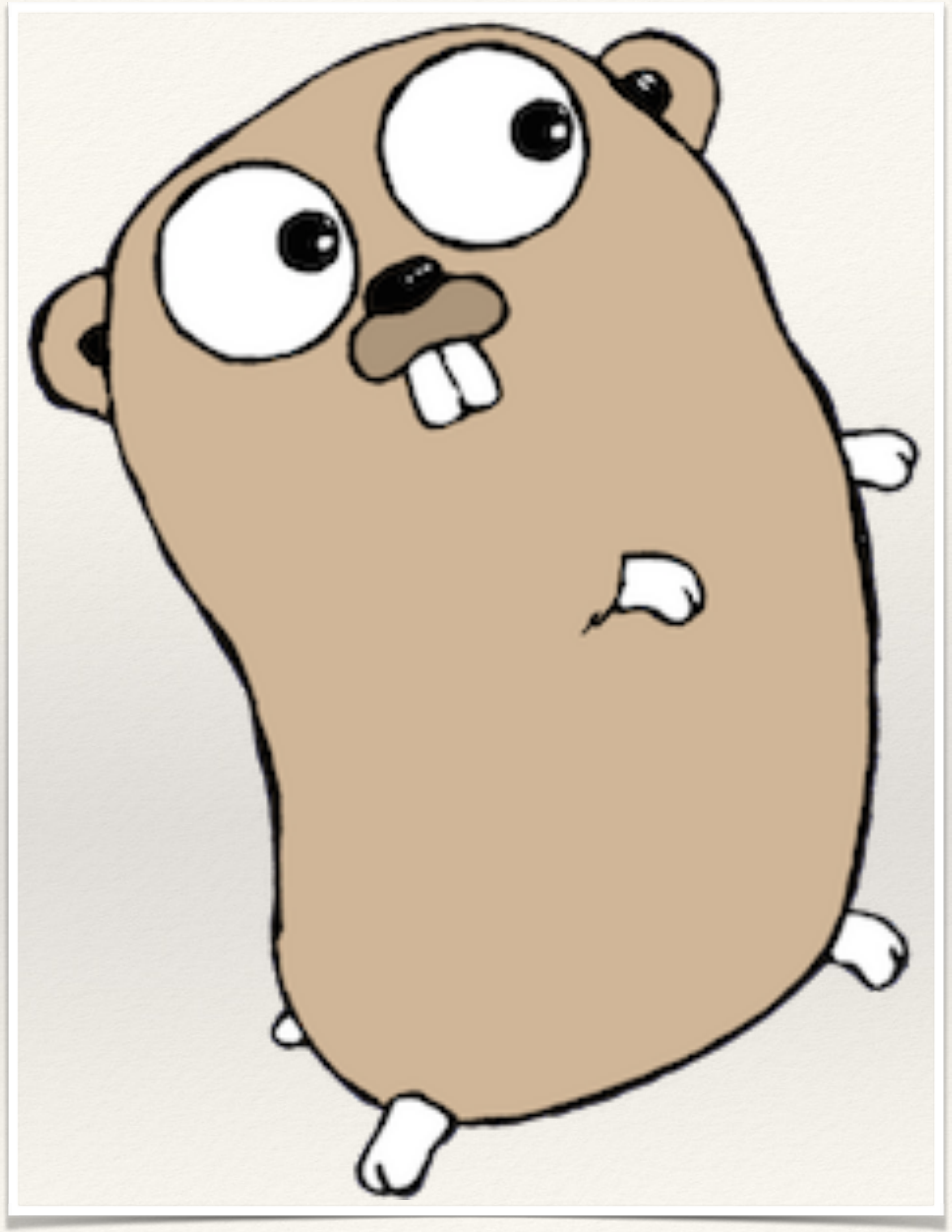
Ruby Loco Hack night

Introduction to Go

April 13, 2015
Andrew Bates



Go…what?

Ruby	Go
Dynamic typing	Static typing
Object oriented	Sort of object oriented
Tricky concurrency	Awesome concurrency
Beautiful syntax	Meh…
Interpretted	Compiled
Both embrace testing!!	

Go…why?

- Great community
- Compile to native code and distribute a single binary
- Easily cross compile from dev environment to any supported platform
    - Windows, MacOS and Linux
    - x86_64, i386 and ARM
- Fantastic concurrency
- Great toolset: go fmt, go get, gofix, go tool cover, etc
- I'm sure there are more reasons 🍺

Go…why not?

