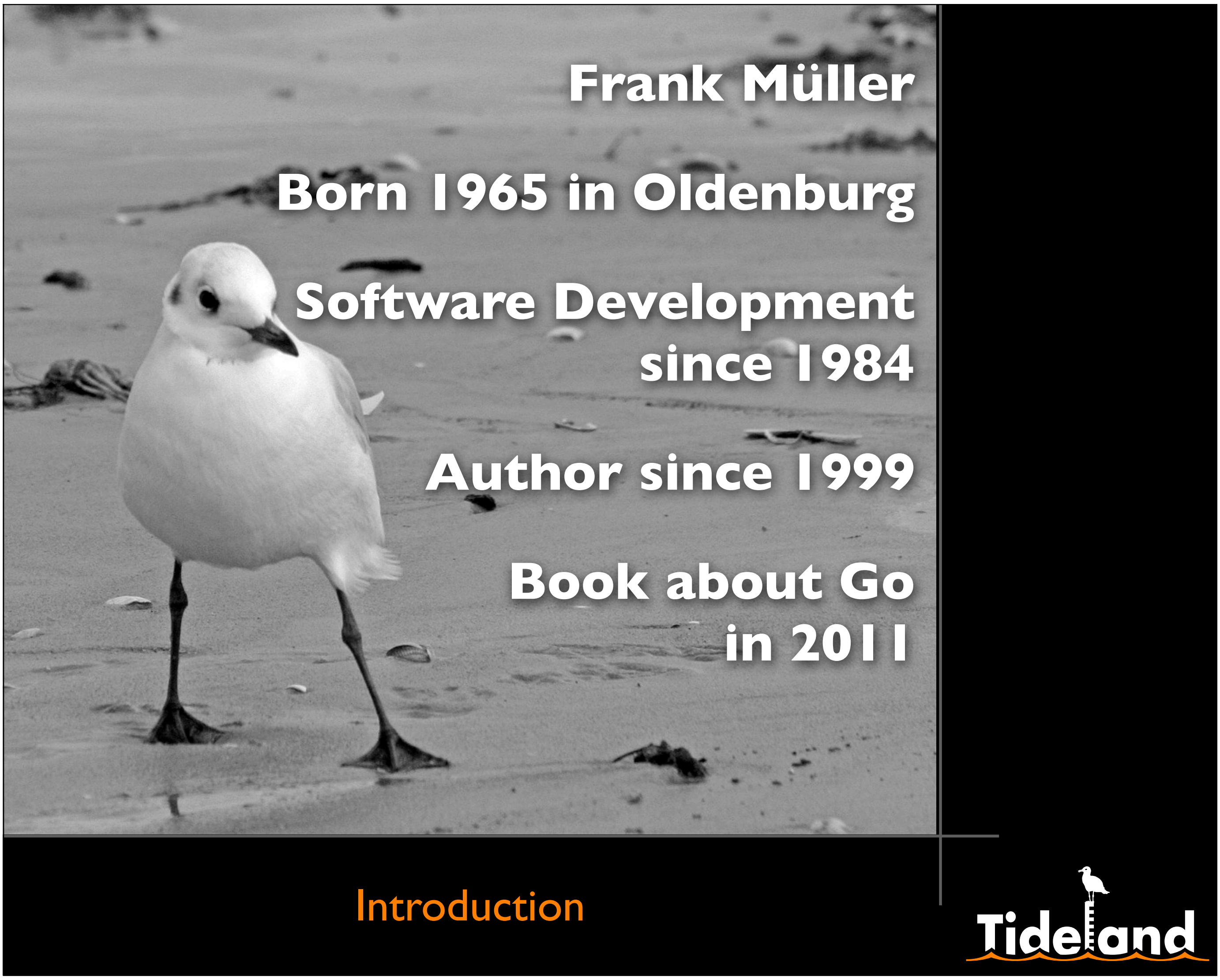
Beauty and Power of Go

Frank Müller / Oldenburg / Germany

Tideland



Tideland is the name of the ecosystem of the German north-sea cost near to Oldenburg

Way to Google Go mostly influenced by Pascal, C, ReXX, Java, Python, Smalltalk and Erlang

Articles and reviews about programming languages and development processes for the German iX magazine

Time is too short for a full tutorial

Concentration on most important features

Example code for some practice later

- End of 2007 start by Rob Pike, Ken Thompson and Robert Griesemer
- Implementation started after design in mid 2008
- Going public in November 2009
- Release of Go 1 in March 2012

History

Tideland

Design decisions had to be made unanimous

Sometimes this forced longer discussions

As a result the core language has been quite stable when going public

- Rob Pike
  - Unix, Plan 9, Inferno, acme, Limbo, UTF-8
- Ken Thompson
  - Multics, Unix, B, Plan 9, ed, UTF-8
  - Turing Award

Famous parents

Tideland

Multics, Unix, Plan 9, Inferno: operating systems

B, Limbo: programming languages

acme, ed: editors

Go aims to combine the **safety** and **performance** of a statically typed compiled language with the **expressiveness** and **convenience** of a dynamically typed interpreted language.

