Reading from a File

This lesson provides coding examples and their explanations for reading from a file.

WE'LL COVER THE FOLLOWING

Reading data from a file

Files in Go are represented by pointers to objects of type `os.File`, also called **file handles**. The standard input `os.Stdin` and output `os.Stdout` we used in the previous lesson are both of type `*os.File`. This is used in the following program:

main.go

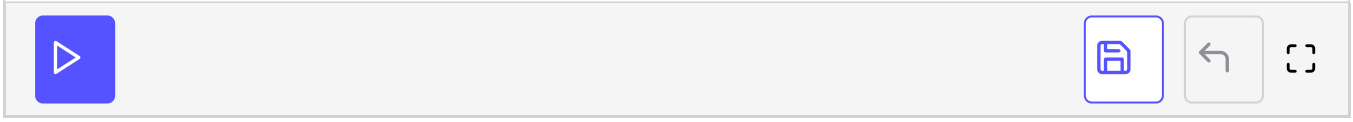
input.dat

```
package main
import (
"bufio"
"fmt"
"io"
"os"
)

func main() {
  inputFile, inputError := os.Open("input.dat")
  if inputError != 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 inputFile.Close()
  inputReader := bufio.NewReader(inputFile)
  for {
    inputString, readerError := inputReader.ReadString('\n')
    if readerError == io.EOF {
      return
  }
    fmt.Printf("The input was: %s", inputString)
  }
}
```



Input from file

In the code above, at **line 10**, a call to `os.Open` creates a variable `inputFile` of type `*os.File`: this is a struct that represents an open file descriptor (a filehandle). The `Open` function accepts a parameter `filename` of type *string* (here as **input.dat**) and opens the file in read-only mode.

This can, of course, result in an error when the file does not exist or the program does not have sufficient rights to open the file: `inputFile, inputError = os.Open("input.dat")`. From **line 11** to **line 17**, we are handling the errors.

The `defer` keyword is very useful for ensuring that the opened file will also be closed at the end of the function with `defer inputFile.Close()` at **line 18**. Here is a code snippet, where this is used in a function `data()`:

```
func data(name string) string {
  f := os.Open(name, os.O_RDONLY, 0)
  defer f.Close() // idiomatic Go code!
  contents := io.ReadAll(f)
  return contents
}
```

Then, at **line 19**, we apply `bufio.NewReader` to get a *reader* variable. Using `bufio's` reader (and the same goes for writer), as we have done here, means that we can work with convenient high-level string objects, completely insulated from the raw bytes which represent the text on disk.

Then, from **line 20** to **line 26**, we read each line of the file (delimited by '\n' or '\r\n')) in an infinite for-loop with the method `ReadString('\n')` or

`ReadBytes('\n')`. `ReadString` returns an `io.EOF` error when the end of the input file is reached, which we test from **line 22** to **line 24**.

