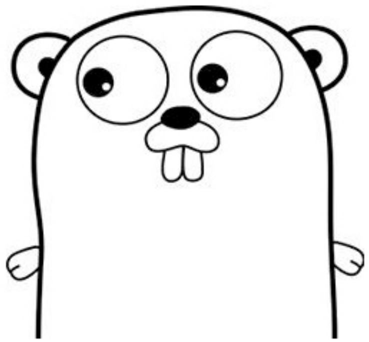


Google GO

An introduction

Paolo Baldan

Languages for Concurrency and Distribution

Some ideas

- dynamic language, statically typed
- compiled
- garbage collected
- powerful and light type system
- OO features
- concurrent

Ken Thompson
Rob Pike
Robert Griesemer

Ken Thompson

Why should we Go?

- Compiled, statically-typed languages (like C, C++, Java) require too much typing and too much typing:
  - verbose, lots of repetition
  - types get in the way too much
- compiles takes far too long
- poor concurrency

Why should we Go?

- Dynamic languages (Python, JavaScript) fix some problems (no more types, no more compiler) but introduce others:
  - errors at run time that should be caught statically
  - no compilation means slow code

Taking the best?

- Google Go tries to take the best of the two worlds (l'oste dice sempre che il suo vino è buono)
  - compiled to machine code
  - static types
  - some type inference (not full, as in ML)
  - complete (semi-formal) specification
  - wonderful concurrency primitives

History

- Project starts at Google in 2007 (by Griesemer, Pike, Thompson)
- Open source release in November 2009
- More than 2000 contributors
- Version 1.0 release in March 2012
- Version 1.20 (March, 2023)

Popularity?

- Language of the year in 2009 and 2016 (by Tiobe)
- 10th position popularity ranking
- Used in several organisations

Basics

Packages

- Programs are organised in packages
- Special package main with special function main()

- Go Standard Library: packages supplied with Go

Example

epoch.go

```go
package main

import (
   "fmt"
   "time"
)

func main() {
   fmt.Println(time.Now().Unix(),
                    "sec since Unix epoch\n")
}
```

```
$ go run epoch.go

Output:
1682505658 sec since Unix epoch
```

The go tool

- compile

```
$ go build [ epoch.go ]
```

- compile and run

```
$ go run epoch.go
```

- format

```
$ go fmt epoch.go
```

Lexical structure

- Source is UTF-8

```
π = 3.14
```

- C-like comments, number and strings

```
/* This is a comment; no nesting */
// So is this.
```

```
23      0x0FF       1.234e7
```

```
"Hello, world\n"  `\nabc\t` == "\\nabc\\t"
```

Back to packages

- Possibly split in several files
- In a **directory** which conventionally has the same **name** as the package

Example

epochSplit/epochSplit.go

```go
package main

import "fmt"

func message(val int64) string {
   return fmt.Sprintf("%d",val) + " sec since Unix epoch"
}
```

epochSplit/epochSplit1.go

```go
package main

import (
"time"
"fmt"
)

func main() {
   fmt.Println(message(time.Now().Unix()))
}
```

Exported names

- A name is exported when it starts with capital letters

```
func message(val int) string { // not visible to
                                // importers
   ...
}


func Message(val int) string { // visible to importers
   ...
}
```

Where are my semicolons?

- “;” terminate statements but ... inserted by the lexer at end of line if the previous token could end a statement.

```
func message(s int64) string {
   return fmt.Sprintf("%d",s)
          + " sec since Unix epoch"
}
```

!NO!

```
func message(s int64) string {
   return fmt.Sprintf("%d",s) +
          " sec since Unix epoch"
}
```

!OK!

Hello Who?

helloWho.go

```go
import (
	"fmt"
	"os"
	"strings"
)

func main() {
	who := "World!"          /* implicit declaration */
	if len(os.Args) > 1 { /* os.Args[0] is "helloWho" */
		who = strings.Join(os.Args[1:], " ")
	}
	fmt.Println("Hello ", who)
}
```

Basic types

C-like types

- Various integers

```
var a int               // integer
var b int32             // integer (32 bit), rune
var c int64             // integer (64 bit)
var d byte              // unsigned integer (8 bit)
```

- Cannot be mixed up: strict typing!

```
c = b                   // illegal!
```

- And more, more or less familiar types ...

Booleans

- Boolean `bool`
- Usual boolean type, with values `true` and `false` (predefined const).
- Strict! Pointers and integers are not booleans.

More types

- Floats

```
var f float32      // float (32 or 64 flavor)
```

- Complex numbers are built in

```
var c complex64    // complex nums (64 or 128)
c = 3 + 2i
```

Strings and pointers

- Strings
  - built in
  - ~ immutable arrays

```
var s string

c = s[1]       // char
s[2] = ...     // illegal
```

- Pointers
  - but no pointer arithmetic!

```
var p *int
```

Arrays ...

- Array (and slices)

```
var a [9]int                  // array (fixed length)
len(a) = 9
```

- Arrays are values (not pointers)

```
func fval(a  [3]int, val int) { a[0]= val }
func fptr(a *[3]int, val int) { a[0]= val }

func main() {
  var ar [3] int
  fval(ar,3)
  fmt.Println(ar) // passes a copy of ar    (out: [0,0,0])
  fptr(&ar,3)
  fmt.Println(ar) // passes a pointer to ar (out: &[3,0,0])
}
```

Creating values

- Similar syntax for all composite types

```
[3]int{1,2,3}    // Array of 3 integers
```

```
[10]int{1,2,3}   // Array of 10 integers
                 // first three not zero
```

```
[...]int{1, 2, 3}        // ... -> self count
```

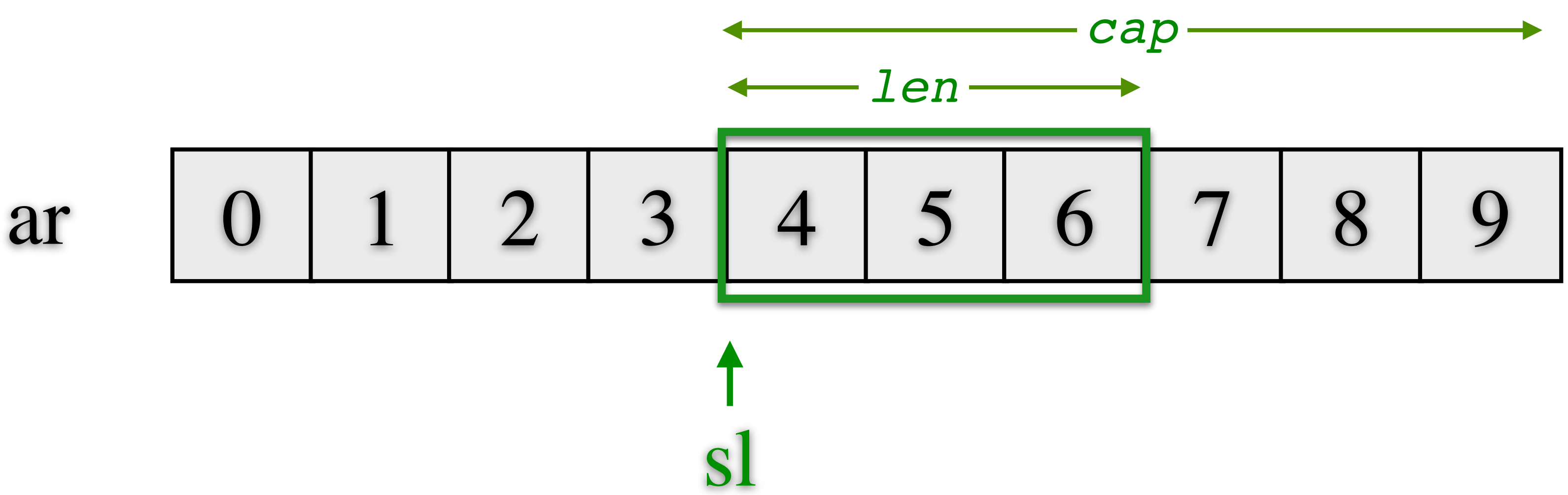
```
[10]int{2:1, 3:1, 5:1} // key:value pairs
```

Slices

- Slices

```
var sl []int  // slice (~ ptr to int array)
```

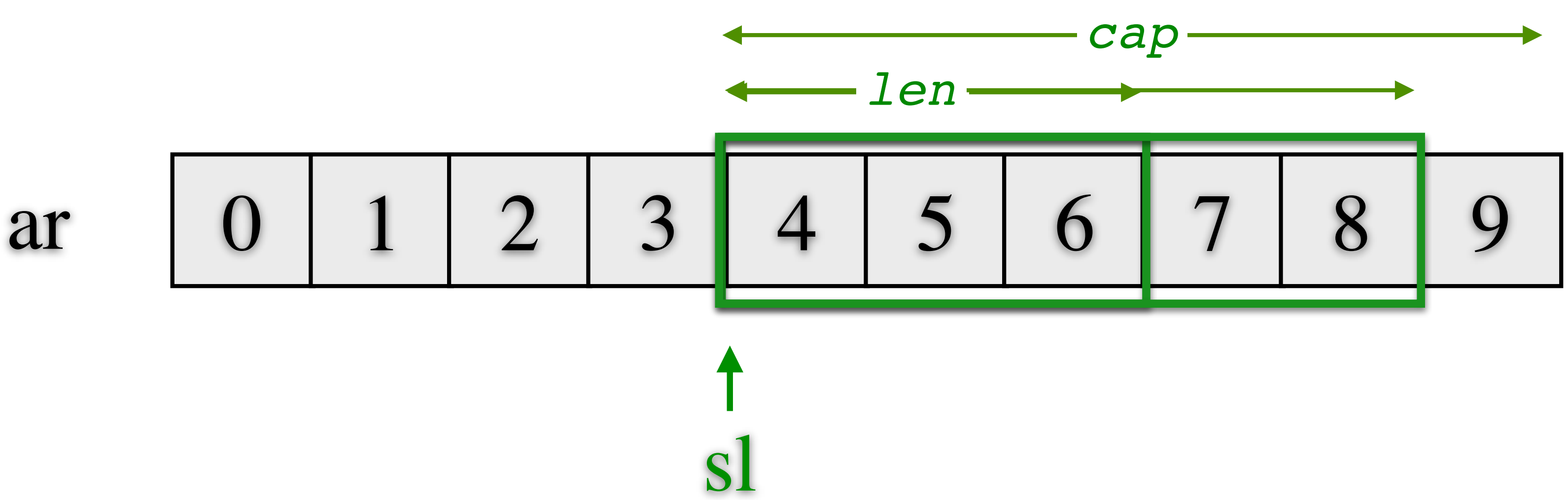
```
var ar = [...]int{0,1,2,3,4,5,6,7,8,9}
sl = ar[4:7]          // ref to subarray {4,5,6}
                      // len(sl) = 3, cap(sl) = 6
```



Slices

- Reslicing

```
sl = sl[0:5]          // ref to subarray {4,5,6,7,8}
                      // len(sl) = 5, cap(sl) = 6
```



Maps

- ~ associative arrays (reference type)

```
m := make(map[string]int) // map with string keys
                          // and integer values
m["cat"] = 3
m["dog"] = 2
```

- testing for presence (comma ok idiom) and deleting

```
wolfnum, ok := m["wolf"]   // ok false if absent
delete (m, "wolf")         // delete
```

Structs

- Usual syntax

```
var p struct {
     x, y, z float64
     name    string
}
```

- More to come ...