It also aims to be suitable for modern systems - large scale - programming.

– Rob Pike

Tideland

```
// Organized in packages.
package main

// Packages can be imported.
import "fmt"

// Simple function declaration.
func sayHello(to string) {
   // Package usage by prefix.
   fmt.Printf("Hello, %s!\n", to)
}

// Main program is function main() in package main.
func main() {
   sayHello("World")
}
```

Hello, World!

Tideland

Still too complex example of a main program (smile)

main is the name of the program package, main() is the main function

Arguments and return values are handled via packages

Directly using println(“Hello, World!“) in main is the shortest way

- Roots in C and Plan 9
- Native binaries with garbage collection
- Static typing
- Concurreny
- Interfaces
- Powerful tools
- No fantastic new pur language

Some quick facts

Tideland

Very fast compilation has been a design goal

Static typing is strict

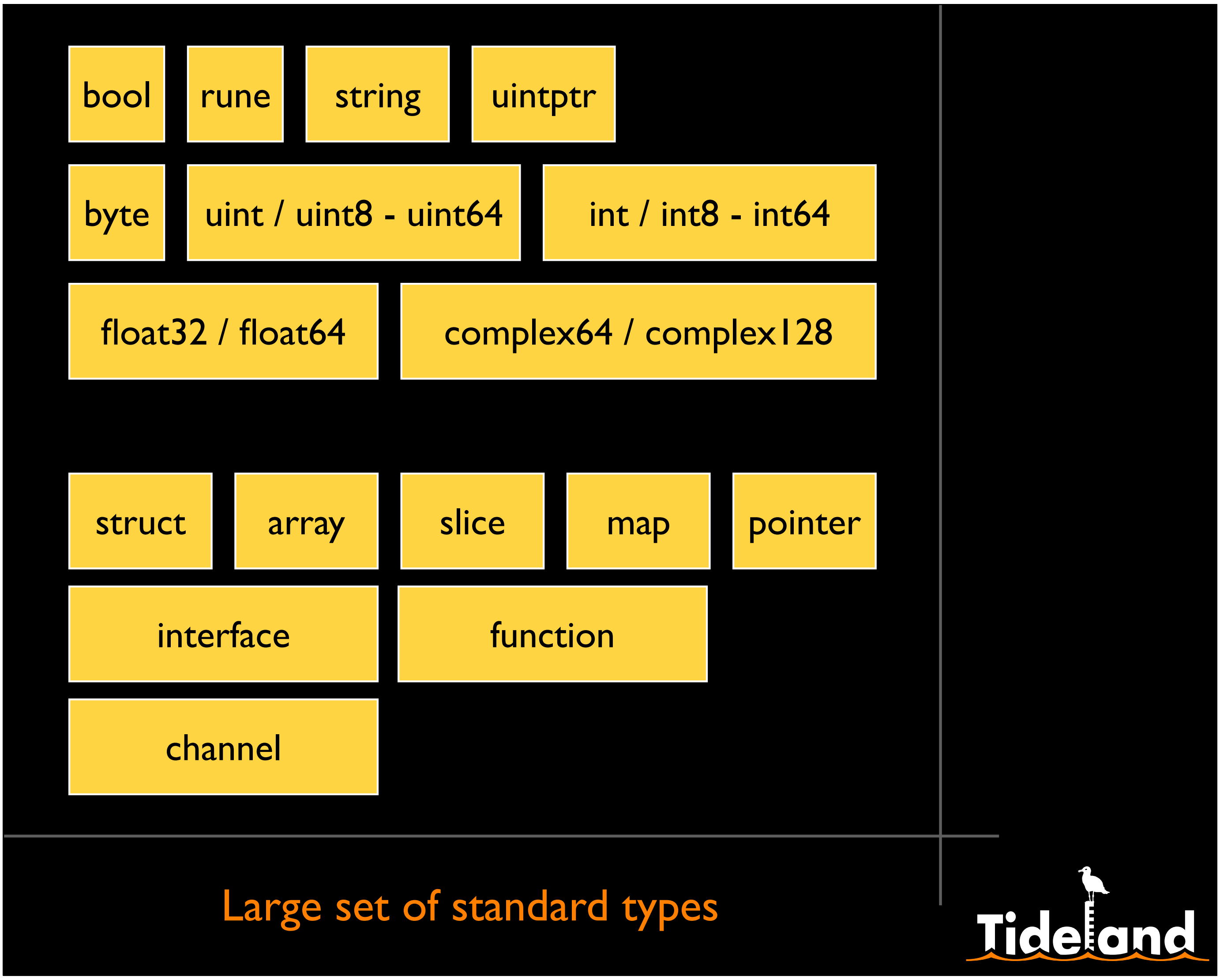
Concurreny originates in Tony Hoare's Communicating Sequential Processes (CSP)

