

Introduction to Go

Moncton User Group

15 March 2016

Serge Léger

Prototype Developer, National Research Council

About Me

- Bachelor of Computer Science from Udm.
- Started my career in 1994 with a local startup - Dovico.
- Joined the public service in 1998 to help transition internal applications to Y2K.
- In 2007, joined National Research Council where I work with researchers in a number of different fields.
- C, C++, Perl, HTML+Javascript+CSS, Java, Python, Go

Go Language (golang)

Go is:

- Lightweight, avoids unnecessary repetition
- Object Oriented, but not in the usual way
- Concurrent, in a way that keeps you sane
- Designed for working programmers
- Compiled
- Statically typed
- Garbage collected
- Cross platform (Linux, Mac OSX, Windows, iOS, Android)

What's Missing

- No type inheritance
- No method or operator overloading
- No generic programming
- No exceptions

History

- Project starts at Google in 2007 by engineers frustrated with the complexity of C++
- Open source release in November 2009
- Version 1.0 in March 2012
- Version 1.1 in May 2013
- Version 1.2 in December 2013
- Version 1.3 in June 2014
- Version 1.4 in December 2014
- **Version 1.5** in August 2015 - Compiler & tools now written in Go. GC is now concurrent.
- Version 1.6 in February 2016

Tutorial

Hello World

```
package main

import "fmt"

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

Run

Types and Variables

Go has many simple types:

```
int, int8, int16, int32, int64  
uint, uint8, uint16, uint32, uint64, uintptr  
float32, float64  
complex64, complex128  
bool, byte, rune, string, error
```

But these are the ones you'll use most often:

```
int, float64, bool, byte, string, error
```

`int` and `uint` are the natural or most efficient size for integers on a particular platform, either 32 or 64 bits.

All types have a **zero value**, which is used to initialize variables. `0` for numbers, `false` for booleans, `""` for strings and `nil` for everything else.

Variable Declaration

```
package main

import "fmt"

var name string = "Serge"

func main() {
    var age int = 40
    fmt.Printf("Hello, World! My name is %s, and I'm %d (ish).\n", name, age)
}
```

Run

- implicit type declaration
- short variable declaration
- unused variables

Control Structures

If statement

A simple if statement:

```
if x > 0 {  
    return y  
}
```

- Braces are mandatory, note the missing parentheses.
- The expression must evaluate to a boolean.
- There is also an optional else statement:

```
if x > 0 {  
    return y  
} else {  
    return z  
}
```

If statement (continued)

- Cascading if / else statements

```
if x > 0 {  
    return y  
} else if x < 0 {  
    return z  
} else if x == 0 {  
    return x  
} else {  
    // Can't reach this condition?  
}
```

- if statements can also contain an initialization statement:

```
if x = someFunction(); x != 0 {  
    return z*x  
}
```

If statement (playground)

```
func main() {  
    percent := someFunction()  
    if percent < 50 {  
        fmt.Println("You're just starting.", percent)  
    } else {  
        fmt.Println("Almost done.", percent)  
    }  
}
```

Run

Switch statement

A simple C-like switch statement:

```
func main() {  
    var heads, tails int  
  
    switch coinFlip() {  
    case "heads":  
        heads++  
  
    case "tails":  
        tails++  
  
    default:  
        fmt.Println("landed on edge!")  
    }  
  
    fmt.Println(heads, tails)  
}
```

Run

- No break statement required, by default cases do not fall through.
- Use the `fallthrough` statement if that behavior is needed.

Switch statement (continued)

Instead of using `fallthrough`, case statements can be combined:

```
func main() {  
    dayOfWeek := today()  
  
    switch dayOfWeek {  
    case "Saturday", "Sunday":  
        fmt.Println("It's the weekend!")  
  
    default:  
        fmt.Println("Get up and go to work.")  
    }  
}
```

Run

Switch statement (continued)

The switch's expression is optional, so a switch statement can be used to replace long and complex if / else cascades:

