Quick Introduction to System Tools Programming with Go

Chris McEniry
Sony Network Entertainment
@macmceniry
http://blog.corgalabs.com/
Administivia
Goals

- Gain a novice understanding of the Go Programming Language
- Know how to apply that understanding to common work needs that Sysadmins see
Some things to know before we get started

- Go
  - Why?
  - Opinionated
  - How to get started

- The Tutorial
  - Conventions
  - Examples
Why Go?

- Simple enough language to keep it in your head
- Built-in Concurrency Primitives
- Fast Compilation
- Simplified Deployments
- Flexibility of a dynamic language. Safety of a static
- Built in garbage collection - avoid memory games
Opinionated

- There’s a way to do whitespace: `gofmt`
- Very few control structures (8 statement terminating keywords)
- Just one loop: `for`
- No semicolons
- No extra parenthesis in conditionals
- All errors, not warnings
- Must use what you declare
- Everything is initialized
- “Order not specified” really does mean you can’t rely on the return order
Getting Started

- Download go

- Set Environment Variables
  - `GOROOT` : Location of the Go tools
  - `GOPATH` : Location of the Workspace

- Workspace: Working directory for a project
  - `bin`, `pkg`, `src`
  - Workspace is not what goes under version control
Invocations

- **go run**: Execute a go file
- **go build**: Build a standalone object file or executable
- **go install**: Install from the source tree a package or binary
- **go doc**: Print out embedded documentation for a package
- **godoc**: Extracts and generates documentation for a package (and can host a web server for it)
- **go fmt**: Applies standard formatting to a package
- **gofmt**: Applies standard formatting to stdin
- **go get**: Download module src from an upstream repository
- **go help**: Command line help
Conventions used in this presentation

- Informational text - Arial
- Code - Chalkboard

```javascript
var (  
  foo = 1  
  bar = 2
)
```

- Shell - Courier

```
echo "hello FOO" | \  
  sed -e 's/FOO/world;
```

Kata

- Using a real example to convey the concepts
- Broken into multiple sections
  - Each focusing on a particular method
- Four ways to follow along:
  - Read along on the slides
  - Read the github code [https://github.com/cmceniry/gotutorial](https://github.com/cmceniry/gotutorial)
  - Run the VM being passed around on USB stick. Login: `golang/golang`
  - `vagrant up` from the main code repository
- Keep it up to date with the tutorial:
  ```
  go get -u github.com/cmceniry/gotutorial
  ```
- Example name will be noted in the bottom right corner of the slide. Corresponds with the directory of the main file of the exercise (and optionally : to indicate a module file)
The Real World Example

- Mapping IO activity for Oracle ASM
- Problem
  - Oracle ASM uses disk labels (e.g. ASM001, ASM002)
  - Linux (and performance tools) uses the device names (e.g. dm-2, loop2)
  - Always have to do the mental conversion

```
[golang@golang ~]$ iostat
...
Device: tps Blk_read/s Blk_wrtn/s Blk_read Blk_wrtn
loop0 0.24 1.87 0.00 1409 0
loop1 0.24 1.87 0.00 1409 0
loop2 0.24 1.87 0.00 1409 0
loop3 0.24 1.87 0.00 1409 0
loop4 0.24 1.87 0.00 1409 0
loop5 0.24 1.87 0.00 1409 0
loop6 0.24 1.87 0.00 1409 0
loop7 0.24 1.87 0.00 1409 0
```
The Real World Example

- Would be nice to...
  - short circuit the mental conversion (i.e. iostat but with the device name substituted)
  - have some other output formats

[golang@golang ~]$ asmiostat

Device:          tps  Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
ASM001           0.25   2.01      0.00        1409      0
ASM002           0.25   2.01      0.00        1409      0
ASM003           0.25   2.01      0.00        1409      0
ASM004           0.25   2.01      0.00        1409      0
Katas

- #1: Exiting (comments, package, import, os, main, exported functions)
- #2: Output (fmt)
- #3: Reading stdin (bufio, short variable declaration, if, multiple return value pattern, panic, for, break)
- #4: Regular Expressions (data types, functions, regexp, map)
- #5: ENV Variables (arrays, slices, syscall)
- #6: Command Line Arguments (os.Args, flag, )
- #7: Executing Commands (structs, fields, methods, os/exec)
- #8: Line Parsing (scanf)
- #9: Reading files (packaging)
- #10: Files and Directories (pointers, nil, interfaces, type assertions, goroutines, channels)
Kata #1: Exiting

