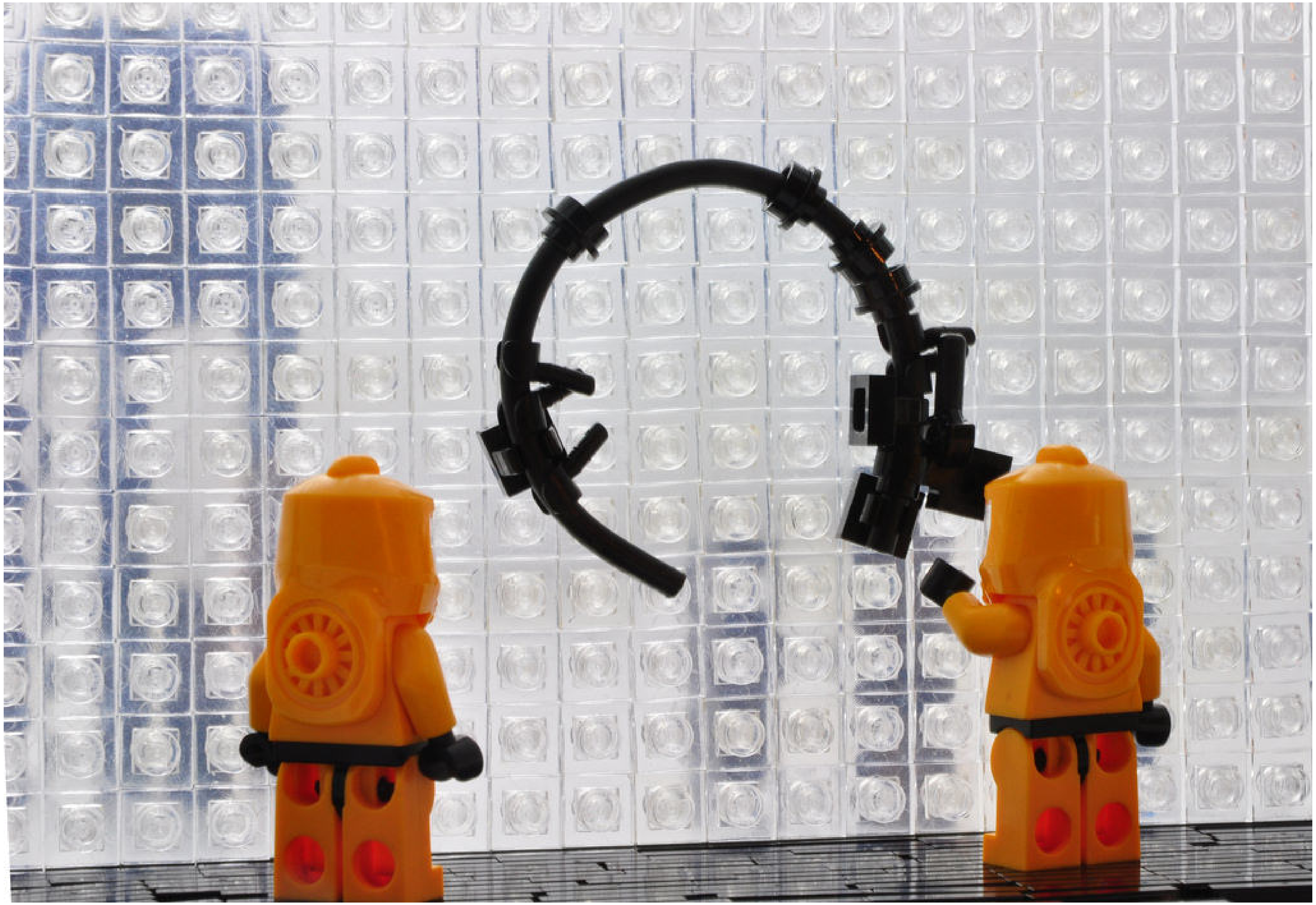
Rapid Introduction to Go

12 May 2017

Tim Schulte
Albert-Ludwigs-Universität Freiburg

How to learn a new language from scratch in limited time?



Agenda

- Gathering data
- Entering the ship
- Session 01: Establishing a basic understanding
- Session 02: Interfaces
- Session 03: Concurrency

Gathering data

What is Go?

Go is an open-source, general-purpose programming language.