```
func main() {  
    dayOfWeek := today()  
    weekNumber := week()  
  
    switch {  
    case dayOfWeek == "Saturday" || dayOfWeek == "Sunday":  
        fmt.Println("It's the weekend!")  
  
    case dayOfWeek == "Tuesday" && weekNumber%2 == 0:  
        fmt.Println("Payday!")  
  
    default:  
        fmt.Println("Get up and go to work.")  
    }  
}
```

Run

For statement

Go has a single loop statement; the for loop has multiple forms:

```
// C style
for initialization; condition; post {
}
```

Both, *initialization* and *post* are optional:

```
// A traditional "while" loop
for condition {
}
```

condition is also optional and defaults to true:

```
// an infinite loop
for {
}
```

For statement (continued)

- break statement terminates the execution of the inner-most loop
- continue statement begins the next iteration of the inner-most loop
- There is also a "for-each" operator: range which operates on strings, arrays, slices, maps and channels.

```
func main() {  
    // C style loop  
    for i := 0; i < 7; i++ {  
        fmt.Println(dwarves[i])  
    }  
  
    fmt.Println("-----")  
  
    // Using the range operator  
    for index, name := range dwarves {  
        fmt.Println(index, name)  
    }  
}
```

Run

Constants, Custom Types, and Enumerated Types

Constants

Constants are defined with the `const` keyword:

```
const Pi = 3.141592
const (
    a = 1
    b = 2
)
const c, d = 3, 4

func main() {
    fmt.Println(Pi, a, b)
    // c = 2
}
```

Run

iota

Go provides the `iota` constant generator to help construct constant values:

```
const (  
    Sunday    = 0  
    Monday    = 1  
    Tuesday   = 2  
    Wednesday = 3  
    Thursday  = 4  
    Friday    = 5  
    Saturday  = 6  
)  
  
func main() {  
    fmt.Println("Sunday = ", Sunday)  
    fmt.Println("Saturday = ", Saturday)  
}
```

Run

Custom Types

Using the type keyword we can create custom types:

```
type Color uint64
```

```
func main() {  
    var red Color = 1  
    fmt.Println(red)  
  
    // Same rules apply, you cannot assign a Color value to an uint64 variable  
    // var value uint64  
    // value = red  
}
```

Run

Enumerated Types

By combining constants and custom types we can create enumerated types:

```
type Color uint64

const (
    Red Color = iota
    Green
    Blue
)
```

or bit masks (from Go's net package)

```
type Flags uint
const (
    FlagUp Flags = 1 << iota
    FlagBroadcast
    FlagLoopback
    FlagPointToPoint
    FlagMulticast
)
```

Collections and Complex Types

Arrays

Arrays is a **fixed-length** sequence of zero or more elements. Since they are fixed-length arrays are rarely used.

```
// An IP address
var ip [4]byte
ip[0] = 192
ip[1] = 168
ip[2] = 1
ip[3] = 1

// The same declaration
var ip = [4]byte{ 192, 168, 1, 1 }

// Another, the array size is calculated by the compiler
var ip = [...]byte{ 192, 168, 1, 1 }
```

- The built-in `len` function returns the number of elements in the array.
- Arrays are always passed by value, so use with care.

Arrays (playground)

```
package main

import "fmt"

func main() {
    var daysOfWeek = [7]string{
        "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday",
        "Saturday",
    }

    fmt.Println("Number of entries: ", len(daysOfWeek))
    fmt.Println("First day:", daysOfWeek[0])
    fmt.Printf("%T", daysOfWeek)
}
```

Run

Slices

Slices represents variable-length sequences, like arrays, slices are indexable and have a length. Slices have another property: capacity. When a slice reaches its capacity, the Go runtime allocates more memory.