- All good scripts return some useful exit code
- 0 means “everything’s OK”
- Anything else means “something’s not OK”
- Bash checking:

```bash
test $? -eq 0
```

- Without an explicit exit, go returns 0
1. Comments can be on any line and begin after // Also can use /* */ for multiline comments

2. Go uses packages for namespaces to avoid conflicts
The Go Package Model

- Packages break up pieces of work
- Each file requires `package` definition
- Multiple files can be in the same package
- `import` tells Go to use another package
- Definitions inside a package can be internal only or allowed to be used externally
  - Internal only definitions are named variable name starting with a lowercase character
  - External definitions are named starting with an uppercase character
- Package name matches directory to all
3. `import` references a library external to this package

`os` is the platform-independent interface to operating system functionality

Dir, File, Pid, UID, GID, etc.
4. Function definition for `main.main`. Execution as a command requires `main.main`

5. Exporting functions to other packages, and calling externally exported functions requires Capital letter
Run It!

```bash
[golang@golang gotutorial]$ go install  
> github.com/cmceniry/gotutorial/commands/exit
[golang@golang gotutorial]$ ./bin/exit
[golang@golang gotutorial]$ echo $?  
27
[golang@golang gotutorial]$ go run  
> src/github.com/cmceniry/gotutorial/commands/exit/exit.go
exit status 27
```
Kata #2: Output

- Two ways to output from a command
  - Return code
  - stdout

- In general, think about what’s going to use the output to decide how to format it. Human parsing and machine parsing is very different (as we’ll see in a later Kata).
fmt - All things formatting

- `fmt.Print()` : Outputs a line of text. Spaces added between operands.
- `fmt.Println()` : Mostly the same as above, but adds a newline to the end (also has some subtle differences on spacing between operands).
- `fmt.Printf()` : Output formatted text (only does newline if you include \n). If you really care about the output, this is the one to use.
package main
import "fmt"
import "os"

func main() {
    fmt.Println("Welcome to Go!") // (1)
    os.Exit(0)
}

1. Output the string “Welcome to Go!” with a new line.
Run It!

[golang@golang gotutorial]$ go install \
> github.com/cmceniry/gotutorial/commands/helloworld
[golang@golang gotutorial]$ $GOPATH/bin/helloworld
Welcome to Go!
Kata #3: Reading stdin

- Reading from the stdin allows you to read from other commands via pipe
- Example `cat`
package main
import "fmt"
import "os"
import "bufio"

func main() {
    stdin := bufio.NewReader(os.Stdin)                  // (1)
    if line, err := stdin.ReadString('
'); err == nil { // (2)
        fmt.Print(line)
    } else { // (3)
        panic(err) // (4)
    }
    os.Exit(0)
}
var

- All identifiers/variables have to be defined before you can use them (or defined when you use them).

    "var" name type [= initialvalue]

    `var line string = "my string"`

- Can also use the "short variable declaration" :=
  - Compiler infers the type
  - More idiomatic

    `line := "my string"`

- Note: Can’t redeclare a variable inside the same scope
1. := short declaration of stdin to be whatever type bufio.NewReader returns

**bufio** - Buffered I/O package

Regular file reads/writes happen here

**os.Stdin** - package os variable which reflects the underlying stdin
if

"if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ]

- Simple two branch conditional (if/else)
- Has optional simple statement
  - Heavily used for call and error checking
- Scope of variables in if continue till the end of the if block (including the else statement)
  - Watch out for trying to call/error check and use it later
  - Watch out for redeclaring and expecting them to be the same
2. See the if statement in action

```
stdin := bufio.NewReader(os.Stdin) // (1)
if line, err := stdin.ReadString('
'); err == nil { // (2)
    fmt.Print(line)
} else { // (3)
    panic(err) // (4)
```

**SimpleStmt**: `line, err := stdin.ReadString('\n')`

`stdin.ReadString` reads up to and including that character

Multi-value return: Returns the string, and error
stdin := bufio.NewReader(os.Stdin)  // (1)
if line, err := stdin.ReadString(\n'); err == nil { // (2)
    fmt.Println(line)
} else { // (3)
    panic(err) // (4)
}

3. **else** is the second branch option
4. **panic**

Handles “extreme” runtime errors

Could be used as exception handling. Unlike other languages, this isn’t a common pattern in Go. Use the `err` return pattern and resolve internally.
Kata #3, part II

- So far, we’ve only read the first line of input
- Go has a single looping control word: `for`
for

- Go has a single looping control word

  ```go
  for [ InitStmt ] ";;" [ Condition ] ";;" [ PostStmt ]
  ```

  ```go
  for x := 0; x < 10; x ++ {
  for ; true ; {
  for {
  ```

