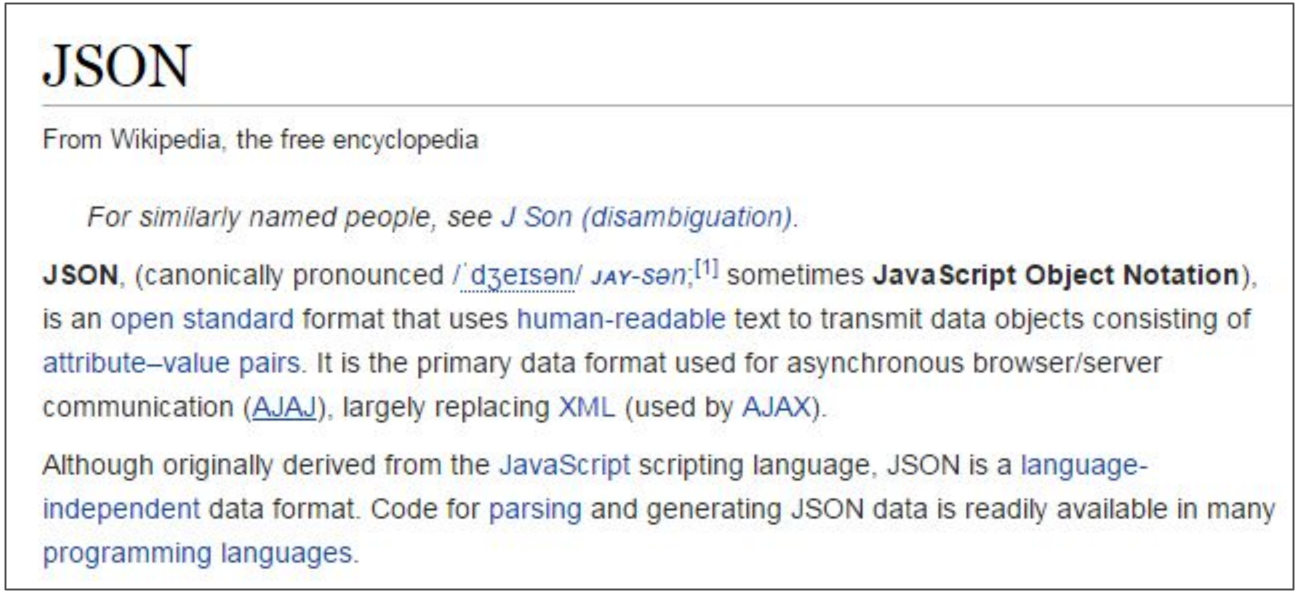
JSON

JavaScript Object Notation



JSON

From Wikipedia, the free encyclopedia

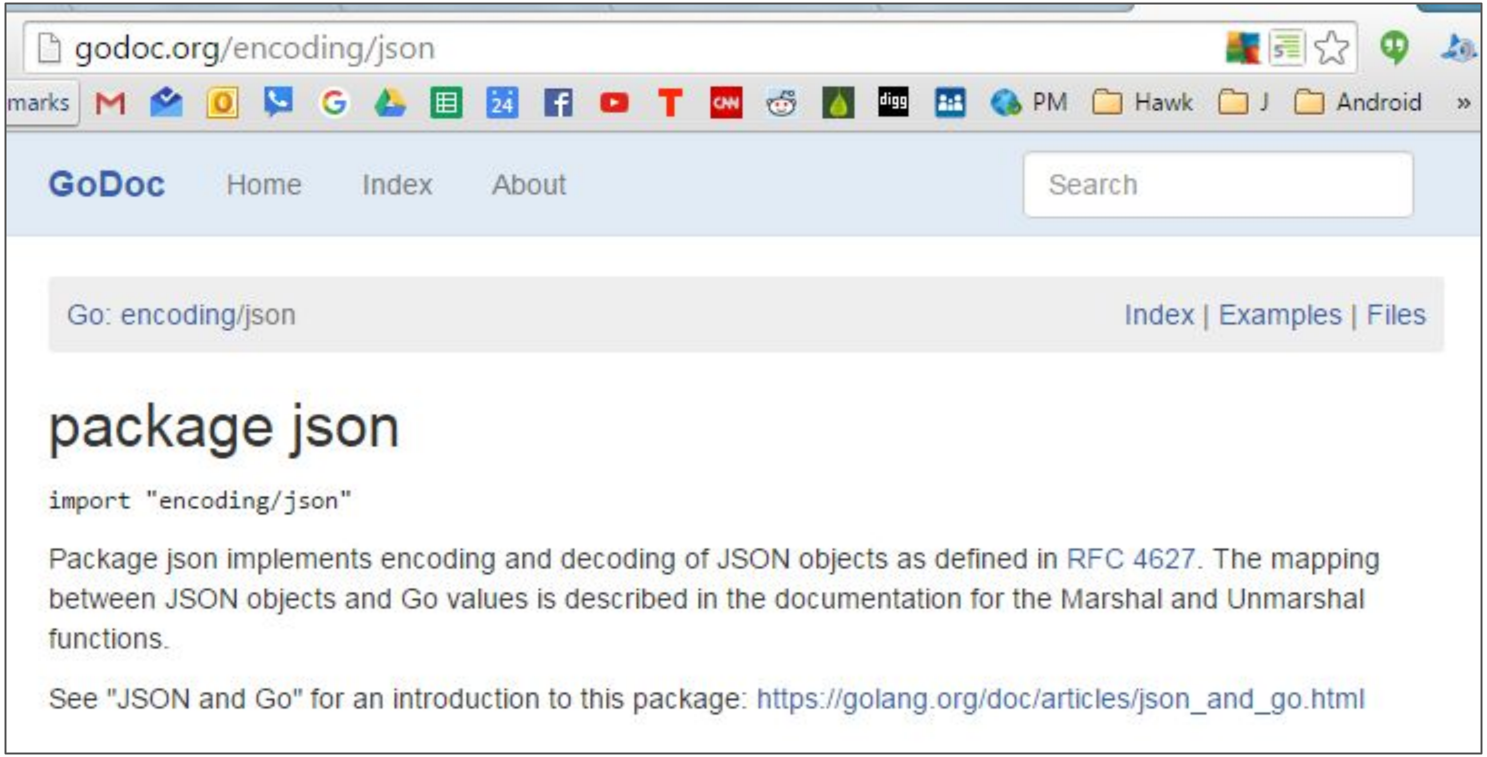
For similarly named people, see J Son (disambiguation).

JSON, (canonically pronounced /ˈdʒeɪsən/ *JAY-sən*;[1] sometimes **JavaScript Object Notation**), is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the primary data format used for asynchronous browser/server communication (AJAJ), largely replacing XML (used by AJAX).

Although originally derived from the JavaScript scripting language, JSON is a language-independent data format. Code for parsing and generating JSON data is readily available in many programming languages.

http://www.jsoneditoronline.org/

http://json.org/



package json

```
import "encoding/json"
```

Package json implements encoding and decoding of JSON objects as defined in RFC 4627. The mapping between JSON objects and Go values is described in the documentation for the Marshal and Unmarshal functions.

See "JSON and Go" for an introduction to this package: https://golang.org/doc/articles/json_and_go.html