```
// An IP address, using slices
var ip []byte
ip = append(ip, 192)
ip = append(ip, 168)
ip = append(ip, 1)
ip = append(ip, 1)

// The same declaration
var ip = []byte{192, 168, 1, 1}
```

- `make` is a built-in function to create a new slice.
- `len` returns the length of the slice.
- `cap` returns the capacity of the slice.
- `append` appends elements to the slice.

Slice Construction

A slice can be constructed using the `make` built-in function, providing a length and optionally an initial capacity.

```
func main() {  
    ids := make([]int, 5)  
    ids[0] = 2  
    ids[1] = 15  
    fmt.Println("len=", len(ids), "cap=", cap(ids))  
    fmt.Println(ids)  
}
```

[Run](#)

Slices are backed by an array which Go grows to accommodate new elements.

Length and Capacity

```
func main() {  
    var ids = make([]int, 0, 10)  
    for i := 0; i < 100; i++ {  
        fmt.Println(len(ids), cap(ids))  
        ids = append(ids, i)  
    }  
  
    // The in-memory location of the array changes as Go allocates more memory for the  
    // growing slice.  
    // slicePtr(ids)  
}
```

Run

Slice Operators

Go provides slice operators to create different views of the same in-memory array:

```
var newSlice = slice[start:end] // start is inclusive and end is exclusive.
```

```
func main() {  
    var daysOfWeek = [7]string{  
        "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",  
    }  
  
    // Slice from an array  
    weekdays := daysOfWeek[1:6]  
  
    // Range operator over a slice  
    for i, v := range weekdays {  
        fmt.Println(i, v)  
        weekdays[i] = strings.ToLower(v)  
    }  
  
    fmt.Println(weekdays)  
  
    // But the weekdays and daysOfWeek slices are backed by the same array:  
    // fmt.Println(daysOfWeek)  
}
```

Run

Maps

Maps are Go's implementation of associative arrays. Maps are expressed as follows:

```
map[KeyType]ValueType
```

```
func main() {  
    var capitals = map[string]string{  
        "NB": "Fredericton",  
        "NS": "Halifax",  
        "QC": "Quebec",  
    }  
  
    fmt.Println(capitals["NB"])  
    fmt.Println(capitals["NS"])  
  
    // Check if a key exists  
    city, ok := capitals["ON"]  
    if ok {  
        fmt.Println("Ontario's capital is", city)  
    }  
}
```

Run

Map Construction

Maps must be created using the `make` built-in function:

```
func main() {  
    capitals := make(map[string]string, 13) // initial space for 13 provinces  
    capitals["NB"] = "Fredericton"  
    capitals["NS"] = "Halifax"  
    capitals["QC"] = "Quebec"  
    capitals["ON"] = "Toronto"  
  
    for province, city := range capitals {  
        fmt.Printf("%s's capital is %s\n", province, city)  
    }  
  
    // Remove an entry from the map  
    delete(capitals, "NB")  
  
    fmt.Println(len(capitals))  
    fmt.Println(capitals)  
}
```

Run

- `make` is a built-in function to create a new map.
- `len` returns the length of the map.

- `delete` removes an entry from the map.

Structures

A structure is a sequence of fields, each of which has a name, a type and optionally a `field tag` which is metadata encoded in key/value pairs.

```
type Person struct {  
    Name    string  
    Street  string  
    City    string  
}
```

Structures (playground)

```
func main() {  
    var bob Person  
    bob.Name = "Bob"  
    bob.Street = "123 Street"  
    bob.City = "Somewhere"  
  
    // Using struct literals  
    var charles = Person{  
        Name: "Chuck",  
        Street: "456 Street",  
        City: "Somecity",  
    }  
  
    fmt.Printf(fmtString, "Bob", "Charles")  
    fmt.Printf(fmtString, "-----", "-----")  
    fmt.Printf(fmtString, bob.Name, charles.Name)  
    fmt.Printf(fmtString, bob.Street, charles.Street)  
    fmt.Printf(fmtString, bob.City, charles.City)  
  
    // fmt.Printf("%+v\n", bob)  
}
```

Run

Embedding structures and anonymous fields

Instead of doing the following (which says `Employee` as a field named `Person` of type `Person`):

```
type Employee struct {  
    Person Person  
    Id      string  
}
```

We can omit the field name, this provides a convenient shortcut that allows the following shorter notation: `Employee.Name` instead of `Employee.Person.Name`.