Variables

- Declared explicitly (possibly with initializers)

```
var a int
var b int = 2

var (
     a, b string = "stringa", "stringb"
     c    float32
)
```

- or implicitly (inside functions)

```
b := "string"     // type is inferred
```

Initialisation

- Variables not initialised are "zero"
  - numeric 0, bool false, string ""
  - nil pointer, map, slice, channel, ...

```
for i := 0; i < 5; i++ {
    var v int
    fmt.Printf("%d ", v)
    v = 5
}
```

```
outputs: 0 0 0 0 0
```

Constants

- Declared like variables, but with the const keyword.
- Can be character, string, boolean, or numeric values.

```
const Pi = 3.14
const ApproxPi = float64(Pi)
const Vero = true
const VeroString  = "true"
```

User defined types

- Keyword type

```
type Point struct {
     x, y, z float64
     name string
}


type SliceOfIntPointers []*int

type Operator func(a, b int) int
```

Functions

Adder

adder.go

```go
package main

import "fmt"

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

func main() {
    fmt.Println(add(12, 13))
}
```

Multiple return values

swap.go

```go
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a := "first"
    b := "second"
    a, b = swap(a, b)
    fmt.Println(a, b)
}
```

Blank identifier

- What if you care only about the first value of swap?
- Solution: blank identifier _ (underscore).

```go
func main() {
  a := "first"
  b := "second"
  a, _ = swap(a, b)
  fmt.Println(a)
}
```

Exercise

- Simpler way of swapping?

```
func main() {
  a := "first"
  b := "second"
  a, b = b, a
}
```

Named results

swapNamed.go

```go
package main

import "fmt"

func swap(x, y string) (w string, z string) {
   w = y
   z = x
   return
}

func main() {
   a := "first"
   b := "second"
   a, b = swap(a, b)
   fmt.Println(a, b)
}
```

Functions are values

executor.go

```go
type MyFun func(int) int
```

```go
func execute (fun MyFun, x int) int {
  return fun(x)
}
```

```go
func iterate (fun MyFun, n, x int) int {
  if n == 0 { return x }
  return fun(iterate(fun, n-1, x))
}
```

Functions are values

executor.go

```
type MyFun func(int) int
```

```
func double (inFun MyFun) MyFun {
   var outFun MyFun =
     func(x int) int {
               return 2*inFun(x)
      }
    return outFun
}
```

```
func main() {
  succ := func(x int) int { return x+1  }

  fmt.Println(execute(succ, 2))       Out:  3
  fmt.Println(iterate(succ, 10, 2))  Out: 12
  fmt.Println(double(succ)(1))        Out:  4
}
```

Fun values as closures

```go
type MyFun func(int) int                    closure.go
```

```go
func closure() MyFun {
   var x int = 3
   return
      func(delta int) int {
         x += delta
         return x
      }
}
```


```go
func main () {
   f := closure()
   fmt.Println(f(1))
   fmt.Println(f(10))
   fmt.Println(f(100)) }

/* Prints 4 14 114 - accumulating in f's x */
```

Defer

- Execute a function when the enclosing function returns

```
func data(fileName string) string {
   f := os.Open(fileName)
   defer f.Close()
   contents := io.ReadAll(f)
   return contents
}
```

- Useful for closing fds, unlocking mutexes, etc.

One call for each defer

- Each defer that executes stacks a function call to execute, in LIFO order, so

```
func f() {
  for i := 0; i < 5; i++ {
     defer fmt.Printf("%d ", i)
  }
}
```

prints 4 3 2 1 0.

Control structures

Control structures

- `if`, `for` (in variants), `switch` ... and more

```
if a == b { return true }
else { return false }

for i = 0; i < 10; i++ { ... }

switch val {
  case 1:  ...
  case 2:  ...
  default: ...
}
```

If

- Initialization statement allowed

```go
if v := count(); v < 10 {
    fmt.Printf("%d less than 10\n", v)
} else { ... }
```

- Avoid longer version

```go
{
  v := count();
  if v < 10 {
    fmt.Printf("%d less than 10\n", v)
  } else { ... }
}
```

If

- Useful with multivariate functions:

```
if n, err = fd.Read(buf); err != nil {
    ...
    use n
    ...
}
```

- Missing condition means true (not too useful here but handy in "for", "switch")

For

- Basic form is familiar:

```
for i := 0; i < 10; i++ { ... }
```

- While:

```
for Cond { ... }
```

- Missing condition = true

```
for { fmt.Printf("never stop") }
```

For

- Iterating over arrays, slices, maps (and more); get key/value pairs ...

```
m := map[string]float64{"e":2.718, "pi":3.142}
for key, value := range m {
    fmt.Printf("key %s, value %g\n", key, value)
}
```

range.go

- .. or only keys

```
for key := range m {
    fmt.Printf("key %s", key)
}
```

Exercise

- Iterating over values?

```
for _, val := range m {
    fmt.Printf("value %g", val)
}
```

Switch

```
switch instrCod % 4 {
  case 0, 1: nArgs = 1  /* zero or succ instr */
  case 2:    nArgs = 2  /* transfer instr */
  case 3:    nArgs = 3  /* jump instr */
}
```

```
switch instrCod % 4 {
  case 2:    nArgs = 2  /* transfer instr */
  case 3:    nArgs = 3  /* jump instr */
  default:   nArgs = 1  /* zero or succ instr */
}
```

Switch

- Expressions can be of any type and missing expression means true

```
switch {
  case a <  b: return 1
  case a == b: return 0
  case a >  b: return -1
}
```

Noo, a goto ...

- break, continue ... almost goto ...

```
Loop: for i := 0; i < 10; i++ {
          switch v:=f(i); v {
             case 0, 1, 2: break Loop
             case 8: return v
          }
        }
```

Summary

- Go programs organised in packages
- Strict type discipline, type inference
- Standard basic types and structured types
  - arrays, structs
  - slices, maps, function types
- Pointers
- Standard control structures (with some sugar)

Type System

Type System

- Not an OO language, but some OO features
  - Custom types can have "methods"
  - Type embedding
  - Interfaces and generics for polymorphism

Methods

No classes, (custom) types can "have" methods

```go
type Vertex struct {
   X, Y float64
}

// self: explicit method receiver
func (self *Vertex) Abs() float64 {
   return math.Sqrt(self.X*self.X+self.Y*self.Y)
}
```

```go
func main() {
   v := &Vertex{3, 4}     // Vertex{x,y}: create val of
                          // type Vertex, initialised
   fmt.Println(v.Abs())
}
```

Implicit (de)referencing

```go
// pointer receiver
func (self *Vertex) Abs() float64 {
  return math.Sqrt(self.X*self.X+self.Y*self.Y)
}

v := Vertex{3, 4}
fmt.Println(v.Abs()) // need a pointer? take the addr
```

```go
// non pointer receiver
func (self Vertex) Abs() float64 {
  return math.Sqrt(self.X*self.X+self.Y*self.Y)
}

vp := &Vertex{3, 4}
fmt.Println(vp.Abs()) // need a value? dereferenced
```

New

- Built in **`new(T)`**: allocate memory for T value and returns a pointer

```
var v *Vertex = new(Vertex)
v.X, v.Y = 3, 4   // Variable of type *Vertex
                  // object allocated and
                  // then initialised same as
                  // v := &Vertex{3,4}

var p := new(int) // p has type *int
```

- No free / delete / dispose (garbage collected)

Custom New

- Initialisation method normally called New

```
package vertex

type Vertex struct {
   X, Y float64
}
...
// creation
func New (x, y float64) *Vertex {
   var v *Vertex = new(Vertex)
   v.X, v.Y = 3, 4
   return v                    // same as return &Vertex{3,4}
}
```

```
func main() {
   v := vertex.New(3,4)
   fmt.Println(v.Abs())
}
```

Methods for all

- (Almost) all types can have methods

```go
type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

func main() {
	f := MyFloat(-math.Sqrt(2))
	fmt.Println(f.Abs())
}
```

Visibility

- Package scope
  - Capitalization determines **exported**/ **local**
  - Structs in the same pkg have full access to one another's fields and methods
  - A "local" type can "export" its fields and methods

```go
type vertex struct { X, Y float64 }
```

Methods for types

vertex.go

```
package vertex

type Vertex struct {
   X, Y float64
}
```

```
func (self *Vertex) Abs() float64 {
   return math.Sqrt(self.X*self.X+self.Y*self.Y)
}
```

```
func (self *Vertex) SetX(x float64) {
   self.X = x
}
```

```
func New (x, y float64) *Vertex {
   var v *Vertex = new(Vertex)
   v.X, v.Y = 3, 4
   return v                    // same as return &Vertex{3,4}
}
```

Methods for types

```go
package main

import (
   "..."
   "vertex"
)

func main() {
   v := vertex.New(3,4)
   v.SetX(5)
   fmt.Println(v.Abs())
}
```

Embedding

- No classes, no inheritance, but embedding/composition

embedding.go

useColoredVertex.go

```
import "vertex"

type ColoredVertex struct {
   vertex.Vertex
   Color    int
}
```

```
func New (x, y float64, color int) *ColoredVertex{
   return &ColoredVertex{*vertex.New(x,y), color}
}
```

Delegation

- Fields and methods of embedded types are accessible from the embedder

```
import "vertex"

type ColoredVertex struct {
  vertex.Vertex
  Color    int
}
```

```
cv := coloredVertex.New(3, 4, 2)
fmt.Println(cv.X)
fmt.Println(cv.Abs())
```

Delegation rules

- (Unqualified) fields/methods refer to the "closer" type in the "embedding hierarchy"

```
type StrangeVertex struct {
  vertex.Vertex
  X float
}

sv = strangeVertex.New(...)
sv.X
```

- "Overridden" fields still accessible

```
sv.Vertex.X
```

Embedding conflicts

- Still there can be conflicts

```
import ( "vertex" ; "vertexC" )

type doubleVert struct {
   vertex.Vertex
   vertexC.VertexC // vertexC just a copy of vertex
                   // with additional Col field
}
```

```
func main() {
   dv := doubleVert{*vertex. New(1,2),
                    *vertexC.New(3,4,0)}
   fmt.Println(dv.Color) // working fine
   fmt.Println(dv.X)    // error (ambiguous selector)
   fmt.Println(dv.Vertex.X) // working fine
}
```

Even more OO

- Use functions as first-class values

```
type Vertex struct {
   X, Y float64
   Abs func ()float64
}
```

vertexObj.go

```
func New(x, y float64) *Vertex {
   v := new(Vertex)

   v.X = x
   v.Y = y
   /* problems? */
   v.Abs = func () float64 {
      return math.Sqrt(v.X*v.X+v.Y*v.Y)
   }
   return v
}
```

Even more OO

- Methods local to each instance and modifiable
- Still no subclassing
- Against the ideas of the designers ...

Interfaces

- User type which corresponds to a **set of types**
  - **Basic**: defined by a list of methods
  - **Embedded**: intersection of interfaces
  - **General**: general type expressions with union, underlying types

Basic Interfaces

- Defined by a **list of methods** signatures (abstracting away from the receiver)
- Implemented **implicitly** by any type that realises the specified methods (**duck typing**)
- A type may implement an arbitrary number of different interfaces

Recall

vertex.go

```go
package vertex

type Vertex struct {
   X, Y float64
}
```

```go
func (self *Vertex) Abs() float64 {
   return math.Sqrt(self.X*self.X+self.Y*self.Y)
}
```

```
....
```

Interfaces

```
type Abser interface {
   Abs() float64      // no receiver
}
```

```
func DoSomething (a Abser) {
   fmt.Println(a.Abs())
}
```

```
func main() {
   v := vertex.New(3, 4) // returns a *Vertex

   DoSomething(v)   // a *Vertex implements Abser
   DoSomething(*v)   // a Vertex does not: error!
}
```

Interfaces

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
  if f < 0 {
    return float64(-f)
  }
  return float64(f)
}
```

```
var f MyFloat = -1
DoSomething(f)  // also MyFloat implements Abser
```

Interfaces

- Satisfied (implemented) implicitly
- A type can satisfy multiple interfaces
- E.g.: `*Vertex` satisfies

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

- Empty interface referred as **`any`**

Containers and empty interface (the old way)

- Empty interface values can be of any type

```
package stack
import "errors"

type Stack [] any

func (stack *Stack) Push(x any) {
 *stack = append(*stack, x)
}
```

Stack (cont.)

- Once a value is in a Stack, it's stored as an (empty) interface value.