- Compiled
- Statically typed
- Concurrent
- Garbage collected
- Simple and productive

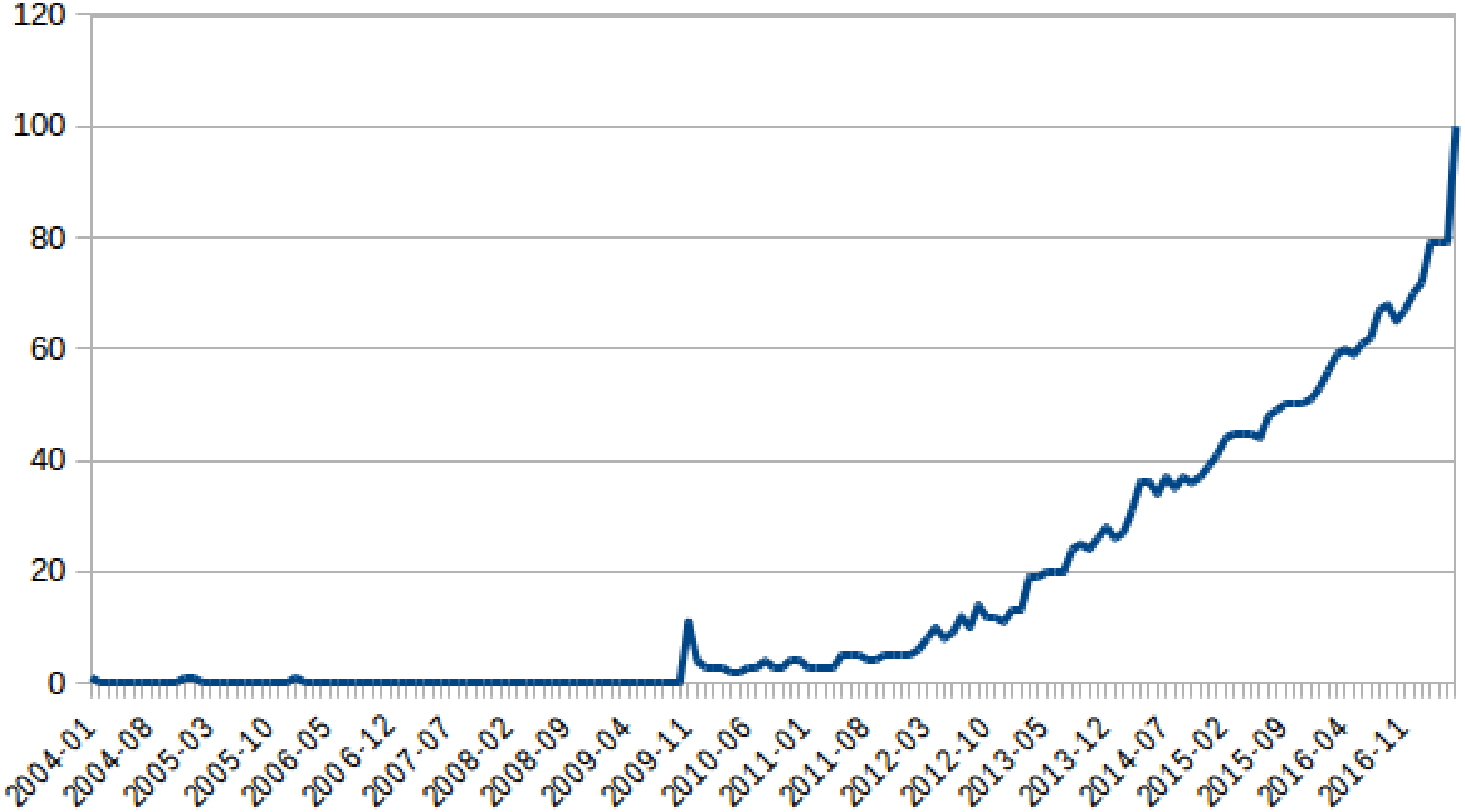
What is different?

No classes, no inheritance, no generics

No function overloading, no default values

No exception handling

Public reception



Google Trends for golang (https://trends.google.com/trends/explore?date=all&q=golang)

Entering the ship

Installing the Go tools

Get the archive for your system from golang.org/doc/install (https://golang.org/doc/install).