- There are some more options (we’ll come back to that)
func main() {
    bio := bufio.NewReader(os.Stdin)
    for { // (1)
        if line, err := bio.ReadString('\n'); err == nil {
            fmt.Print(line)
        } else {
            break // (2)
        }
    }
}

1. Using the implied `for ; true ; {` form

Which means it would never get out
func main() {
    bio := bufio.NewReader(os.Stdin)
    for { // (1)
        if line, err := bio.ReadString('\n'); err == nil {
            fmt.Println(line)
        } else {
            break // (2)
        }
    }
}

2. **break** exits out of the current (innermost) loop

Not shown here, but **continue** immediately goes to the next iteration of the current loop
Kata #4: Regular Expressions

- Always need to replace text data
- Regular Expressions used everywhere
- To help us with this Kata, we’re introducing a couple additional Go concepts
  - Types
  - Functions
Go Types (Part I)

- Every variable has a Type
  - Can represent an actual memory structure
  - Or a behavior contract (where Go dynamics come from)
Go Types (Part I)

- Multiple categories of types
- First look at traditional data types
  - Boolean (bool): true, false
  - Numeric
    - String (string): immutable series of bytes
    - len (mystring): Builtin function that returns the mystring’s size
    - stringa + stringb: Creates a third string
- Can create user defined types
  - type mytype uint32
Numeric Types

- `uint8` the set of all unsigned 8-bit integers (0 to 255)
- `uint16` the set of all unsigned 16-bit integers (0 to 65535)
- `uint32` the set of all unsigned 32-bit integers (0 to 4294967295)
- `uint64` the set of all unsigned 64-bit integers (0 to 18446744073709551615)
- `int8` the set of all signed 8-bit integers (-128 to 127)
- `int16` the set of all signed 16-bit integers (-32768 to 32767)
- `int32` the set of all signed 32-bit integers (-2147483648 to 2147483647)
- `int64` the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
- `float32` the set of all IEEE-754 32-bit floating-point numbers
- `float64` the set of all IEEE-754 64-bit floating-point numbers
- `complex64` the set of all complex numbers with float32 real and imaginary parts
- `complex128` the set of all complex numbers with float64 real and imaginary parts
- `byte` alias for `uint8`
- `rune` alias for `int32`
Go Functions

- Identifier, arguments, return value(s)
- Pass by value (creates a copy every time)

```
“func” FunctionName Signature FunctionBody

Signature = “(“ [ParamIdentifier Type [“,“ ParamIdentifier Type]+) “)” [Type [“,“ Type]+]

FunctionBody = “{“ [Statements]+ “}”
```

- Exits at end of body or with `return`
  - If return types are declared in signature, must be returned
  - `return` can be (almost) anywhere in body
- Package exporting rules apply (capitalize when you want to make available in a library)
Go Functions

```go
func noop()
func funca(stra string) string
func concat(stra string, strb string) string
  * Can combine similar if they’re in the same order:
    func concat(stra, strb string) string
func read(filename string) (int, string, error)
  * Multiple return values: # of bytes read, actual bytes as string, error
```
Regular Expression

- import “regexp”
- **MatchString** : regexp grep which returns true or false
- **Compile** : Used for any higher level regexp actions (Findall, Replace, etc)
  - returns a **Regexp** type to be operated on
- All Regexp functions have two forms: bytes and string
  - We’re working on string. bytes are left as an exercise for the reader.
- No syntactic sugar (i.e. */FO*/ doesn’t work)
func substitute(before string) string { // (1)
    if re, err := regexp.Compile("loop1 "); err == nil { // (2)
        return re.ReplaceAllString(before, "ASM001") // (3)
    }
    return before
}

fmt.Print(substitute(line)) // (4)

1. Function declaration with Signature and beginning of block

In this case, we’re just wrapping the regexp actions so we can more readily use it in the line.
```go
func substitute(before string) string {
    if re, err := regexp.Compile("loop1 "); err == nil {
        return re.ReplaceAllString(before, "ASM001")
    }
    return before
}

fmt.Println(substitute(line))
```

2. Compile to generate the regexp to work with
3. ReplaceAllString operates such as s///g

Return the string with substitutions or fall through to return the unaltered string

ReplaceAllString, not ReplaceAllStrings - versus bytes
Bonus: Variadic Functions

- Functions which take a variable number of parameters
- Go only allows this as the final argument
- Denoted with ... preceding the last argument type

```go
func Printf(format string, a ...string) string

fmt.Printf("%s", "string1", "string2")
fmt.Printf("%s%s", "string1", "string2", "string3", "string4", "string5")
```
Kata #4, Part II

