

Go in Three Easy Pieces

Robert Griesemer
Google Inc.

Lang.NEXT 2012

<http://golang.org>



What is Go

- Go is an open source programming environment.
- It is a general purpose programming language:
 - Compact yet expressive
 - Statically typed, garbage-collected
 - Object- but not type-oriented
 - Strong concurrency support
 - Efficiently implemented
- It comes with a powerful standard library:
 - <http://golang.org/pkg>
- And a growing set of development tools:
 - go tool (cgo, cov, fix, prof, vet, ...), gofmt, godoc

"Hello, World"

```
package main
```

```
import "fmt"
```

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

"Hello, World" server

```
package main
```

```
import "fmt"
```

```
import "net/http"
```

```
func main() {  
    http.HandleFunc("/hello", helloHandler)  
    http.ListenAndServe(":7777", nil)  
}
```

```
func helloHandler(  
    w http.ResponseWriter, req *http.Request,  
) {  
    fmt.Fprintln(w, "Hello, 世界")  
}
```

- <http://localhost:7777/hello>

Recognizing Go code

- Declarations use a cleaned-up Pascal style.

```
// keyword name [type] [initial value]
import "fmt"
const digits = "0123456789abcdef"
type Point struct { x, y int }
var s [32]byte
var msgs = []string{"Hello, 世界", "Ciao, Mondo"}
```

- Statements use a cleaned-up C style.

```
i := len(s)
for x != 0 {    // Look, ma, no ()'s!
    i--
    s[i] = digits[x%base]
    x /= base
}
```

How is Go different?

- **Object-orientation:**
 - Methods without classes
 - Interfaces without hierarchies
 - Code reuse without inheritance
- **Concurrency and communication:**
 - Built into the language rather than the library (go, select, chan)
 - Concurrency support comes with "batteries included"
- **Handling of exceptional situations:**
 - Functions with multiple return values
 - Defer statement for deferring function invocation
 - Handling of exceptional situations without try-catch

1. Methods and Interfaces

Methods

```
type Point struct { x, y int }
```

```
func PointToString(p Point) string {  
    return fmt.Sprintf("Point{%d, %d}", p.x,  
        p.y)  
}
```

```
PointToString(myPoint) // static dispatch
```

- A method is a function with a receiver:

```
func (p Point) String() string {  
    return fmt.Sprintf("Point{%d, %d}", p.x,  
        p.y)  
}
```

```
myPoint.String() // static dispatch
```

Methods can be attached to any type

```
// Celsius, Fahrenheit, float32 are different types!
```

```
type Celsius float32  
type Fahrenheit float32
```

```
func (t Celsius) String() string {  
    return fmt.Sprintf("%f°C", t)  
}
```

```
func (t Fahrenheit) String() string {  
    return fmt.Sprintf("%g°F", t)  
}
```

```
func (t Celsius) ToFahrenheit() Fahrenheit {  
    return Fahrenheit(t*9/5 + 32) // conversion  
}
```

- No need for a class (there are no classes); just define methods as they become necessary.

Interfaces

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

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

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

- An interface defines a set of methods.
- A type that implements all methods of an interface is said to implement the interface.
- All types implement the empty interface `interface{}`.

Dynamic dispatch

```
var corner = Point{1, 1}
var boiling Celsius = 100
```

```
var v Stringer
```

```
v = corner
v.String()    // Point{1, 1} -- dynamic dispatch
```

```
v = boiling.ToFahrenheit()
v.String()    // 212°F      -- dynamic dispatch
```

- A value of a type that implements an interface can be assigned to a variable of that interface type.
- Point, Celsius, and Fahrenheit implement Stringer.

Composition and chaining

- Typically, interfaces are small (1-3 methods).
- Pervasive use of key interfaces in the standard library make it easy to chain APIs together.

```
package io
```

```
func Copy(dst Writer, src Reader) (int64, error)
```