```
type Employee struct {  
    Person  
    Id      string  
}
```

Embedding structures (playground)

```
func main() {  
    var bob Employee  
    bob.Id = "123"  
    bob.Name = "Bob"  
    bob.Street = "123 Street"  
    bob.City = "Somewhere"  
  
    // Using struct literals  
    var charles = Employee{  
        Person: Person{Name: "Chuck", Street: "456 Street", City: "Somecity"},  
        Id: "456",  
    }  
  
    fmt.Printf(fmtString, "Bob", "Charles")  
    fmt.Printf(fmtString, "-----", "-----")  
    fmt.Printf(fmtString, bob.Id, charles.Id)  
    fmt.Printf(fmtString, bob.Name, charles.Name)  
    fmt.Printf(fmtString, bob.Street, charles.Street)  
    fmt.Printf(fmtString, bob.City, charles.City)  
  
    // fmt.Printf("%+v\n", bob)  
}
```

Run

Functions

Functions

Function declaration has a name, a list of parameters, an optional list of results, and a body:

```
func name(parameter-list) (result-list) {  
    body  
}
```

```
func max(x int, y int) int {  
    if x > y {  
        return x  
    } else {  
        return y  
    }  
}  
  
func main() {  
    fmt.Println(max(10, 5))  
}
```

Run

Functions with multiple return values

Functions can return multiple values, this is often used to return error conditions.

```
func addAndProduct(x, y int) (int, int) {  
    return x + y, x * y  
}  
  
func safeAddAndProduct(x, y int) (int, int, error) {  
    if x < 0 || y < 0 {  
        return 0, 0, errors.New("can't work with negative numbers")  
    }  
    return x + y, x * y, nil  
}  
  
func main() {  
    a, b := addAndProduct(2, 5)  
    fmt.Println(a, b)  
  
    // x, y, err := safeAddAndProduct(-2, 5)  
    // if err != nil {  
    //     fmt.Println("error occurred:", err)  
    // } else {  
    //     fmt.Println(x, y)  
    // }  
}
```

Run

Functions are first-class values

```
func square(n int) int    { return n * n }
func negative(n int) int  { return -n }
func product(n, m int) int { return n * m }

func main() {
    var fn func(int) int

    // assign the square function to f
    fn = square
    fmt.Println(fn(5))

    // assign the negative function to f
    fn = negative
    fmt.Println(fn(5))

    // assign the product function to f
    // fn = product
    // fmt.Println(fn(5, 2))
}
```

Run

Closures

```
func main() {  
    var fn func(int) int  
  
    // assign the square function to f  
    fn = square  
    fmt.Println(fn(5))  
  
    // assign the negative function to f  
    fn = negative  
    fmt.Println(fn(5))  
  
    // assign the product function to f  
    // fn = product  
    // fmt.Println(fn(5, 2))  
}  
  
func newProductFunc(n int) func(int) int {  
    return func(m int) int {  
        return product(n, m)  
    }  
}
```

Run

Variadic functions

```
func sum(numbers ...int) (total int) {  
    log.Printf("%T len=%d\n", numbers, len(numbers))  
  
    for _, v := range numbers {  
        total += v  
    }  
    return  
}  
  
func main() {  
    fmt.Println("Sum is", sum())  
    fmt.Println("Sum is", sum(2, 4, 6, 8, 10))  
}
```

Run

Deferred Functions

The execution of a function can be deferred until the enclosing function returns. This is normally used with paired operations like **open/close** or **lock/unlock**.

