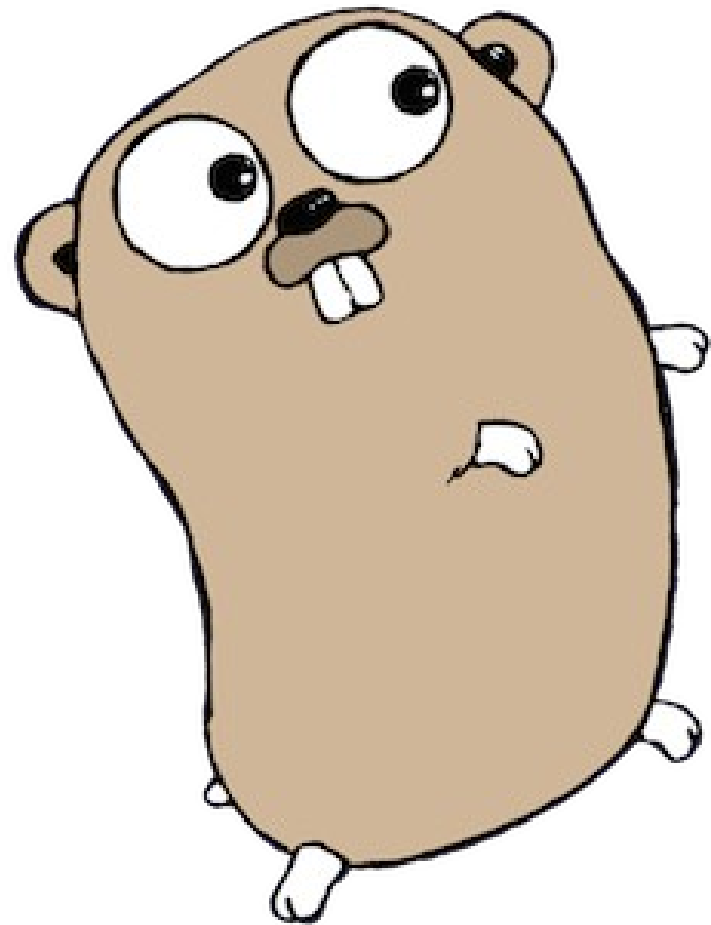
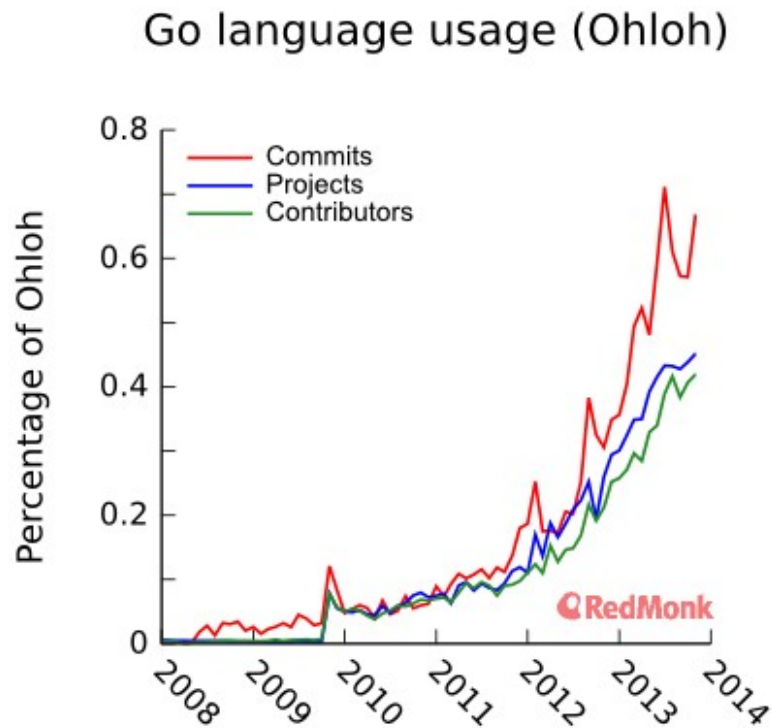


