Programming in Go

Matt Holiday
Christmas 2020

Input/Output

Standard I/O

Unix has the notion of three standard I/O streams

They're open by default in every program

Most modern programming languages have followed this convention:

- Standard input
- Standard output
- Standard error (output)

These are normally mapped to the console/terminal but can be *redirected*

```
find . -name '*.go' | xargs grep -n "rintf" > print.txt
```

Formatted I/O

We've been using the `fmt` package to do I/O

By default, we've been printing to standard output

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("printing a line to standard output")

    fmt.Fprintln(os.Stderr, "printing to error output")
}
```

A whole family of functions

The `fmt` package uses reflection and can print anything;
some of the functions take a *format string*

```
// always os.Stdout

fmt.Println(...interface{}) (int, error)
fmt.Printf(string, ...interface{}) (int, error)

// print to anything that has the correct Write() method

fmt.Fprintln(io.Writer, ...interface{}) (int, error)
fmt.Fprintf(io.Writer, string, ...interface{}) (int, error)

// return a string

fmt.Sprintln(...interface{}) string
fmt.Sprintf(string, ...interface{}) string
```

Format codes

The `fmt` package uses format codes reminiscent of C

```
%s   the uninterpreted bytes of the string or slice
%q   a double-quoted string safely escaped with Go syntax
%c   the character represented by the corresponding Unicode code point

%d   base 10
%x   base 16, with lower-case letters for a-f
%f   decimal point but no exponent, e.g. 123.456
%t   the word true or false

%v   the value in a default format
     when printing structs, the plus flag (%+v) adds field names

%#v  a Go-syntax representation of the value
%T   a Go-syntax representation of the type of the value

%%   a literal percent sign; consumes no value [escape]
```

Read the godoc, Luke: https://golang.org/pkg/fmt/

Format code examples

A few examples:

```
a := 12
b := 345
c := 1.2
d := 3.45

fmt.Printf("%d %d\n", a, b)            //  12 345
fmt.Printf("%#x %x\n", a, b)           //  0xc 159
fmt.Printf("%f %.2f\n", c, d)          //  1.200000 3.45

fmt.Println()

fmt.Printf("|%6d|%6d|\n", a, b)        //  |    12|   345|
fmt.Printf("|%06d|%06d|\n", a, b)      //  |000012|000345|
fmt.Printf("|%-6d|%-6d|\n", a, b)      //  |12    |345   |
fmt.Printf("|%6.2f|%6.2f|\n", c, d)    //  |  1.20|  3.45|
```

Format code examples

%#v and %T are very useful for describing what something is:

```
s := []int{1, 2, 3}
a := [3]rune{'a', 'b', 'c'}
m := map[string]int{"and":1, "or":2}

fmt.Printf("%T\n", s)   // []int
fmt.Printf("%v\n", s)   // [1 2 3]
fmt.Printf("%#v\n", s)  // []int{1, 2, 3}

fmt.Printf("%T\n", a)   // [3]int32
fmt.Printf("%q\n", a)   // ['a' 'b' 'c']
fmt.Printf("%v\n", a)   // [97 98 99]
fmt.Printf("%#v\n", a)  // [3]int32{97, 98, 99}

fmt.Printf("%T\n", m)   // map[string]int
fmt.Printf("%v\n", m)   // map[and:1 or:2]
fmt.Printf("%#v\n", m)  // map[string]int{"and":1, "or":2}
```

File I/O

Package `os` has functions to open or create files,
list directories, etc. and hosts the `File` type

Package `io` has utilities to read and write; `bufio` provides the
buffered I/O scanners, etc.

Package `io/ioutil` has extra utilities such as reading an entire file
to memory, or writing it out all at once

Package `strconv` has utilities to convert to/from string representations

Con•cat•enate

```
package main
import ("io"; "log"; "os")

func main() {
    for _, fname := range os.Args[1:] {
        file, err := os.Open(fname)

        if err != nil {
            log.Fatal(err)
        }

        if _, err = io.Copy(os.Stdout, file); err != nil {
            log.Fatal(err)
        }

        file.Close()
    }
}
```

Reading a file

Wait, what's going on here?

```
if f, err := os.Open(fname); err != nil {
    fmt.Fprintln(os.Stderr, "bad file:", err)
} . . .
```

We often call functions whose 2nd return value is a possible error

```
func Open(name string) (*File, error)
```

where the `error` can be compared to `nil`, meaning no error

Always check the error — the file might not really be open!

Reading a file and calculating its size

```
package main

import ("fmt"; "io/ioutil"; "os")

func main() {
    fname := os.Args[1]
    if f, err := os.Open(fname); err != nil {
        fmt.Fprintln(os.Stderr, "bad file:", err)
    } else if d, err := ioutil.ReadAll(f); err != nil {
        fmt.Fprintln(os.Stderr, "can't read:", err)
    } else {
        fmt.Printf("The file has %d bytes\n", len(d))
    }
}
```

If run on itself (the source file), it prints “The file has 333 bytes”

```
package main

import ("bufio"; "fmt"; "os"; "strings")

func main() {
    for _, fname := range os.Args[1:] {
        var lc, wc, cc int

        file, err := os.Open(fname)

        if err != nil {
            fmt.Fprintln(os.Stderr, err)
            continue
        }
```

```
        scan := bufio.NewScanner(file)

        for scan.Scan() {
            s := scan.Text()

            cc += len(s)
            wc += len(strings.Fields(s))
            lc++
        }

        fmt.Printf(" %7d %7d %7d %s\n", lc, wc, cc, fname)
        file.Close()
    }
}
```