Encoder, Decoder
Marshal, Unmarshal

to JSON from JSON

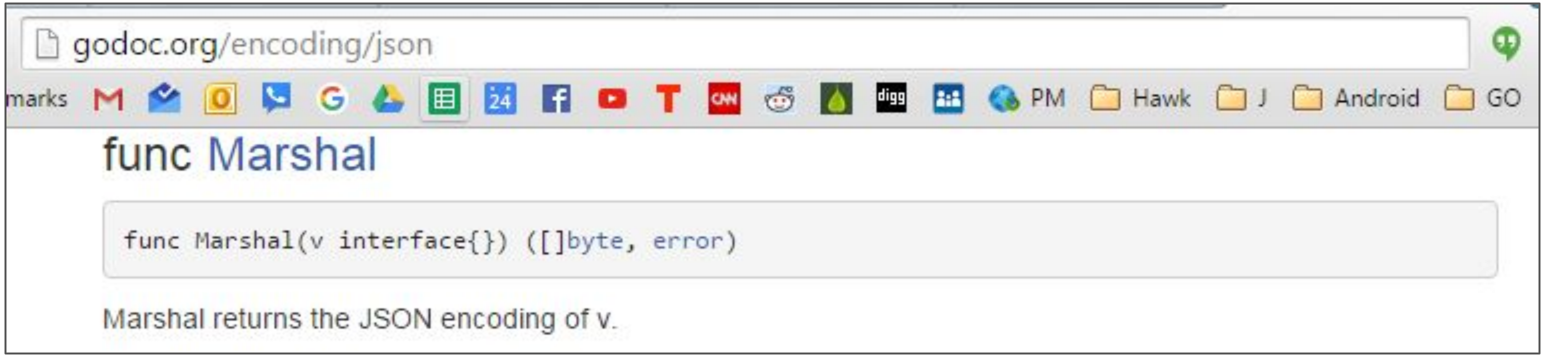
Encoder, Decoder

Marshal, Unmarshal

to JSON from JSON

Encoder, Decoder reader, writer

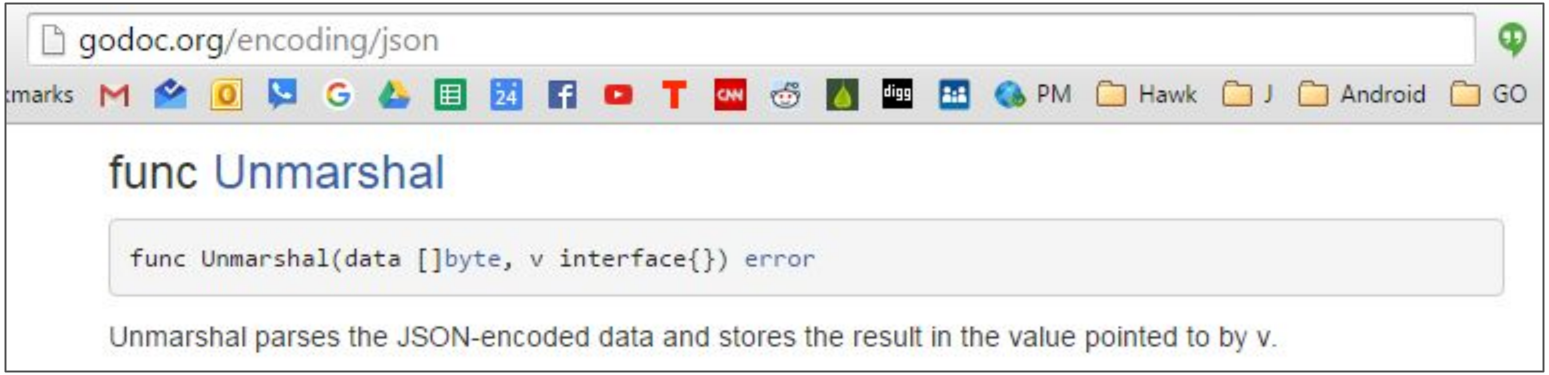
Marshal, Unmarshal []bytes



func Marshal

```
func Marshal(v interface{}) ([]byte, error)
```

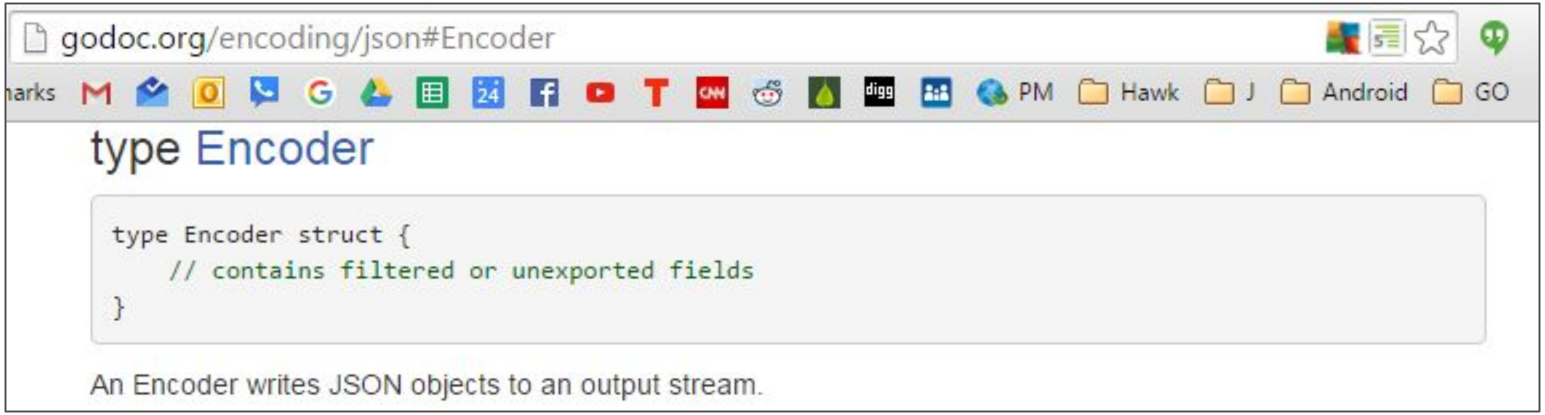
Marshal returns the JSON encoding of v.


