Reading Arguments from the Command-Line

This lesson explains two packages of Go that let us read arguments from the command line.

WE'LL COVER THE FOLLOWING

- With the `os` package
- With the `flag` package
- `go run main.go -h` command
- `go run main.go` command
- `go run main.go -lang="V" -num=42` command

With the `os` package

The package `os` also has a variable `os.Args` of type slice of *strings* that can be used for elementary command-line argument processing, which is reading arguments that are given on the command-line when the program is started. Look at the following greetings-program:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
"fmt"
"os"
"strings"
)

func main() {
  who := "Alice "
  if len(os.Args) > 1 {
    who += strings.Join(os.Args[1:], " ")
```

```
  fmt.Println("Good Morning", who)
}
```

When we run this program, the output is: **Good Morning Alice**. The same output we get when we run it on the command-line as:

```
go run main.go
```

But, when we give arguments on the command-line like:

```
go run main.go John Bill Marc Luke
```

we get **Good Morning Alice John Bill Marc Luke** as an output.

When there is at least one command-line argument, the slice `os.Args[]` takes in the arguments (separated by a space), starting from index 1 since `os.Args[0]` contains the name of the program, os_args in this case.

At **line 10**, we test if there is more than one command-line argument with `len(os.args) > 1`. In that case, all the remaining arguments given by the slice `os.Args[1:]` are joined together at **line 11** and added to **Alice**. The `strings.Join` function glues them all together with space in between. The string `who`, which contains the names of all the people to be greeted, is then printed from **line 13**.

With the `flag` package

The package `flag` has extended functionality for parsing of command-line options, but it is also often used to replace ordinary constants, e.g. when we want to give a different value for our constant on the command-line. A `Flag` is defined in the package `flag` as a struct with the following fields:

```
type Flag struct {
  Name string // name as it appears on command line
  Usage string // help message
  Value Value // value as set
  DefValue string // default value (as text); for usage message
}
```

The usage of the `flags` package is demonstrated in the following program:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
"flag"
"fmt"
)

func main() {
  strPtr := flag.String("lang", "Go", "a string")
  numPtr := flag.Int("num", 108, "an int")
  boolPtr := flag.Bool("truth", false, "a bool")
  var str string
  flag.StringVar(&str, "str", "Crystal", "a string variable")
  flag.Parse()
  fmt.Println("lang:", *strPtr)
  fmt.Println("num:", *numPtr)
  fmt.Println("truth:", *boolPtr)
  fmt.Println("str:", str)
  fmt.Println("tail:", flag.Args())
}
```

Click the **RUN** button, and wait for the terminal to start. There are *three* ways to run the program:

- Type `go run main.go` and press ENTER.
- Type `go run main.go -h` and press ENTER.
- Type `go run main.go -lang="V" -num=42` and press ENTER.

`go run main.go -h` command

The flags are defined for *strings*, *integers* or *booleans* with the respective functions `flag.String`, `flag.Int` and `flag.Bool`. They all return a pointer as you can see from **line 8** to **line 10**. Their first argument is the *flag name*, then comes the *default value*, and then a *help info* string. This is shown when you call this program with `-h`, like:

```
go run main.go -h
```

The following output is shown on the screen:

```
Usage of /tmp/go-build001915860/command-line-arguments/_obj/exe/main:
```

```
  -lang string
        a string (default "Go")
  -num int
        an int (default 108)
  -str string
        a string variable (default "Crystal")
  -truth
        a bool
```

`go run main.go` command

You can also use `flag.StringVar` to define a string flag, with the first argument as a pointer to the string that contains the value. If you call the program without arguments, you get the default values:

```
lang: Go
num: 108
truth: false
str: Crystal
tail: []
```

`go run main.go -lang="V" -num=42` command

By giving flags on the command-line as:

```
go run main.go -lang="V" -num=42
```

the values can be changed:

```
lang: V
num: 42
truth: false
str: Crystal
tail: []
```

All the flags that this program can handle are defined from **line 8** to **line 12**. A call to `flag.Parse()` at **line 13** is necessary to scan the argument list (or list of constants) and set up the flags. After parsing, the flags or constants can be used.

From **line 14** to **line 17**, we dereference the pointers (`*strPtr` and so on) to get their value and print it out.

You can iterate over the flags like this:

```
var s string
for i := 0; i < flag.NArg(); i++ {
  s += flag.Arg(i)
}
```

`flag.Arg(i)` is the i-th argument. All `flag.Arg(i)` are available after `Parse()`. The `flag.Arg(0)` is the first real flag, not the name of the program in contrast to `os.Args(0)`. The `flag.NArg()` is the number of arguments, and `flag.Args()` is an array containing the remaining arguments.

`flag.VisitAll(fn func(*Flag))` is another useful function: it visits the set flags in lexicographical order, calling `fn` for each.

With `flag.Bool` you can make *boolean* flags, which can be tested against in your code. For example, define a flag `processedFlag` which:

```
var processedFlag = flag.Bool("proc", false, "nothing processed yet")
```

Test it later in the code by dereferencing it:

```
if *processedFlag {  // found flag -proc
  r = process()
}
```

Reading arguments from the command line makes programming so much more flexible. In the next lesson, you'll learn how to use a buffer.