Reading and Writing

This lesson introduces a package provided by Golang, that has implemented interfaces for reading and writing the information.

WE'LL COVER THE FOLLOWING

Introduction

Reading and writing are universal activities in software: reading and writing files comes to mind first, whereas reading and writing to buffers (e.g., to slices of bytes or to strings), and to the standard input, output, and error streams, network connections, pipes, and so on (or to our custom types) comes second. To make the codebase as generic as possible, Go takes a consistent approach to reading and writing data.

The package `io` provides us with the interfaces for reading and writing, `io.Reader` and `io.Writer`:

```
type Reader interface {
  Read(p []byte) (n int, err error)
}

type Writer interface {
  Write(p []byte) (n int, err error)
}
```

Reading with `io` package

You can read from and write to any type as long as the type provides the methods `Read()` and `Write()` necessary to satisfy the reading and writing

interfaces. For an object to be readable, it must satisfy the `io.Reader` interface. This interface specifies a single method with the signature, `Read([]byte) (int, error)`. The `Read()` method reads data from the object it is called on and puts the read data into the given byte slice. It returns the number of bytes read and an error object, which will be nil if no error occurred or `io.EOF` ("End Of File") and the end of the input was reached, or some other non-nil value if an error occurred.

Writing with `io` package

Similarly, for an object to be writable, it must satisfy the `io.Writer` interface. This interface specifies a single method with the signature, `Write([]byte) (int, error)`. The `Write()` method writes data from the given byte slice into the object the method was called on and returns the number of bytes written and an error object (which will be nil if no error occurred).

The `io` Readers and Writers are unbuffered; the package `bufio` provides for the corresponding buffered operations and so are especially useful for reading and writing **UTF-8** encoded text files. We see this in action in the many examples in the next chapter.

Through using as much as possible these interfaces in the signature of methods, they become as generic as possible: every type that implements these interfaces can be used by other methods requiring these Reading and Writing operations.

For example, the function which implements a JPEG decoder takes a Reader as a parameter and thus can decode from a disk, network connection, gzipped http, and so on.

We'll discuss the reading and writing processes in detail in the next chapter. For now, the next lesson brings us a new concept of empty interfaces.