```
func readFile(filename string) {  
    fh, err := os.Open(filename)  
    if err != nil {  
        return  
    }  
  
    defer fh.Close()  
}
```

Deferred example

Deferred calls are stacked, so deferred functions are called in LIFO order:

```
func sum(numbers ...int) (total int) {  
    defer log.Printf("Deferred? %T len=%d\n", numbers, len(numbers))  
  
    for _, v := range numbers {  
        defer log.Println(v, total)  
        total += v  
    }  
  
    log.Println("leaving sum()")  
    return  
}  
  
func main() {  
    fmt.Println(sum(2, 4, 6, 8, 10))  
}
```

Run

Objects

Objects

In Go an object is simply a variable or value that has methods, a method is a function associated with a particular type.

Here is generic method definition:

```
func (receiver) name(parameter-list) (result-list) {  
    body  
}
```

- There is no inheritance in Go instead types are composed.
- There are no constructors or destructors either.

Objects Example

```
type Point struct {  
    X, Y float64  
}  
  
func (point Point) Distance(q Point) float64 {  
    return math.Hypot(point.X-q.X, point.Y-q.Y)  
}  
  
func main() {  
    var p = Point{3, 5}  
  
    dist := p.Distance(Point{1, 0})  
  
    fmt.Println(dist)  
}
```

Run

Methods with Pointer Receivers

The `Point.Distance` method uses a pass-by-value receiver meaning that any changes made to the point variable will be lost. Instead we can use a **pointer receiver**

```
func (point *Point) ScaleBy(n float64) {  
    point.X *= n  
    point.Y *= n  
}  
  
func main() {  
    var p = Point{3, 5}  
  
    fmt.Println("Before:", p)  
    p.ScaleBy(3)  
    fmt.Println("After:", p)  
  
    dist := p.Distance(Point{1, 0})  
    fmt.Println(dist)  
}
```

Run

Object Composition

```
type ColoredPoint struct {  
    Point  
    Color string  
}  
  
func main() {  
    cpRed := ColoredPoint{Point{1, 2}, "red"}  
    cpBlue := ColoredPoint{Point{3, 6}, "blue"}  
  
    cpRed.ScaleBy(3) // ColoredPoint "inherits" the Point methods  
  
    fmt.Println(cpRed.Distance(Point{1, 0}))  
    fmt.Println(cpBlue.Distance(Point{1, 0}))  
}
```

Run

Interfaces

Interfaces

Interfaces in Go are similar to interfaces from Java. Here is a generic interface definition:

```
type Name interface {  
    // List of functions ...  
    fnName1(parameter-list) (result-list)  
    fnName2(parameter-list) (result-list)  
    fnName3(parameter-list) (result-list)  
  
    // ...or list of other interfaces  
    InterfaceName2  
}
```

- There is no requirement to declare that an object *implements* an interface. The compiler ensures that an object properly implements the interface.
- Interfaces in Go are normally small with only a few methods.

Interfaces from Go's io package

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

```
type Closer interface {  
    Close() error  
}
```

// Interface embedding

```
type ReadCloser interface {  
    Reader  
    Closer  
}
```

Implementation of the io.Reader interface

```
type myReader uint8

func (r myReader) Read(buf []byte) (int, error) { // io.Reader "implementation"
    for i := range buf {
        buf[i] = byte(r)
    }
    return len(buf), nil
}

func main() {
    var reader myReader = 65

    // Read 10 bytes from the reader...
    var buf = make([]byte, 10)
    reader.Read(buf)

    fmt.Println(buf)

    io.Copy(os.Stdout, reader)
}
```

Run

Interfaces

```
type Shape interface {  
    Area() float64  
}
```

```
type Square struct{ side float64 }  
  
func (sq *Square) Area() float64 {  
    return sq.side * sq.side  
}
```

```
type Circle struct{ radius float64 }  
  
func (c *Circle) Area() float64 {  
    return math.Pi * math.Pow(c.radius, 2)  
}
```

Interfaces (playground)