- Must like the straight cat-like behavior of Kata #3, we’re only handling one substitution
- Could copy multiple substitutes - ugh

```go
if re, err := regexp.Compile("loop1 "); err == nil {
    if re, err := regexp.Compile("loop2 "); err == nil {
        if re, err := regexp.Compile("loop3 "); err == nil {
            if re, err := regexp.Compile("loop4 "); err == nil {

- Should use `for` but that depends on being able to iterate over `something`
```
Maps (Go Types, Part II)

- A map is your traditional key/value structure
- “Unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type.”
- `var subs map[string]string`

Key Type  Element Type
Maps (Go Types, Part II)

- Initializing can be done using the `{}` form

```go
type map[string]string = map[string]string

var subs map[string]string = map[string]string{
    "loop1": "ASM001",
    "loop2": "ASM002",
    "loop3": "ASM003",
    "loop4": "ASM004",
}
```

- Note: The comma on the last item is mandatory (also nicer on line diffs)
Maps (Go Types, Part II)

- Initialization can also use the **make** function
  - **make** allocates the memory for it
  - maps are flexible in that they can grow or shrink dynamically without more **make** calls

- Additions are done by referencing [key]

```go
var subs map[string]string
subs = make(map[string]string)
subs["loop1"] = "ASM001"
subs["loop2"] = "ASM002"
subs["loop3"] = "ASM003"
...```

```
Maps (Go Types, Part II)

- Initialization can be done with the short variable declaration and make

```go
subs := make(map[string]string)
subs["loop1 “]: “ASM001”
subs["loop2 “]: “ASM002”
subs["loop3 “]: “ASM003”
```

- Remove entries by `delete(mapvar, key)`

```go
delete(subs, “loop1 “)
delete(subs, “loop2 “)
```
Maps (Go Types, Part II)

- Can dereference by keys. If it doesn’t exist, returns “zero” initialized value for that type.

```go
element := mapvar["key"]
```

- Can check for existence using the same form (with an extra return value)

```go
if element, present := mapvar["key"]; present {
```

- Can iterate over using the range operator

```go
for key, element := range mapvar {
    fmt.Println(key, element)
}
```
1. Using the range form to iterate over all
2. Note that `before` is being used as the temporary variable throughout the entire function.

   Go is pass-by-value so it is a locally scoped copy of the original value.

   *Parameters are not constants, and do not affect the value of what the function was called with.*

```go
func substitute(before string) string {
    for needle, sub := range subs { // (1)
        if re, err := regexp.Compile(needle); err == nil {
            before = re.ReplaceAllString(before, sub) // (2)
        }
    }
    return before
}
```
Kata #5: ENV Variables

- The ENV Variables
- Can be used as another method of process communication (invoker -> invokee)
Handling single ENV variable

- syscall.Getenv() : Retrieves one ENV variable
- syscall.Setenv() : Sets an ENV variable
- syscall.Environ() : Returns an array of all of the ENV variables
- os.Environ() : Same as syscall.Environ()
1. If you know the variable name, you get value directly.

    Behaves like a map in this perspective, but it is a function call in `syscall`.

That works for one variable, but how do you get *everything*?
An **array** is a numbered sequence of a single type

- E.g. int, int, int, int, int, int, int

- It has its own type signature (and that is effectively its own type)

Variable type signature is `[size]singletype`


Variable value is allocated at creation

```go
define a [10]int // [0,0,0,0,0,0,0,0,0,0]
```
Arrays, Slices (Go Types, Part III)

- Get size with the `len()` function
  ```go
  fmt.Println(len(a)) // 10
  ```

- Indexed by integers 0 thru `len(array)`-1
  ```go
  fmt.Println(a[1]) // 0
  a[1] = 42
  ```

- Outside that == panic
Arrays, Slices (Go Types, Part III)

- A subset of an array is a **slice**
- Basically a reference to which element to start with, and a length
- Type signature is `[singletype]` - NOTE: No size value in the signature
- Created by
  - The `[:]` syntax
    ```go
    b := a[5:8]
    ```
- The **make** function
  ```go
  var c []int
c = make([]int, 10)
  ```
- Separates the interaction to an array from the storage of the array
- Useful when you want to reuse code where the specific length of the array doesn’t matter, only the fact that it is an array of a specific type
  - Equivalent to how you treat arrays in ruby, python, perl
Arrays, Slices (Go Types, Part III)

- `len()` is similar to arrays
- Slices also have a `cap()` - capacity: how many elements the slice can hold
- All slices have an underlying array to them - the slice is a reference
1. Dereference the first one
2. Iterate over the array
3. Causes a panic (unless your ENV is really, really large)

Note: You can get the variables via `Environ()`, but you have to set them using `Setenv()`
More info on Arrays, Slices

- There are some subtle behaviors here. The underlying representations can have some gotchas for what really is the slice versus what really is an array.