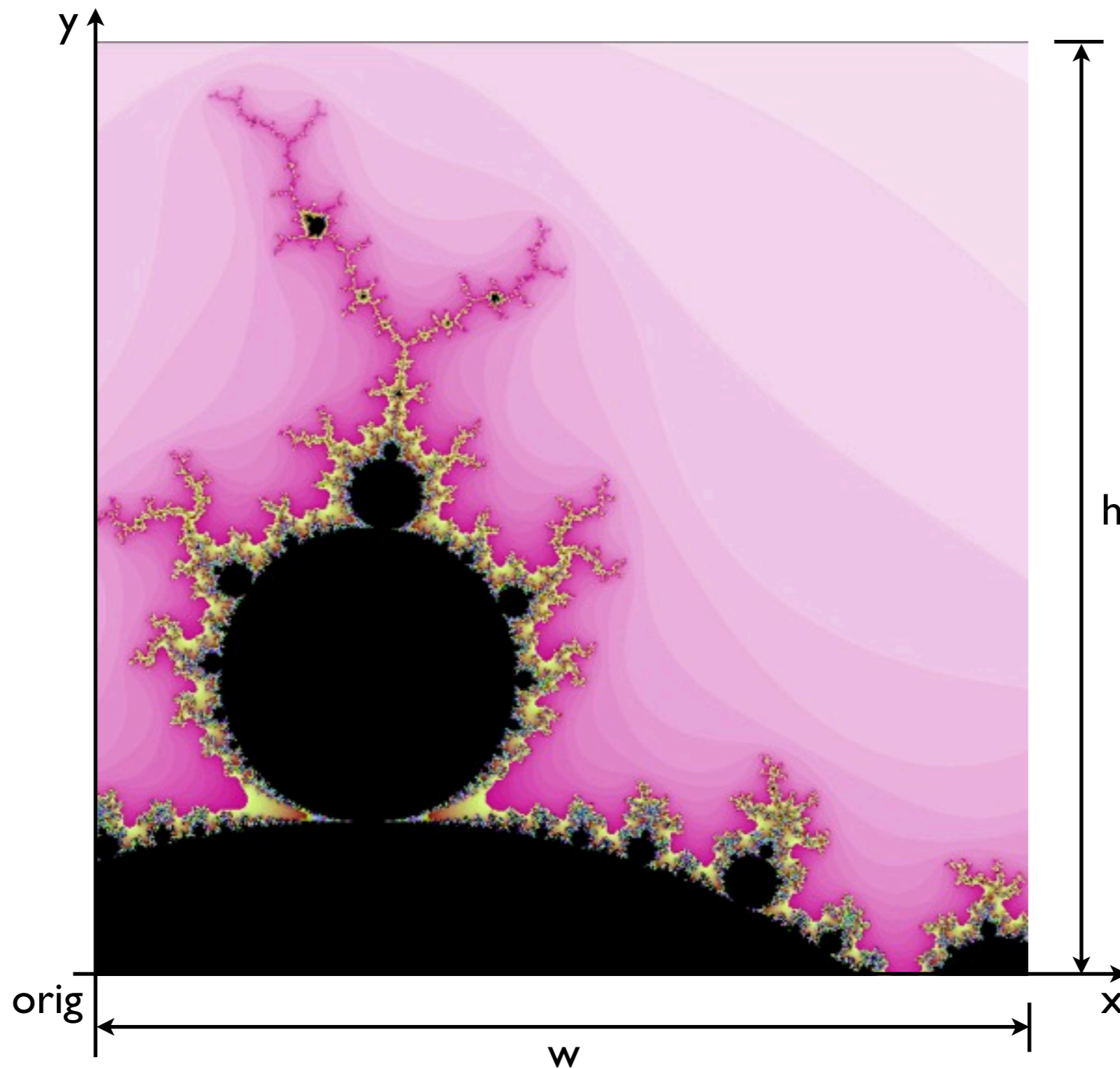
- The `io.Copy` function copies by reading from any `Reader` and writing to any `Writer`.
- Interfaces are often introduced ad-hoc, and after the fact.
- There is no explicit hierarchy and thus no need to design one!

A file server

```
func fileHandler(  
    w http.ResponseWriter, req *http.Request,  
) {  
    f, err := os.Open(req.FormValue("name"))  
    if err != nil {  
        fmt.Fprintln(w, err)  
        return  
    }  
    io.Copy(w, f)  
    f.Close()  
}
```

- `io.Copy` copies data by reading from file `f` which implements an `io.Reader` and writing to `w` which implements an `io.Writer`.
- `http://localhost:7777/file?name=server.go`

A Mandelbrot server



Representation of a Mandelbrot image

```
type mandel struct {  
    orig complex128 // left-bottom corner  
    d      float64   // pixel size in complex plane  
    w, h   int       // image size  
    n      int       // max. number of iterations  
}  
  
// mandel implements image.Image.  
func (m *mandel) ColorModel() color.Model {  
    return color.RGBAModel  
}  
func (m *mandel) Bounds() image.Rectangle {  
    return image.Rect(0, 0, m.w, m.h)  
}  
func (m *mandel) At(x, y int) color.Color {  
    y = m.h - y - 1 // image y is growing down  
    p := complex(float64(x)*m.d, float64(y)*m.d)  
    return colorFor(iterate(m.orig+p, m.n))  
}
```

