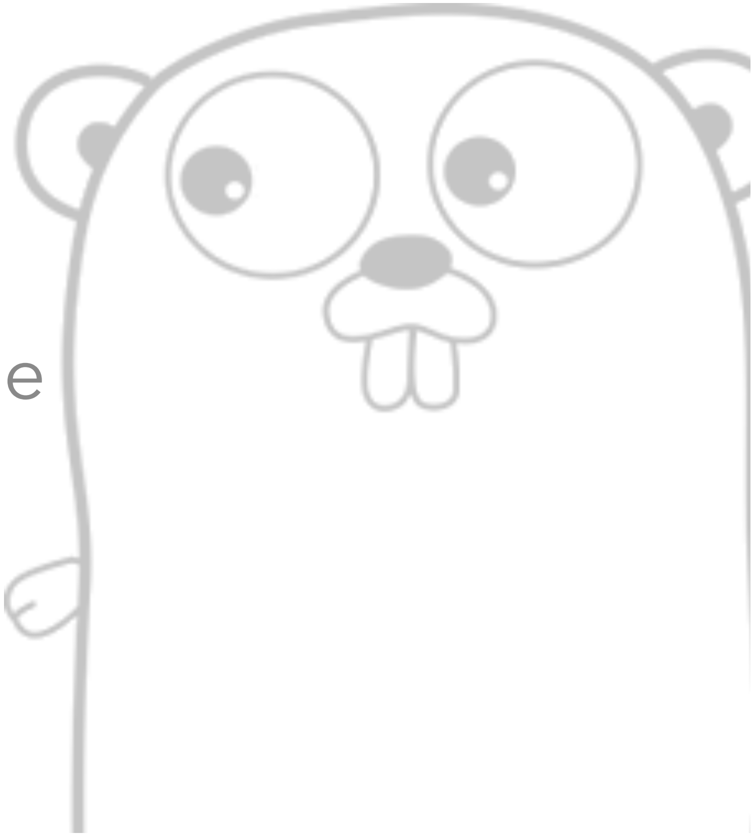
Introduction to Go

Ric Glassey

glassey@kth.se



Outline

- Go
  - **Aim: "Survey and discuss the core language syntax with reference to Java and other languages"**
  - Motivation and features of Go
  - Hello world and Go tools
  - Tour of Language syntax
  - Packages in Go
    - fmt & strings packages
  - Simple concurrency
    - using goroutines
    - using channels

MOTIVATION AND FEATURES

Why Go?