- Here’re some useful references:
Bonus : Variadic Functions (Part II)

- Can flip the behavior of the variable arguments when calling variadic functions
  - Can supply a *slice* in place of a variadic argument
- Only allowable for the variadic - final - argument

```go
func Printf(format string, a ...string) string
fmt.Printf("%s%s%s%s%s", "string1", "string2", "string3", "string4", "string5")

args := []string{"string1", "string2", "string3", "string4", "string5"}
fmt.Printf("%s%s%s%s%s", args...)
```
Kata #6: Command Line Arguments

- Parameters of the command line
  - Give flexibility to commands
- In our example, going to use it to specify outputs
  - First pass, just turning on and off processing
Go’s command line parsing

- 3 main options
  - `os.Args`: Array of the command name and all of the command line arguments. Can parse it yourself
  - `getopt` packages: Opinionated Go of the stdlib says “these are ambiguous,” but they have been ported independently
  - `flag` package: Go’s command argument parser
Pointers (Go Types, Part IV)

- A variable is a placeholder for a chunk of memory
- A pointer is a placeholder to the *address* of that chunk of memory.
- nil is the pointer that points nowhere
- c-style syntax
  - Declare: `*type`
  - Reference: `*var` - On a pointer to get the value at the address
  - Dereference: `&var` - On a nonpointer to get the value of the address

```go
var a int, b *int
da = 5
b = &a
c := &a
```
Pointers? Why?

- Functions are pass-by-value: A copy of the value is passed in to the parameter variable.
- If you make changes to the variable inside of the function, it doesn’t impact the original variable.
- (To make a long story short…) Sometimes you want to do that; sometimes you don’t.
- Since the pointer is the address and the address is passed by value, it is still pointing to the same spot. When you make changes to the spot that that pointer is pointing at, the underlying data changes.

=> If you want to make changes, pass around pointers. If not, continue on as normal.
flag

- Multi-character
- Typed (bool, int, string, int64, float64, etc) and Valued
  - e.g. “-x=1234”, “-y=mysting”
  - bool allows for traditional flag (“-x”)
- Only 1 argument allowed per dash
  - e.g. “-x” and “-y”, not “-xy”
- Doesn’t distinguish between single and double dash
  - e.g. “-x” == “—x”
- All flags need to be first on the command line
  - Once it hits an arg that it doesn’t know, it stops
1. Initialize the flag system. `flag.ExitOnError` indicates that when it fails what to do (exits -2)

2. Create a boolean command line parameter named `dontprocess`, bind it to the variable `dontprocess`, set default to false, and set the help message.

3. Parse using the flag system. Still uses `os.Args` to get the command line args, but strips off the command name using the slice operator (from 1 until the end).
Kata #7 : Executing Commands

- Every script writer at some point needs to be able to call out to other scripts
- Like regexp, no syntactic sugar for executing commands
- Two main mechanisms
  - High level: `os/exec`
  - Low level: `os.StartProcess()`
struct (Go Types, Part V)

- The **struct** is used to create combinations of other related types

```go
type MyType struct {
    MyItem string
    MyCount uint32
}
```

- Can reference internal fields. With an instance of **MyType** named **myinstance**:

```go
myinstance.MyItem
myinstance.MyCount
```
struct (Go Types, Part V)

- Struct is also used for part of Go’s OOP(ish) style with functions
- We saw this in the Regular Expressions, Flags

```go
type Regexp struct {
    // contains filtered or unexported fields
}
```

- `re` is a (pointer to a) Regexp struct. `f` is a (pointer to a) FlagSet struct.

```go
re, err := regexp.Compile("FO*")
```

- This is used EVERYWHERE
struct (Go Types, Part V)

- structs can also have functions associated with them, called **methods**.
  - Again, already been using these (e.g. **ReplaceAllString**)
- Function signature refers to the struct involved
  - Can be pointer in addition to raw struct
  - Referenced in the same way when using (so not too many * or -> scattered around)
- Can operate on data in the struct - where the OOP comparison comes from

```go
func (re *Regexp) ReplaceAllString(src, repl string) string
```
import "os/exec"