```
type MyInt int

func (self MyInt) Triple () MyInt {
                              return 3*self     }

var s stack.Stack
var m MyInt = 3

s.Push(3)      // int stored as interface val
s.Push(m)      // MyInt stored as interface val
```

Type assertions

- Once a value is in a Stack, it's stored as an interface value.
- Need to “unbox” it to get the original back with a type assertion

```
interfaceValue.(typeToExtract)
```

Type assertions

MyInt	3
int	3

```
i,_ := s.Pop()  // MyInt retrieved as any
fmt.Println("%d", i.Triple()) // compile error
if i, ok := i.(MyInt); ok {
     fmt.Printf("Here is my int: %d\n", i.Triple()/3)
}
```

```
Out: Here is my int: 3
```

```
j,_ := s.Pop()   // int retrieved as any
   // fmt.Println("%d", j.Triple()) -> compile error
   if j, ok := j.(MyInt); ok {
     fmt.Printf("Here is my int: %d\n", j.Triple()/3)
   }
```

```
Out: none
```

Type Switches (and variadic functions)

```
func classifier(items ...interface{}) {
  for i, x := range items {
      switch x.(type) {
      case bool:
         fmt.Printf("param #%d is a bool\n", i)
      case float64:
         fmt.Printf("param #%d is a float64\n", i)
      case int, int8, int16, int32, int64:
         fmt.Printf("param #%d is an int\n", i)
      case uint, uint8, uint16, uint32, uint64:
         fmt.Printf("param #%d is an unsigned int\n", i)
      case string:
         fmt.Printf("param #%d is a string\n", i)
      default:
         fmt.Printf("param #%d's type is unknown\n", i)
      } } }
```

Type Switches (and variadic functions)

```
classifier(5, -17.9, "ZIP", true, complex(1, 1))

param #0 is an int
param #1 is a float64
param #2 is a string
param #3 is nil
param #4 is a bool
param #5's type is unknown
```

Finer checks

- Does a value val implement a method?

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

if strVal, ok := val.(Stringer); ok {
     fmt.Printf("the value implements String: %s",
                strVal.String()
```

Generics

- Type parameters for **function** and **types**
- Interface types as sets of types (with embedding and union)
- Type inference, which permits omitting type arguments in many cases

Generic functions

```
func sel[T any](first, second T, i int) T {
   var result T
   switch i {
   case 0: result = first
   case 1: result = second
   default:
   }
   return result
}
```

```
  sel[int](12, 23, 1)
```

```
  selString := sel[string]
  selString(“first", "second", 1)

  sel("first", "second", 1)
```

Generic types: Stack

```
package stack
import "errors"

type Stack[T any] []T

func (stack *Stack[T]) Push(x T) {
 *stack = append(*stack, x)
}
```

Errors!

- Multivalued functions for managing errs

error.go

```
func  div(a, b int) (int, error)  {
   if b != 0 {
      return a/b, nil
   }
   return 0, errors.New("Cannot divide by 0")
}

func  main() {
  var a, b
  ...
   if res, err := div(a,b); err != nil {
      fmt.Println(err)
   } else {
      fmt.Println(res)
   }
}
```

More packages ...

`webServer.go`

- net/http for creating web-based apps

Concurrency

Goroutines

- A function can be launched in a separate lightweight thread

```
go f(x, y, z)
```

- Goroutines run in the same address space (sync primitives in sync package)

Old "bad" stuff

- Conditions (wait, signal, broadcast)
- Mutexes (lock, unlock)
- RW Mutexes
- Wait group
- Atomic

Message from Thompson

- Do not communicate by sharing memory.
- Instead, **share memory by communicating**.

(Typed) Channels

- Very similar to CCS (pi-calculus) channels
- Any type can be transmitted on a channel (also structs, functions, … and channels!)

```
ch := make(chan int)    // Create channel for
                        // integer values
```

- Reference type!

```
ch1 = ch                // ch1 and ch access
                        // the same channel
```

Input/Output

- `<-` send value to channel on the left

```
ch <- 1          // Send 1 to channel ch

ch <- v          // Send v to channel ch
```

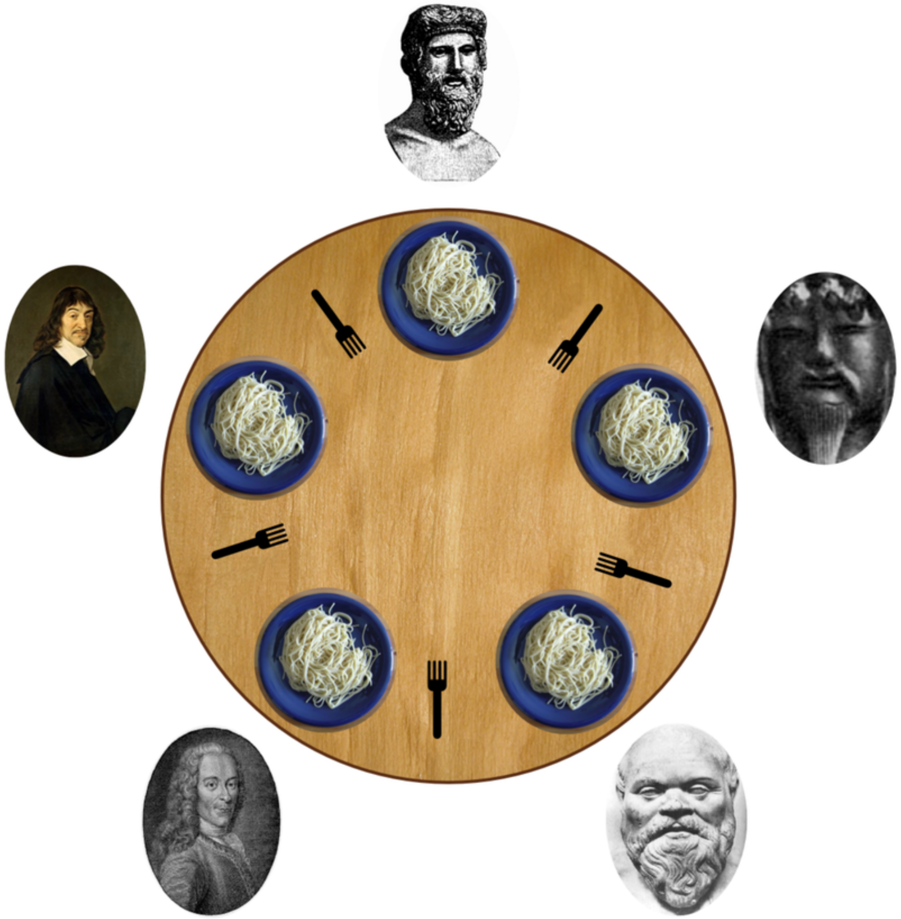
- `<-` receive value from channel on the right

```
v := <- ch       // Receive from ch, and
                 // assign value to v

<- ch            // Receive value and
                 // throw it away
```

Semantics

- By default communication is **synchronous**
- Send and receive are **blocking**
  - **Send** on a channel blocks until a receiver is available for the same channel.
  - **Receive** on a channel blocks until a sender is available for the same channel.

Channels, not locks



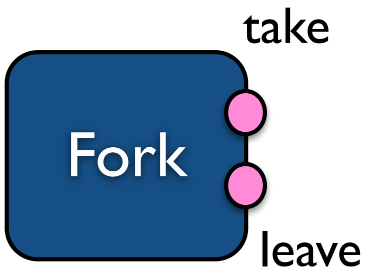
Dining philosophers, a classical problem

- Philosophers think and then eat
- For eating they need the left and right fork
- Hence their standard behaviour consists of thinking, taking the forks, eating, releasing the forks and so on ...

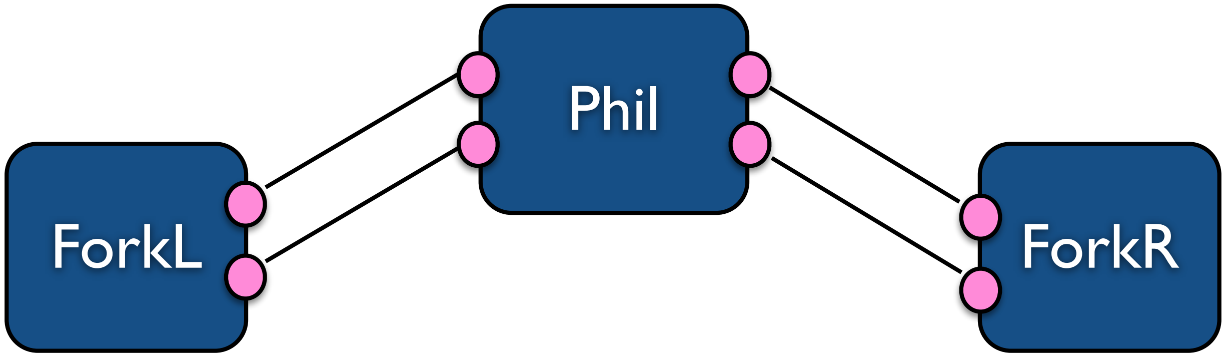
Channels, not locks

- **Dining philosophers**:
  forks can be implemented
  - via shared-memory locks (not the Go philosophy ;-)
  - as active entities (as in CCS!)

Dining philosophers



Fork = take. leave. Fork



Phil = think. 'takeL. 'takeR.
eat.
'leaveL.'leaveR. Phil

Forks

```go
// Fork type
type Fork struct {
  name string            //  fork name
  free bool              //  is it free?

  // (public) channels
  Take  chan int         //  request of taking
  Leave chan int         //  release
}
```

Forks

```go
// Fork behaviour
func (self *Fork) Run () {
   for {
      if self.free {
         <- self.Take
         self.free = false
         fmt.Printf("Fork %s taken\n", self.name)
      } else {
         <- self.Leave
         self.free = true
         fmt.Printf("Fork %s released\n", self.name)
      }
   }
}
```

“Stateless” Forks

```
// Fork behaviour
func (self *Fork) Run () {
  for {
     <- self.Take
     fmt.Printf("Fork %s taken\n", self.name)
     <- self.Leave
      fmt.Printf("Fork %s released\n", self.name)
  }
}
```

Philosophers

```
// Philosopher behaviour
func Phil (id int, left *Fork, right *Fork) {
   for {
      fmt.Printf("Phil %d is thinking ...\n", id)
      time.Sleep(time.Duration(rand.Int63n(2*1e9)))
      left.Take <- 1
      time.Sleep(time.Duration(rand.Int63n(2*1e9)))
      right.Take <- 1
      fmt.Printf("Philosopher %d is eating ...\n", id)
      left.Leave  <- 1
      right.Leave <- 1
   }
}
```

Main

```
func main () {
   var forks [NPhil]Fork

   for i:=0; i<NPhil; i++ {

      // create the ith-fork
      forks[i] = Fork {
              fmt.Sprintf(“F%d",i),
              make(chan int),
              make(chan int)}
      go forks[i].Run()
   }
   for i:=0; i<NPhil-1; i++ {
      go Phil(i, &forks[i], &forks[(i+1)%NPhil])
   }

   // left-handed philosopher: no deadlock!
   // part of main (since when main terminate
   //              all goroutines are killed)
   Phil(NPhil-1, &forks[0], &forks[NPhil-1])
}
```

Directionality

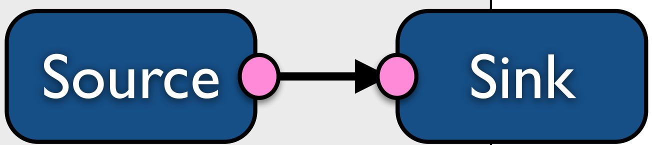
- Channel types can be annotated with directionality

```
var recCh <-chan int      // only for receive
var sndCh chan<- int      // only for send
```