Extract it somewhere, e.g. `/usr/local`:

```
$ tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Export the path and set the `GOROOT` environment variable:
(You may want to add commands like the following to `$HOME/.profile`)

```
$ export GOROOT=/usr/local/go
$ export PATH=$PATH:$GOROOT/bin
```

Test your installation

```
$ go version
$ go env GOROOT
```

Creating a Workspace

A workspace is a directory hierarchy with three directories at its root:

```
gocode/
├── src/  # contains Go source files,
├── pkg/  # contains package objects, and
└── bin/  # contains executable commands.
```

The go `tool` builds source packages and installs the resulting binaries to the pkg and `bin` directories.

Set the `GOPATH` environment variable to specify the location of your workspace, e.g.:

```
$ export GOPATH=$HOME/gocode
```

Creating and Running Go Programs

Create a new package directory inside your workspace

```
$ mkdir -p $GOPATH/src/github.com/user/hello
```

Create a file named `hello.go` inside that directory, containing the following Go code

```
package main

import "fmt"

func main() {
    fmt.Printf("Why are you here?\n")
}
```

Run the program

```
$ go run hello.go
```

Session 01: Establishing a basic understanding

Packages

```go
package main

import (
	"fmt"
	"math/rand"
)

func main() {
	fmt.Println("My favorite number is", rand.Intn(10))
}
```

Run

Every Go program is made up of packages.

Imports can be grouped into a paranthesized import statement.

Programs start running in package `main`.

Exported names

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(math.pi)
}
```

Run

A name is exported if it begins with a capital letter.

When importing a package, you can refer only to its exported names.

Functions

```
package main

import "fmt"

func greet(name string) {
    fmt.Println("Hello,", name+"!")
}

func main() {
    greet("Gopher")
}
```

Run

Functions can take zero or more arguments.

Notice that the type comes *after* the variable name.

Functions

```
func add(x, y int) int {
    return x + y
}
```

```
func swap(i int, j bool) (bool, int) {
    return j, i
}
```

If multiple arguments are of the same type, it must only be specified once.

Functions can return any number of results.

Higher-Order Functions

```go
package main

import (
    "fmt"
    "math"
)

func compose(f, g func(float64) float64) func(float64) float64 {
    return func(x float64) float64 {
        return f(g(x))
    }
}

func main() {
    sincos := compose(math.Sin, math.Cos)
    fmt.Printf("%T: 0.5 -> %v\n", sincos, sincos(0.5))
}
```

Run

Functions are values too.

Variables

```go
package main

import "fmt"

var a int
var b, c string = "alice", "bob"

func main() {
    host, port := "localhost", 3035

    fmt.Println(a, b, c, host, port)
}
```

Run

The `var` statement declares a list of variables (at package or function level).

Variable declarations can include initializers (one per variable).

Inside a function, the short assignment statement `:=` can be used.

Zero values

```
package main

import "fmt"

func main() {
    var i int
    var f float64
    var b bool
    var s string
    fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
```

Run

Variables declared without an explicit initial value are given their zero value.

Types

```
// Basic types
bool

string

int, int8, int16, int32, int64
uint, uint8, uint16, uint32, uint64, uintptr

byte // alias for uint8
rune // alias for int32 (Unicode point)

float32, float64

complex64, complex128
```

```
// Pointers
*bool, *string, *int, *int8, ...
```

For-Loop

```
for i := 0; i < 100; i++ {
}
```

```
// When neither init- nor post-statements are present, semicolons can be omitted
for isEven(x) {
}
```

```
// Without the condition-expression `for` loops forever
for {
}
```

Go has only one looping construct, the `for` loop.

Init-statement, condition-expression, and post-statement are optional

If-Statement

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    e := 2.71828183
    if y := math.Pow(e, math.Pi); y < 24 {
        fmt.Printf("e^π = %v\n", y)
    } else {
        fmt.Printf("%v\n", y)
    }
    // y is out of scope
}
```

Run

Switch

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd, plan9, windows...
        fmt.Printf("%s.", os)
    }
}
```

Run

A case body breaks automatically, unless it ends with a `fallthrough` statement.

