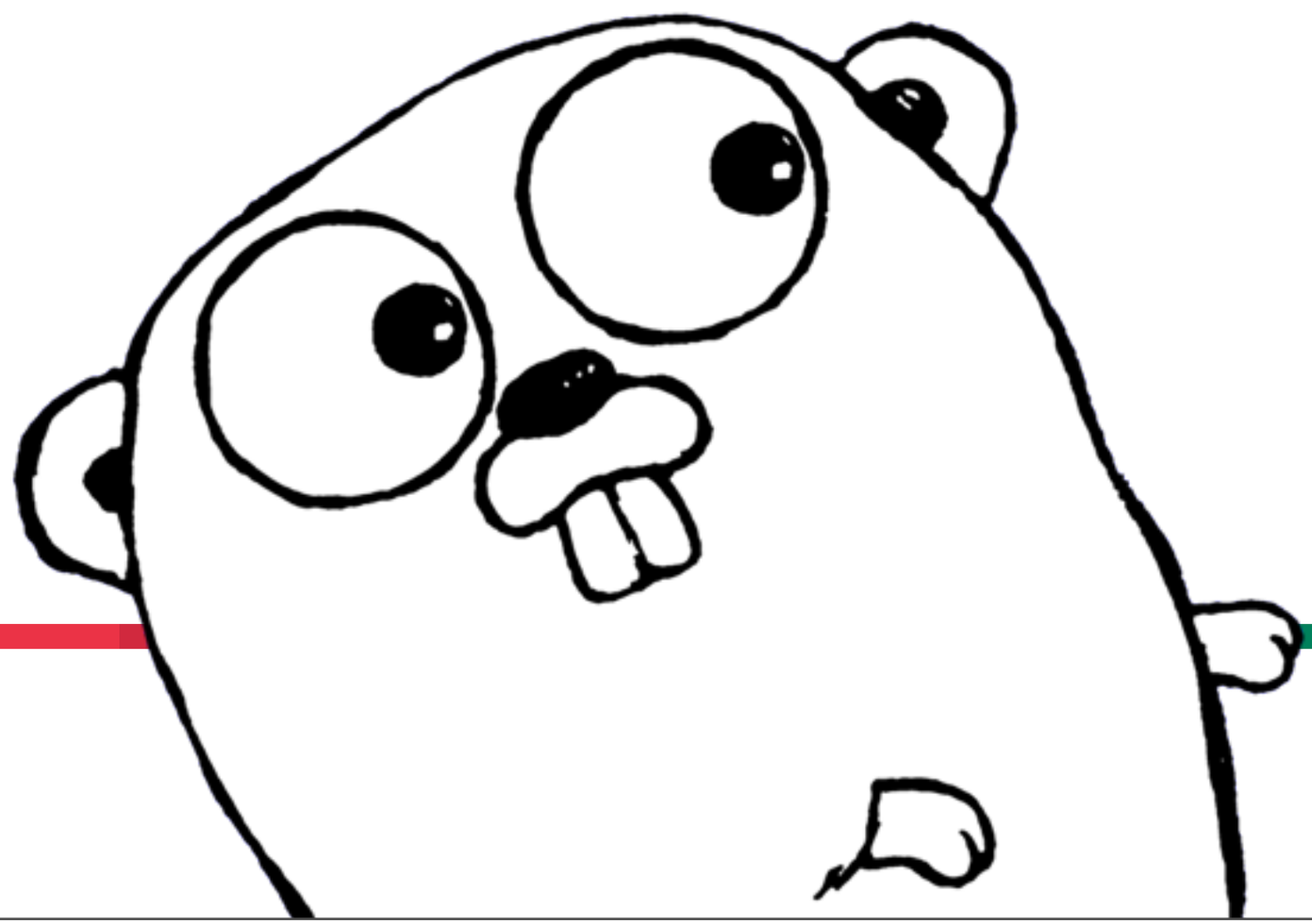
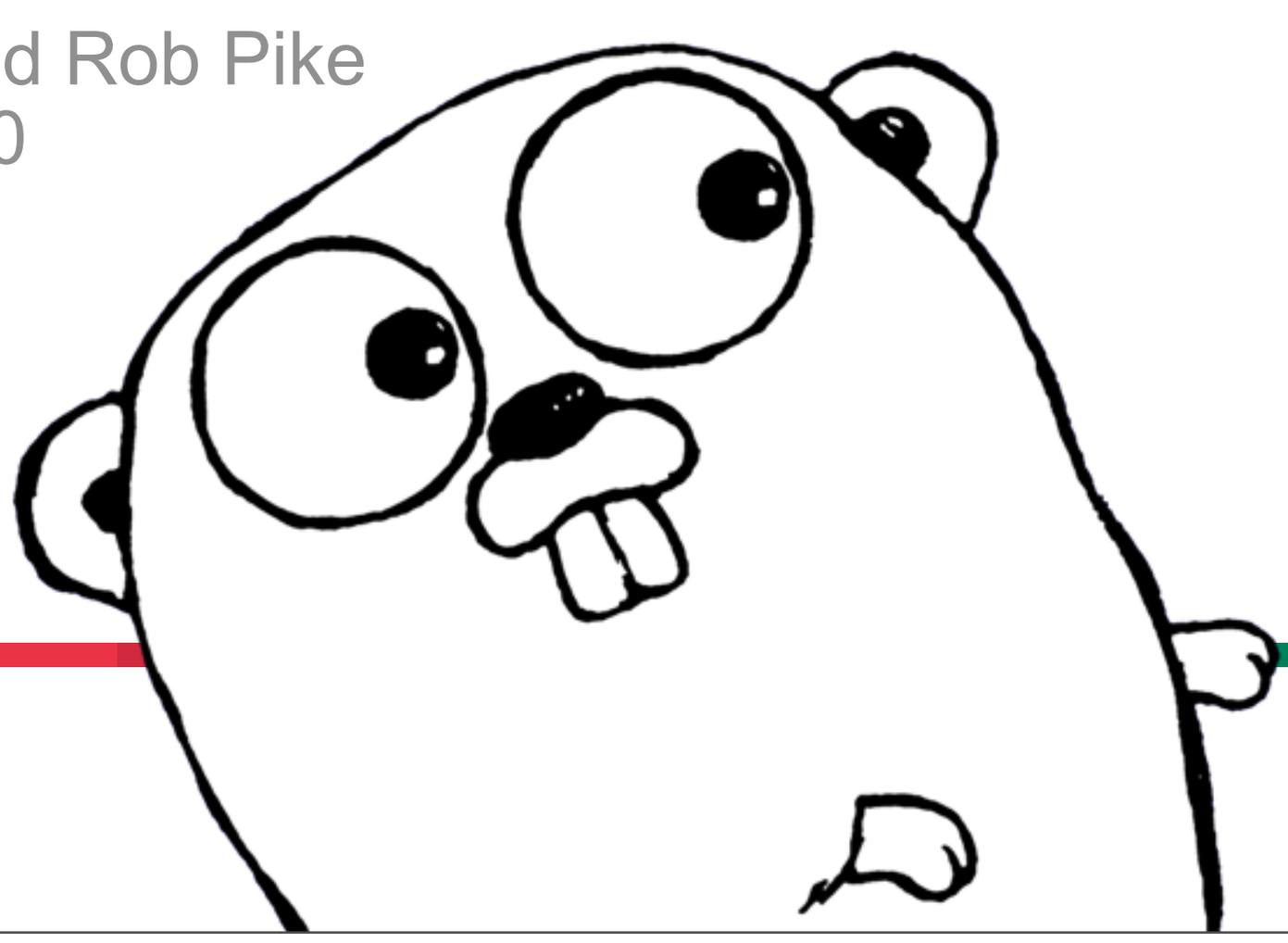
Google™ 10 I/O