Directionality

- Channels are created bidirectional, they can be assigned/passed to unidirectional vars

```go
func sink(ch <-chan int) {
    for { <-ch }
}

func source(ch chan<- int) {
    for { ch <- 1 }
}

c := make(chan int)        // bidirectional
go source(c)
go sink(c)
```



Directionality

- Forks

```
type Fork struct {
   name string        // fork name

   Take  <-chan int   // in channels: requests of take
                                      &
   Leave <-chan int   //              release
}
```

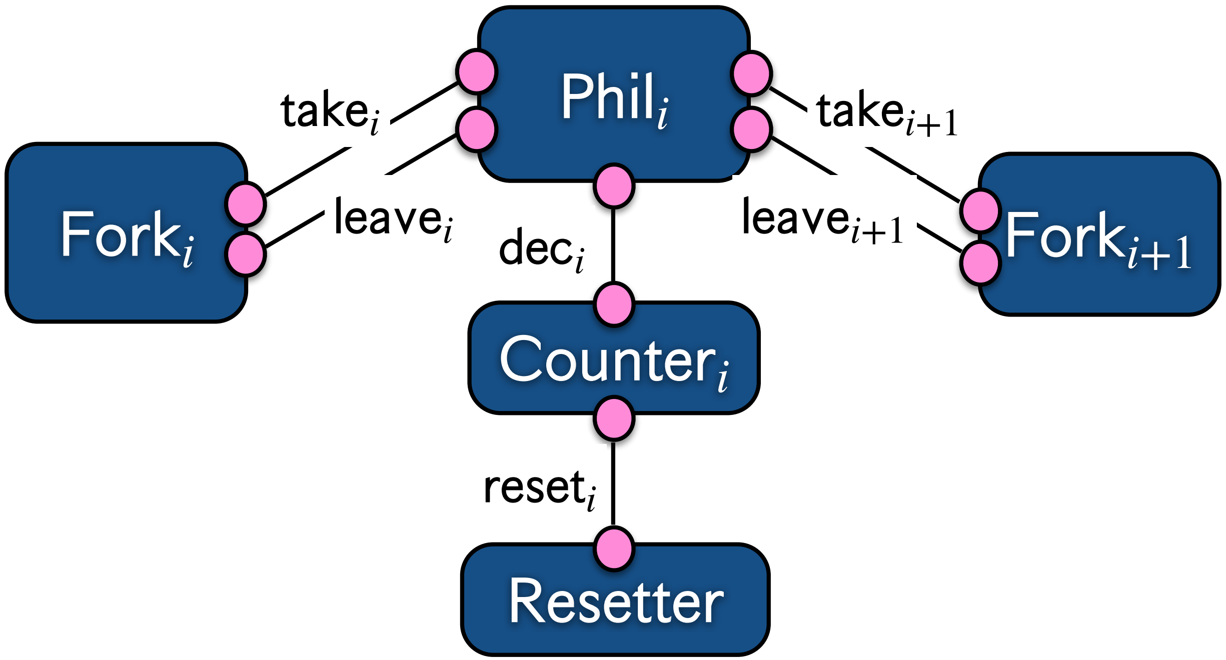
- Philosophers

```
func Phil (id int,
           LTake, LLeave, RTake, RLeave chan<- int)
{ ... }
```

Fair philosophers

- As seen in CCS
  - for each philosopher there is a companion counter process, initially at k
  - it can take a token decrementing the counter and, either disregard it or use it to eat (at least one used)
  - counter, when 0, is reset by a process which cyclically restores all counters.

Fair philosophers



$$\mathrm{Phil}_i = \mathrm{think} \,.\, \overline{\mathrm{dec}_i} \,.\, \mathrm{take}_i \,.\, \mathrm{take}_{i+1} \,.\, \mathrm{eat} \,.\, \mathrm{leave}_i \,.\, \mathrm{leave}_{i+1} \,.\, \mathrm{Phil}_i$$

$$\mathrm{Counter}_i = \mathrm{dec}_i \,.\, \ldots \,.\, \mathrm{dec}_i \,.\, \mathrm{reset}_i \,.\, \mathrm{Counter}_i$$

$$\mathrm{Resetter} = \overline{\mathrm{reset}_1} \,.\, \ldots \,.\, \overline{\mathrm{reset}_5} \,.\, \mathrm{Resetter}$$

Fair Phils: Counter

```
type Counter struct {
   // top value
   top int

   // channels
   Dec chan Signal        //  use a token
   Res chan Signal        //  restore
}
```

```
// Counter behaviour
func (self *Counter) Run () {
   var curVal int
   for {
      curVal = self.top
      for curVal > 0 { // if there tokens available, provide one
         <- self.Dec
         curVal = curVal- 1
      }
      // when value goes down to zero, wait to be reset
      <- self.Res
   }
}
```

Fair Phils: Resetter

```
/* resetter (takes in input a slice of counters) */
func Resetter (counters []Counter) {
   for {
      for _,counter := range counters {
         counter.Res <- sigVal
      }
   }
}
```

Fair Phils: Philosopher

```
func Phil (id int, left *Fork, right *Fork, counter *Counter) {

   /* number of tokens disregarded consecutively */
   disregarded := 0

   for {
      /* decrement the counter, if possible */
      counter.Dec <- sigVal

      /* the probability of eating increases with hunger */
      if (disregarded < MaxToken) && (rand.Intn(disregarded+1) == 0) {
         disregarded++  /* decide not to eat */
      } else {

         /* std philosopher behaviour: take forks, eat, leave forks */
         ...
         ...

         // starts again, reset disregarded tokens
         disregarded = 0

      }}}
```

Recap

- Go Concurrency
  - Goroutines
  - Channel-based communication
  - Synchronous by default

Semantics: Async

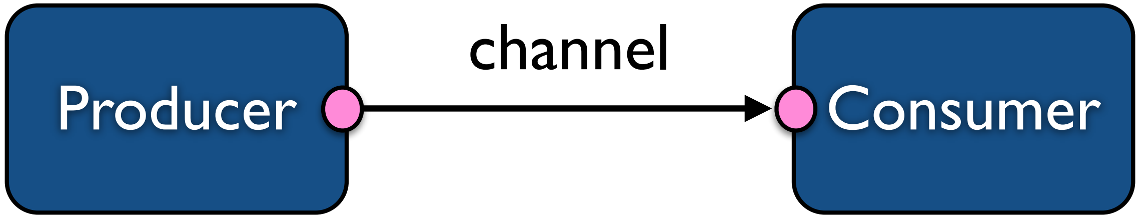
- Asynchronous communication (blocking receive, non-blocking send) with buffered channels

```
ch := make(chan int, 100)   // send blocks only
                            // if buffer full
```

- Buffering is a property of the value, not of the type

```
ch1 := make(chan int)
ch1 = ch
```

Producer - Consumer



Producer-Consumer

```
// Producer
func producer (channel chan<- int) {
  for {
    value := rand.Intn(100) // produce a value
    channel <- value        // send
    fmt.Println("Producer: Sent value", value)
  }
}

// Consumer
func consumer (channel <-chan int) {
  for {
    value <- channel       // receive
    fmt.Println("Consumer: Got value", value)
  }
}
```

Producer-Consumer/2

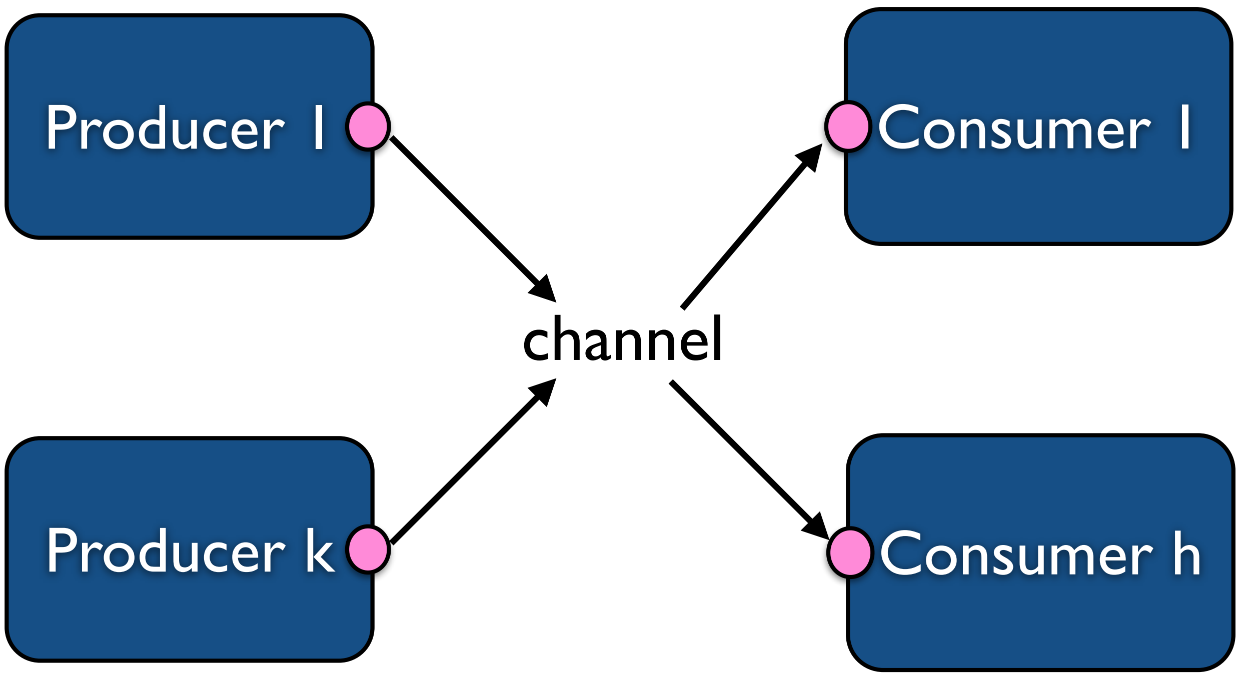
```go
// Main
func main() {
  channel := make(chan int, BUF_SIZE)
  go producer(channel)
  go consumer(channel)
  …
}
```

Producer-Consumer/3

- What if the channel were synchronous?

```
// Main
func producer (channel chan int) {
  channel := make(chan int)
  go producer(channel)
  go consumer(channel)
  …
}
```

Many Producers
Many Consumers



Many Producers Many Consumers

prodCons.go

```
// Main
func main () {

  channel := make(chan int, BUF_SIZE)

  go producer(channel)  // Producer 1
  …
  go producer(channel)  // Producer k

  go consumer(channel)  // Consumer 1
  …
  go consumer(channel)  // Consumer h
  …
}
```

"Active" channels

pump.go

```
func pump() chan int {
  ch := make(chan int)
  go func() {
    for i := 0; ; i++ { ch <- i }
  }()
  return ch
```

```
func main () {
  stream := pump()
  fmt.Println(<-stream) // prints 0
  fmt.Println(<-stream) // prints 1
  fmt.Println(<-stream) // prints 2
}
```

Summary

- Processes (Goroutines) & Channel-based interaction
- Send (`c<-exp`) and receive (`v:=<-c`)
- Communication is synchronous by default
- It can be made asynchronous (via buffered channels)

Select

- Construct for waiting on multiple channels

```
select {
    case   v:= <- ch1:
           command1
    case   ch2 <- exp:
           command2
    default:
           command3
}
```

- ... or guarded sums in Google Go (~)

```
ch1(v). command1 + ‘ch2(exp). command2 + τ.command3
```

Semantics

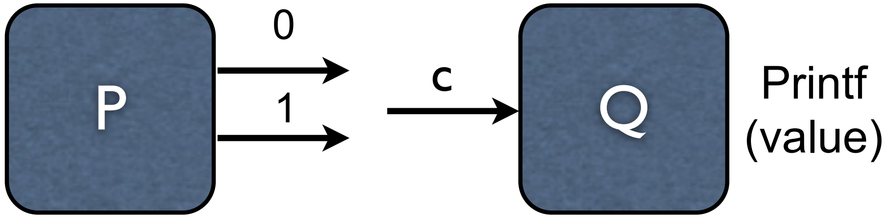
- Every case is a (possibly :=) communication
- If multiple cases are ready, one is selected with a uniform pseudo-random choice ("pseudo-random, fairly")
- If none is ready:
  - If there is a default clause, that runs
  - If there is no default, select blocks until one communication can proceed

Semantics (cont.)