Arrays and Slices

```go
package main

import "fmt"

func main() {
    primes := [5]int{2, 3, 5, 7, 11} // array of size 5

    var s []int = primes[2:4] // slice
    fmt.Printf("primes: %v\ns: %v\n", primes, s)

    s[0] = 42
    fmt.Printf("primes: %v\ns: %v\n", primes, s)
}
```

Run

An `array` has a fixed size.

A `slice` is a dynamically-sized, flexible view into the elements of an `array`.

Creating a Slice with Make

```
package main

import "fmt"

func main() {
    a := make([]float32, 5)     // a = [0 0 0 0 0] len=5 cap=5
    b := make([]float32, 0, 5)  // b = []          len=0 cap=5
    c := b[1:4]                 // c = [0 0 0]     len=3 cap=4
    fmt.Printf("%v\n%v\n%v\n", a, b, c)
}
```

Run

make allocates a zeroed array and returns a slice that refers to that array.

Appending to a Slice

```
package main

import "fmt"

func main() {
    var v []int
    v = append(v, 1, 2, 3)
    fmt.Println(v)

    w := []int{4, 5, 6}
    w = append(w, v...)
    fmt.Println(w)
}
```

Run

append takes a slice s of type T, and T values to append to the slice.

Maps

```go
package main

import "fmt"

func main() {
    // make returns a map of the given type, initialized and ready to use
    m := make(map[int]int)
    m[0] = 1
    m[1] = 2

    v, ok := m[1]
    if ok {
        fmt.Println(v)
    }

    delete(m, 1)
    fmt.Println(m)
}
```

Run

A map maps keys to values.

Range

```
package main

import "fmt"

func main() {
    orange := map[string]byte{"red": 0xff, "green": 0x99, "blue": 0x00}

    for k, v := range orange {
        fmt.Printf("%v: %v\n", k, v)
    }

    pow := []int{1, 2, 4, 8, 16, 32, 64, 128}

    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

Run

The `range` form of the `for` loop iterates over a slice or map.

Exercise: Fibonacci closure

```go
package main

import "fmt"

// fibonacci is a function that returns a function that returns an int.
func fibonacci() func() int {
    // TODO: IMPLEMENT
}

func main() {
    f := fibonacci()
    for i := 0; i < 10; i++ {
        fmt.Println(f())
    }
}
```

Run

Exercise: Exponential function

```
// Exp computes the exponential function e^x
Exp(x, epsilon float64) float64 {
    // TODO: IMPLEMENT
}
```

```
// You may use the following sum to approximate the exponential function:
// Stop when two consecutive sums differ by less than epsilon.
       ∞
e^x =  Σ (x^n)/n!
      n=0
```

Session 02: Interfaces

Interfaces

"*Go's most distinctive and powerful feature.*" - Rob Pike

"*If I could export one feature of Go into other languages, it would be interfaces.*" - Ross Cox

OK... BUT WHY?

Structs

```go
package main

import "fmt"

type Vertex struct {
    x float64
    y float64
}

func main() {
    v1 := Vertex{1, 2}    // has type Vertex
    p := &Vertex{1, 2}    // has type *Vertex
    v2 := Vertex{x: 35.0} // y:0 is implicit
    v3 := Vertex{}        // x:0 and y:0

    fmt.Println(v1, p, v2, v3)
    v1.x = 42
    fmt.Println(v1.x)
}
```

Run

A `struct` is a collection of fields (and a `type` declaration does what you'd expect.)

Methods

```
type Vertex struct {
    x, y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.x*v.x + v.y*v.y)
}

func main() {
    v := Vertex{3, 5}
    fmt.Println(v.Abs())
    fmt.Println(v)
}
```

Run

A `method` is a function with a special *receiver* argument.

Methods with pointer receiver

```
type Vertex struct {
    x, y float64
}

func (v *Vertex) Scale(f float64) {
    v.x *= f
    v.y *= f
}

func main() {
    v := Vertex{3, 5}
    v.Scale(3)
    fmt.Println(v)
}
```

Run

Interfaces

```
type Abser interface {
    Abs() float64
}
```

```
func main() {
    var a Abser
    a = Vertex{3, 5}
    fmt.Printf("Abs: %f\n", a.Abs())
}
```

Run

An *interface type* is defined as a set of method signatures.

A value of interface type can hold any value that implements those methods.

Types implicitly satisfy an interface if they implement all required methods.

Type switch