Share by communication

It is better to have a **permanent income** than to be **fascinating**.

– Oscar Wilde

Tideland



Upper half: predeclared types

Lower half: composite types

```
// Declaration can be done this way:
var i int

i = myIntReturningFunction()

// More usual is:
i := myIntReturningFunction()

// Even for complex types:
s := []string{“Hello“, “World“}
m := map[string]string{
   “foo“: “bar“,
   “baz“, “yadda“,
}

// Or multiple values:
v, err := myFunctionWithPossibleError()
```

Implicit declaration

Tideland

Explicit declaration of variable and type with var is allowed

Mostly the short declaration and initializing with := is used

```
// Alias for a simple type.
type Weight float64

func (w *Weight) String() string {
   return fmt.Sprintf(“%.3f kg“, w)
}

// Struct with hidden fields.
type Name struct {
   first string
   last  string
}

func (n *Name) String() string {
   return n.first + “ “ + n.last
}

func (n *Name) SetLast(l string) {
   n.last = l
}
```

Own types can have methods

String() is the only method of the Stringer interface allowing types to return a natural representation as a string

Implicit export: lower-case is package-private, upper-case is exported

When setter and getter are needed, the convention is SetFoo() and Foo()

Asterisk in front of type passes variable as reference instead of a copy (important for settings)

```go
// Two-dimensional object.
type BasePlate struct {
   width float64
   depth float64
}

func (bp *BasePlate) Area() float64 {
   return bp.width * bp.depth
}

// Use the base plate.
type Box struct {
   BasePlate
   height float64
}

func (b *Box) Volume() float64 {
   return b.Area() * b.height
}
```

No inheritence, but embedding

Tideland

Embedding may cause naming troubles with fields or methods, e.g. Box defines Area() itself

Here the embedded field or method has to be addressed full qualified: b.BasePlate.Area()

When the embedded type is from external package no access to private fields or methods is possible

Using of embedding type by embedded type is possible by interface implementation and passing to embedded type (tricky)

```
// Create a base plate.
func NewBasePlate(w, d float64) *BasePlate {
   return &BasePlate{w, d}
}

// Create a box.
func NewBox(w, d, h float64) *Box {
   return &Box{BasePlate{w, d}, h}
}

// Create an illuminated box.
func NewIlluminatedBox() *IlluminatedBox {
   ib := &IlluminatedBox{
      box:           NewBox(1.0, 1.0, 1.0),
      lightOnHour:   22,
      lightDuration: 8 * time.Hour,
   }
   go ib.backend()
   return ib
}
```

No contructors, simple functions

Tideland

In simple cases often directly creations like &BasePlate{} or &Box{} are used

When using the short initialization of a struct all fields in order or some fields with name has to be passed

Interfaces

- Important abstraction
- Duck typing à la Go
- Set of method signatures
- No explicit implementation

Duck test: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck

No "type foo struct implements bar, baz { ... }"

Just provide the methods of the interface

```
// Interface declaration.
type Duck interface {
   Quack() string
}

// First duck implementation.
type SimpleDuck Name

func (sd *SimpleDuck) Quack() string {
   return fmt.Sprintf("Quack, my name is %s", sd)
}

// Second duck implementation.
type ComplexDuck struct {
   name Name
   box  *IlluminatedBox
}

func (cd *ComplexDuck) Quack() string {
   return fmt.Sprintf("Quack, my name is %s " +
      "and my box has %.3f m³.", cd.name,
      cd.box.Volume())
}
```