- Values to be sent evaluated at the beginning
- No re-evaluation of channels or values

Select

- When multiple possibilities are available, a pseudo-random choice is taken



randomGen.go

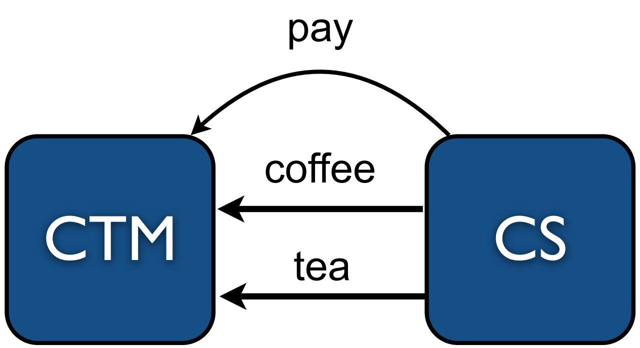
```
zeros := ones := 0
for {
   select {
   case c <- 0:
      zeros +=1
   case c <- 1:
      ones +=1
   default:
   // do nothing
}}
```

```
for {
   msg := <- c
}
```

The "office" example

`office.go`

Coffee and tea machine CTM, interacting with a user CS (computer scientist)



CTM
- Inputs some money
- Provides coffee or tea

CS
- Publish and earn some money
- Insert money in CTM (pay)
- Asks for coffee or tea

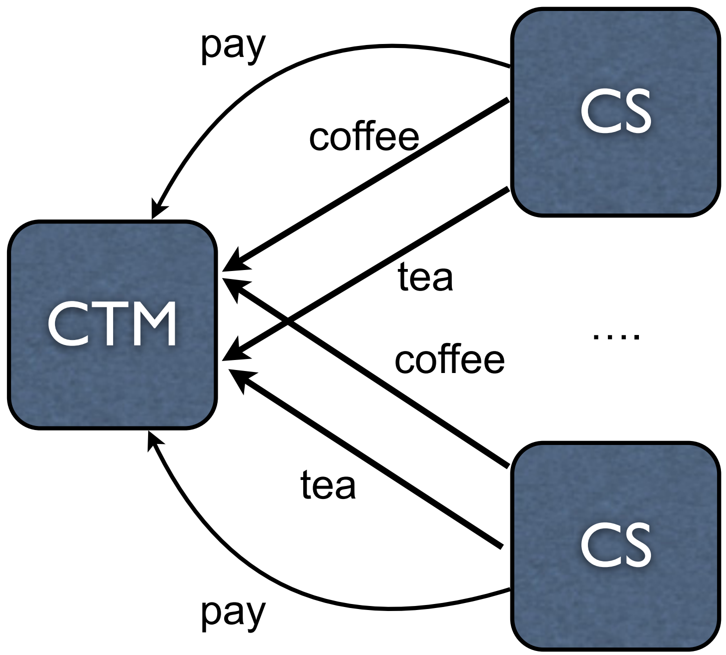
```
// Coffee & Tea Machine (CTM structure)
type CTM struct {
  Pay     chan int   //  get money
  Coffee chan int   //  coffee or tea request
  Tea     chan int
}
```

```
// CTM behaviour
func (self *CTM) Run () {
  var choice string
  for {
    <- self.Pay      // get the money
    select {        // wait for the choice
    case <- self.Tea:
      choice = "Tea"
    case <- self.Coffee:
      choice = "Coffee"
    }
  }}
```

```
// Computer scientist
func CS (machine *CTM) {

  for {
    // internal behaviour
    ...
    machine.Pay <- 1             // pay

    // randomly choose coffee or tea
    if (rand.Intn(2) == 0) {
        machine.Coffee <- 1
      } else {
        machine.Tea <- 1
      }

}
```

Using select, and more

- The "Office example"
  - multiple CSs



Test for communicability

- Idiom for non-blocking receive

```
select {
    case v := <-ch:
      fmt.Println("received", v)

    default:
      fmt.Println("ch not ready for receive")
}
```

Timeouts

- Try a communication with a timeout

```
select {

case v := <-ch:
    fmt.Println("received", v)

case <-time.After(30*time.Millisecond):
    fmt.Println("timed out after 30 ms")
}
```

Go Concurrency summary

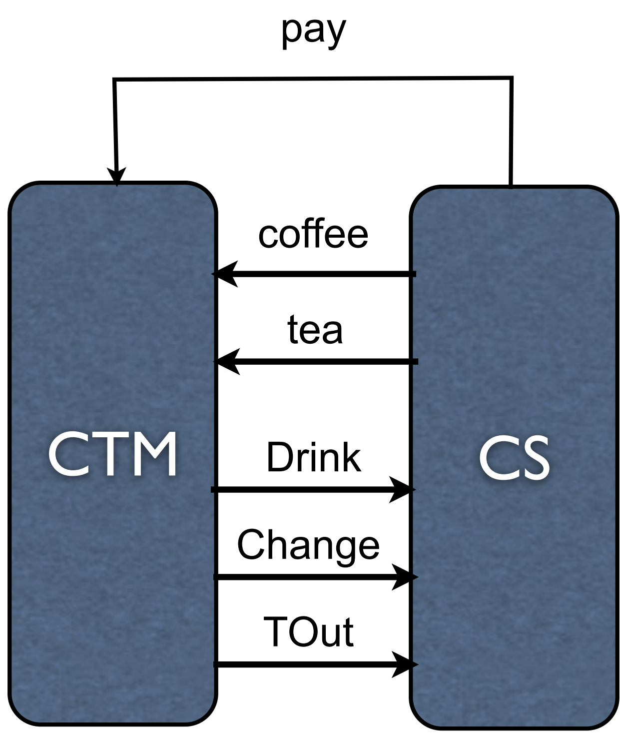
- Processes (Goroutines) & Channel-based interaction
- Buffering (for asynchronous communication)
- Select for communication-centric choices
- Tests for communicability and Timeout

The "office" example, revised

- CS makes a publication & gets some money
- CS tries to buy something
- Puts money, waits before choosing so that the CTM could time out and gets the money back
- if money not sufficient, gets all the money back

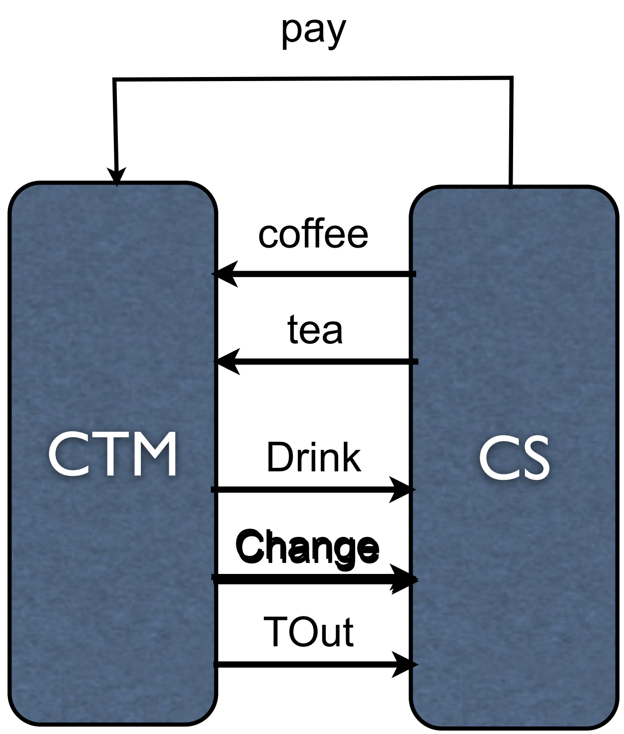
Using select, and more

- The "Office example"

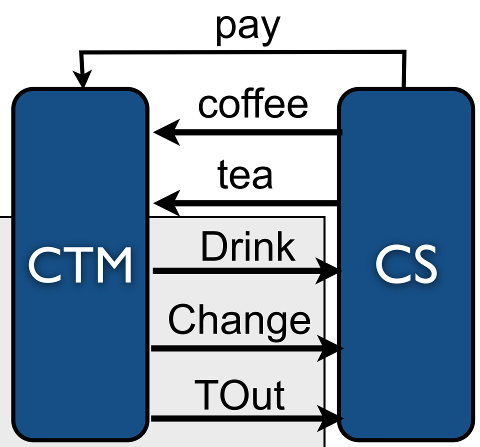


Using select, and more

- The "Office example"



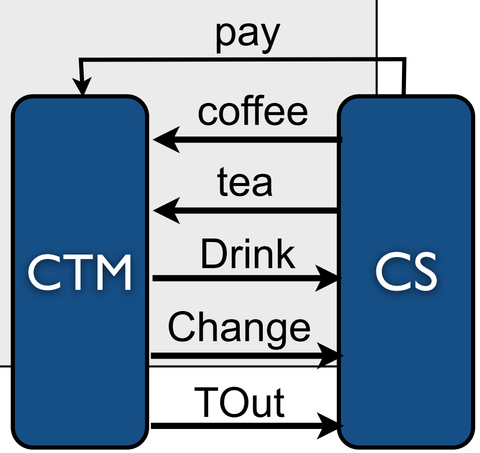
CTM




```
for {
    money = <- self.Pay  // get the money
    select {             // wait for the choice
    case <- self.Tea:    // (with timeout)
      choice = "Tea"
    case <- self.Coffee:
      choice = "Coffee"
    case <- time.After(5 * time.Second):
      choice = "nothing"
      self.TOut <- 1   // if timed out, tell the user
    }
    if choice != "nothing" {       // if chosen
          if Price[choice] <= money {      // money is enough
            self.Drink <- choice             // provide beverage
            money = money - Price[choice]
          } else {
            fmt.Printf("CTM: %d cents not enough\n", money)
          }
       }
    self.Change <- money  // give back the change
}
```

CS

```
for {
  gain := rand.Intn(5)       // publish and get some random money
  money = money + gain
  machine.Pay <- money       // pay: put all money in the machine
  time.Sleep(time.Duration(rand.Intn(5))*time.Second) // wait random
  select {          // randomly choose coffee or tea (or rec. timed-out)
  case machine.Tea <- 1:          // coffee
  case machine.Coffee <- 1:       // tea
  case <- machine.TOut:           // timeout
  }
  select {         // can get the beverage or only the money back
  case beverage = <-machine.Drink:
    money = <- machine.Change

  case money = <- machine.Change:
  }
  fmt.Printf("CS change back %d \n", money)
}}
```



Does it work?

- Deadlock?
- More properties?
  - If the CS asks for beverage he will eventually get it
  - If the CS asks for beverage he will eventually get it or get the money back

Let's build a model

```
set CHANS = {pay, coffee, tea, tout, drink, change};
```

```
MACHINE
CTM    = pay. (tea.SERVE + coffee.SERVE
                 + tau.'tout.CHANGE);
SERVE  = tau. 'drink. CHANGE + tau. CHANGE;
CHANGE = 'change. CTM;
```

```
COMPUTER SCIENTIST
CS      = 'publish. 'pay. CHOOSE;
CHOOSE  = 'coffee. GET + 'tea. GET + tout. GET;
GET     = drink. change. CS + change. CS;
```

```
SYS = (CTM | CS) \ CHANS;
```

Are we done?

- I can use the channel-based communication of CCS, PI and the like ...
- This is the paradigmatic communication style suggested by the designers
- Are all my old shared-memory concurrency style problems solved?

Concurrency + shared objs = danger!