```
type Stringer interface {
    String() string
}
```

```
func ToString(any interface{}) string {
    if v, ok := any.(Stringer); ok {
        return v.String()
    }
    switch v := any.(type) {
    case Abser: return fmt.Sprintf("Abs: %v", v.Abs())
    case int: return strconv.Itoa(v)
    case float32: fmt.Sprintf("%f", v)
    }
    return "???"
}
```

Check dynamically whether a particular interface value has an additional method.

Example

```
func main() {
    fmt.Println(ToString(35))
    fmt.Println(ToString(Vertex{3, 5}))
    fmt.Println(ToString(2.3 + 1.7i))
    fmt.Println(ToString(time.Minute + time.Second*10))
}
```

Run

`time.Duration` satisfies `Stringer`.

Even though package `time` does not know about `Stringer`.

```
func (d Duration) String() string { ... }  // package time
```

golang.org/pkg/time/#Duration (http://golang.org/pkg/time/#Duration)

fmt.Stringer

```go
func main() {
    fmt.Println(35)
    fmt.Println(Vertex{3, 5})
    fmt.Println(2.3 + 1.7i)
    fmt.Println(time.Minute + time.Second*10)
}
```

Run

We don't need `ToString`. `fmt.Println` and `fmt.Printf` work similarly.

```go
type Stringer interface {  // package fmt
    String() string
}
```

golang.org/pkg/fmt/#Stringer (http://golang.org/pkg/fmt/#Stringer)

Errors

```
type error interface {
    Error() string
}
```

```
func doStuff() (int, error) { ... }

func main() {
    res, err := doStuff()
    if error != nil {
        // handle error
    } else {
        // all is good, use result
    }
}
```

Go code uses `error` values to indicate an abnormal state.

Errors

```
e1 := errors.New("This is an error")
```

```
e2 := fmt.Errorf("IndexError: %d", i)
```

```
// Implementing the Error interface on a custom type
type HTTPError int

func (h HTTPError) Error() string { return fmt.Sprintf("Error. Code: %d\n", int(h)) }

e3 := HTTPError(403)
```

There are at least three ways to create your own `errors`.

Embedding types

```go
package main

import "fmt"

type A struct{}

func (a A) Foo() { fmt.Println("Foo on A") }
func (a A) Bar() { fmt.Println("Bar on A") }

type B struct {
    A
}

func (b B) Bar() { fmt.Println("Bar on B") }

func main() {
    b := B{}
    b.Foo()
    b.Bar()
}
```

Run

Embed types within a struct to "borrow" pieces of an implementation.

Embedding interfaces

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

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

```
// ReadWriter is the interface that combines the Reader and Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

Hello, net

```go
package main

import (
    "fmt"
    "net"
)

func main() {
    l, _ := net.Listen("tcp", "localhost:3333")
    defer l.Close()

    for {
        conn, _ := l.Accept()
        fmt.Fprintln(conn, "Hello!")
        conn.Close()
    }
}
```

Run

Hello, net

We just used `Fprintln` to write to a net connection.

That's because a `Fprintln` writes to an `io.Writer`, and `net.`Conn is an `io.Writer`.

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)  // package fmt
```

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

```
type Conn interface {                                            // package net
    Read(b []byte) (n int, err error)
    Write(b []byte) (n int, err error)
    Close() error
    ...
}
```

Summary

Go's interfaces are satisfied implicitly.

Enables true component architectures.

Implicit conversions are checked at compile time.

Explicit interface-to-interface conversions can inquire about method sets at run time.

Exercise: Sortable ByteSlice

For a new type `ByteSlice`, implement the methods needed to satisfy sort.Interface.

```go
package main

import (
    "fmt"
    "sort"
)

type ByteSlice []byte

// TODO: implement methods of sort.Interface
// func (b ByteSlice) Len() int { ... }
// ...

func main() {
    b := ByteSlice{2, 0, 133, 44, 10, 200}
    sort.Sort(b)
    fmt.Println(b)
    // Output: [0 2 10 44 133 200]
}
```

Run

Exercise: IntReader

```
convert int to string: strconv.Itoa(int) string
convert string to byte-slice: []byte(string)
```

Session 03: Concurrency

Concurrency

Concurrency is the ability to write your program as independently executing pieces.

Concurrency is not parallelism!

But

- What are *goroutines*?
- What are *channels*?
- What's a *select-statement*?

Goroutines