```
type Rect struct{ width, height int }

func (r *Rect) Area() int {
    return r.width * r.height
}

func main() {
    var c = &Circle{5}
    var s = &Square{7}
    var r = &Rect{7, 5}

    // boring... call the Area method directly
    fmt.Println(c.Area())
    fmt.Println(s.Area())
    fmt.Println(r.Area())

    // call the Area method via the Shape interface
    // shapes := []Shape{c, s, r}
    // for _, shape := range shapes {
    //     fmt.Println(shape.Area())
    // }
}
```

Run

Empty Interface

An empty interface is an interface with no methods -- meaning that everything in Go satisfies the empty interface. This is similar to Java's `Object`. The empty interface type is written `interface{}`:

```
var f interface{}
```

```
func describe(v interface{}) {  
    fmt.Printf("(%T, %v)\n", v, v)  
}  
  
func main() {  
    var f32, f64 = float32(21), float64(42)  
    var i, s = 10, "A string"  
  
    describe(f32)  
    describe(f64)  
    describe(i)  
    describe(&i)  
    describe(s)  
}
```

Run

Type Assertions

Sometimes it is useful to either convert an interface to its underlying type or to convert it to another interface.

```
func readAndClose(r io.Reader) {
    // ... read data from r ...

    // Close the file if it implements the io.Closer interface
    if closer, ok := r.(io.Closer); ok { // Type Assertion to another interface
        fmt.Println("Closing the file")
        closer.Close()
    }

    // Is this a myReader?
    if _, ok := r.(myReader); ok { // Type Assertion to the implementation type
        fmt.Println("You've been using a dumb reader.")
    }
}

func main() {
    var reader myReader = 65
    readAndClose(os.Stdin)
    readAndClose(reader)
}
```

Run

A complex type assertion example

```
func double(x interface{}) interface{} {  
    if i, ok := x.(int); ok {  
        return i * 2  
    } else if iPtr, ok := x.(*int); ok {  
        return (*iPtr) * 2  
    } else if f32, ok := x.(float32); ok {  
        return f32 * 2  
    } else if f64, ok := x.(float64); ok {  
        return f64 * 2  
    } else {  
        log.Printf("unsupported type: %T\n", x)  
    }  
  
    panic("unsupported type")  
}
```

From Type Assertions to Type Switches

```
func main() {  
    fmt.Println(double(float32(5.5)))  
    fmt.Println(double(float64(5.5)))  
  
    i := 15  
    fmt.Println(double(i))  
    fmt.Println(double(&i))  
}
```

Run

Type Switches

```
func triple(x interface{}) interface{} {  
    switch x := x.(type) {  
    case int:  
        return x * 3  
    case float32:  
        return x * 3  
    case float64:  
        return x * 3  
    case *int:  
        return *x * 3  
    default:  
        log.Printf("unsupported type: %T\n", x)  
    }  
  
    panic("unsupported type")  
}
```


Type Switches

```
func main() {  
    fmt.Println(double(float32(5.5)))  
    fmt.Println(double(float64(5.5)))  
  
    i := 15  
    fmt.Println(double(i))  
    fmt.Println(double(&i))  
}
```

Run

Concurrency

Go supports concurrency

Go provides:

- concurrent execution (goroutines)
- synchronization and messaging (channels)
- multi-way concurrent control (select)

Goroutines are not threads

They're a bit like threads, but cheaper. Small and resizable stacks (~2KB versus ~1MB for threads).

Common to launch a large number of goroutines during the life of an application.

The Go runtime multiplexes goroutines across OS threads.

Goroutines (launch and forget)

```
func say(str string) {  
    fmt.Println(str)  
}  
  
func main() {  
    go say("Hello, World!")  
  
    say("All Done!")  
  
    time.Sleep(500 * time.Millisecond)  
}
```

Run

Channels

Channels provide communication and synchronization mechanisms between goroutines. Each channel provides a conduit for values of a particular types.