Google I/O 10

Go Programming

Russ Cox and Rob Pike
May 20, 2010

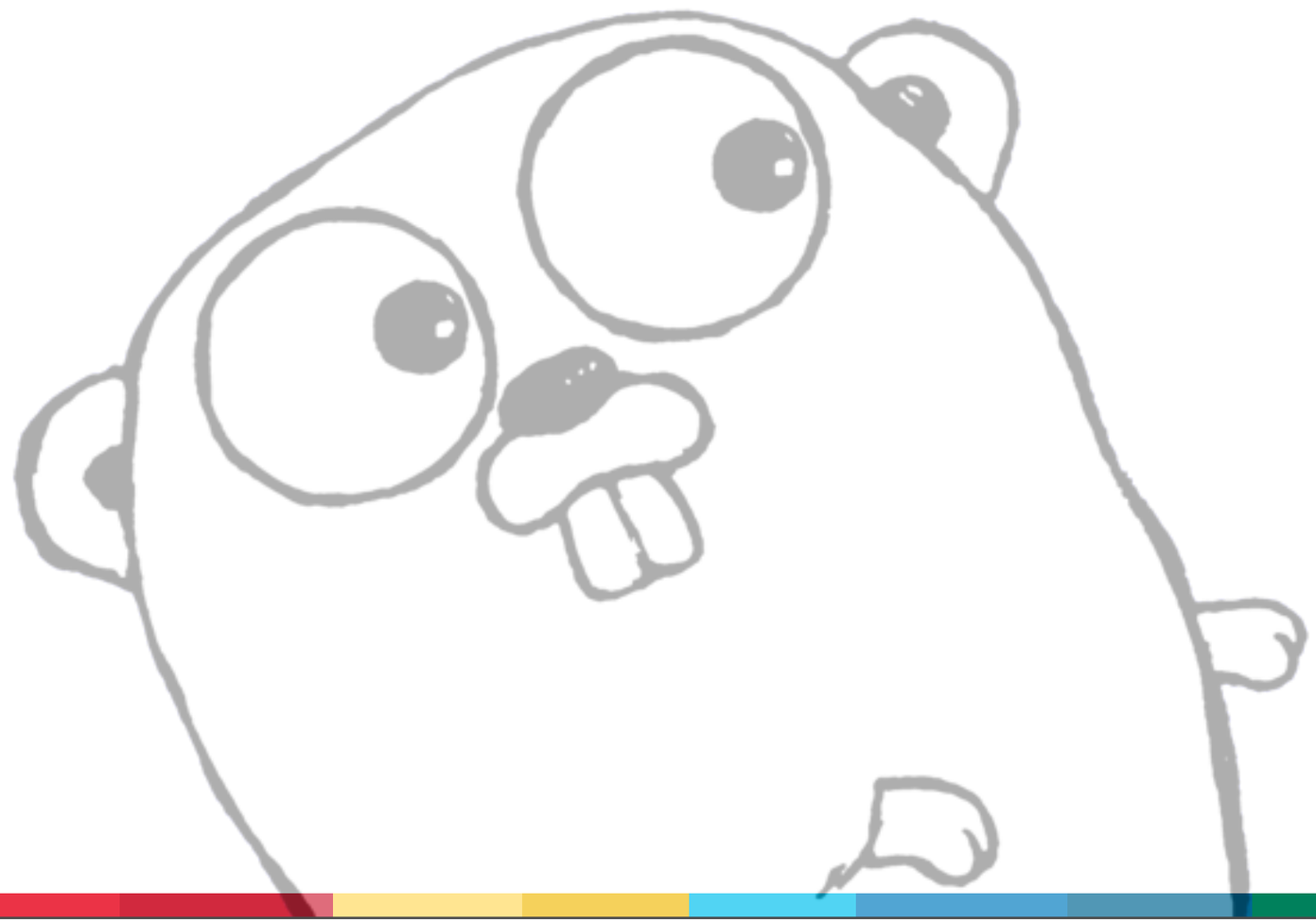


Google I/O 10

Live waving

View live notes and ask questions about this session on Google Wave:

`http://bit.ly/go2010io`



Go is different

Go is more unusual than you might first think.

Programming in Go is different from programming in most procedural languages.

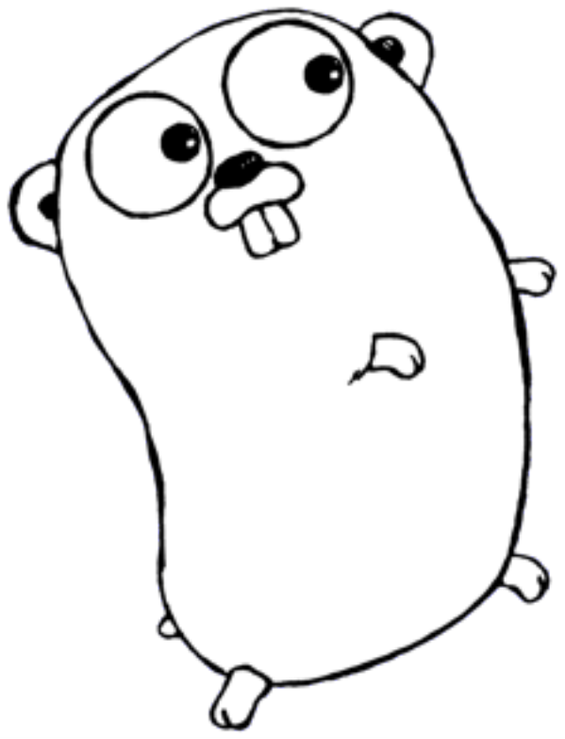
If you try to write Java* programs in Go, you may become frustrated. If you write Go programs in Go, you will be much more productive.

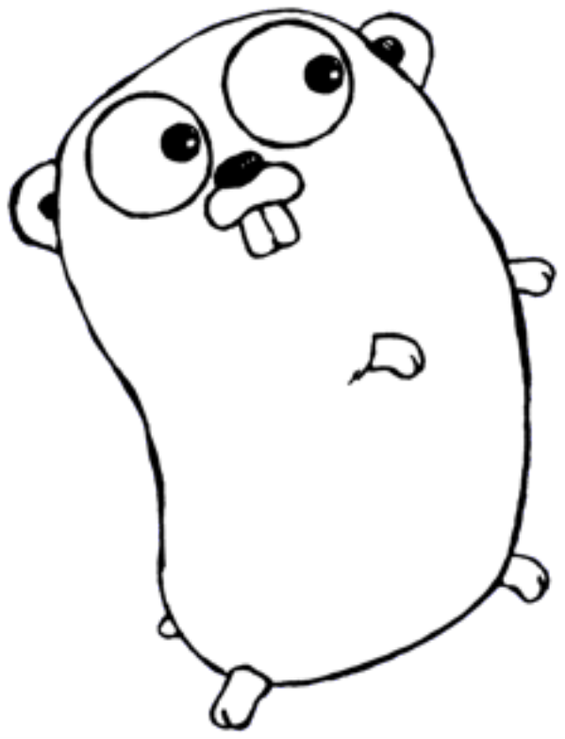
Our goal today is to teach you to think like a Go programmer.

*Sometimes we'll use Java as a reference for comparison but we could make the same points comparing Go to a number of other languages.