Declare and implement interfaces

Tideland

Interface Duck only defines one the method Quack()

Very often interfaces only contain few methods describing what the implementor is able to do

```
// A pond with ducks.
type Pond struct {
   ducks []Duck
}

// Add ducks to the pond.
func (p *Pond) Add(ds ...Duck) {
   p.ducks = append(p.ducks, ds...)
}

// Add ducks to the pond.
func main() {
   p := &Pond{}
   huey := NewSimpleDuck(...)
   dewey := NewComplexDuck(...)
   louie := NewAnyDuck(...)

   p.Add(huey, dewey, louie)

   for _, duck := range p.ducks {
      println(duck.Quack())
   }
}
```

Use interfaces

Tideland

Focus on what, not how

Pond.Add() shows usage of variadic final parameter, which internally is a slice of the given type

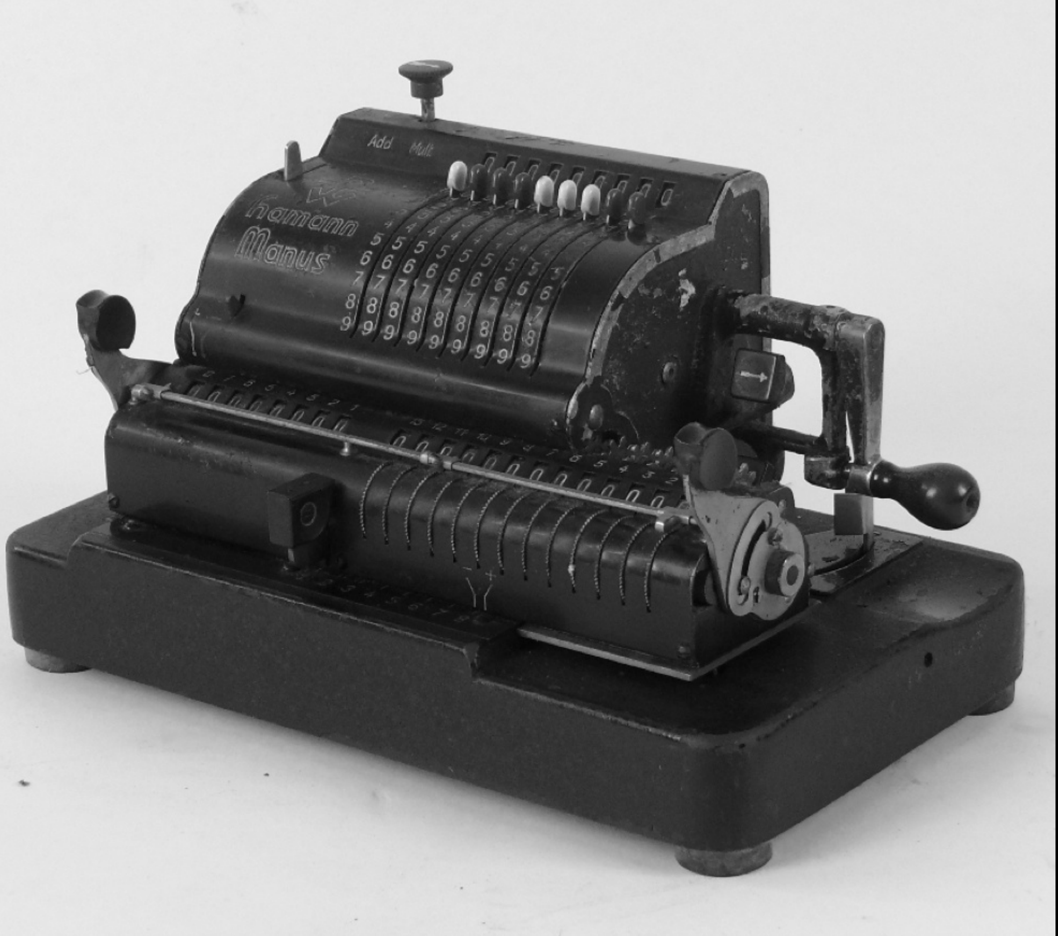
The build-in functions append() and copy() support the work with slices

```
// An empty interface has no methods.
type Anything interface{}

// Type switch helps to get the real type.
func foo(any interface{}) error {
   switch v := any.(type) {
   case bool:
      fmt.Printf(“I‘m a bool: %v“, v)
   case ComplexType:
      fmt.Printf(“I‘m a complex type: %v“, v)
   default:
      return errors.New(“Oh, type is unexpected!“)
   }
   return nil
}

// Or even shorter with type assertion.
v, ok := any.(WantedType)
if !ok {
   return errors.New(“Expected a ‘WantedType‘!“)
}
```