Implementation of the server

```
func (m *mandel) getFormValues(r *http.Request){  
    m.d = 1 / floatFormValue(r, "zoom", 200)  
    m.w = intFormValue(r, "w", 600)  
    m.h = intFormValue(r, "h", 600)  
    ...  
}
```

```
func mandelHandler0(  
    w http.ResponseWriter, req *http.Request,  
) {  
    var m mandel  
    m.getFormValues(req)  
    png.Encode(w, &m)  
}
```

- `png.Encode` encodes an `image.Image` to an `io.Writer`.
- `http://localhost:7777/mandel0`

Digression: Functions are values, too

```
func inc1(x int) int { return x+1 }
```

```
f1 := inc1
```

```
f1(1) // 2
```

```
f2 := func(x int) int { return x+2 }
```

```
f2(1) // 3
```

```
// genInc creates an "incn" function (closure)
```

```
func genInc(n int) func(x int) int {
```

```
    return func(x int) int {
```

```
        return x+n
```

```
    }
```

```
}
```

```
f3 := genInc(3)
```

```
f3(1) // 4
```

A logging HTTP handler

```
func makeLoggingHandler(  
    h http.HandlerFunc) http.HandlerFunc {  
  
    return func(  
        w http.ResponseWriter, r *http.Request,  
    ) {  
        start := time.Now()  
        h(w, r)  
        log.Printf(  
            "%s\t%s\t%s", r.RemoteAddr, r.URL,  
            time.Since(start)  
        )  
    }  
}
```

- Use: `makeLoggingHandler(mandelHandler0)`
- `http://localhost:7777/mandel0`

Interfaces in practice

- Methods on any types and ad-hoc interfaces make for a light-weight OO programming style.
- Go interfaces enable post-facto abstraction
 - Existing code may not know about a new interface
- No explicit type hierarchies:
 - Coming up with the correct type hierarchy is hard
 - They are often wrong
 - They are difficult and time-consuming to change
- "Plug'n play" in a type-safe way

2. Goroutines and Channels

The go statement

- The `go` statement turns a function invocation into a **goroutine**.
- `go f(x)` invokes `f` to run concurrently.

```
// launch n goroutines f
for i := 0; i < n; i++ {
    go f(...)
}
```

- Often, goroutines are closures: `go func() { ... }()`.
- Stacks grow dynamically, no need to pre-allocate space per goroutine.
- Goroutines are very light-weight.

Channels

- A channel permits goroutines to communicate by sending and receiving messages.
- A channels acts like a FIFO queue with synchronized operations.

```
c := make(chan int, 1) // create a new buffered  
                        // channel of int values
```

```
c <- 42                // send the value 42 to c; blocks  
                        // until c can send the value
```

```
result = <-c           // receive a value from c; blocks  
                        // until a value can be received
```

- Channels may be synchronous (unbuffered) or asynchronous (buffered).

Goroutines and channels

```
c := make(chan, resultType)
go func() {
    // do some work concurrently
    c <- result // send result
}()
// do some more work
x := <-c // receive result, store in x
```

- Typical Go concurrency pattern:
 - use a channel to communicate result
 - use a closure as a goroutine
 - no explicit synchronization
- Mantra:
Do not synchronize to communicate, instead communicate to synchronize.

A simple work queue scheduler

```
func scheduler(n int) chan<- func() {  
    queue := make(chan func(), n)  
    for i := 0; i < n; i++ {  
        go func() {  
            for {  
                (<-queue)()  
            }  
        }()  
    }  
    return queue  
}
```

```
queue := scheduler(4) // use 4 goroutines  
result := make(chan int)  
queue <- func() { result <- ... }  
queue <- func() { result <- ... }  
...
```

A better Mandelbrot server

- Approach: implement sequential draw and then develop it into a concurrent version.

```
func mandelHandler1(  
    w http.ResponseWriter, req *http.Request,  
) {  
    var m mandel  
    m.getFormValues(req)  
    png.Encode(w, m.draw1())  
}
```

- draw1 draws a Mandelbrot image as defined by m.
- As before, png.Encode encodes an image.Image to an io.Writer.