Go is and Go is not

- Go is object-oriented not type-oriented
  - inheritance is not primary
  - methods on any type, but no classes or subclasses
- Go is (mostly) implicit not explicit
  - types are inferred not declared
  - objects have interfaces but they are derived, not specified
- Go is concurrent not parallel
  - intended for program structure, not maximum performance
  - but still can keep all the cores humming nicely
  - ... and some programs are nicer even if not parallel at all





1: Evaluating expressions

Google I/O 10

A specification of behavior

Expression evaluators often define a type, called `Value`, as a parent class with integers, strings, etc. as child classes.

In Go, we just specify what a `Value` needs to do. For our simple example, that means: do binary operations and be printable.

```
type Value interface {
    BinaryOp(op string, y Value) Value
    String() string
}
```

Implement those methods and you have a type that the evaluator can use.

Integer values

Dead easy; just write the methods. (No extra bookkeeping.)

```
type Int int  // A basic type (not a pointer, struct, or class).
func (x Int) String() string { return strconv.Itoa(int(x)) }
func (x Int) BinaryOp(op string, y Value) Value {
    switch y := y.(type) {
    case Int:
        switch op {
        case "+":  return x + y
        case "-":  return x - y
        case "*":  return x * y
            ...
        }
    case Error:  // defined on the next slide
        return y
    }
    return Error(fmt.Sprintf("illegal op: '%v %s %v'", x, op, y))
}
```

Errors

For error handling, define an `Error` type that satisfies `Value` that will just propagate the error up the evaluation tree.

```
type Error string
func (e Error) String() string {
    return string(e)
}
func (e Error) BinaryOp(op string, y Value) Value {
    return e
}
```

No need for "implements" clauses or other annotations. `Ints` and `Errors` are `Values` because they satisfy the `Value` interface implicitly.

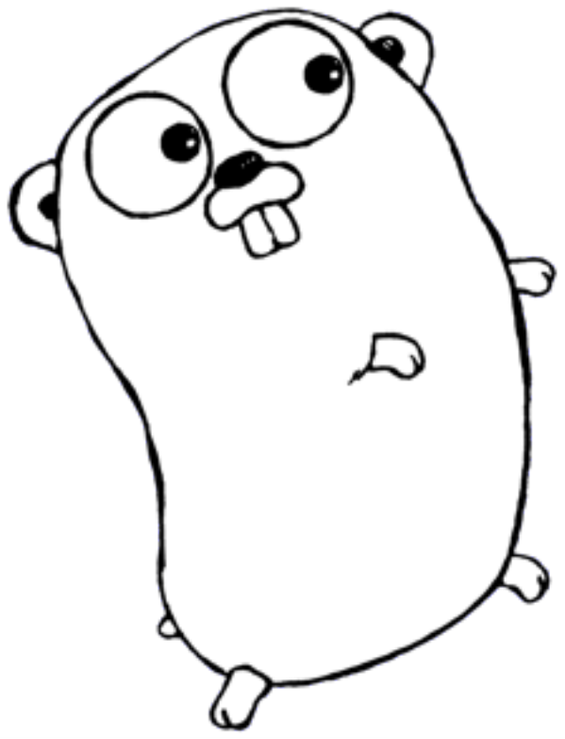
Input

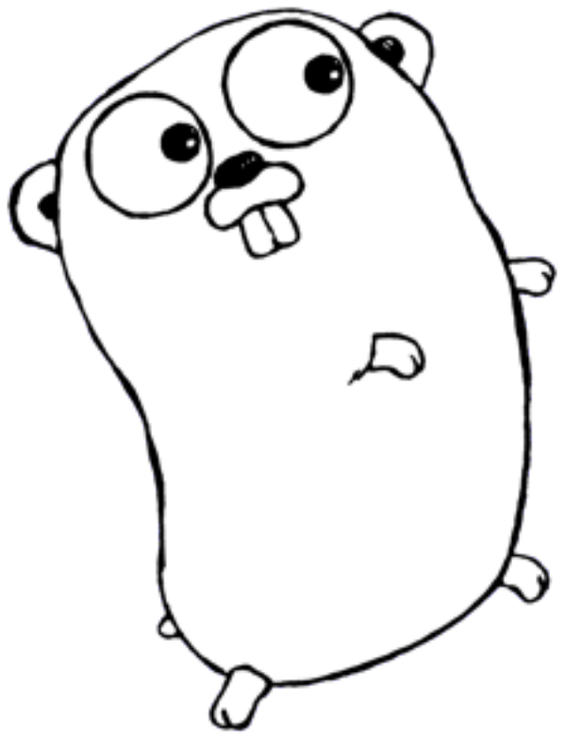
We need a basic scanner to input values. Here's a simple one that, given a string, returns a `Value` representing an integer or error.

```
func newVal(lit string) Value {
    x, err := strconv.Atoi(lit)
    if err == nil {
        return Int(x)
    }
    return Error(fmt.Sprintf("illegal literal '%s'", lit))
}
```

The evaluator just tokenizes, parses and, in effect, calls

```
fmt.Println(newVal("2").BinaryOp("+", newVal("4")).String())
```





Demo

Let's add strings

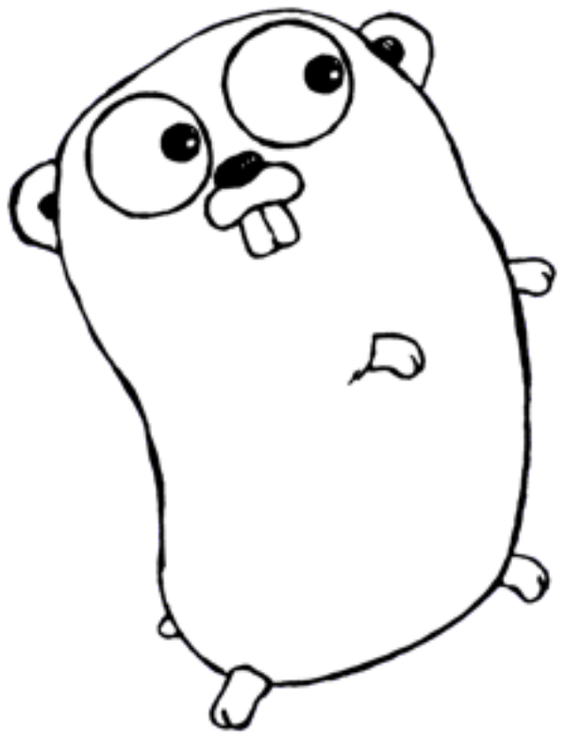
```
type String string
func (x String) String() string { return strconv.Quote(string(x)) }
func (x String) BinaryOp(op string, y Value) Value {
    switch y := y.(type) {
    case String:
        switch op {
        case "+": return x + y
            ...
        }
    case Int:  // String * Int
        switch op {
        case "*": return String(strings.Repeat(string(x), int(y)))
            ...
        }
    case Error:
        return y
    }
    return Error(fmt.Sprintf("illegal op: '%v %s %v'", x, op, y))
}
```

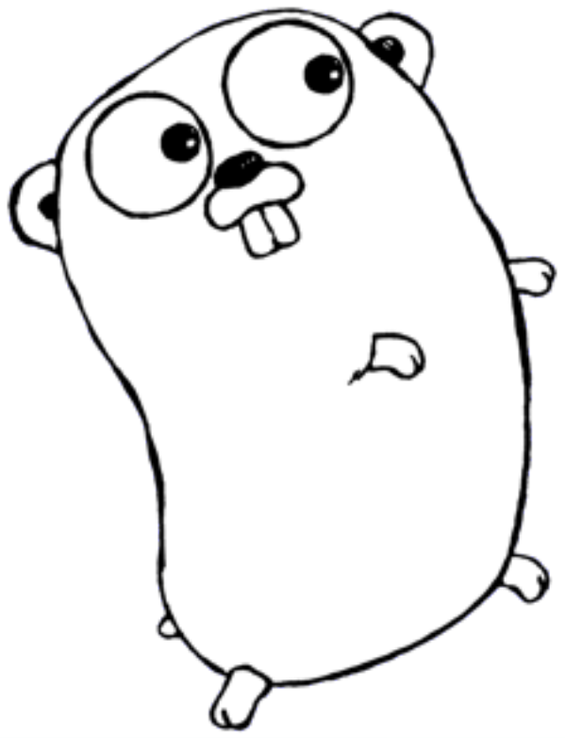
String input

Just a few more lines in `newVal`.

```
func newVal(lit string) Value {
    x, err := strconv.Atoi(lit)
    if err == nil {
        return Int(x)
    }
    s, err := strconv.Unquote(lit)
    if err == nil {
        return String(s)
    }
    return Error(fmt.Sprintf("illegal literal '%s'", lit))
}
```

We've added strings by just *adding strings*. This happens because of Go's implicit interface satisfaction. No retroactive bookkeeping.