godoc.org/encoding/json

func Unmarshal

```
func Unmarshal(data []byte, v interface{}) error
```

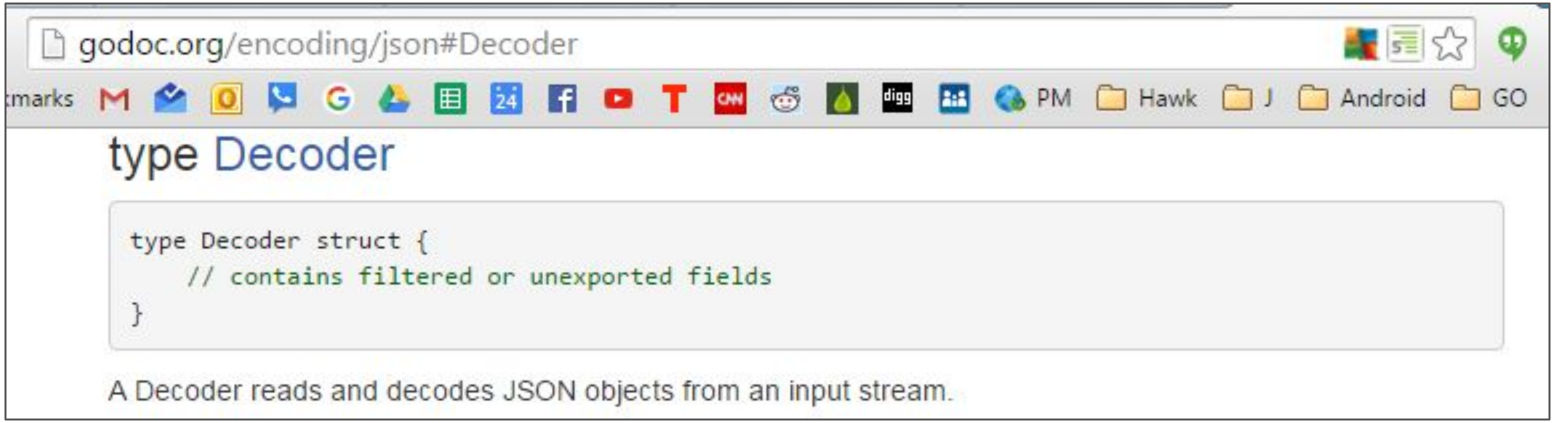
Unmarshal parses the JSON-encoded data and stores the result in the value pointed to by v.



type Encoder

```
type Encoder struct {
    // contains filtered or unexported fields
}
```