Sequential draw

```
func (m *mandel) draw1() *image.RGBA {  
    img := image.NewRGBA(  
        image.Rect(0, 0, m.w, m.h))  
  
    // loop over all lines  
    for y := 0; y < m.h; y++ {  
        // loop over all pixels of a line  
        for x := 0; x < m.w; x++ {  
            img.Set(x, y, m.At(x, y))  
        }  
    }  
  
    return img  
}
```

- <http://localhost:7777/mandel1>

Concurrent draw

- Idea: Use as many CPUs as available to draw the image.
- Approach: Treat each line as an independent unit of work.

```
type line struct {  
    y      int  
    data []color.Color  
}
```

- Use a goroutine to compute the pixel data for each line.
- Use a channel to send back a computed line.

```
lines := make(chan *line)
```

Concurrent draw

```
func (m *mandel) draw2() <-chan *line {
    lines := make(chan *line)

    // loop over all lines
    for y := 0; y < m.h; y++ {
        // launch goroutine for each line
        go func(y int) {
            data := make([]color.Color, m.w)
            for x := range data {
                data[x] = m.At(x, y)
            }
            lines <- &line{y, data}
        }(y)
    }

    return lines
}
```

Concurrent Mandelbrot server

```
func mandelHandler2(  
    w http.ResponseWriter, req *http.Request,  
) {  
    var m mandel  
    m.getFormValues(req)  
    lines := m.draw2()  
  
    img := image.NewRGBA(image.Rect(0, 0, m.w, m.h))  
  
    for i := 0; i < m.h; i++ {  
        l := <-lines  
        for x, col := range l.data {  
            img.Set(x, l.y, col)  
        }  
    }  
  
    png.Encode(w, img)  
}
```

Goroutines and channels in practice

- Trivial to launch multiple threads of control
 - Number of goroutines only limited by memory
- Channels make communication between goroutines straightforward
 - Communication instead of synchronization primitives
 - Much easier to reason about concurrent programs
- Goroutine and channel notation is "light on the page"
 - Program logic is not drowned out by technicalities

3. Deferred Calls and Exceptional Situations

The defer statement

- A `defer` statement invokes a function whose execution is deferred to the moment the surrounding function returns.

```
func hello() {  
    defer fmt.Println("world")  
    fmt.Println("hello")  
}
```

- calling `hello()` prints:

```
hello  
world
```

- Crude analogy: Imagine a `try-catch-finally` without the `try-catch`, and the `finally` being a function-scoped statement rather than block-scoped control-flow structure.

Deferred calls are function-scoped

```
func f() {  
    fmt.Println("start")  
    for i := 0; i < 10; i++ {  
        defer fmt.Println(i)  
    }  
    fmt.Println("end")  
}
```

- Calling `f()` will print: start, end, 9, 8, 7, ...
- Multiple defers are stacked and executed in LIFO order.
- `defer`, like `go`, is often used with closures.
- Common uses: cleanup tasks such as closing files, etc.

```
f, err := os.Open("gopher.png")  
if err != nil { return err }  
defer f.Close() // close file upon function exit
```

Panics and deferred calls

- `panic("foo")` causes the run-time panic "foo".
- Some panics are generated by the run-time system (e.g., division by zero).
- A run-time panic causes a goroutine stack to unwind.
- Deferred function calls are run when panicking.

```
func f(n int) {  
    defer func() { print(n) }()  
    if n == 0 {  
        panic(0)  
    }  
    f(n-1)  
}
```

- Calling `f(5)` will print:

012345panic(0)

Panics and recoveries

- `recover()`, called inside a deferred function, recovers from a panic (if any), and returns the panic value (or `nil`).

```
func div(x, y int) (z int) {  
    defer func() {  
        if err := recover(); err != nil {  
            fmt.Println(err)  
            z = 0 // set return value!  
        }  
    }()  
    return x / y // may cause div-zero panic  
}
```

- calling `div(7, 0)` will return `z == 0` and print:

runtime error: integer divide by zero

- `defer`, like `go`, is often used with closures.

Application-specific error handling

- `formError` is an application-specific error:

```
type formError string
func (e formError) Error() string {
    return string(e)
}
```

- In `loggingHandler`:

```
defer func() {
    if err := recover(); err != nil {
        if err, ok := err.(formError); ok {
            fmt.Fprintf(w, "Error: %s", err)
            return
        }
        panic(err)
    }
}()
```

But there's much more!

- Mathematically precise constants
- Composite literals
- Embedding of types
- `Type switch` statement
- `select` statement (multiplexing of channels)
- Variadic functions
- Program initialization support
- Reflection (via library)
- etc.

Go 1

- Go 1 was released last week.
 - Culmination of 4+ years of language and library work.
 - Stable release for the foreseeable future.
 - Selected binary and complete source code release.
 - Excellent documentation.
 - Support for OS X, FreeBSD, OpenBSD, Linux, and Windows, 32bit and 64bit x86, as well as ARM (Linux only).
 - <http://golang.org/>
- Go 1 is also available for App Engine.
 - <https://developers.google.com/appengine/>

<http://golang.org/>