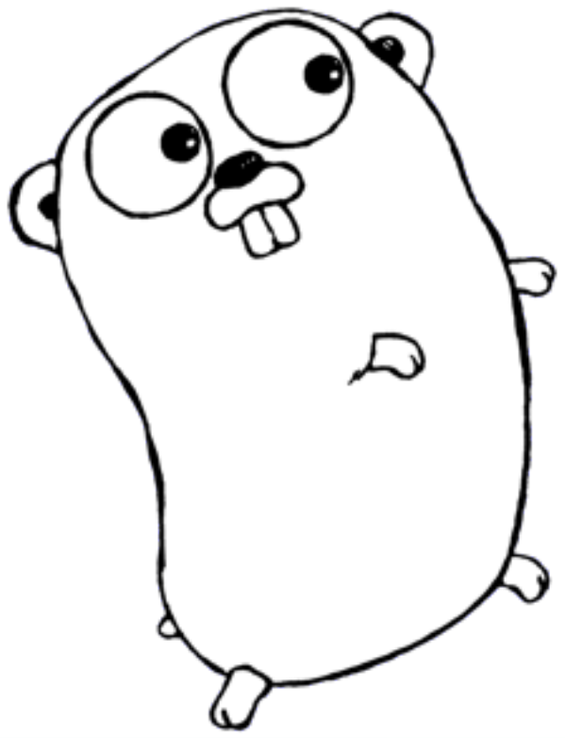
Demo

Objects but no hierarchy

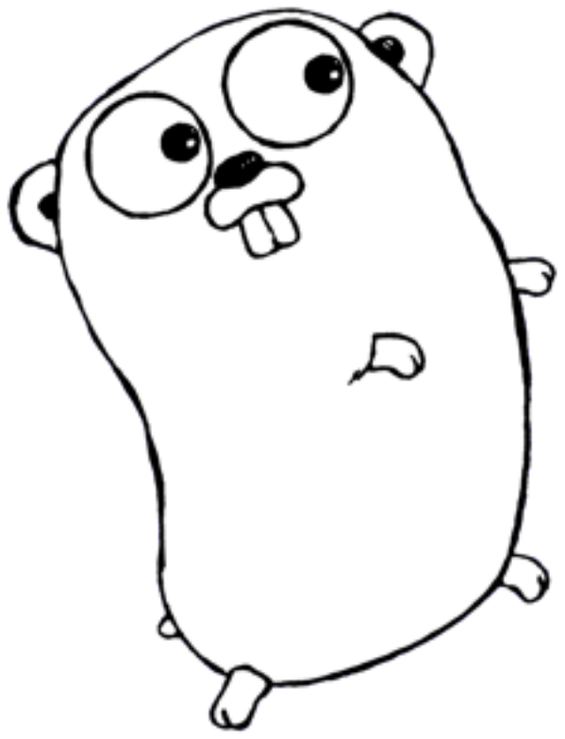
In Java, the type hierarchy is the foundation of the program, which can be hard to change as the design evolves. (Can be easier to compromise the design than change the foundation.)

Programming in Go is not primarily about types and inheritance. *There is no type hierarchy.* The most important design decisions don't have to be made first, and it's easy to change types as the program develops because the compiler infers their relationships *automatically*.

Go programs are therefore more flexible and adaptable.



2: Not inheritance



Google I/O 10

Java: Compressing using Buffers, given byte[] (1)

Suppose we have a zlib compressor:

```
public static class ZlibCompressor {

    public int compress(byte[] in, int inOffset, int inLength,
                        byte[] out, int outOffset) {
        ...
    }
    ...
}
```

and we want to support `Buffer` in a way that will generalize to other compressors.

Java: Compressing using Buffers, given byte[] (2)

Define an abstract compressor class.

```
public abstract class AbstractCompressor {
  /** Compresses the input into the output buffer. */
  abstract int compress(byte[] in, int inOffset, int inLength,
                        byte[] out, int outOffset);
  /**
   * Compresses byte buffers using abstract compress method.
   * Assumes Buffers are based on arrays.
   */
  public void compress(Buffer in, Buffer out) {
    int numWritten = compress(in.array(), in.arrayOffset() +
        in.position(), in.remaining(), out.array(),
        out.arrayOffset() + out.position());
    out.position(out.position() + numWritten);
  }
}
```

Java: Compressing using Buffers, given byte[] (3)

Subclass the abstract class to create a compression class.

```
public static class ZLibCompressor extends AbstractCompressor {

  public int compress(byte[] in, int inOffset, int inLength,
                      byte[] out, int outOffset) {
    ...
  }
}
```

This is common Java style: Inherit the abstract behavior.

Go: Compressing using Buffers, given []byte (1)

Again, we have a zlib compressor:

```
type ZlibCompressor struct { ... }

func (c *ZlibCompressor) Compress(in, out []byte) int
```

Again, we want to support `Buffer` in a way that will generalize to other compressors.

Go: Compressing using Buffers, given []byte (2)

Define an interface for the compressor and write a *function*.

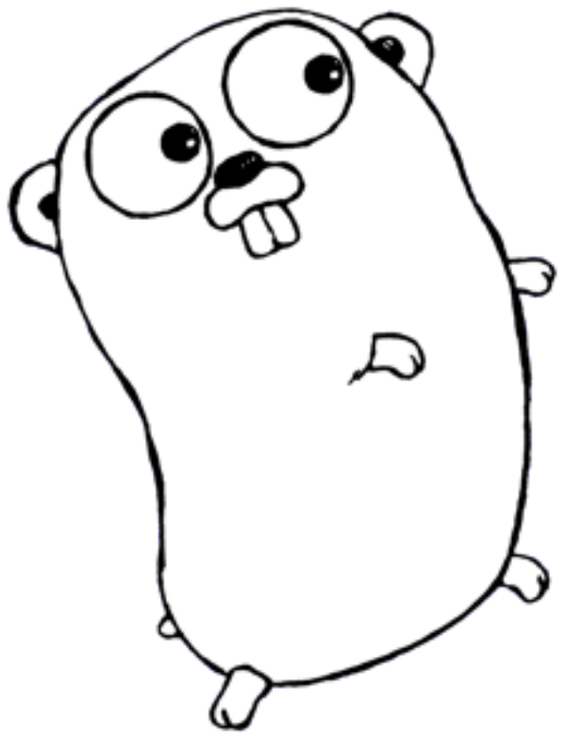
```
type Compressor interface {
    Compress(in, out []byte) int
}

func CompressBuffer(c Compressor, in, out *Buffer) {
    n := c.Compress(in.Bytes(), out.Bytes())
    out.Advance(n)
}
```