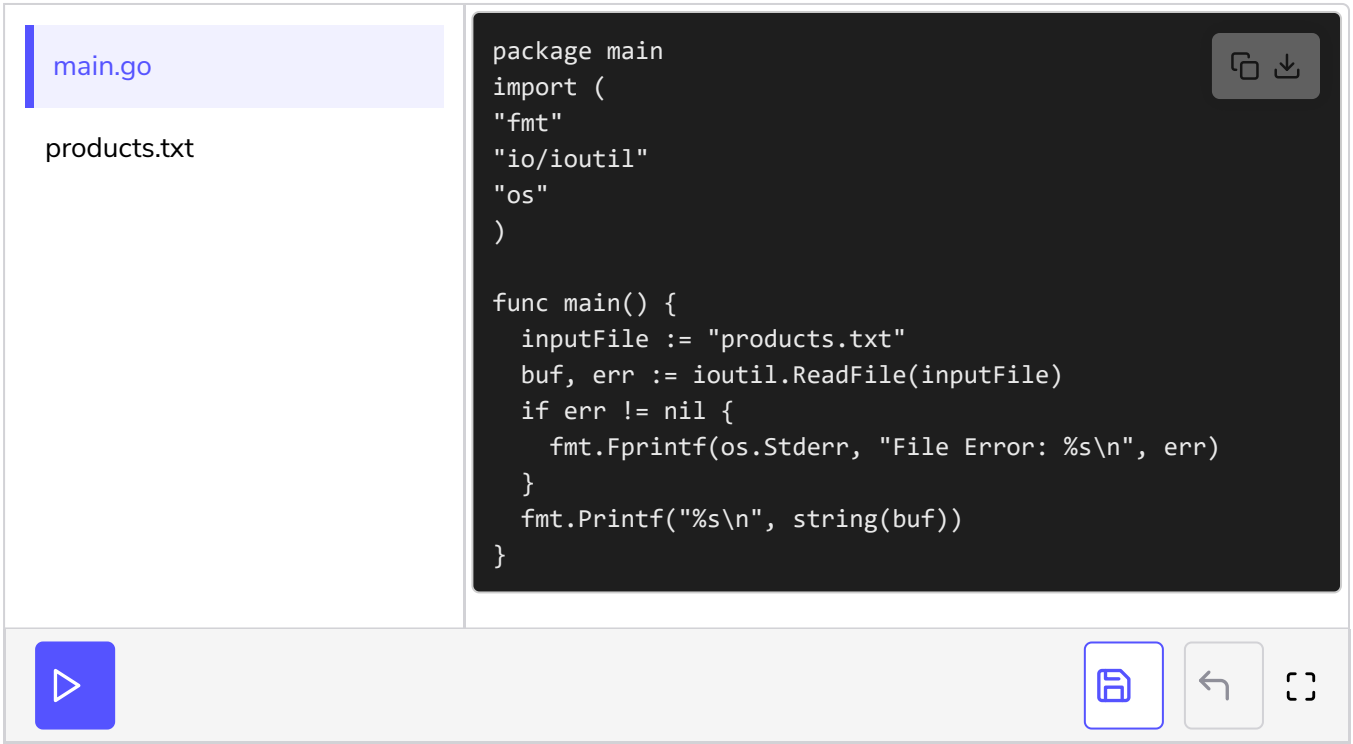
> **Remark**: In a previous example, we saw text-files in Unix end on \n but in Windows; this is \r\n. By using the method `ReadString` or `ReadBytes` with \n as a delimiter, you don't have to worry about this. The use of the `ReadLine()` method could also be a good alternative.

When we read the file past the end, the `readerError` is not nil (actually `io.EOF` is true), and the infinite for-loop is left through the return statement.

Some alternatives

Reading contents of an entire file in a string

If this is sufficient for your needs, you can use the `ioutil.ReadFile()` method from the package `io/ioutil`, which returns a *[]byte* containing the bytes read and nil or a possible error.



Reading from and Writing in File

The name of the file is stored in the variable `inputfile` at **line 9**. At **line 10**, we use the `ReadFile` method from the package `ioutil` to read in the file as a whole into a variable `buf`. If there is an error, we print this out (see **line 12**).

Finally, at **line 14**, we convert `buf` to a string with `string(buf)` and print it out.

> **Remark**: Don't use `ReadFile` for big files because one large string uses a lot of memory!

Buffered read

Instead of using `ReadString()`, in the more general case of a file not divided in lines or a binary file, we could have used the `Read()` method on the `bufio.Reader`, with a slice of bytes to read into as an input parameter:

```
buf := make([]byte, 1024)
...
n, err := inputReader.Read(buf)
if (n == 0) { break}
```

`n` is the number of bytes read.

Reading columns of data from a file

If the data columns are separated by a space, you can use the `FScan`-function series from the `fmt` package. This is applied in the following program:

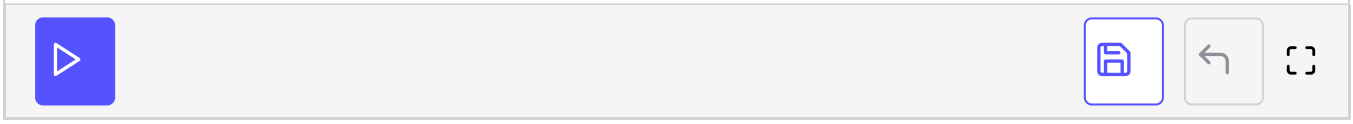
main.go

products2.txt

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

func main() {
  file, err := os.Open("products2.txt")
  if err != nil {
    panic(err)
  }
  defer file.Close()
  var col1, col2, col3 []string
  for {
    var v1, v2, v3 string
    _, err := fmt.Fscanln(file, &v1, &v2, &v3) // scans until newline
```