- myexec := exec.Command(name, arg, arg, arg, ...)  
  - returns `Cmd` type after setup
  - NOTE: `arg` is passed to the kernel exec, not shell interpreted (i.e. `/etc/*` doesn’t get expanded)

- myexec.Start()  
  - Starts the command in the background and continues on
  - Can wait for it to finish using `myexec.Wait()`

- myexec.Run()  
  - Starts the command but waits for it to complete
import "os/exec"

func main() {
    args := []string{"-xt", "/dev/loop1", "/dev/loop2", "/dev/loop3", "/dev/loop4"} // (1)
    cmd := exec.Command("/usr/bin/iostat", args...) // (2)
    cmd.Run()
}

Notice: There’s no output from this run.

The inputs and outputs for this command were never setup. Go does not automatically assume stdin/stdout and attach it to the current terminal.

1. Here we’re setting up a slice for the command line args.
   Would probably have done /dev/loop* but the arguments are not shell interpreted
2. Using the variadic form of exec.Command
Getting to the output

- Can set input/output to stdin/stdout by accessing `cmd`'s field `Stdout` and connect it to this running process’s STDOUT

```go
cmd := exec.Command("/usr/bin/iostat", args...)
cmd.Stdout = os.Stdout
cmd.Run()
```

- Or can create an internal pipe by invoking `cmd`'s method `StdoutPipe()`
  - which is what we want since we’re going to have to manipulate the output

```go
cmd := exec.Command("/usr/bin/iostat", args...)
stdout, err := cmd.StdoutPipe()
cmd.Run()
```
1. Use the Start method to execute in the background
2. Want to wait for a bit of time for iostat to send output.
buf := make([]byte, 2048)
cmd := exec.Command("/usr/bin/iostat", args...)
stdoutpipe, err := cmd.StdoutPipe()
if err := cmd.Start(); err != nil { // (1)

time.Sleep(1*time.Second) // (2)

n, err := stdoutpipe.Read(buf) // (3)

... cmd.Wait() // (4)
fmt.Print(string(buf[:n])) // (5)

3. Read from the pipe into a buffer (stdout).
   Careful: Have to read before the command finishes
   When the command finishes, the pipe will close

4. Wait for the command to finish for good hygiene.
5. Slice out only the actual returned output, convert that from a byte array to a string, then output it. (Otherwise, it'll output as an array format).
Kata #8: Line Parsing

- Regex is good, but sometimes you want to pull in more
- Can use Regexp grouping, but not always the best route
- Welcome to `scanf`
  - In `fmt` (complementary to `printf`)
iostat -xt /dev/loop[1234]

- Choose the extended format and adding a time reading since it's a bit more descriptive and identifiable (even if we're only using a subset)

- Two areas to try to parse
  - Timestamp
  - Device metrics

```
Linux 2.6.32-358.el6.x86_64 (golang) 02/23/2014 _x86_64_ (1 CPU)

02/23/2014 05:27:38 PM
avg-cpu: %user %nice %system %iowait %steal %idle
         0.16  0.00  0.12  0.03  0.00  99.68

Device:     rrqm/s  wrqm/s  r/s  w/s  rsec/s  wsec/s avgrq-sz avgqu-sz  await  svctm  %util
loop1  0.05  0.40  0.18  0.40   7.91   5.81   23.41   0.00  1.28   0.99   0.06
loop2  0.00  0.00  0.15  0.73   7.34   5.81  14.90   0.00  2.35   0.63   0.06
loop3  0.00  0.00  0.00  0.00   0.04   0.00   7.96   0.00  1.33  1.33   0.00
loop4  0.00  0.00  0.01  0.00   0.07   0.00   7.98   0.00  1.21  1.10   0.00
```
1. Set the matching format. `%` options are largely the same as printf formatting:
   http://golang.org/pkg/fmt/
2. Call `scanf` and use use the pointer dereference so that it updates the values for each variable.
3. `fmt.Sprintf` is used to copy to a formatted result to a string (stored so it can be used later).

4. `continue` is used to go to the next iteration of this loop. No need to continue processing since we're set one value from this line.
5. Again, pointers are used to set the variables. Heavy use of the `ign`ore variable to discard the data

6. `ts` used from earlier to complete the output of the line.

   Slightly different use of `subs` - doesn’t include spaces to match up
Kata #9 : Reading Files

- Always have some kind of file to read
- Behaves just like stdin, but you have to read it first
  - “Everything’s a file”
- We’re going to use it to move our compiled in definitions of the device mapping out into a mapping file.
1. **os.Open** is the general form for getting a file handle.