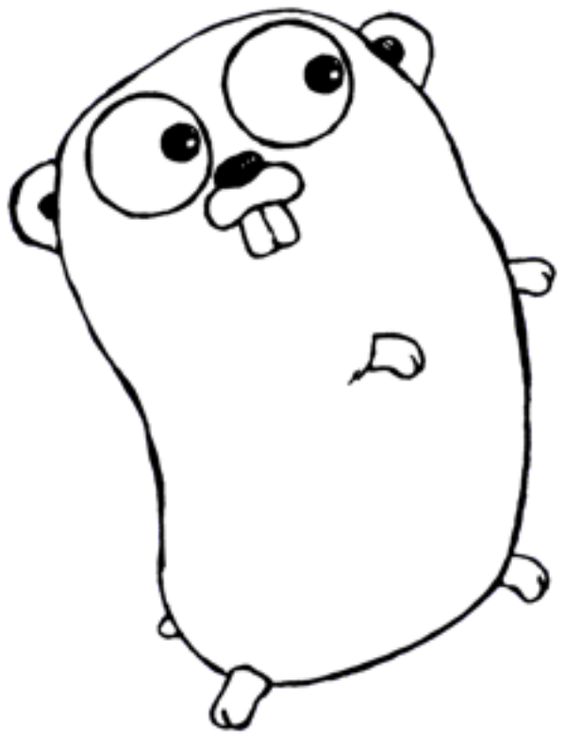
This is good Go style: just use the abstract behavior.
It's easy and much less typing (in two senses).

You can use this approach in Java but it's not usual Java style.
Java (like many languages) puts type inheritance first.

Implicitness means flexibility

- In Go, could use as many wrappers as you like.
  - A type can satisfy many interfaces and therefore be used by any number of 'abstract wrappers' like `CompressBuffer`.
- In Java, can only extend one abstract class.
  - Could use Java interfaces but still need to annotate the original implementation — that is, edit the existing code.
  - What if it's not yours to edit?
- In Go, `Compressor`'s implementers do not need to know about `CompressBuffer` or even the `Compressor` interface.
  - The `Buffer` type might be private yet the type with the `Compress` method could be in a standard library.





3: Lightweight interfaces

Interfaces are lightweight

- A typical Go interface has only one or two methods.
  - (In fact, the commonest interface has zero, but that's another story.)
- Programmers new to Go see interfaces as a building block for type hierarchies and tend to create interfaces with many methods.
- But that's the wrong way to think about them. They are:
  - small
  - nimble
  - often ad hoc

Problem: Generalizing RPC

The RPC package in Go uses package gob to marshal objects on the wire. We needed a variant that used JSON.

Abstract the codec into an interface:

```
type ServerCodec interface {
    ReadRequestHeader(*Request) os.Error
    ReadRequestBody(interface{}) os.Error
    WriteResponse(*Response, interface{}) os.Error
    Close() os.Error
}
```

Problem: Generalizing RPC

Two functions must change signature:

```
func sendResponse(sending *sync.Mutex, req *Request,
        reply interface{}, enc *gob.Encoder, errmsg string)
```

becomes

```
func sendResponse(sending *sync.Mutex, req *Request,
        reply interface{}, enc ServerCodec, errmsg string)
```

And similarly for requests.
That is almost the whole change to the RPC implementation.
The bodies of the functions needed a couple of tiny edits. In other such examples, often no editing would be required.

We saw an opportunity: RPC needed only `Read` and `Write` methods. Put those in an interface and you've got abstraction. Post facto.

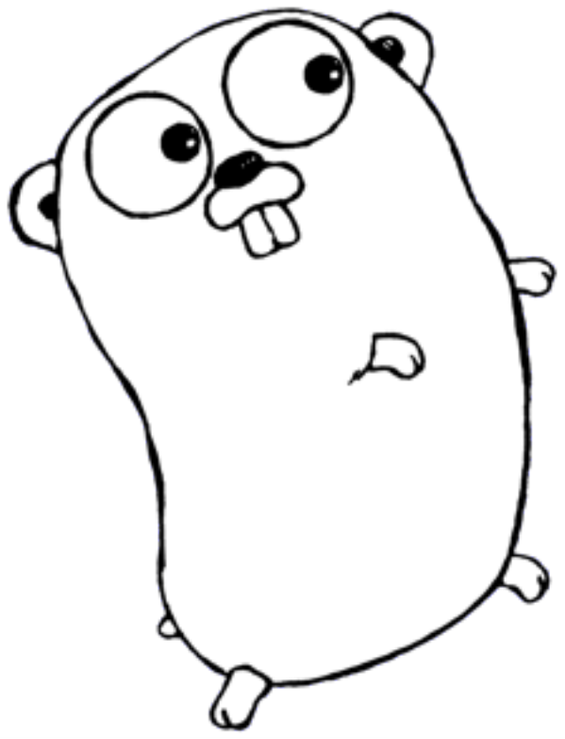
Total time: 20 minutes

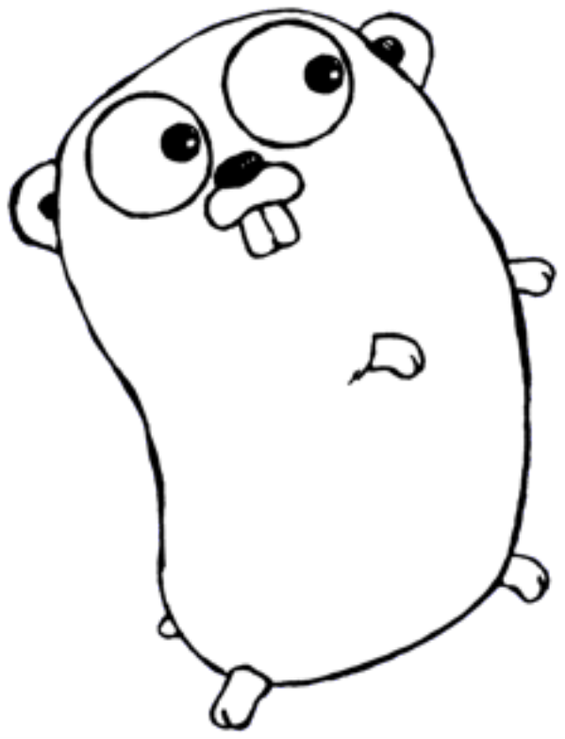
And that includes writing and testing the JSON implementation of the interface.

(We wrote a trivial `gobServerCodec` type to implement the new `rpc.ServerCodec` interface.)

In Java, RPC would be refactored into a half-abstract class, subclassed to create `JsonRPC` and `GobRPC`.

In Go, there is no need to manage a type hierarchy: just pass in a codec interface stub (and nothing else).





4: Common interfaces

Post facto abstraction

In the previous example, we were in charge of all the pieces.

But it's common for interfaces to arise as codifications of existing patterns.

Such interfaces often include only one or two methods: `io.Reader`, `io.Writer`, etc.

It is vital that such interfaces do not need retrofitting to work with existing code. They work automatically.