- (see sharePointers.go)

Example

```go
type MyObj struct {
   X       int //  a value
   DoubleX int //  its double
}
```

```go
func New(X int) *MyObj {
   return &MyObj{X, 2 * X}
}
```

```go
func (self *MyObj) Inc() {
   self.X++

   //clearly, not the best way to do it
   self.DoubleX = self.X

   self.DoubleX = 2 * self.DoubleX
   fmt.Println("object: ", self)
}
```

Example (cont.)

```
func child (comm chan *MyObj, end chan bool) {
    val := <-comm
    val.Inc()
    end <- true
}
```

```
func main() {
    obj := New(2)
    fmt.Println("object:", obj)

    c := make(chan *MyObj)  // channel for sending objects

    wait := make(chan bool) // chan for waiting end of child

    go child(c, wait) // adhere to chan-based communication

    c <- obj          // send the child a ref to obj on c

    obj.Inc()         // act on the object

    <-wait            // wait for child termination
```

Things can go wrong!

```
Inc() {
   X++

   DoubleX = X

   DoubleX = 2 * DoubleX
}
```

```
Inc() {
   X++

   DoubleX = X

   DoubleX = 2 * DoubleX
}
```

```
X++
X++
DoubleX = X
DoubleX = X
DoubleX = 2 * DoubleX
DoubleX = 2 * DoubleX
```

X	DoubleX
2	4
3	4
4	4
4	4
4	4
4	8
4	16

Synchronising?

- Use locks

```
import "sync"

var mu sync.Mutex

fun (self *MyObj) Inc() {
    mu.Lock()
    ...
    mu.Unlock()
}
```

- Reentrant/Recursive?
- Just avoid them ... and stick to channel style!

Change perspective

- "Active objects" manage their internal state, share only by communicating

```go
type MyObj struct {
   X         int        //  a value
   DoubleX   int        //  its double
   IncReq    chan bool  // inc requests
}
```

```go
// increment no longer public
func (self *MyObj) inc () {
   self.X++
   self.DoubleX=2*self.X
   fmt.Println("object:", self)
}
```

Change perspective

- Expose `inc` as a "service"

```
func New (X int) *MyObj {
   IncReq:=make(chan bool)
   obj:=MyObj{X, 2*X,IncReq}

   go func () {
      for {
         <-IncReq
         obj.inc()
      }
   }()
   return &obj
}
```

- Parallel inc's are suitably serialised by obj

Channels of channels

- Channels can be sent over channels (in pi-calculus style)

```
var chanOfChan = make(chan chan int)
```

- More generally structs (including channels) can be sent.

Server

- The client supplies, with its request, the channel to which reply (or the server can provide a channel where the client will have a reply)

```
type Request struct {
    arg1, arg2  someType
    replyc      chan ReplyType
}
```

- See: server.go

Request

```go
// client request
type Request struct {
   a,b int                    // operands
   replyChan chan int         // channel for answer
}
```

```go
// waits for the answer to the request and returns it
func (r *Request) GetResult() int {
   return <-r.replyChan
}
```

```go
// get the parameters of the request
func (r *Request) GetPars() (int, int) {
   return r.a, r.b
}

// function for sending the result on the reply channel
func (r *Request) SendRes(val int) {
   r.replyChan <- val
}
```

Server

```
// the type of a possible operator to apply
type binOp func(a, b int) int

// server: parametrised by op to exec and input port for reqs
func server(op binOp, service <-chan *Request) {
   for {
      req := <-service      // requests arrive here
      go serve(op, req)     // don't wait for op
   }
}
```

```
// serving the request
func serve(op binOp, req *Request) {
   req.SendRes(op(req.GetPars()))
}
```

```
// starting the server: returns a port for sending reqs
func StartServer(op binOp) chan<- *Request {
   service := make(chan *Request)
   go server(op, service)
   return service
}
```

Server: Start & Client

```go
// server instance, performing additions
sum := func(a, b int) int { return a + b }
serverChanSum  := StartServer(sum)

// same for multiplication
prod :=  func(a, b int) int { return a * b }
serverChanProd := StartServer(prod)
```

```go
// client side
reqS := &Request{rand.Intn(100), rand.Intn(100), make(chan int)}
reqP := &Request{rand.Intn(100), rand.Intn(100), make(chan int)}

// send the requests
serverChanSum  <- reqS
serverChanProd <- reqP
```

```go
// Can retrieve results in any order
fmt.Printf("Request 2: Operands: %d %d = %d\n",
            reqP.GetPars(),      reqP.GetResult())

fmt.Printf("Request 1: Operands  %d %d = %d\n",
            reqS.GetPars(),      reqS.GetResult())
```

Functions over chans

- Functions are first class values and can be sent over channels
- Actually, what is sent is a closure

- See: chanFun.go

Ranging over channels

- A for loop can range over the values received on a channel (until closed)

```
for v := range ch { fmt.Println(v) }
```

Ranging over channels

- The sender closes the channel with

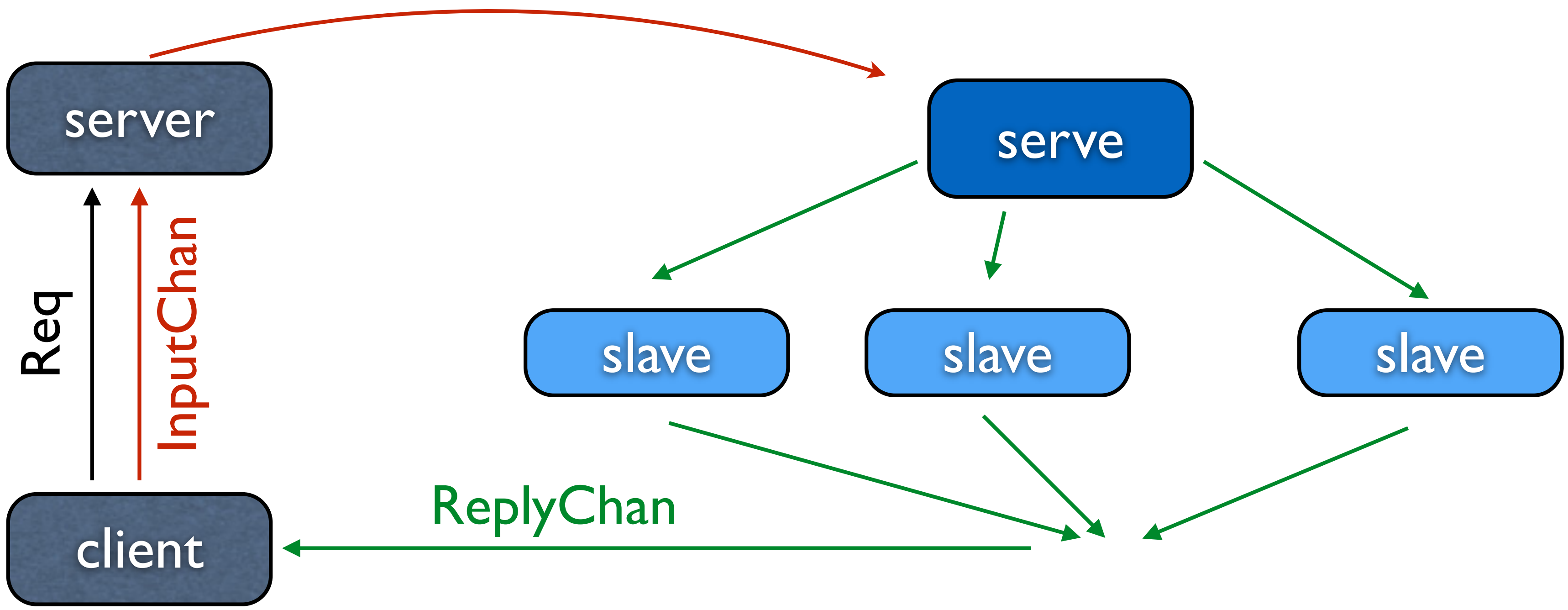
```
close(ch)
```

- The receiver can test if channel is closed with comma ok

```
val, ok := <-ch
        // result is either (value, true)
        //                        or (0, false)
```

Example: Task splitting

- Process a set of elements (parallelize.go)



Code (server)

```
type Operator func (int) int

type Request struct {
   Op        Operator  // operator to be applied
   InputChan chan int  // chan to input data to be processed
   ReplyChan chan int  // chan for returning processed data
}
```

```
func server(service <-chan *Request) {
   for {
      req := <-service     // requests arrive here
      go serve(req)        // served by a child goroutine
   }
}
```

Code (server)

```go
func serve(req *Request) {

   // channel for waiting slaves termination
   done := make(chan bool, BUF_SIZE)

   numSlaves := 0
   for val := range req.InputChan {
      // concurrent or non-concurrent?
      go slave(req.Op, val, req.ReplyChan, done)
      numSlaves++
   }

   // waiting for slaves completion
   for (numSlaves > 0) {
      <-done
      numSlaves--
   }

   // close replyChan (client is notified)
   close(req.ReplyChan)
}
```

Code (server)

```go
func slave(op Operator, val int,
           reply chan<- int, done chan<- bool) {

   reply <- op(val)
   done <- true
}
```

```go
// creates a server instance and returns the channel
// where reqs are accepted
func startServer() chan<- *Request {
   service := make(chan *Request)
   go server(service)
   return service
}
```

Code (client)

```
// client side
// create the request
req := &Request{
   useIO/useCPU,                    // task to be executed
   make(chan int, BUF_SIZE),   // inputChan
   make(chan int, BUF_SIZE)}   // replyChan


// send the requests to the server
serverChan <- req

// send data on the input channel
for val := range some_data {
   req.InputChan <- val
}
close(req.InputChan)

// retrieve the results ranging on the reply channel
for v := range req.ReplyChan {
   do_something_with(v)

}
```

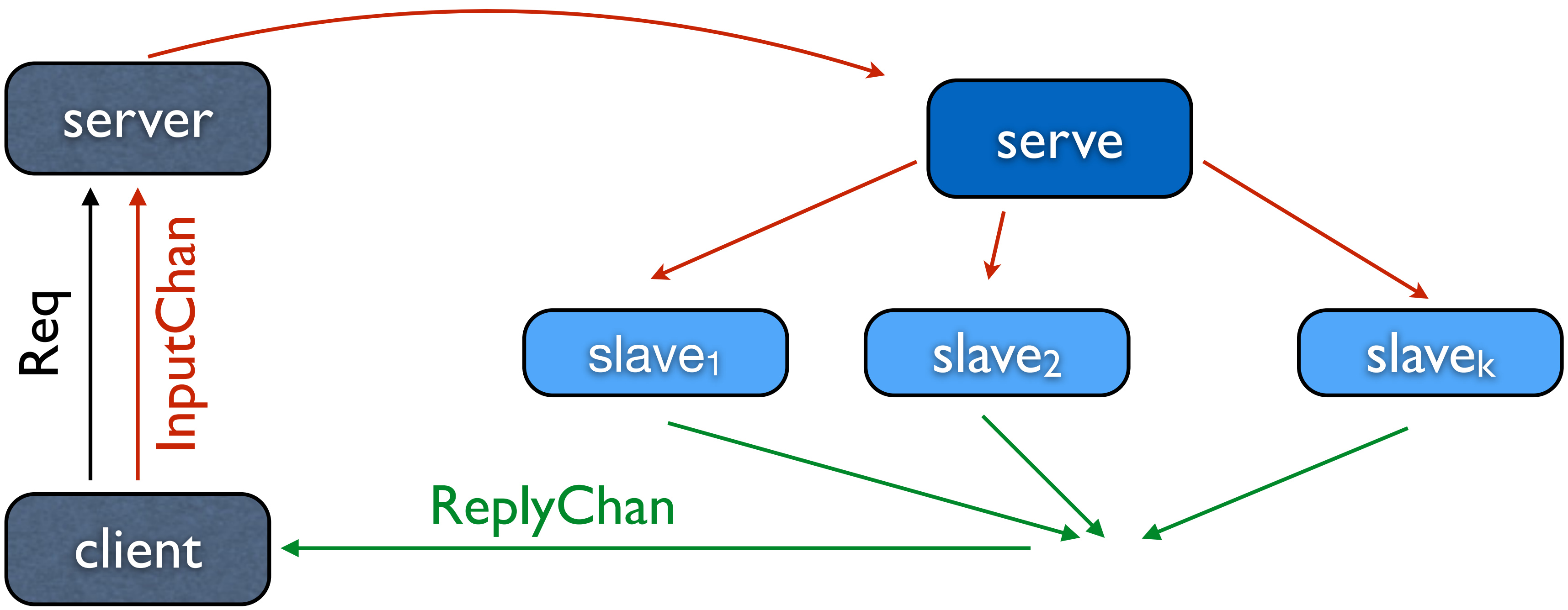
Code (tasks)

```go
// do some lengthy, not CPU intensive work: IO over network
func useIO (n int) int {
   ...
   resp, _ := http.Get(url)    // get request

   defer resp.Body.Close()

   // read html as a slice of bytes (ignore a possible error)
   html, _ := ioutil.ReadAll(resp.Body)
   return len(html)
}
```

```go
// some CPU intensive task (theta(n^2))
func useCPU (n int) int {
   for i:=0; i< n; i++ {
      for j:=0; j< n; j++ {
         ... i*j ...
      }
   }
   return ...
}
```