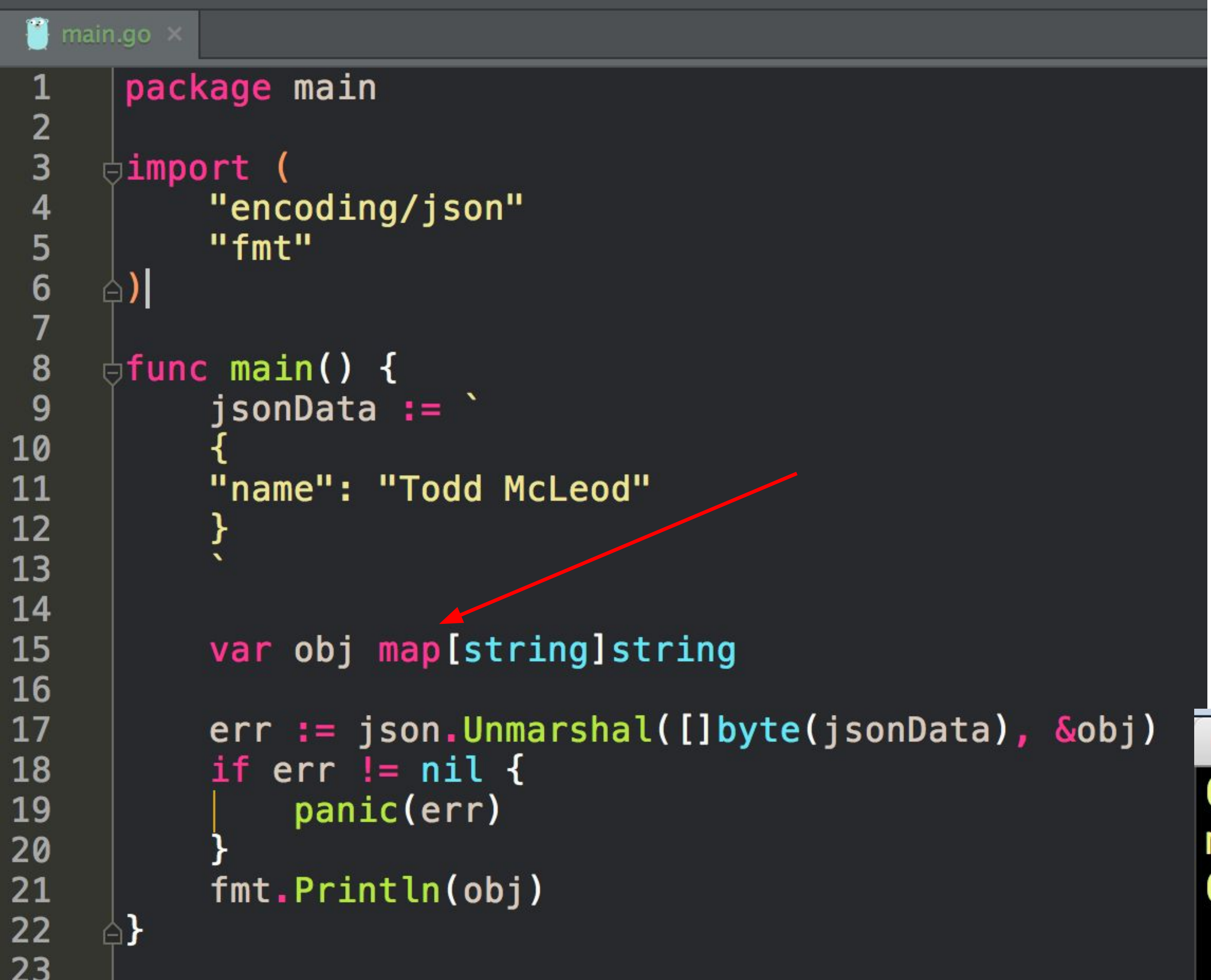
An Encoder writes JSON objects to an output stream.



type Decoder

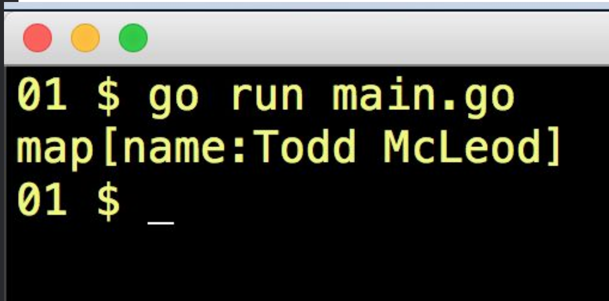
```
type Decoder struct {
    // contains filtered or unexported fields
}
```

A Decoder reads and decodes JSON objects from an input stream.