Simple interfaces are widely used

The type `aes.Cipher` has methods:

```
func (c *Cipher) BlockSize() int     // size of encryption unit
func (c *Cipher) Decrypt(src, dst []byte)  // decrypt one block
func (c *Cipher) Encrypt(src, dst []byte)  // encrypt one block
```

So do `blowfish.Cipher`, `xtea.Cipher` and others. We can make ciphers interchangeable by defining an interface:

```
type Cipher interface {
    BlockSize() int
    Decrypt(src, dst []byte)
    Encrypt(src, dst []byte)
}
```

Chaining ciphers

We define block cipher modes using the interface.

```
// cipher-block chaining
func NewCBCDecrypter(c Cipher, iv []byte, r io.Reader) io.Reader

// cipher feedback
func NewCFBDecrypter(c Cipher, s int, iv []byte, r io.Reader)
                     io.Reader

// output feedback
func NewOFBReader(c Cipher, iv []byte, r io.Reader) io.Reader
```

Want AES CBC mode?

```
    // (For brevity, we cheat a bit here about error handling)
    r = block.NewCBCDecrypter(aes.NewCipher(key), iv, r)
```

Blowfish CBC?

```
    r = block.NewCBCDecrypter(blowfish.NewCipher(key), iv, r)
```

No need for multiple CBC implementations.

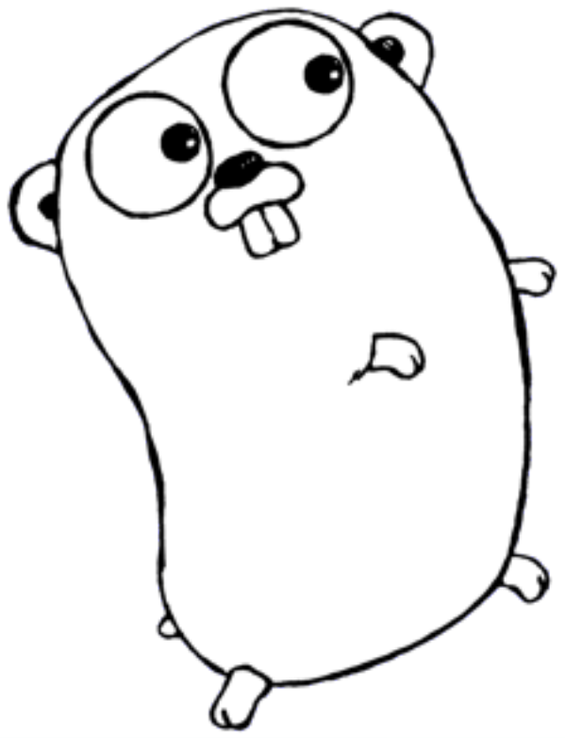
Chain away

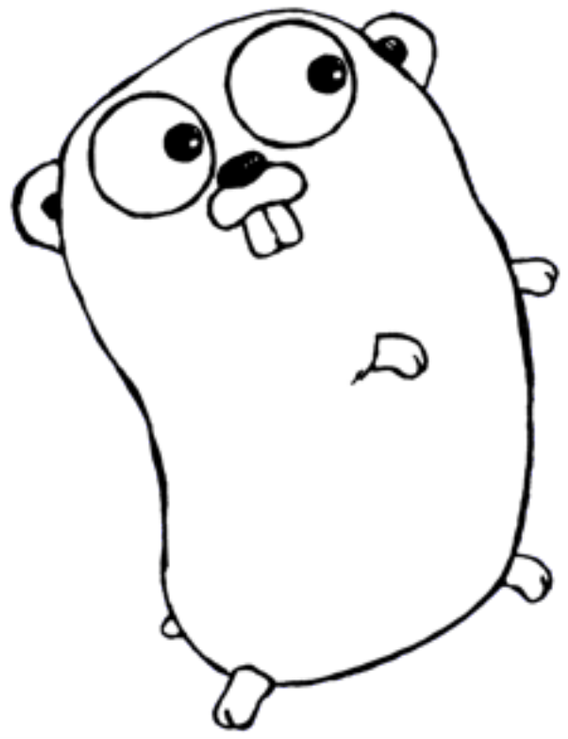
Libraries in other languages usually provide an API with the cross product of all useful ciphers and chainers. Go just needs to provide the building blocks.

And more! Decrypt and decompress by chaining further.

```
func DecryptAndGunzip(dstfile, srcfile string, key, iv []byte) {
    f := os.Open(srcfile, os.O_RDONLY, 0)  // open source file
    defer f.Close()
    c := aes.NewCipher(key)                // create cipher
    r := block.NewOFBReader(c, iv, f)      // decrypting reader
    r = gzip.NewReader(r)                  // decompressing reader
    w := os.Open(dstfile, os.O_WRONLY | os.O_CREATE, 0666)
    defer w.Close()
    io.Copy(w, r)                          // copy to output
}
```

(Again, cheating a bit regarding error handling.)





5: Concurrency for structure

Google I/O 10

Concurrent programs

Java programmers use class hierarchies to structure their programs.

Go's concurrency primitives provide the elements of another approach.

It's not about parallelism. Concurrent programming allows parallelism but that's not what it's really for.

It's about expressing program structure to represent independently executing actions.

In short:

- parallelism is about performance
- concurrency is about program design

Example: a load balancer

Imagine you have many processes requesting actions and a few workers that share the load. Assume workers are more efficient if they batch many requests.

We want a load balancer that distributes the load fairly across the workers according to their current workload.

In real life we'd distribute work across many machines, but for simplicity we'll just focus on a local load balancer.

This is simplistic but representative of the core of a realistic problem.

Life of a request

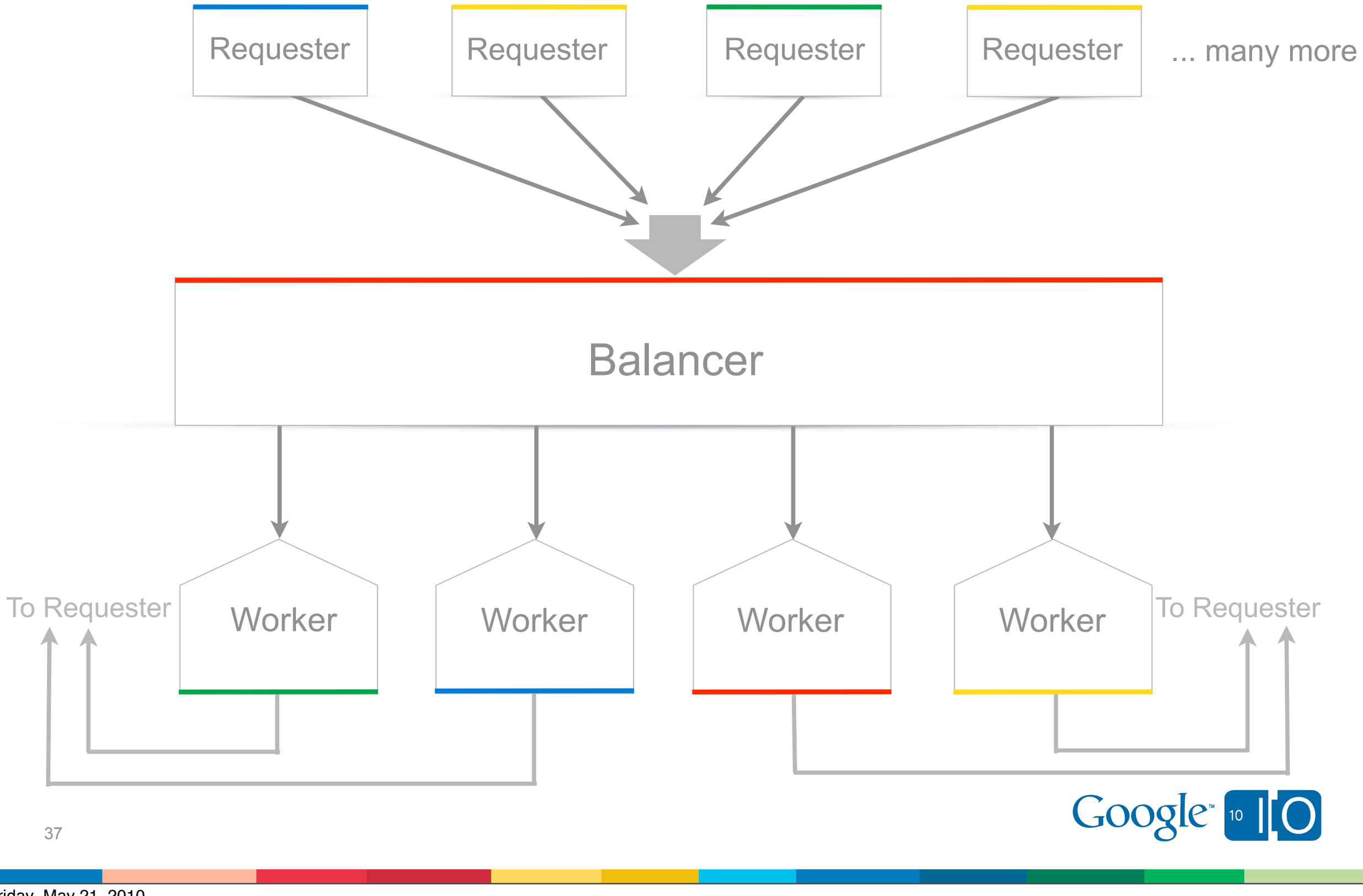
Requesters make a request to the load balancer.