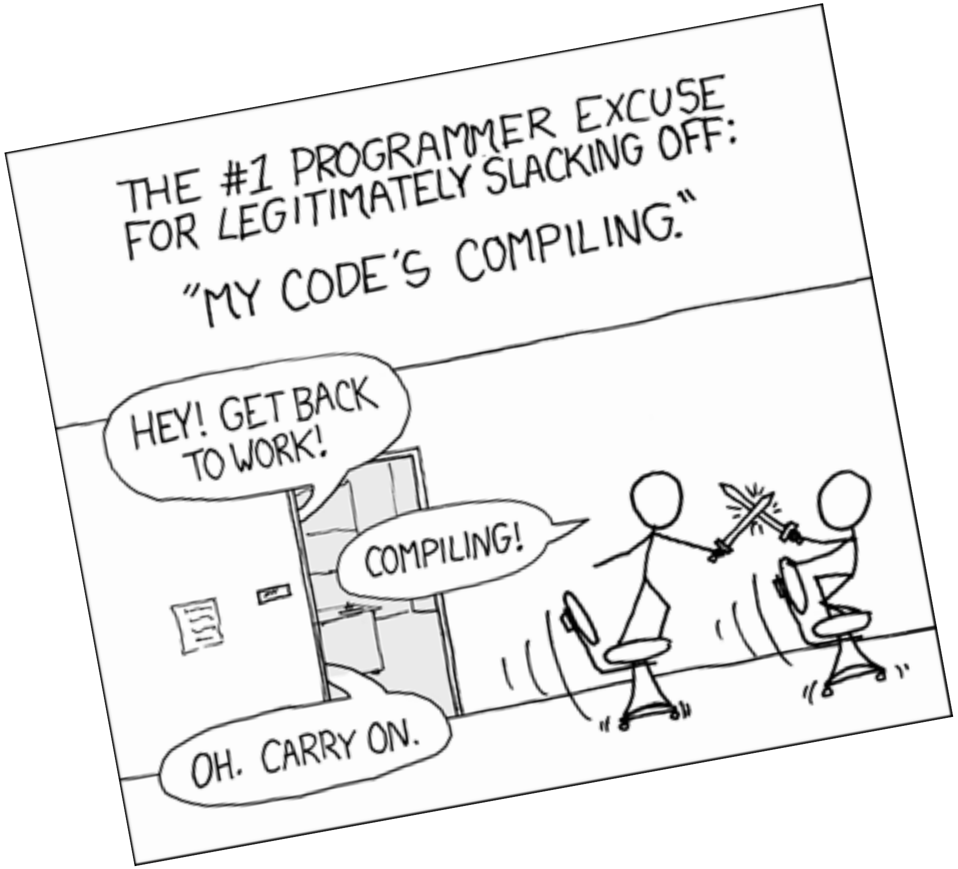
- "No major systems language has appeared in over a decade", despite the trends:
  - Computers faster; software engineering no faster
  - Dependency management is important
  - Rebellion against cumbersome type systems
  - Popular systems languages lack garbage collection and parallel computation
  - Emergence of multicore has completely changed computer architecture

Go goals

- It must work at **scale**
  - inspired by large scale server work at Google
- It must be **familiar**
  - roughly C-like
- It must be **modern**
  - built-in garbage collection
  - built-in concurrency
- It must be **clear**
  - dependencies
  - semantics
  - syntax

Features

- Statically-typed language
- Type inference
- Fast compilation



https://xkcd.com/303/

Features

- Remote package management
  - Using the delightful invocation **go get**
- Garbage collection
  - automatic memory management
  - unrequired object's memory is reclaimed
- Composition over inheritance
  - Less time worrying about type hierarchies

Features

- Built-in concurrency
  - Intuitive spawning of multiple concurrent computations
- Concurrency management
  - tests for race conditions†

HELLO WORLD & GO TOOLS

```go
package main

import "fmt"

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

Observations: Package

- **`package main`**

- No classes or objects, or sub-packages
- Packages are the basic building block
- Programs run from “main” package

Observations: Importing Packages

- **`import “fmt”`**

- Access to other packages
- Contains types, functions, etc
- No circular dependencies permitted
- Packages are identified by a string
  - Standard libraries live at project root
  - No *java.blah.more.what.something.whereami.*;*

Observations: Functions

- **`func main( ) { ...`**

- Function declaration
- Same format for any type of function
  - Regular functions
  - Anonymous functions
  - Methods of types
  - Closures
- main.main( ) is our starting point
  - compare to public static void main (String args) { ...

Observations: main( )

- We do not mention return type
- We do not mention return value
- We do not have String args for command line paramaters
  - use the os package (http://golang.org/pkg/os/)

```
package main

import (      // parentheses for multiple imports
   "fmt"
   "os"
)

func main() {
   fmt.Println(os.Args)
}
```

Observation: Braces not Spaces

- Use of braces / brackets should be the same as most of C-style languages
- There are very few semi-colons visible
- Automatic insertion during compile time
- Must be a new line after an opening brace {

- Go on…try to break the rules :-)

Observation: Println

- Note capital letter for function **imported** from fmt package
- Capital letters mean "**exported**" from package
  - think "public access"
- Lowercase indicate "**unexported**"
  - think "private access"
  - cannot be directly accessed

Observation: String

- String is a built-in type
- Strings are UTF-8 encoded
- Strings are immutable