Introduction to Go for Java Developers



László Csontos

Liferay Hungary

Agenda

- **Go's short history and features**
- Packages
- Basic stuff: control structures and built-in types
- Composite types
- Go's approach to OO design
- Concurrency made easy
- Standard library
- Web apps with Go

History

- *“Three of us [Ken Thompson, Rob Pike, and Robert Griesemer] got together and decided that we hated C++.”*
- *“All three of us had to be talked into every feature in the language, so there was no extraneous garbage put into it for any reason.”*
- Development started in 2007
- Open source since 2009
- Stable 1.0 released in 2012

Features

- Statically typed with automatic type infer. (`x := 0 // int x = 0`)
- Garbage collected
- Fast compilation times
- Remote package management (~Maven)
- Built-in concurrency primitives: light-weight processes (goroutines), channels
- An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance.
- A toolchain that, by default, produces statically linked native binaries without external dependencies.

Features

- no type inheritance
- no method or operator overloading
- no circular dependencies among packages
- no pointer arithmetic
- no assertions
- no generic programming
- no implicit type conversions

Agenda

- Go's short history and features
- **Packages**
- Basic stuff: control structures and built-in types
- Composite types
- Go's approach to OO design
- Concurrency made easy
- Standard library
- Web apps with Go

Packages

- Go programs are organized into packages.
- They correspond to packages with classes in Java
- Package `main` with a `main` function is the entry point of the program

```
package main

import "fmt"

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

<https://play.golang.org/p/duRF5gXJEP>

Agenda

- Go's short history and features
- Packages
- **Basic stuff: control structures and built-in types**
- Composite types
- Go's approach to OO design
- Concurrency made easy
- Standard library
- Web apps with Go

Basic stuff

- Control structures look very similar for those who came from C/C++ or Java world.
- Parenthesis after `if`, `for`, `switch`, etc. isn't mandatory
- Semicolons are also optional at the end of the line (except if you want to place multiple statements in a single line)
- Unused imports cause compiler error
- Array operator comes before the type: `[]int`
- Types declaration comes after the variable name:
`func add(a, b int) int { return a + b; }`

Control structures

FOR is the only way of looping.

```
s := []string{"a", "b", "c"}
for i := 0; i < len(s); i++ {
    v := s[i]; fmt.Printf("index: %d, value: %v\n", i, v)
}
for i, v := range s {
    fmt.Printf("index: %d, value: %v\n", i, v)
}
```

```
m := map[int]string{1: "a", 2: "b", 3: "c"}
for k, v := range m {
    fmt.Printf("key: %d, value: %v\n", k, v)
}
```

<https://play.golang.org/p/HAdAf-D4a1>

Control structures

IF can have temporary variables

```
s := []int{3, 42, 73, 1}

max := -1

for i := 0; i < len(s); i++ {
    if v := s[i]; v > max {
        max = v
    }
}

fmt.Println(max)
```

<https://play.golang.org/p/hvhWqaw8UG>

Control structures

SWITCH can have cases not just for values, but for types

```
// golang/src/pkg/fmt/print.go
switch f := arg.(type) {
case bool:
    p.fmtBool(f, verb)
case float32:
    p.fmtFloat32(f, verb)
    ...
default:
}
```

Control structures

There is no TRY/CATCH/FINALL

```
// golang/src/pkg/image/png/reader.go
func (d *decoder) decode() (image.Image, error) {
    // TRY
    r, err := zlib.NewReader(d)
    // CATCH
    if err != nil {
        return nil, err
    }
    // FINALLY
    defer r.Close()
    ...
}
```

Types system

- Simple types: `int`, `float`, `bool`, etc.
- Composite types: structures
- Reference types: slice, map, channel, interface and function
- Named types: can refer to any of the above, we'll see later why this matters