Load balancer immediately sends the request to the most lightly loaded worker.

When it completes the task the worker replies directly to the requester.

Balancer adjusts its measure of the workloads.

A load balancer



Requester

The requester sends `Requests` to the balancer.

```
type Request struct {
    fn func() int  // The operation to perform
    c  chan int    // The channel on which to return the result
}
```

An artificial but illustrative simulation of a requester.

```
func requester(work chan Request) {
    c := make(chan int)
    for {
        time.Sleep(rand.Int63n(nWorker * 2e9)) // spend time
        work <- Request{workFn, c}             // send request
        result := <-c                          // wait for answer
        furtherProcess(result)
    }
}
```

Worker

The balancer will send each request to the most lightly loaded worker.

This is a simple version of a `Worker` but it's plausible.

```
func (w *Worker) work(done chan *Worker) {
    for {
        req := <-w.requests  // get Request from load balancer
        req.c <- req.fn()    // call fn and send result to requester
        done <- w            // tell balancer we've finished this job
    }
}
```

The channel of requests (`w.requests`) delivers requests to each worker. The balancer tracks the number of pending requests as a measure of load.

Note that each response goes directly to its requester.

Balancer

The load balancer needs a pool of workers and a single channel to which requesters can send work.

```
type Pool []*Worker
type Balancer struct {
    pool Pool
    done chan *Worker
}
```

At this point, the balancer is very easy.

```
func (b *Balancer) balance(work chan Request) {
    for {
        select {
        case req := <-work: // received a Request...
            b.dispatch(req)  // ...so send it to a Worker
        case w := <-b.done: // a worker has finished a request...
            b.completed(w)   // ...so update its info
        }
    }
}
```

A heap of channels

How do we implement `dispatch` and `completed`?

We can use a heap to choose the most lightly loaded worker by attaching the necessary methods to type `Pool` (`Len`, `Push`, `Pop`, `Swap`, `Less`). That's easy; here for instance is `Less`:

```
func (p Pool) Less(i, j int) bool {
    return p[i].pending < p[j].pending
}
```

And in each `Worker`, we keep a count of pending operations.

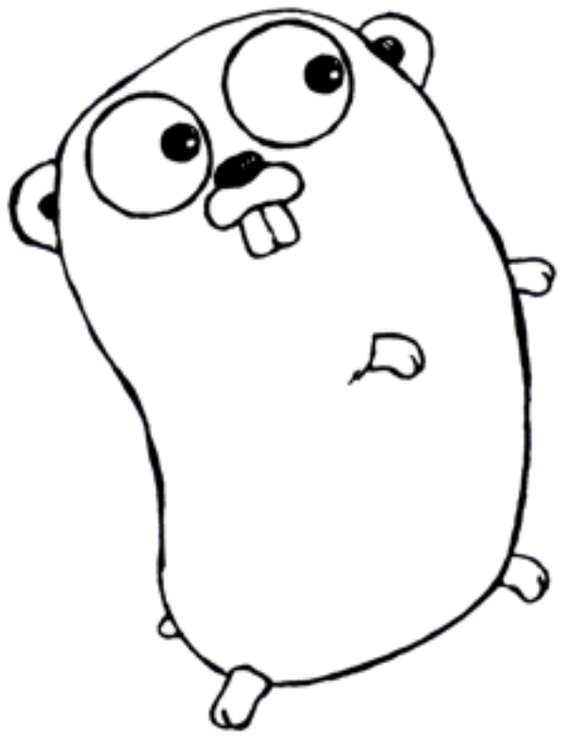
```
type Worker struct {
    requests chan Request   // work to do (a buffered channel)
    pending  int            // count of pending tasks
    index    int            // index in the heap
}
```

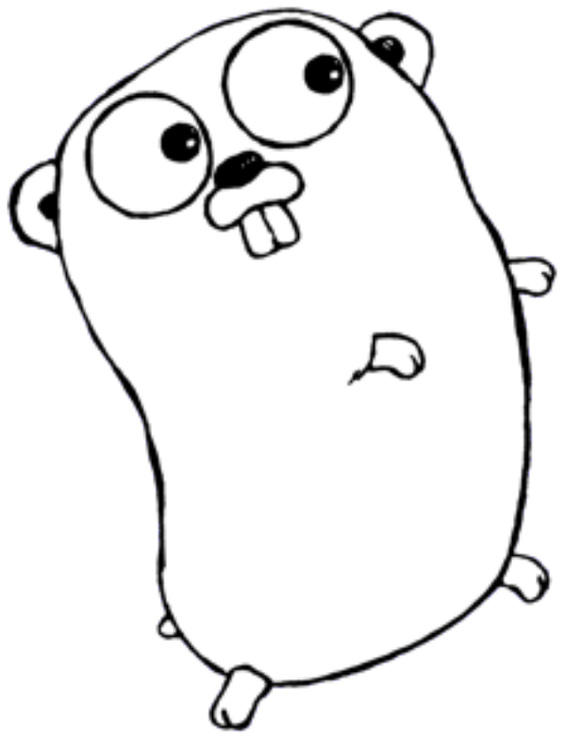
Use the heap to maintain balance

The implementation of `dispatch` and `completed` is easy:

```
// Send Request to worker
func (b *Balancer) dispatch(req Request) {
    w := heap.Pop(&b.pool).(*Worker)  // least loaded worker...
    w.requests <- req                 // ...is assigned the task
    w.pending++                       // one more in its queue
    heap.Push(&b.pool, w)             // put it back in the heap
}

// Job is complete; update heap
func (b *Balancer) completed(w *Worker) {
    w.pending--                       // one fewer in its queue
    heap.Remove(&b.pool, w.index)     // remove it from heap
    heap.Push(&b.pool, w)             // put it back where it belongs
}
```

Channels are first-class values. We've built a heap of channels to multiplex and load balance.