Empty interface

The empty interface may look strange but shows exactly what it is: an interface without methods

Empty interface, type switch and type assertion allow a transparent control flow for generic data



Functions

- First-class functions
- Higher-order functions
- User-defined function types
- Function literals
- Closures
- Multiple return values

Flexible world of functions

```go
// Function type to do something with ducks.
type DuckAction func(d Duck) error

// Pond method to let the ducks do something.
func (p *Pond) LetDucksDo(a DuckAction) error {
   for _, d := range p.ducks {
      if err := a(d); err != nil {
         return err
      }
   }
}

// Use a duck action.
func CollectQuacks(p *Pond) ([]string, error) {
   quacks := []string{}
   if err := p.LetDucksDo(func(d Duck) error {
      quacks = append(quacks, d.Quack)
      return nil
   }); err != nil {
      return nil, err
   }
   return quacks, nil
}
```

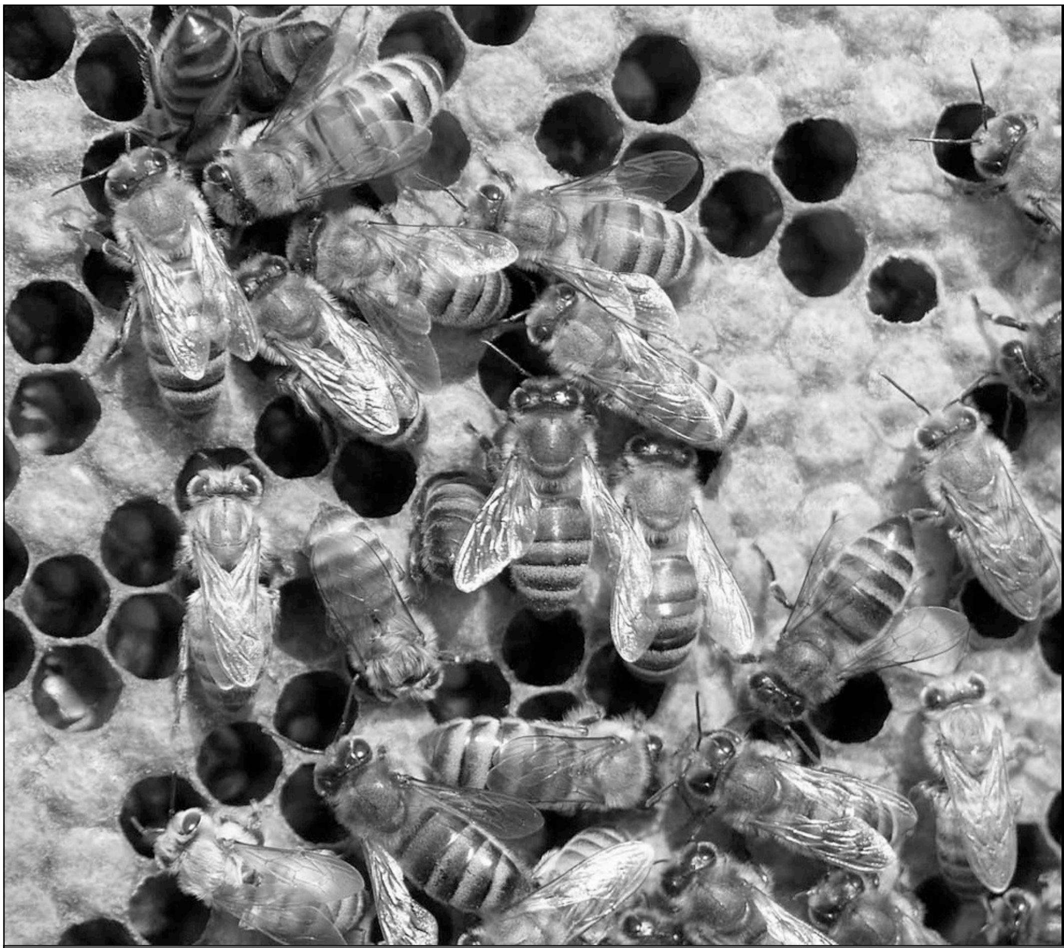
Full bandwidth of function usage

Tideland

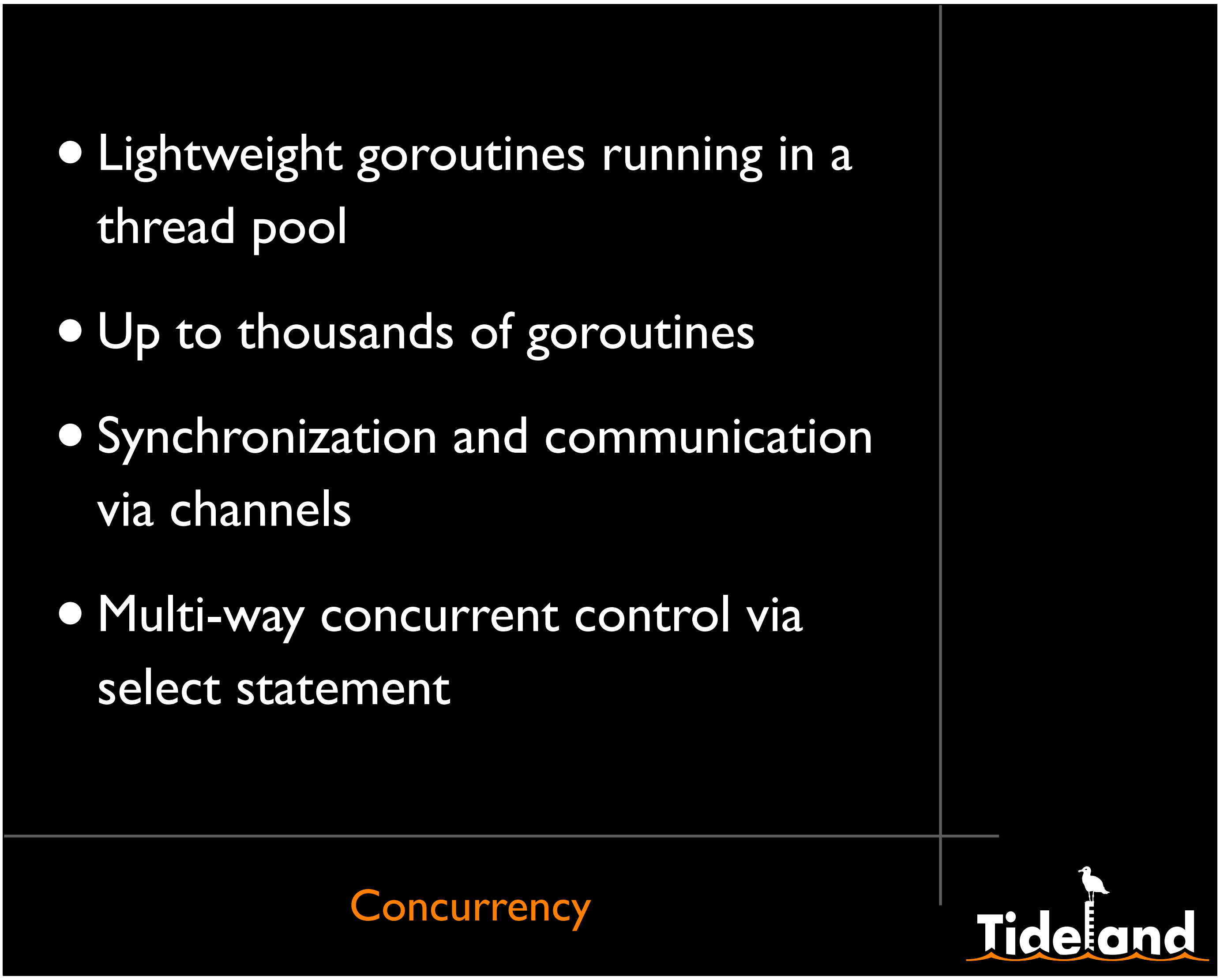
LetDucksDo() uses the user-defined function type DuckAction as higher-order function

First-class function CollectQuacks() passes a function literal to LetDucksDo()

As a closure it has access to the variable quacks of the surrounding function



Concurrency



Memory overhead and context switching very small compared to threads (like Erlang)

Independent of the number of cores (reason will get more clear later)

Sessions with more than a million goroutines have been tested

Goroutines may share multiple channels, not just one mailbox per goroutine (unlike Erlang)

- Concurrency is the composition of independently executing processes
- Parallelism is the simultaneous execution of computations in a context