```
var chInt chan int // A channel of integers
var chPtr chan *int // A channel of integer pointers
```

A channel is created with the built-in make function:

```
chInt = make(chan int)

chInt <- 42 // Send a value on the channel (blocks until another goroutine reads from chInt)

// Reads a value (blocks until another goroutine is writing to chInt)
v := <- chInt

// Channels can be used with for/range loop
for v := range chInt {
    // Do something with v
}

// Close a channel
close(chInt)
```

Ping-Pong

```
type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

More Concurrency

Go has a `select` statement that can multiplex multiple channels, similar to C's `select()` function for file handles or a `switch` statement.

```
select {  
case v := <- ch1:  
    // do something with v  
  
case ch2 <- "a string":  
  
default:  
    // default is optional, if it's absent the whole select blocks until a case is ready  
}
```

Go also has a package `sync` and `sync/atomic` that provides regular synchronization mechanisms:

- `sync.Mutex`, `sync.RWMutex`, `sync.WaitGroup`
- `atomic.CompareAndSwap`, `atomic.AddInt`

Packages

Workspace Organization

The only configuration most users ever need is the GOPATH environment variable.

```
export GOPATH=$HOME/projects/pivottable
```

The GOPATH variable is similar to Java's CLASSPATH variable and can contain more than one entry, separated by : (or ; on Windows).

A workspace contains at least one directory `src` which contains your project's source files (and third-party packages you may have installed). The Go tools will generate two other directories:

- `bin` contains your project's executable files
- `pkg` contains the compiled packages

Package Construction

```
$GOPATH/  
├─ bin  
├─ pkg  
└─ src  
    └─ pt  
        ├── cmd  
        │   └─ pivottable  
        ├── ioutil  
        └─ report  
            └─ text
```

In the example above, the pivottable project defines the following packages:

```
import "pt"  
import "pt/ioutil"  
import "pt/report"  
import "pt/report/text"
```

Package Construction

A package is defined by one or more `.go` files, each declaring the same package namespace using the package statement.

By convention, the directory name and the package name are the same, but it's not a requirement.

```
$GOPATH/  
└─ src  
    └─ pt  
        └─ ioutil  
            ├── colreader.go  
            ├── colreader_test.go  
            ├── cpuprofile.go  
            ├── io.go  
            ├── randseed.go  
            ├── reader.go  
            ├── reader_test.go  
            ├── traceprofile.go  
            ├── writer.go  
            └─ writer_test.go
```

Import Paths

```
import (  
    "fmt"  
    "pt/ioutil"  
    "github.com/ckeekybits/is"  
    "google.golang.org/api/calendar/v3" // actual package name is "calendar"  
)
```

Import paths are only strings and have no meaning to the Go language, they're only used for looking up packages.

Go tools will use GOROOT and GOPATH to find the actual packages:

```
$GOROOT/src/fmt  
  
$GOPATH/src/pt/ioutil  
$GOPATH/src/github.com/ckeekybits/is  
$GOPATH/src/google.golang.org/api/calendar/v3
```

Standard Library

Go's standard library has about 150 packages:

- Input/Output (bufio, io, os, path, path/filepath)
- Networking (net, net/http)
- Types (strings, bytes, errors, time, strconv)
- Testing (testing)
- Reflection (reflect)
- Regular Expression (regexp)
- Many more... <https://golang.org/pkg/> (<https://golang.org/pkg/>)

References

- The Go Programming Language, Alan A. A. Donovan & Brian W. Kernighan
- <https://tour.golang.org/> (<https://tour.golang.org/>)
- https://golang.org/doc/effective_go.html (https://golang.org/doc/effective_go.html)
- <https://golang.org/ref/spec> (<https://golang.org/ref/spec>)
- <https://golang.org/pkg> (<https://golang.org/pkg>)

Thank you

Serge Léger

Prototype Developer, National Research Council

<https://github.com/sergeleger/mug> (<https://github.com/sergeleger/mug>)

[@sergeleger](http://twitter.com/sergeleger) (<http://twitter.com/sergeleger>)