Demo

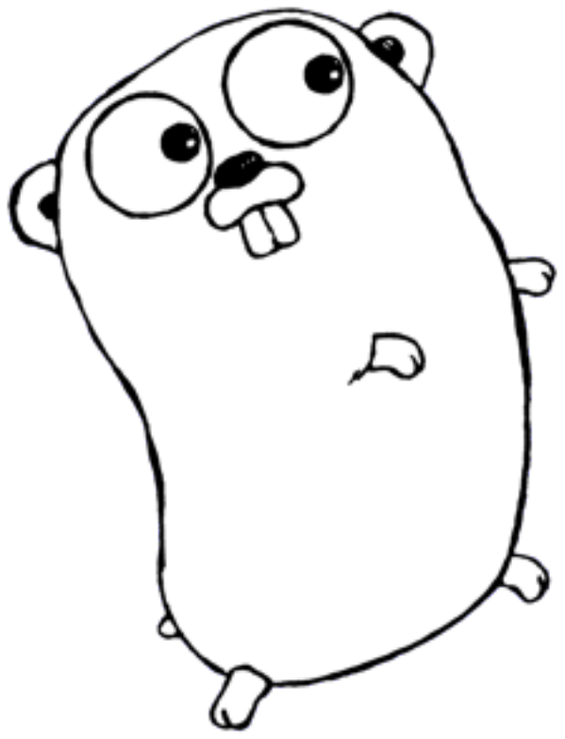
Discussion

Every operation blocks, yet this system is highly concurrent. The combination of communication and synchronization is a powerful tool.

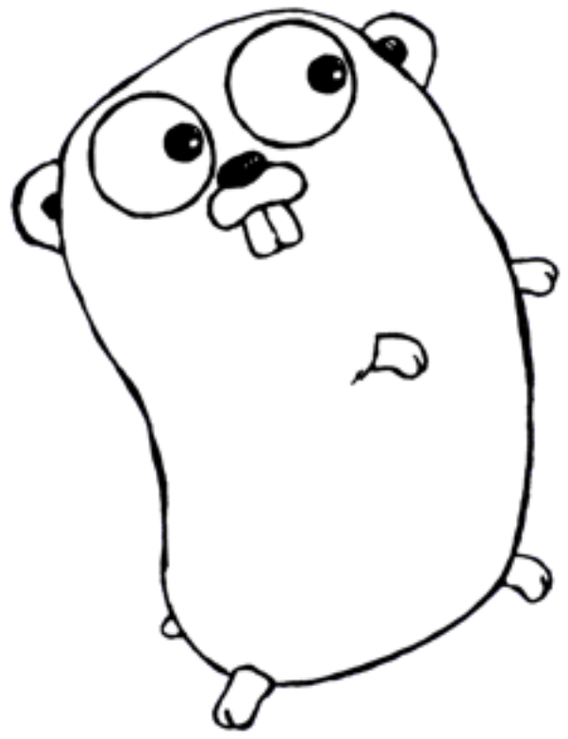
{Closure, channel} pairs are a nice way to pass work around. The closure says what to do; the channel lets the answer flow directly back to the requester.

Channels can be part of other data structures and can be passed between goroutines.

With very little work, you could use network channels or RPCs to make this a distributed, networked load balancer. (Although closures don't work across the network, RPCs help.)



Conclusion



Google I/O 10

Go is different

- Objects are not always classes
  - Inheritance is not the only way to structure a program

- You don't need to specify everything in advance
  - (And you shouldn't need to anyway)
  - Implicitly satisfying behavior leads to pleasant surprises

- Concurrency is not just parallelism
  - But it is a great way to structure software

Go is more productive

Go can make programming very productive:

- any type can be given methods, which opens up interesting design possibilities.

- most of the bookkeeping of type-driven programming is done automatically.

- the structuring power of concurrent programming leads easily to correct, scalable server software.

Such properties make Go programs more malleable, more adaptable, less brittle.

More at `http://golang.org`

Go is more fun

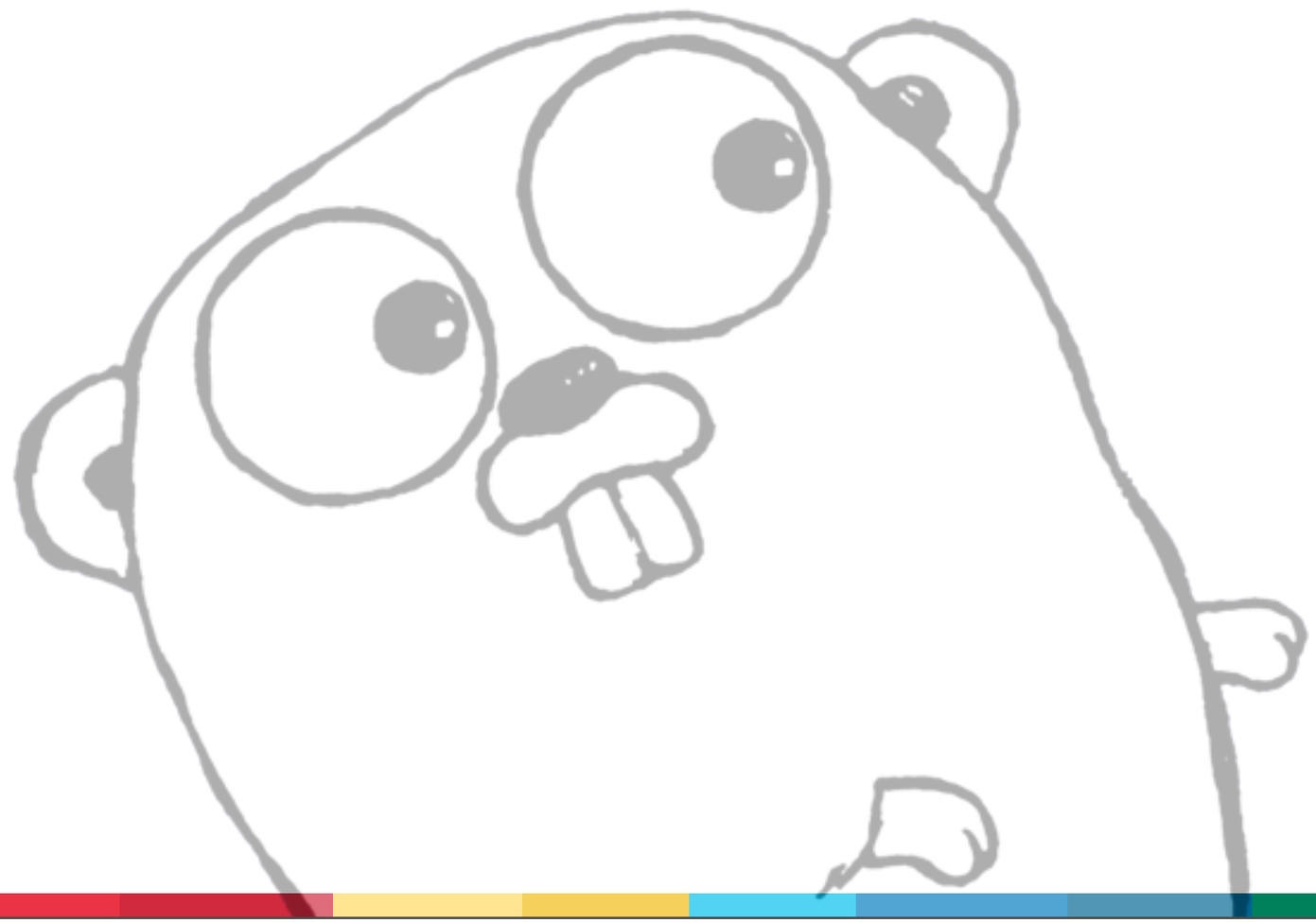
Go comes with T-shirts, tattoos and stickers.

Live waving

Please ask questions about this session on Google Wave:

`http://bit.ly/go2010io`

More about Go at `http://golang.org`



Google™ 10 I/O