```
package main

import "fmt"
import "time"

func say(what string, after int) {
    for {
        time.Sleep(time.Duration(after) * time.Millisecond)
        fmt.Println(what)
    }
}

func main() {
    go say("Marco", 100)
    go say("Polo", 500)
    time.Sleep(3 * time.Second)
}
```

Run

A goroutine is an independently executing function, launched by a go statement.

Goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

```
// Declaring and initializing.
var c chan int
c = make(chan int)
// or
c := make(chan int) // HL
```

```
// Sending on a channel.
c <- 1
```

```
// Receiving from a channel.
// The "arrow" indicates the direction of data flow.
value = <-c
```

Communicating Goroutines

```go
func say(what string, after int, c chan int) {
    for {
        v := <-c
        fmt.Println(what, v)
        c <- v + 1
        time.Sleep(time.Duration(after) * time.Millisecond)
    }
}

func main() {
    c := make(chan int)
    go say("Marco", 100, c)
    go say("Polo", 500, c)
    c <- 0
    time.Sleep(5 * time.Second)
}
```

Select

```
select {
    case v1 := <-c1:
       fmt.Printf("recv %v from channel c1", v1)
    case v2 := <-c2:
       fmt.Printf("recv %v from channel c2", v2)
    case c3 <- 42:
       fmt.Printf("sent %v to channel c3", 42)
    default:
       fmt.Println("no communication")
}
```

`select` works like a `switch` where each case is a communication.

`select` blocks until one communication can proceed, which then does.

A default clause, if present, executes immediately if no channel is ready.

Communicating Goroutines

```
func say(what string, after int, c chan int) {
    var v int
    for {
        select {
        case <-time.After(200 * time.Millisecond):
            fmt.Println(what, "waiting...")
        case v = <-c:
            fmt.Println(what, v)
            v++
        case c <- v:
            v++
            time.Sleep(time.Duration(after) * time.Millisecond)
        }
    }
}

func main() {
    c := make(chan int)
    go say("Marco", 100, c)
    go say("Polo", 500, c)
    c <- 0
    time.Sleep(5 * time.Second)
}
```

Run

Concurrency Patterns

Generator

```go
func say(what string, after int) <-chan string {
    c := make(chan string)
    go func() {
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", what, i)
            time.Sleep(time.Duration(after) * time.Millisecond)
        }
    }()
    return c
}

func main() {
    c1 := say("Marco", 100)
    c2 := say("Polo", 500)
    for i := 0; i < 20; i++ {
        fmt.Println(<-c1)
        fmt.Println(<-c2)
    }
}
```

Run

Multiplexer

```go
func fanIn(in1, in2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case v := <-in1: c <- v
            case v := <-in2: c <- v
            }
        }
    }()
    return c
}
```

