The comma, ok Pattern

This lesson briefly discusses the uses of the comma, ok pattern in Go language.

WE'LL COVER THE FOLLOWING

While studying the Go-language, we encountered several times the so-called *comma, ok* idiom where an expression returns two values: the first of which is a *value* or *nil*, and the second is *true/false* or an error. An if-condition with initialization and then testing on the second-value leads to succinct and elegant code. This is a significant pattern in idiomatic Go-code. Here are all cases summarized:

Testing for errors on function return

```
var value Type_value
var err error
if value, err = pack1.Func1(param1); err != nil {
  fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), par
am1)
  return err
}
// no error in Func1:
Process(value)
```

Other places where it is used:

```
os.Open(file), strconv.Atoi(str)
```

The following code will not compile because the scope of the file variable is

The following code will not compile because the scope of the file variable is the if-block only:

```
if file, err := os.Open("input.dat"); err != nil {
  fmt.Printf("An error occurred on opening the inputfile\n" +
  "Does the file exist?\n" +
  "Have you got access to it?\n")
  return // exit the function on error
}
defer file.Close()
reader := bufio.NewReader(file)
```

In this case, declare the variables upfront:

```
var file *os.File
var err error
```

and then `file, err := os.Open("input.dat")` can be replaced by: `file, err = os.Open("input.dat")`.

The function in which this code occurs returns the error to the caller, giving it the value nil when the normal processing was successful and so has the signature:

```
func SomeFunc() error {
  ...
  if value, err = pack1.Func1(param1); err != nil {
    ...
    return err
  }
  ...
  return nil
}
```

The same pattern is used when recovering from a *panic* with `defer`. A good pattern for clean error checking is using closures.

Testing if a key-value item exists in a map

Does this means that `map1` has a value for `key1`? Look at the following snippet:

```
if value, isPresent = map1[key1]; isPresent {
```

```
  Process(value)
}

// key1 is not present
...
```

Testing if an interface variable is of certain type

```
if value, ok := varI.(T); ok {
  Process(value)
}
// varI is not of type T
```

Testing if a channel is closed

```
for input := range ch {
  Process(input)
}
```

or:

```
var input Type_of_input
var err error
for {
  if input, open = <-ch; !open {
    break // channel is closed
  }
  Process(input)
}
```

This is how this pattern resolves many issues in a single line. Similarly, in the next lesson, we'll be studying the defer pattern, used several times before in this course.