2. **defer** takes a function and invokes it the moment the surrounding function returns
   - Even if panicked
   - It’s good for any cleanup functions

3. And now we can fall back to how we treated **Stdin**

```go
file, err := os.Open(filename) // (1)
if err != nil {
    panic(err)
}
defer file.Close() // (2)
bio := bufio.NewReader(file) // (3)
```
Packaging

- So far, have really been just relying on single files, but our code is getting large enough and we’re adding in and removing pieces modularly (stdin vs cmd, regexp vs formatted output) that it makes sense to start breaking it out.
Packaging

- **package** name corresponds to directory name
- github.com/cmceniry/gotutorial/mapping
  - Would prefix with **package mapping**
- File name inside that directory doesn’t matter as much. All files in a package (in the directory and with the proper **package** header) will be built together.
- Files in a directory that use a different **package** header will be ignored. We’re already been doing this to some degree with the commands directories.
1. In our command file, we have to reference the module that we’re creating. This matches what’s in the src directory in the `$GOPATH` workspace.

2. Reference the module and identifiers in it as you would with the standard libraries.
1. Must specify the **package** name

2. Export (capitalized) function. Following the data, err convention for the return value

3. Same opening of the file as before…

4. But in this case, we’re returning the error if it arises instead of just panicking. Convention is to return the error from libraries so that the program can decide if it needs to panic or not. Avoid panicking directly in libraries
Kata #10: Directories and Filestat

- Need to navigator around the file system
- Need to be able to list files in a directory
- Need to be able to look at file metadata
Interface Type (Go Types, Part VI)

- Go's form of dynamic typing
- It's a description of behavior, not of storage or structure
- Collection of methods which must be satisfied by a type
  - versus struct methods which are what satisfy these
- When a variable is declared of an interface type, it’s saying “where used, this variable will be able to handle these methods somehow”
- The interface is a minimum set of methods. structs can implement more methods, but to use them from an interface variable, you have to explicitly type assert
- All types satisfy the empty `interface{}` so it can be used anywhere (but always has to be asserted before it can be used)
Interface Type (Go Types, Part VI)

- Seen this in `fmt` already

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

func MyPrintf(format string, a ...Stringer) string

MyPrintf(`%s%d%s%f%s`, `string1`, 100, `string3`, 0.001, `string5`)
```

- As long as concrete type has a method to convert it to a string, it can be used inside as an argument to `MyPrintf`
- This allows `MyPrintf` to not care about how that happens, only that it happens
- Actual concrete type has to care about how that happens
Interface Type (Go Types, Part VI)

- `os.Open` used on all files and directory handles

```go
var f *File
f, err = os.Open("/")
```

- (*File) `Readdirnames` gets the list of file names

```go
var filenames []string
filenames, err = f.Readdirnames(0)
```

- (*File) `Readdir` gets `FileInfo` for files in directory

```go
var info []FileInfo
info, err = f.Readdir(0)
```
type FileInfo interface {
    Name() string       // base name of the file
    Size() int64        // length in bytes for regular
                        // files; system-dependent for
                        // others
    Mode() FileMode     // file mode bits
    ModTime() time.Time // modification time
    IsDir() bool        // abbreviation for
                        // Mode().IsDir()
    Sys() interface{}   // underlying data source (can
                        // return nil)
}
Interface Type (Go Types, Part VI)

- `Sys()` in the previous interface returns the empty interface. Go now treats the value returned by `Sys()` as something with no fields or methods, when, in reality, there’s some underlying value that has fields and methods.

- A Type Assertion is a way of telling go to treat that returned value as something else.

```
a := myfileinfo.Sys()
b := a.(*syscall.Stat_t) // forces a into *syscall.Stat_t
fmt.Println(b.Atim, b.Mtim, b.Ctim)
```
Interface Type (Go Types, Part VI)

- The Type Switch can be used to test what the underlying type is an execute conditionally

```go
switch b := a.(type) {
    case *syscall.Stat_t:
        fmt.Println("Stat_t")
    default:
        fmt.Println("Unknown result")
}
```
ASM to device mapping

- /dev/oracleasm/disks
  - Lists all ASM labeled disks
  - Has the same major/minor numbers as /dev devices
- Can map the devices by iterating through all of the /dev/oracleasm/disks entries and all of the /dev entries and matching up the major/minors
1. `os.Open` is used for directories as well as files
2. `file.Readdir` used to get the `[]FileInfo`
3. `Rdev` is not in the empty interface returned by `Sys()`, so we must do a type assertion to `*syscall.Stat_t` to be able to use the data in it. This is checked at compile time and will error if it doesn’t match.
4. `findDev` does the same lookup in the `/dev` directory and matches up the `rdev` value
Kata #10, Part II

- It is unlikely, but the dev mapping can change while the command is running. It would be nice if that was updated.