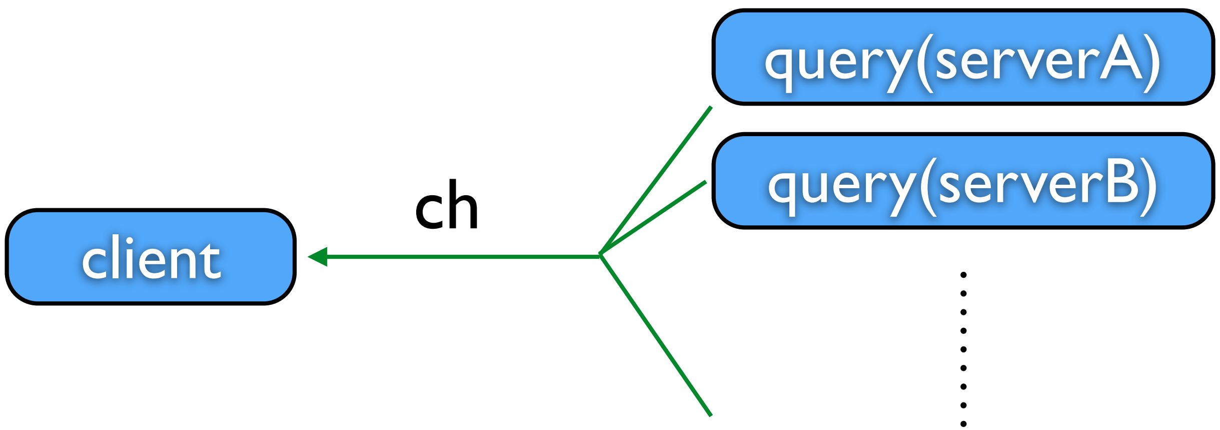
Exercise

- Change `serve` in a way that it creates a fixed number of slaves taking elements directly from InputChan



Replication Idiom

- Replicating a single task on different servers
- Take as a result the first that answers
- Disregard the others



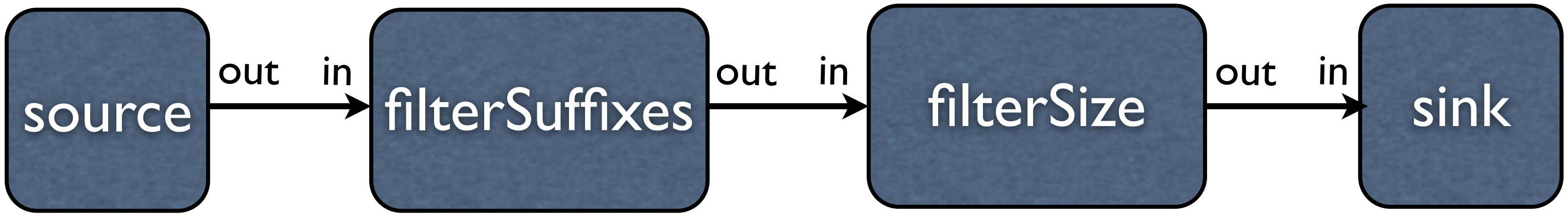
Replication

```
// conns: slice of connections, query: query to execute
func Query(conns []Conn, query string) Result {
    ch := make(chan Result, 1)
    // try in parallel the query on any connection c,
    // result on channel ch
    for _, c := range conns { go execQuery(c, ch) }
    return <- ch
}
```

```
// the first query which succeed will put the result on the channel
// no race as the channel is buffered (capacity 1)
func execQuery(c Conn, ch <-chan Result) {
    select {
      case ch <- c.DoQuery(query):
      default:
     }
}
```

Filter (pipeline)

- Pipelining



- Asynchronous (buffered) communication for decoupling processes

```
source: data -> chan
sink  : chan -> data
```

```
filterSuffixes : chan, params -> chan
filterSize : chan, params -> chan
```

`filter.go`

Code (source)

```go
func source(files []string) <-chan string {
   out := make(chan string, 1000)
   go func() {
      for _, filename := range files {
         out <- filename
      }
      close(out)
   }()
   return out
}
```

Code (filter)

```
func filterSuffixes(suffixes map[string]bool, in <-chan string) <-
chan string {
   // output channel (same size as input to maximize throughput)
   out := make(chan string, cap(in))
   go func() {
      for filename := range in {
         if <filename has suffix in suffixes> {
            out <- filename
         }
      }
      close(out)
   }()
   return out
}
```

Code (main)

```
func main() {
   ....

   // chaining
   sink(
      filterSize(minSize, maxSize,
         filterSuffixes(suffixes,
            source(files))))
}
```

Go Concurrency Summary

- Processes & Channel-based interaction
- Select for communication-centric choices
- Idioms (active channel, source, server, task splitting, replication, pipeline, …)

Peterson

- Shared memory: variables k, b_1, b_2

```
Process Pi
while true do
begin
  bi = true
  k  = j
  while (bj and k=j) do skip
  enteri
    critical section
  exiti
  bi = false
end
```

```
func proc (i int, b []bool, k *int) {
   for {
      j = (i+1)%2   /* other process */
      b[i] = true
      *k = j
      for b[j] && (*k == j) {}
      fmt.Printf("Process %d entered\n", i)
      /* critical section */
      ...
      fmt.Printf("Process %d exited\n", i)
      b[i] = false
   }
}
```

Atomic?

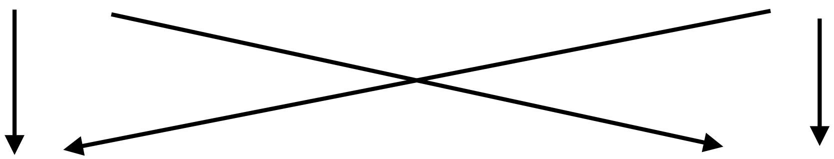
- From FAQS: What operations are atomic?
  - See the package **atomic**
  - other than that, concurrent access to shared memory should be serialised

Memory model

- Overall effect of concurrent goroutines corresponds to some interleaving (**sequential consistency**)?

```
Inc() {
    X++
    DoubleX = X
    DoubleX = 2 * DoubleX
}
```

```
Inc() {
    X++
    DoubleX = X
    DoubleX = 2 * DoubleX
}
```



```
X++
X++
DoubleX = X
DoubleX = X
DoubleX = 2 * DoubleX
DoubleX = 2 * DoubleX
```

```
X++
DoubleX = X
X++
DoubleX = X
DoubleX = 2 * DoubleX
DoubleX = 2 * DoubleX
```

(Weak) Memory model

- R/W can be "reordered" in the same goroutine according to **happens-before** order
- Roughly, reordering in a goroutine is possible if has no "local effect"
- E.g., `a=1;b=2 -> b=2;a=1`

Be careful ...

```go
// not working, in general
var a string = "uninitialised"
var done bool

func setup() {
   a = "hello, world"
   done = true
}

func main() {
   go setup()
   for !done {
   }
   fmt.Printf("a = %s\n", a)
}
```

→

```go
func setup() {
   done = true
   a = "hello, world"
}
```

Suggestion …

Do not mess up with this …

Share memory by communicating!

This works!

```
var a string = "uninitialised"

func setup(done chan bool) {
   a = "hello, world"
   done <- true
}

func main() {
   done := make (chan bool)
   go setup(done)
   <- done
   fmt.Printf("a = %s\n", a)
}
```

Peterson with Channels!

```
Bi(x) = ‘bir(x). Bi(x) + b1w(y). Bi(y)

Pi  = 'biw(t). 'kw(j). Pi1
Pi1 = bjr(f). Pi2 + bjr(t). (kr(i).Pi2 + kr(j). Pi1)
Pi2 = enteri. exiti. 'biw(f). Pi;

Sys = ( P1 | P2 | B1 | B2 | K ) \ L;
```

```
Process Pi
while true do
begin
  bi = true
  k  = j
  while (bj and k=j) do skip
  enteri
    ...
  exiti
  bi = false
end
```

- No busy waiting?

```
Bi(x) = ‘bir(x). Bi(x) + b1w(y). Bi(y)

Pi  = 'biw(t). 'kw(j). Pi1
Pi1 = bjr(f). Pi2 + kr(i).Pi2
Pi2 = enteri. exiti. 'biw(f). Pi;

Sys = ( P1 | P2 | B1 | B2 | K ) \ L;
```

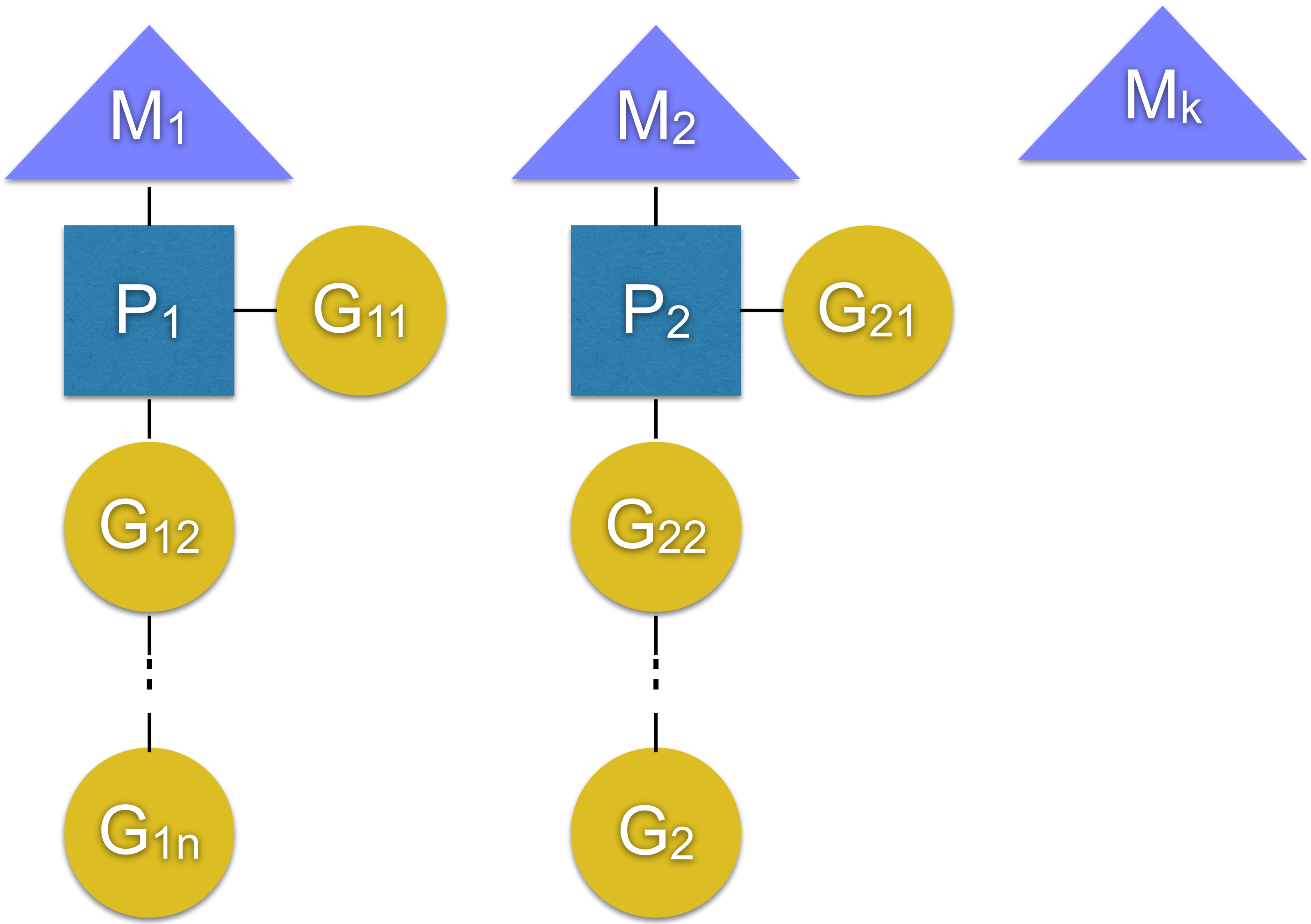
The scheduler

- Older compiler (gccgo) used pthreads, one per goroutine
- This is not the case now [judged too heavy - goroutines are lighter]