```
// golang/src/pkg/time/time.go
```

```
type Duration int64
```

Simple types

- Boolean: `bool`
- Signed integers: `int8`, ..., `int64`
- Unsigned integers: `uint8`, ..., `uint64`
- Floats: `float32`, `float64`
- `byte` -> `uint8`
- `rune` -> `int32` (~char int Java)
- `int` -> `int32/int64`, `uint` -> `uint32/uint64`
- `complex64`, `complex128`
- `string`

Agenda

- Go's short history and features
- Packages
- Basic stuff: control structures and built-in types
- **Composite types**
- Go's approach to OO design
- Concurrency made easy
- Standard library
- Web apps with Go

Composite types

```
type Vertex struct {  
    name string  
  
    incomingEdges []*Edge  
    outgoingEdges []*Edge  
}  
  
type Edge struct {  
    head *Vertex  
    tail *Vertex  
  
    weight int  
}  
  
type Graph struct {  
    verticesMap map[string]*Vertex  
}
```

Composite types

```
type Graph struct {  
    verticesMap map[string]*Vertex  
}  
  
// "g" is the receiver of method AddEdge()  
func (g *Graph) AddEdge(  
    tail, head string, weight int) {  
    ...  
}
```

<https://play.golang.org/p/Ugoowc6fnd>

Reference types

- All reference types have a lightweight “header” which in turn contains pointers to the underlying data structures.
- This makes pass them by value very cheap
- Let's have a closer look at them:
 - Slice
 - Map
 - Channel
 - Interface
 - Function

Reference types / Slice

A slice is just like a `java.util.ArrayList`, it shirks and grows as necessary

```
func printSlice(s []int) {  
    fmt.Printf("len=%d, cap=%d, s=%v\n", len(s), cap(s), s)  
}
```

```
func main() {  
    // empty slice  
    var s []int;    printSlice(s)  
    s = []int {1,2,3,4}; printSlice(s)  
    s = append(s, 5); printSlice(s)  
}
```

<https://play.golang.org/p/hJl7iI-87H>

Reference types / Map

- A map is just like a `java.util.HashMap`

```
func (g *Graph) AddEdge(  
    tailName, headName string, weight int) {  
  
    tail, ok := g.verticesMap[tailName]  
    if !ok {  
        tail = &Vertex{name: tailName}  
        g.verticesMap[tailName] = tail  
    }  
    ...  
}
```

<https://play.golang.org/p/Ugoowc6fnd>

Reference types / Channel

- Channels are like `java.util.concurrent.LinkedBlockingQueue`
- Idiomatic means of communication among co-operating threads (go routines)

```
func main() {  
    // LinkedBlockingQueue<Integer> lbq = new LinkedBlockingQueue<>(1)  
    c := make(chan int, 1)  
    // lbq.offer(1)  
    c <- 1  
    // lbq.poll()  
    fmt.Println(<-c)  
}
```

<https://play.golang.org/p/SaZgJzU0qu>

Reference types / Interface

- Interfaces define behaviour, just like in Java
- However there is no need to formally declare for any given type which interface it implement

```
// golang/src/pkg/io/io.go
// Implementations of Read are discouraged from returning a
// zero byte count with a nil error, and callers should treat
// that situation as a no-op.
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

https://play.golang.org/p/_-zJOHMJ5y

Agenda

- Go's short history and features
- Packages
- Basic stuff: control structures and built-in types
- Composite types
- **Go's approach to OO design**
- Concurrency made easy
- Standard library
- Web apps with Go

Go's approach to OO design

- Go doesn't directly support inheritance, but code reuse can be implemented through composition
- Encapsulation is supposed through methods, data hiding is done with unexported package members
- Polymorphism is provided by interfaces, although formal declaration between concrete types and interfaces isn't necessary

Inheritance

```
class User {  
    private String firstName;  
    private String lastName;  
    private String userName;  
}
```

```
class Admin  
    extends User {  
  
    private String level;  
}
```

```
type User struct {  
    firstName string  
    lastName string  
    userName string  
}
```

```
type Admin struct {  
    User  
    level string  
}
```

Encapsulation

```
class User {  
    private String firstName;  
    public String lastName;  
    ...  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void SetFirstName(...) {  
        ...  
    }  
}
```

```
type User struct {  
    firstName string  
    LastName string  
}  
  
func (u User)  
    GetFirstName() string {  
        return u.firstName  
    }
```