```go
func main() {
    c := fanIn(say("Marco", 100), say("Polo", 500))
    for i := 0; i < 20; i++ {
        fmt.Println(<-c)
    }
}
```

Run

Daisy-chain

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 10000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Run

Quit chan

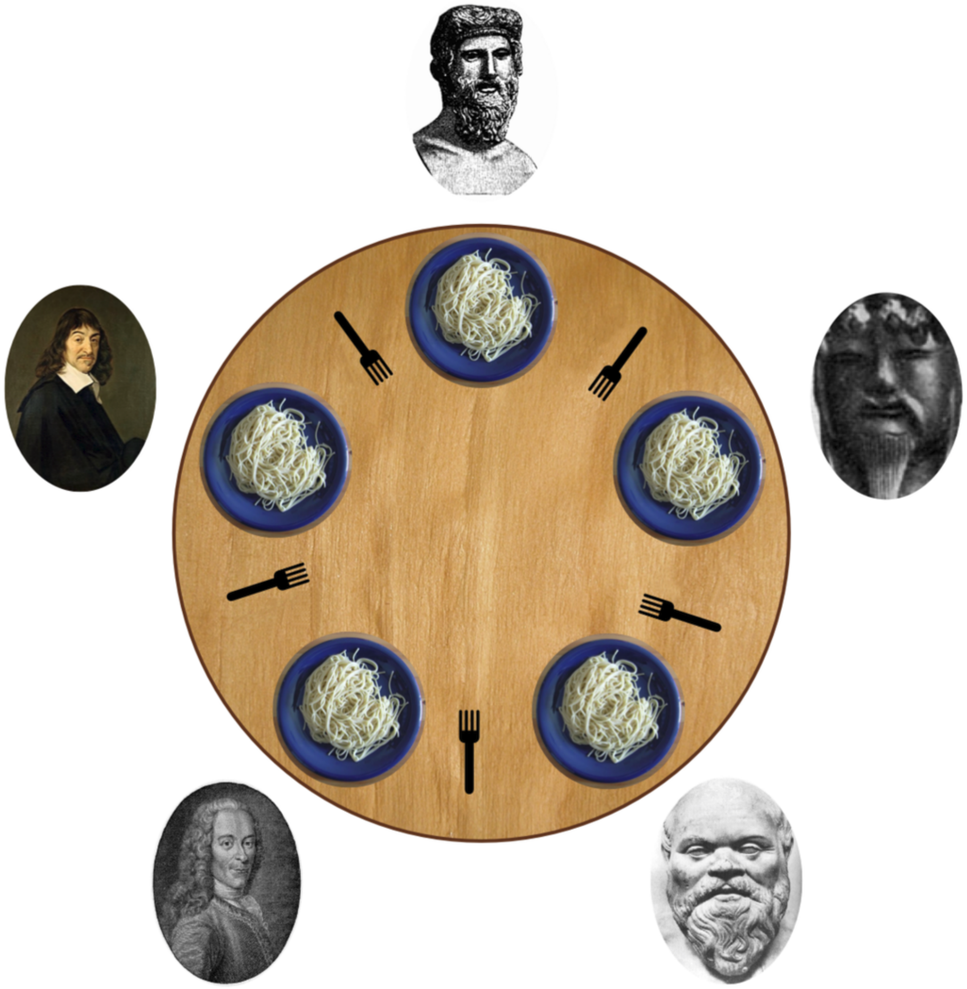
```
func f(c chan int, quit chan bool) {
    var v int
    for {
        select {
        case <-quit:
            close(c)
            return
        case c <- v:
            v++
            time.Sleep(100 * time.Millisecond)
        }
    }
}

func main() {
    c, q := make(chan int), make(chan bool)
    go func() { time.Sleep(3 * time.Second); q <- true }()
    go f(c, q)
    for v := range c {
        fmt.Println(v)
    }
}
```

Run

So...

- *Goroutines* are independently executing functions, launched by go statements.
- *Channels* are typed conduits enabling goroutines to communicate.
- *Select* is a control structure unique to concurrency.

Exercise: Dining Philosophers



Exercise: Dining Philosophers

```go
func dine(name string, left, right chan bool) {
    // TODO: IMPLEMENT
}
```

```go
func main() {
    cs := []chan bool{make(chan bool), make(chan bool), make(chan bool)}
    names := []string{"Hume", "Heidegger", "Kant"}

    for i := range cs {
        go dine(names[i], cs[i], cs[(i+1)%3])
        go func(i int) { cs[i] <- true }(i)
    }
    time.Sleep(5 * time.Second)
}
```

Run

Conclusion

Go is an **efficient, compiled** programming language that feels **lightweight** and **pleasant**

- Implicit satisfaction of interfaces
- Composition instead of inheritance
- Struct embedding
- Concurrency approach: *share memory by communicating*

References

Images (in order of occurence):

- Language is the first weapon drawn in conflict (by *Simon Liu*) (https://www.flickr.com/photos/si-mocs/31131827236) - CC BY-NC-SA 2.0 (https://creativecommons.org/licenses/by-nc-sa/2.0/)
- The Thinker (https://pixabay.com/en/the-thinker-rodin-paris-sculpture-692959/) - CC0 Public Domain
- An illustration of the dining philosophers problem (by *Benjamin D. Esham*) (https://de.wikipedia.org/wiki/Philosophenproblem#/media/File:An_illustration_of_the_dining_philosophers_problem.png) - CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/deed.de)

Thank you

12 May 2017
Tags: Go, golang, go tutorial, introduction, concurrency, interface (#ZgotmplZ)

Tim Schulte
Albert-Ludwigs-Universität Freiburg
schultetp@gmail.com (mailto:schultetp@gmail.com)