```
    if err != nil {
      break
    }
    col1 = append(col1, v1)
    col2 = append(col2, v2)
    col3 = append(col3, v3)
  }
  fmt.Println(col1)
  fmt.Println(col2)
  fmt.Println(col3)
}
```



Reading Columns of Data

This program opens a file **products2.txt** at **line 8**; it panics if there is an error (see the implementation from **line 8** to **line 11**). We make sure the file is closed at **line 12** with `defer`. When you open the file **products2.txt**, you see that it contains *columnar data*, separated by a space:

```
ABC 40 150
FUNC 56 280
GO 45 356
```

We define the variables **col1**, **col2**, and **col3** to hold the data from each column at **line 13**, each as an `[]string`. The contents of the file are read in an infinite loop from **line 14** to **line 19**.

The `Fscanln` function of package `fmt` reads in a line from the file at **line 16**. Because we know that there are *three* columns, `Fscanln` splits the line into the columns and reads their contents in the strings `v1`, `v2` and `v3`. On the left-hand side, you see `_, err`. The `_` means that we neglect the first return value, which is the number of bytes read in. From **line 17** to **line 19**, we detect that we have come at the end of the file: then, `err` is not `nil` and we break from the loop.

From **line 20** to **line 23**, we append the strings `v1`, `v2` and `v3` to the respective column slices `col1`, `col2`, and `col3`. These are then printed out at the end (from **line 24** to **line 26**).

> **Remark**: The sub-package `filepath` of the package `path` provides

functions for manipulating filenames and paths that work across OS

platforms. For example, the function `Base()` returns the last element of a path without trailing separator:

```
import "path/filepath"
filename := filepath.Base(path)
```

Reading from a zipped file

The package `compress`, from the standard library, provides facilities for reading compressed files in the following formats:

- bzip2
- flate
- gzip
- lzw
- zlib

The following program illustrates the reading of a *gzip* file:

main.go

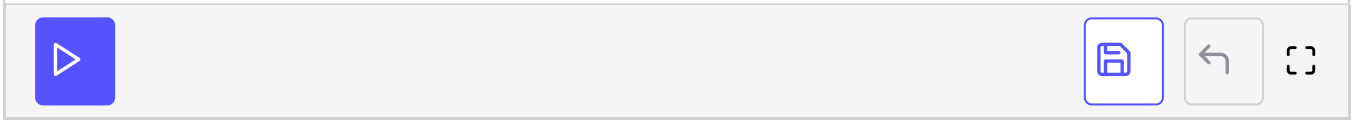
Example.json.gz

```
package main
import (
"fmt"
"bufio"
"os"
"compress/gzip"
)

func main() {
    fName := "Example.json.gz"
    var r *bufio.Reader
    fi, err := os.Open(fName)
    if err != nil {
        fmt.Fprintf(os.Stderr, "%v, Can't open %s: error: %s\n", os.Args[0], fName, err)
        os.Exit(1)
    }
    fz, err := gzip.NewReader(fi)
    if err != nil {
        r = bufio.NewReader(fi)
    } else {
        r = bufio.NewReader(fz)
```

```
    }
    for {
        line, err := r.ReadString('\n')

        if err != nil {
            fmt.Println("Done reading file")
            os.Exit(0)
        }
        fmt.Println(line)
    }
}
```



Reading from a Compressed File

To use that functionality, we need to import the package `compress/gzip` (see **line 6**). The filename of the zipped file is contained in the variable `fName` (see **line 10**). At **line 11**, we open the file returning a filehandle `fi`. The usual error-handling is done from **line 13** to **line 16**.

At **line 17**, we construct with the filehandle `fi`, a `gzip Reader` named `fz`. From **line 18** to **line 22**, we test whether there is an error or not. In both cases, we construct a buffered reader `r`, but we use `fi` as a parameter in case of an error, and `fz` means everything is *ok*.

Then, in an infinite loop (see the implementation from **line 23** to **line 30**), we read line by line from `r` with the `ReadString` method into the `line` variable and print it out. When the end of the file is reached, `err` is not `nil` anymore, and the program is stopped (see the implementation from **line 25** to **line 28**).

Now that you are familiar with the read operations, in the next lesson, you'll learn some write operations in Golang.