```
package main

import "fmt"

func main() {
        fmt.Println("Hello world")
        fmt.Println("Hallå världen")
        fmt.Println("こんにちは")
        fmt.Println("안녕하세요")
        fmt.Println("góðan dag")
        fmt.Println("Grüßgott")
        fmt.Println("hyvää päivää")
        fmt.Println("yá'át'ééh")
        fmt.Println("Γεια σας")
        fmt.Println("Вітаю")
        fmt.Println("გამარჯობა")
        fmt.Println("नमस्ते")
        fmt.Println("你好")
}
```

Go Tools

- For this short course, all you require:

```
$ go run
$ go fmt
```

- Other tools to check out:

```
$ go build
$ go test
```

- In case you lack internet connection…

```
$ go help [command]
$ godoc [package] [topic]
```

Where is BlueJ for Go!?

- Code demos used here:
  - Sublime Text 3 (still unlicensed, trial on-going)
  - GoSublime package
    - Saving invokes **go fmt**
    - Building invokes **go fmt**, saves file, command dialog

- Any programming environment will do:
  - Any text editor + command line tools
  - Plugin for your favourite IDE:
    - Eclipse
    - IntelliJ

TOUR OF LANGUAGE SYNTAX

Language Design

- A member of the C family for basic syntax
  - Great, many things from Java etc apply
  - Not so great, there may be unseen traps

- Basic areas of syntax
  - Types & Variables
  - Control Structures
  - Arrays, slices and maps
  - Functions
  - Pointers
  - Structs and interfaces

Types and variables

```
func main() {

    // `var` declares 1 or more variables.
    var a string = "initial"

    // You can declare multiple variables at once.
    var b, c int = 1, 2

    // Go will infer the type of initialized variables.
    var d = true

    // Variables declared without initialization and zero-valued
    var e int

    // The `:=` syntax is shorthand for declaring and initializing
    f := "short"
}
```

Control Structures: if/else

```
func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    // A statement can precede conditionals
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
 }
```

Control Structures: loops

```go
func main() {

    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    for {
        fmt.Println("loop")
        break
    }
}
```

Arrays

```
func main() {

    // By default an array is zero-valued, fixed length
    var a [5]int

    a[4] = 100
    fmt.Println(a[4])
    fmt.Println(len(a))

    b := [5]int{1, 2, 3, 4, 5}

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
       for j := 0; j < 3; j++ {
          twoD[i][j] = i + j
       }
    }
}
```

Slices

```
    func main() {

        // Unlike arrays, slices can grow
        s := make([]string, 3)

        s[0] = "a"
        s[1] = "b"
        s[2] = "c”
        fmt.Println(len(s))

        // Append returns a new slice value, hence assignment
        s = append(s, "d")
        s = append(s, "e", "f")

        // slice operations [low:high]
        l := s[2:5]
        l = s[:5]
        l = s[2:]

        t := []string{"g", "h", "i"}
    }
```

Maps

```go
func main() {
    // To create an empty map, use the builtin `make`
    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13
    fmt.Println("map:", m)

    v1 := m["k1"]

    // The builtin `len` returns the number of key/value pairs
    fmt.Println("len:", len(m))

    delete(m, "k2")

    // The optional second return value indicates key presence
    _, prs := m["k2"]

    n := map[string]int{"foo": 1, "bar": 2}
}
```

Functions

```go
func plus(a int, b int) int {

    // Go requires explicit returns, i.e. it won't
    // automatically return the value of the last
    // expression.
    return a + b
}

// Omit the type name for the like-typed parameters
func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {

    // Call a function just as you'd expect
    res := plus(1, 2)
    res = plusPlus(1, 2, 3)
}
```

Functions: multiple return values

```
// The `(int, int)` in this function signature shows that
// the function returns 2 `int`s.
func vals() (int, int) {
    return 3, 7
}

func main() {

    // Here we use the 2 different return values from the
    // call with multiple assignment.
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // If you only want a subset of the returned values,
    // use the blank identifier `_`.
    _, c := vals()
    fmt.Println(c)
}
```

Variadic Functions