- Well, it's not Ruby…
  - Syntax isn't as clear
  - Static typing makes marshalling and unmarshalling data a little more difficult (think JSON)
  - No monkey patching or meta-programming
- Currently dependency management is tricky
  - projects usually import directly from master branch of the dependency
  - breaking changes in upstream code are unavoidable with this model (gofix helps with this)
  - I'm sure there are more reasons 🍻

Go…Core Concepts

- For a real training session, check out the **go tour**
- Go syntax is similar to C without the semicolons
- Go has functions, structs, arrays, maps and slices
    - Wait, slices? Effectively a pointer to a section of an array
    - functions are first-class citizens
- Go also has something called channels and go routines which simplify concurrency

Go…Core Concepts

- Full spec available:
  - https://golang.org/ref/spec
- All files start with a package
  - packages generally reflect the directory the code is in
  - package names are always lower case!
- Package statements can be followed by optional package imports

```
package main

import "fmt"
```

Go...Core Concepts

- Declarations follow any imports that are specified.
- Declarations can include new type definitions, function definitions and variable declarations
- Go determines which names are exported by examining the first letter of the declaration
  - Upper case names are exported, lower case names are not
- Non-exported definitions are not accessible outside the package (you could say they are private)

```go
/**
 * Create a new type called Meetup that
 * contains 3 exported fields and one
 * non-exported field
 */
type Meetup struct {
  Name     string
  Date     string
  Location string
  open     bool
}
```

Go…Core Concepts

- Functions work pretty much how you'd expect
- Note that arrays (and slices) are fixed in size. They cannot be expanded.
  - Append returns a new slice with the item appended to the given slice
  - This would seem like a slow solution, but it doesn't seem to cause a problem until you get to millions or billions of items

```
func main() {
   /* Create an empty slice of meetups */
   meetups := make([]Meetup, 0)

   /* Append a meetup and get a new slice */
   meetups = append(meetups, Meetup{
      "Ruby Loco Hack Night",
      "2015-04-13",
      "Phish Me",
      false,
   })

   /* Append a meetup and get a new slice */
   meetups = append(meetups, Meetup{
      "Ruby Loco Lunch",
      "2015-04-24",
      "Alamo Draft House",
      true,
   })

   fmt.Printf("%v\n", meetups)
}
```

Go… Methods and Interfaces

- Types can be given methods, just assign a variable name and the type between the func keyword and the argument list
- Interfaces are a collection of method definitions
- Any type that implements a method matching an Interface meets the requirements of implementing the Interface

```
/**
 * The Stringer interface specifies a
 * single String()function that takes no
 * arguments and returns a single string
 * value
 */
func (m Meetup) String() string {
  return "" +
    "    Name: " + m.Name + "\n" +
    "    Date: " + m.Date + "\n" +
    "Location: " + m.Date + "\n"
}

/* The %v format specifier will call
 * String() on the object if it implements
 * the Stringer interface
 */
fmt.Printf("%v\n", meetups)
```

Go… Channels and Routines

- Channels are like pipes. Put something in one end, take it out of the other
- Channels can block program execution
- Go routines are lightweight threads
  - Enable concurrency, but not necessarily parallelism
  - Parallelism can be enabled

```go
func main() {
   meetups := make([]Meetup, 0)

   c := make(chan Meetup)

   go func() {
      for {
         meetups = append(meetups, <-c)
      }
   }()

   c <- Meetup{
      "Ruby Loco Hack Night",
      "2015-04-13",
      "Phish Me",
      false,
   }

   c <- Meetup{
      "Ruby Loco Lunch",
      "2015-04-24",
      "Alamo Draft House",
      true,
   }
}
```

Go… Errors

- Go doesn't throw errors
- The Go convention is to return an error from the function
- If the error is nil, then there is no problem
- Otherwise, try to recover from the error

```
var meetups []Meetup
var c chan Meetup

func AddMeetup(name, location, date string, open bool) error {
   if len(meetups) < 10 {
      c <- Meetup{name, location, date, open}
      return nil
   }
   return fmt.Errorf("Meetups are full!")
}

…
/* Add two items to the channel */
err := AddMeetup("Ruby Loco Hack Night", "2015-04-13", "Phish
Me", false)
if err != nil {
   fmt.Printf("Error adding Meetup: %v", err)
   return
}

err = AddMeetup("Ruby Loco Lunch", "2015-04-24", "Alamo Draft
House", true)
if err != nil {
   fmt.Printf("Error adding Meetup: %v", err)
   return
}
```

That’s all folks!