Polymorphism

```
type (  
    Shape interface {  
        Area() float64  
    }  
  
    Triangle struct {  
        a, b, c float64  
    }  
)  
  
func (t Triangle) Area() float64 {  
    s := (t.a + t.b + t.c) / 2  
    return math.Sqrt(s * (s - t.a) * (s - t.b) * (s - t.c))  
}  
  
func printShape(s Shape) {  
    a := s.Area()  
  
    fmt.Printf("%T's area is %f\n", s, a)  
}
```

<https://play.golang.org/p/8KLa9b8Q-E>

<https://play.golang.org/p/t-yMdpVgC1>

Agenda

- Go's short history and features
- Packages
- Basic stuff: control structures and built-in types
- Composite types
- Go's approach to OO design
- **Concurrency made easy**
- Standard library
- Web apps with Go

Concurrency made easy

- Concurrency is provided OOTB, in a similar way as we would have an internal `java.util.concurrent.ThreadPoolExecutor`
- A goroutine is like concrete implementation of `java.lang.Runnable` submitted for async execution
- Internal scheduler sets blocked goroutines aside, so that runnable ones are able to execute
- Channels are the idiomatic way of sharing data among goroutines, although package `sync` provides similar functionality like `java.util.concurrent`.

Concurrency made easy

```
func sum(a []int, c chan int) {  
    ...  
    c <- sum // send sum to c  
}
```

```
func main() {  
    ...  
    c := make(chan int)  
  
    for i := 0; i < N/K; i++ {  
        ...  
        go sum(a[start:end], c)  
    }  
  
    total := 0  
    for i := 0; i < N/K; i++ {  
        total += <- c  
    }  
  
    fmt.Println(total)  
}
```

Agenda

- Go's short history and features
- Packages
- Basic stuff: control structures and built-in types
- Composite types
- Go's approach to OO design
- Concurrency made easy
- **Standard library**
- Web apps with Go

Standard Library

- Go's standard library provides support for implementing numerous functionalities
- In Go 1.3, there are 176 built-in packages

Standard Library

```
import ("errors"; "fmt"; "http"; "io"; "encoder/json")

func (c *Client) newJiraRequest(method, path string, reader io.Reader) (*http.Request, error) {
    req, err := http.NewRequest(
        method, fmt.Sprintf("%s/rest/%s", c.uri, path), reader)

    if err != nil { return nil, err; }

    req.Header.Set("Content-Type", "application/json")

    return req, nil
}

func (c *Client) performJiraRequest(method, path string, reader io.Reader, output interface{}) error {
    req, err := c.newJiraRequest(method, path, reader)
    if err != nil { return err; }

    resp, err := c.httpClient.Do(req); defer resp.Body.Close()
    if err != nil { return err }

    if resp.StatusCode != 200 { return errors.New(resp.Status); }

    err = json.NewDecoder(resp.Body).Decode(output)
    if err != nil { return err; }

    return nil
}
```

Agenda

- Go's short history and features
- Packages
- Basic stuff: control structures and built-in types
- Composite types
- Go's approach to OO design
- Concurrency made easy
- Standard library
- **Web apps with Go**

Building Web apps with Go

```
package main
```

```
import ("fmt"; "log"; "net/http"; "time")
```

```
type timeHandler struct{}
```

```
func (th timeHandler) ServeHTTP(w http.ResponseWriter, req *http.Request) {  
    currentTime := time.Now()
```

```
    fmt.Fprint(w, currentTime.Format(time.RFC1123Z))  
}
```

```
func main() {  
    err := http.ListenAndServe("localhost:4000", timeHandler{})  
  
    if err != nil {  
        log.Fatal(err)  
    }  
}
```

Building Web apps with Go

- There are many frameworks which support Web app development
- The most notable ones are:
 - Beego
 - Martini
 - Gorilla
 - GoCraft
 - Standard net/http

References

- Manning: Go in Action
- A Tour of Go
- Go Playground
- Go package reference
- Golang for Java programmers
- A Survey of 5 Go Web Frameworks

Questions?