The scheduler

- Goroutines are multiplexed (as needed) into threads
- Model:
  - some threads M
  - contexts/runqueues P
  - each containing some goroutines G

Idea



The scheduler

- A **runqueue** contains goroutines ready to be scheduled and is associated to a thread M
- Up to a fixed number GOMAXPROCS of contexts/runqueues (threads can be a bit more)
- Default is (related to the) number of cores

Context switch

- A context picks up a new goroutine whenever the current one:
  - finishes
  - makes a blocking Go runtime call (op on a channel)
  - invokes a function (sometimes, if not inlined … limited form of preemption)
  - makes a blocking system call (read file, net op)

Go losing control

- If an active goroutine G invokes a system call the queue is moved to a new thread;
- G is active, but out of the local scheduler control
- when it returns, it is put in some runqueue

What if we do not trust the scheduler?

- The scheduler can be influence by using the package **runtime**
  - runtime.Gosched()
    yields the thread
  - runtime.GOMAXPROCS(n)
    sets the max number of threads

Why should we go? Afterthoughts

- Modern well-designed imperative language in the C, C++ family
  - Statically typed, with type inference
  - Garbage collected
  - Access to low-level details (pointers), but reasonably clean
  - Embedding & interfaces
  - Initial support to generics

Criticisms

- No assertions
- Garbage collection is great but can it cause problems in contexts were efficiency is critical?

Why should we go?

- Designed with concurrency in mind
  - Clean and solidly grounded **channel-based message passing** paradigm
  - Part of the language (not a library)
  - Functional values + goroutines gives a nice combination

Why should we go?

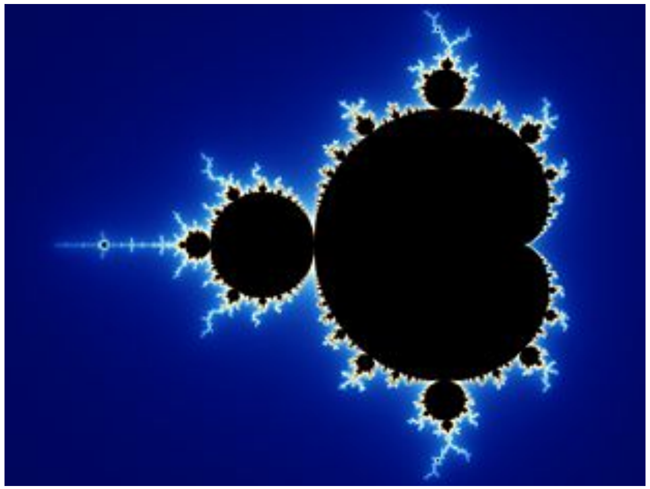
- Good for exploiting **local parallelism**
- with concurrency used as a structuring principle … explicitly (it must be programmed, but often easy)

Example: Mandelbrot

- Set of complex values c for which the **orbit** of 0 under iteration of the map

$$f_c(z) = z^2 + c$$

is bounded



Example: Mandelbrot

- Define the sequence

$$f_c^0 = 0 \quad f_c^1 = f_c(0) \quad f_c^2 = f_c(f_c(0)) \quad \ldots$$

- Then

$$c \in M \quad \Leftrightarrow \quad \sup\{\, |f_c^n| \mid n \in \mathbb{N}\} < \infty$$

- It can be shown

$$c \in M \quad \Leftrightarrow \quad \sup\{\, |f_c^n| \mid n \in \mathbb{N}\} \leq 2$$

Example: Mandelbrot

- Idea: Color points not in M according to which term in the sequence is greater than a certain cutoff value (typically 2)

Check if point in M

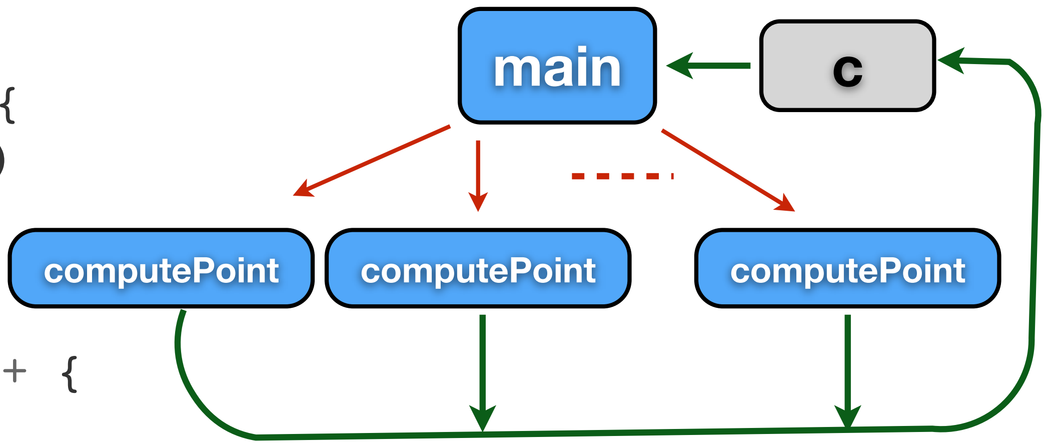
- Simplified

```
func check(z complex128, maxIters int) int {
    var iters int = 0
    var zn complex128 = complex(0,0)

    var len float64 = 0
    for ((len <= 2) && (iters < maxIters)) {
        zn = zn*zn + z
        len = cmplx.Abs(zn)
        iters++
    }

    // the divergence speed is measured as the number of
    // iterations needed to exceed 2
    return iters
}
```

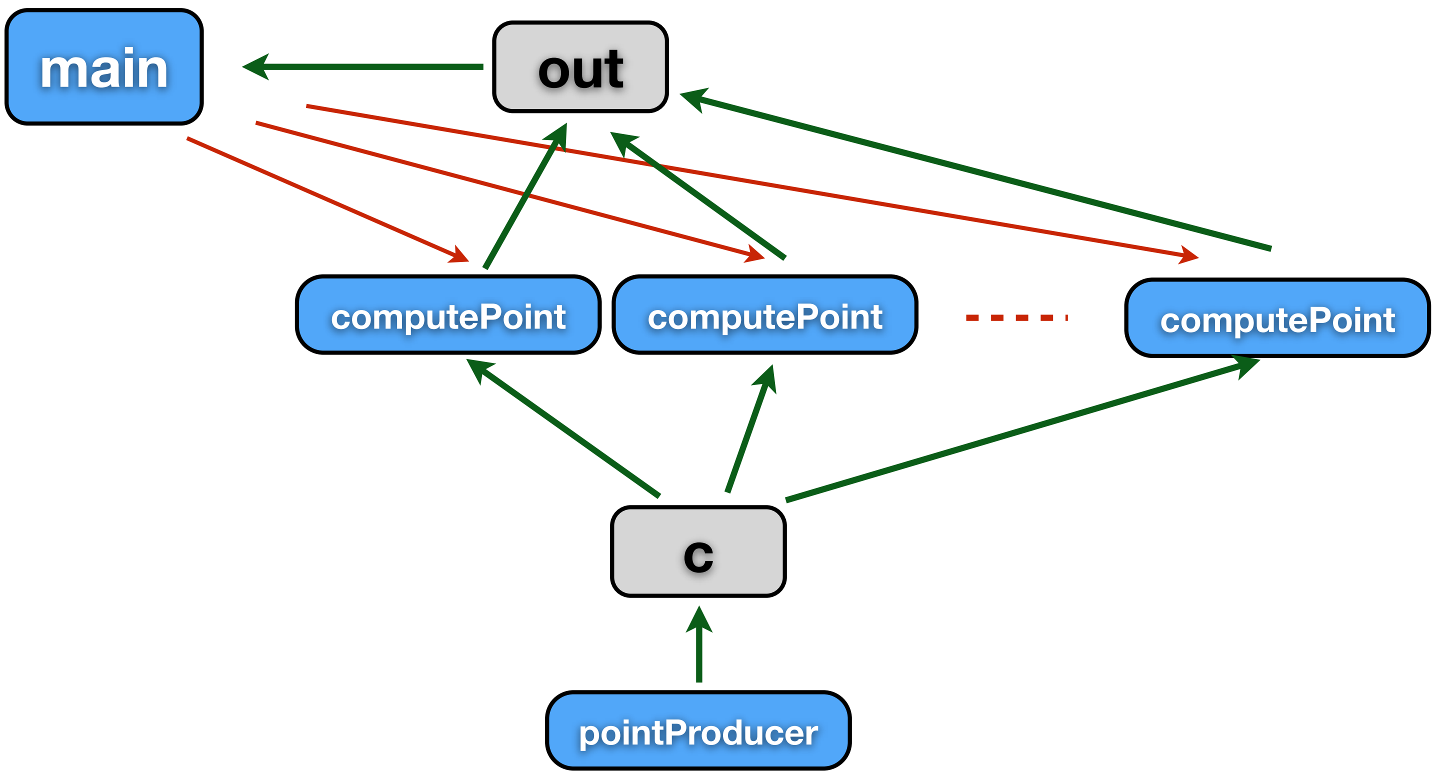
Compute Mandelbrot

- A goroutine per point

```
// imgW and imgH width and height in pixels of the image

// channel where colored points are inserted once produced
c := make(chan *mandelpkg.ColoredPoint, imgW*imgH)

// for each pixel, elaborate the corresponding color,
// result in channel c
for i := 0; i < imgW; i++ {
   for j := 0; j < imgH; j++ {
      go computePoint(i, j, c)
   }
}

for i:=0; i < imgW * imgH; i++ {
   point := <-c
   image.Set(point.GetComponents())
}
```



Fixed number of workers



Compute Mandelbrot/2

- Fixed number of workers

```
// channel for communication producer->consumers
c := make(chan *mandelpkg.ColoredPoint, imgW*imgH)
// channel for outputs
out := make(chan *mandelpkg.ColoredPoint, imgW*imgH)

numWorkers := 8
for i := 0; i < numWorkers; i++ {
   go computePoint(c, out)
}

// fill the channels with uncolored points
go pointProducer(imgW, imgH, c)


for i:=0; i < imgW * imgH; i++ {
   point := <-out
   mandelImg.Set(point.GetComponents())
}
```

Why should we go?

- Goroutine model is suggested but not forced, sharing is possible, structures (arrays, maps, structs, …) can be shared and ops are not safe.
- Distribution and fault tolerance?