Concurrency is not parallelism

Tideland

Concurreny is the base for parallelism

- Modern applications deal with lots of things at once
- Concurrency allows to structure software for these kinds of applications

Structuring applications with concurrency

Tideland

Multicore computers

Cluster and clouds of CPUs

Networks

User sessions

```go
// A goroutine is just a function.
func fileWrite(filename, text string) {
   f, err := file.Open(filename)
   if err != nil {
      log.Printf("fileWrite error: %v", err)
   }
   defer f.Close()

   f.WriteString(text)
}

// It is started with the keyword 'go'.
func main() {
   f := "..."
   t := "..."

   // Write file in the background.
   go fileWrite(f, t)

   // Meanwhile do something else.
   ...
}
```

Simple goroutine

Tideland

The keyword go starts a function in the background

No id or handle returned

fileWrite() introduces defer which executes functions when the surrounding function will be left

Here when file opening shows no error the closing gets automated

```go
// Read strings and convert them to upper case.
func pipe(in <-chan string, out chan<- string) {
   for s := range in {
      out <- strings.ToUpper(s)
   }
}

// Use buffered channels for async. communication.
func main() {
   in := make(chan string, 50)
   out := make(chan string, 50)

   go pipe(in, out)

   in <- "lower-case string"
   s := <-out

   // Prints “LOWER-CASE STRING“.
   fmt.Println(s)
}
```

Pipeline goroutine using range statement

Channels have to be created with make()

By default unbuffered, but can have a buffer

```go
// Job with four parts.
func job(s *Source, t *Target) {
   a := partA(s)
   b := partB(a)
   c := partC(b)
   d := partD(c)
   t.Set(d)
}

// Start job multiple times.
func parallel(s *Source, t *Target) {
   for i := 0; i < 4; i++ {
      go job(s, t)
   }
}
```

Naive approach leads to syncing problems

Tideland

Reading of data from source has to be managed

Writing into target has to be managed

Order has to be kept

parallel() returns after start of goroutines, but jobs aren‘t yet done

```go
// Start a pipeline of parts.
func concurrent(s *Source, t *Target) {
   abChan := make(chan *AResult, 5)
   bcChan := make(chan *BResult, 5)
   cdChan := make(chan *CResult, 5)
   waitChan := make(chan bool)

   go partA(s, abChan)
   go partB(abChan, bcChan)
   go partC(bcChan, cdChan)
   go partD(cdChan, t, waitChan)

   <-waitChan
}
```

Pipelined approach

More like a production line

No syncing in source and target needed

Each part signals when everything is done

Last part signals via waitChan that work has been finished

```go
// Type X managing some data.
type X struct { ... }

// Backend of type X.
func (x *X) backend() {
   for {
      select {
      case r := <-w.requestChan:
         // One request after another.
         w.handleRequest(r)
      case c := <-w.configChan:
         // A configuration change.
         w.handleConfiguration(c)
      case <-w.stopChan:
         // X instance shall stop working.
         return
      case <-time.After(5 * time.Second):
         // No request since 5 seconds.
         w.handleTimeout()
      }
   }
}
```

Complex scenarios using select statement

Tideland

Many different channels are possible (check exactly if needed due to complexity)

Timeout not needed, but may be used for safety aspects

select also knows default branch, taken if no other data received

```go
// A possible request.
type Request struct {
   f            func() *Response
   responseChan chan string
}

// Public method to add an exclamation mark.
func (x *X) Exclamation(in string) string {
   // Also x could be modified here.
   f := func() { return in + "!" }
   req := &Request{f, make(chan string)}
   x.requestChan <- req
   return <-req.responseChan
}

// Using the requests closure in the backend.
func (x *X) handleRequest(req *Request) {
   res := req.f()
   x.responses = append(x.responses, res)
   req.responseChan <- res
}
```

Request and response handling

Tideland

Function passed in a request allows to serialize access to state (see x.responses)

```go
// Start 10 pipes to do the work.
func balancedWorker(ins []string) []string {
   in, out := make(chan string), make(chan string)
   outs := []string{}
   // Start worker goroutines.
   for i := 0; i < 10; i++ {
      go worker(in, out)
   }
   // Send input strings in background.
   go func() {
      for _, is := range ins {
         in <- is
      }
      close(in)
   }()
   // Collect output.
   for os := range out {
      outs = append(outs, os)
      if len(outs) == len(ins) {
         break
      }
   }
   return outs
}
```