```
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    sum(1, 2)
    sum(1, 2, 3)
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Pointers

- Pointers reference a location in memory where a value is stored

- Consider this example - what happens to x?

```
package main

import "fmt"

func zero(x int) {
   x = 0
}

func main() {
   x := 5
   zero(x)
   fmt.Println(x)
}
```

Pointers

- In Go ***type** is used to declare a pointer
  - e.g. *int
- **&name** is used to find the address of a variable
- What happens this time?

```
package main

import "fmt"

func zero(xPtr *int) {
    *xPtr = 0
}

func main() {
    x := 5
    zero(&x)
    fmt.Println(x)
}
```

Structs

- Useful to group fields together as records

```
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
   // next slide
}
```

Structs

```
// This syntax creates a new struct.
fmt.Println(person{"Bob", 20})

// You can name the fields when initializing a struct.
fmt.Println(person{name: "Alice", age: 30})

// Omitted fields will be zero-valued.
fmt.Println(person{name: "Fred"})

// An `&` prefix yields a pointer to the struct.
fmt.Println(&person{name: "Ann", age: 40})

// Access struct fields with a dot.
s := person{name: "Sean", age: 50}
fmt.Println(s.name)

// Structs are mutable.
sp.age = 51
fmt.Println(sp.age)
```

PACKAGES

Stdlib Packages

- Very little software engineering happens in complete isolation
  - If you have a problem; someone already solved it
  - Solutions are glued together from components
  - Always **refer to the standard library** first
  - Caveat: Go is quite new and the standard library and third party library ecosystem are still maturing

- http://golang.org/pkg/

Package: fmt

- Handles with scanning input and printing output

```go
package main

import "fmt"

func main() {
    var i int
    // prompt for input
    fmt.Scan(&i)
    fmt.Println("your number", i)
}
```

Package: strings

```go
package main

import s "strings"
import "fmt"

// we can alias `fmt.Println` with a shorter name
var p = fmt.Println

func main() {

    // we need pass the string as the first argument
    p("Contains:  ", s.Contains("test", "es"))
    p("Count:     ", s.Count("test", "t"))
    p("HasPrefix: ", s.HasPrefix("test", "te"))
    p("HasSuffix: ", s.HasSuffix("test", "st"))
    p("Index:     ", s.Index("test", "e"))
    p("Join:      ", s.Join([]string{"a", "b"}, "-"))
```

Package: strings

```
    // continued
    p("Replace:   ", s.Replace("foo", "o", "0", -1))
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", s.ToLower("TEST"))
    p("ToUpper:   ", s.ToUpper("test"))
    p()

    // Not part of `strings` but worth mentioning
    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

SIMPLE CONCURRENCY

goroutine

- A function that is capable of **running concurrently** with other functions

- Simply use the ‘go’ keyword followed by a function

```go
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    // create goroutine with f()
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Channels

- Communication ‘channel’ between goroutines

```go
package main

import "fmt"

func main() {

    // Create a new channel with `make(chan val-type)`
    messages := make(chan string)

    // Send a value into a channel using the `channel <-` syntax.
    go func() { messages <- "ping" }()

    // The `<-channel` syntax receives a value
    msg := <-messages
    fmt.Println(msg)
 }
```

Readings

- Fundamentals of Concurrent Programming
  - by S. Nilsson
  - Required Reading
  - **Sections 1 + 2**
  - http://www.nada.kth.se/~snilsson/concurrency/#Thread
- Go for Java Programmers
  - by S. Nilsson
  - **Work through text in line with course**
- Go website
  - http://golang.org/
  - Lots of useful reference and rationale for Go

Survey!

- As ever, your feedback is appreciated :)
  - New survey service!
  - Thoughts on Go
  - Go or Java for concurrency?
  - https://www.surveymonkey.com/s/KMQD2KX