```
for {
  if buf, err := out.ReadString("\n"); err != nil {
    panic(err)
  } else {
    fmt.Print(substitute(buf))
  }
}
```

- We could call `GenerateSubs()` every time through the loop, or check a counter or timing.
Goroutines

- A goroutine is an independent concurrent thread of control within the same address space
- Ideal for parallelization
- Also good for background routines
- Easy to start

```go
go myfunc()
```

- Main program control is handled by the `main()` routine
  - I.e. if it finishes, the program will exit, and all other routines die
func UpdateMappingsRegularly() { // (1)
    for {
        if err := GenerateSubs(); err != nil {
            panic(err)
        }
        time.Sleep(5*time.Second) // (2)
    }
}
We’ve introduced a race condition. The timing really depends on how long that first mapping lookup takes versus how fast iostat starts up.

We can fix this by call \texttt{GenerateSubs()} before anything else, or we can send a signal from the subs goroutine to the main one.
Channels

- It’s a pipe
- “Mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type”
- It’s also a Type of its own and can be passed around
Channels

- Must be made first

```go
mychan := make(chan int)
```

- Sending into the channel (sender)

```go
mychan <- 5
```

- Receiving from the channel (receiver)

```go
myvar := <-mychan
```
Blocking

- Without anything else, a channel will block until the other side has something
  - Senders block until a receiver takes
  - Receivers block until a sender transmits
- Can create buffered channels so that it will not block until the buffer is full/empty

```go
mychan := make(chan int, 10)
```

- Can put 10 entries before a sender will block
1. Create a new channel that passes bool. Since we didn’t specify a size to the channel (e.g. `make(chan bool, 42)`) this will block.

2. Start the updater and pass it the channel.

3. Receiver a value from that channel. As said, this will block until it gets a value.

4. Continue on with everything
func SignalUpdater(done chan bool) { // (1)
    var first = true
    for {
        ...
        if first { // (2)
            done <- true // (3)
            first = false
        }
    }

1. Updated function signature which shows it getting passed a channel that passes bool
2. Only run this the first time through the loop.
3. Send **true** into the channel. If nothing is listening on the other side, this will block, which is why we only want to do it once.
Run It!

[golang@golang devs3]$ devs3
Linux 2.6.32-358.el6.x86_64 (golang) 02/24/2014 _x86_64_ (1 CPU)

02/24/2014 01:10:50 PM
...
ASM001     0.00  0.00  0.04  0.00  0.30  0.00  7.96  0.00  1.55  1.55  0.01
ASM002     0.00  0.00  0.04  0.00  0.30  0.00  7.96  0.00  0.80  0.80  0.00
ASM003     0.00  0.00  0.04  0.00  0.30  0.00  7.96  0.00  1.78  1.78  0.01
ASM004     0.00  0.00  0.04  0.00  0.30  0.00  7.96  0.00  1.76  1.76  0.01
...
02/24/2014 01:10:55 PM
...
ASM001     0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
ASM002     0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
ASM003     0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
ASM004     0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

- Much better
Bonus: Go Concurrency

- Like any time concurrency is introduced, complexity comes as well
- Don’t directly manipulate
  - Add channels for passing signals around (kill channel)
- Channels do block
  - But can use `select` to work around
Go Concurrency Articles

- http://blog.golang.org/pipelines
- http://dave.cheney.net/2013/04/30/curious-channels
- http://rcrowley.org/articles/golang-graceful-stop.html
- http://stackoverflow.com/questions/6807590/how-to-stop-a-goroutine
Homework

- We’ve created a bunch of separate tools
- We’ve use command line options to invoke different input and output options
- Part I: Convert the various commands into packages that could be all called from the same command using flag to select
- Part II: Using go routines and channels, abstract out inputs and processing/outputs into their own goroutines and use channels to pass the lines from one to the other
Additional Resources

- http://golang.org/ref/spec
- http://golang.org/doc/faq
- http://tour.golang.org/
- http://gobyexample.com
- http://blog.golang.org/
- https://gophercasts.io/
- https://gocasts.io/
Thank You!

Chris McEniry
Sony Network Entertainment
@macmceniry
http://blog.corgalabs.com/