Simple load balancing

Tideland

Only two channels for input and output

Multiple worker process the data independently

Input is written in the input channel in background too

Received results may have a different order, depending on individual processing duration

```go
// Receiving data.
select {
case data, ok := <-dataChan:
   if ok {
      // Do something with data.
   } else {
      // Channel is closed.
   }
case <-time.After(5 * time.Seconds):
   // A timeout happened.
}

// Sending data.
select {
case dataChan <- data:
   // Everything fine.
case <-time.After(5 * time.Seconds):
   // A timeout happened.
}
```

Adding some safety

Tideland

While receiving channel could be closed or it may last too long

While sending the receiver(s) may have a deadlock so data can‘t be sent and the program hangs

Timeouts may allow graceful termination

No exceptions?

- error is just an interface
- Errors are return values
- defer helps to ensure cleanup tasks
- panic should be used with care
- recover can handle panics
- Leads to clear and immediate error handling

Error, panic and recover

Tideland

error only contains the method Error() string

Packages errors allows to create a standard error with New()

Package fmt has Errorf() to return errors containing variable text information

panic() not for control flow but for real logical failures

```go
// Error for non-feedable ducks.
type NonFeedableDuckError struct {
   Duck Duck
   Food *Food
}

// Fullfil the error interface.
func (e *NonFeedableError) Error() string {
   return fmt.Sprintf(“It seems %v dislikes %v, “,
      “maybe a rubber duck?“, e.Duck, e.Food)
}

// Error as only return value.
func FeedDucks(p *Pond, f *Food) error {
   return p.LetDucksDo(func(d Duck) error {
      // Type assert for ‘FeedableDuck‘ interface.
      if fd, ok := d.(FeedableDuck); ok {
         return fd.Feed(f)
      }
      return &NonFeedableDuckError{d, f}
   })
}
```

Definition and usage of own error type

Tideland

If Duck and Food not needed the generic fmt.Errorf() could be used

Own types and fields allow the error handling to use type switching and get more informations about the error reason

```
// Retrieve data from a server.
func Retrieve(addr, name string) (*Data, error) {
   conn, err := net.Dial(“tcp“, addr)
   if err != nil {
      return nil, err
   }
   defer conn.Close()
   // Do the retrieving.
   ...
   return data, nil
}

// Somewhere else.
data, err := Retrieve(“localhost:80“, “/foo/bar“)
if err != nil {
   // Handle the error.
   ...
}
```

Error in multi-value return, cleanup with defer

Tideland

Linear control flow with error checks is typical

Symetric operations like Open()/Close() or Dial()/Close() often use defer (fire and forget)

defer statements are function or method calls executed in LIFO order

This way work is done step by step, no need for cleanup later

```
// Save execution of function ‘f‘.
func safeExec(f func()) (err error) {
   defer func() {
      if r := recover(); r != nil {
         err = fmt.Sprintf(“Ouch: %v!“, r)
      }
   }
   f()
   return nil
}

// Somewhere else.
var result float64

err := safeExec(func() {
   // Divide by zero.
   result = 1.0 / (1.0 - 1.0)
})
if err != nil {
   // Do the error management.
   ...
}
```

panic and recover

Tideland

Unhandled panics lead to a program abort printing a stack trace (typically readable enough to get a good hint about the error)

Direct divide by zero isn‘t possible, compiler detects it (smile)

Packages

- Network / HTTP / Template
- Encodings (JSON, XML, ...)
- I/O, Formatting
- Crypto
- Images
- Time
- Reflection

Large set of useful packages

Tideland

Packages are, like the language specification, very good documented at http://golang.org

- Import of external packages
- Namespace based on SCM
  - BitBucket, GitHub
  - Google Code, Launchpad
- Example:
  - “code.google.com/p/tcgl/redis“

External packages

Tideland

Retrieving of external packages uses labels/tags matching the local Go version

The go tool

Tideland

- build - build the software
- fmt - format the sources
- test - run unit tests
- install - install the package
- get - retrieve and install an external package
- doc - source documentation

Most important go subcommands

Tideland

- http://golang.org/
- http://tour.golang.org/
- http://blog.golang.org/
- http://godashboard.appspot.com/
- http://go-lang.cat-v.org/
- http://www.reddit.com/r/golang/
- #go-nuts on irc.freenode.net
- Google Group golang-nuts

Some links

Tideland

Thank you for listening.

Questions?

And now some practice ...

Tideland