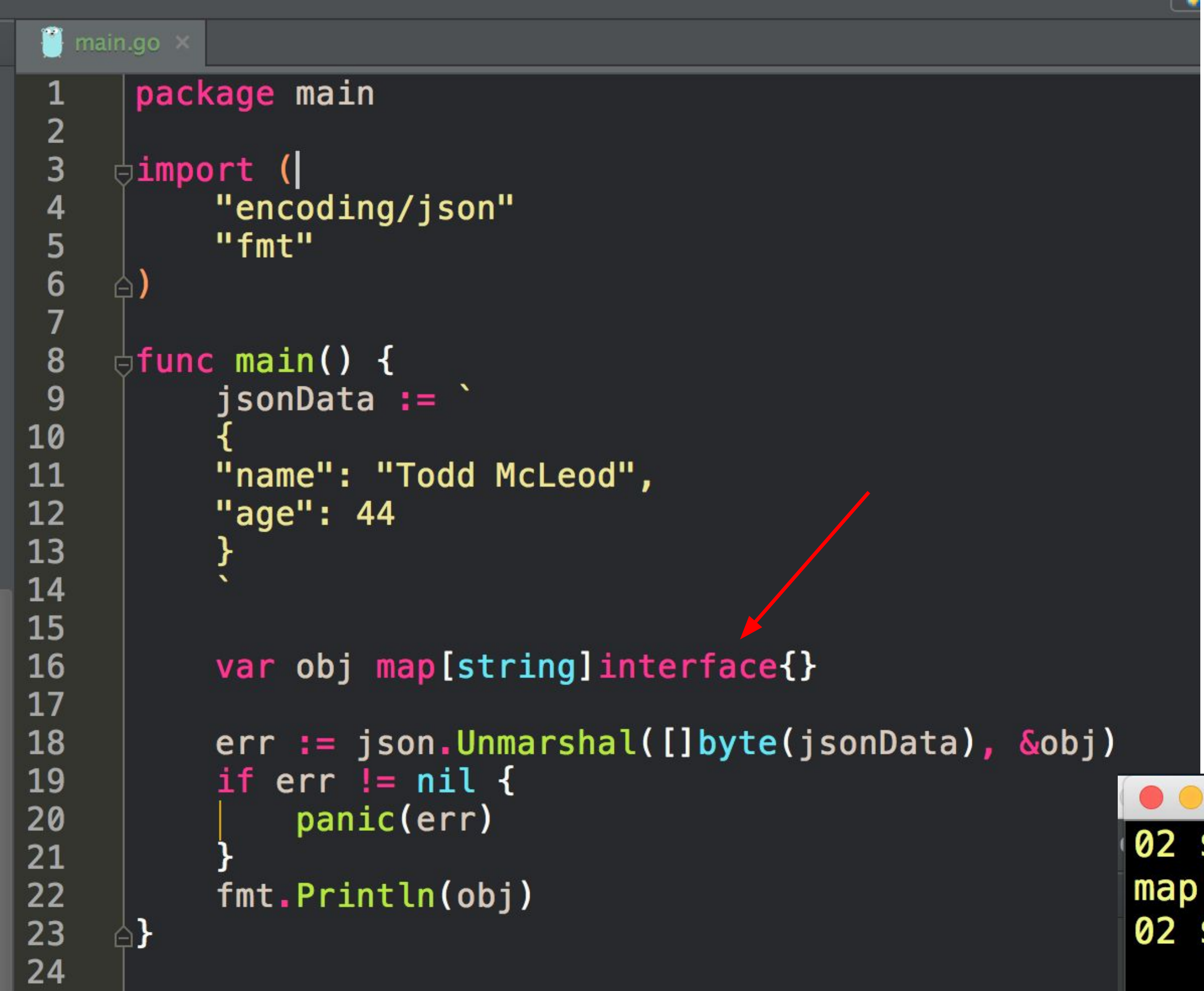


```go
package main

import (
	"encoding/json"
	"fmt"
)

func main() {
	jsonData := `
	{
	"name": "Todd McLeod"
	}
	`

	var obj map[string]string

	err := json.Unmarshal([]byte(jsonData), &obj)
	if err != nil {
		panic(err)
	}
	fmt.Println(obj)
}
```

Unmarshal

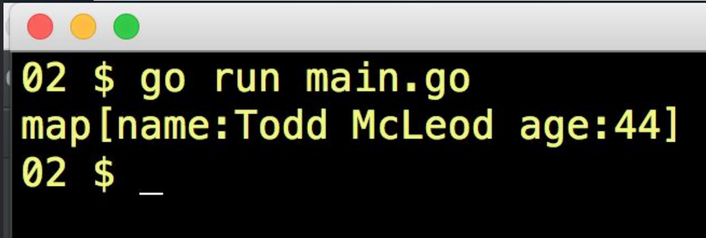


```
01 $ go run main.go
map[name:Todd McLeod]
01 $ _
```

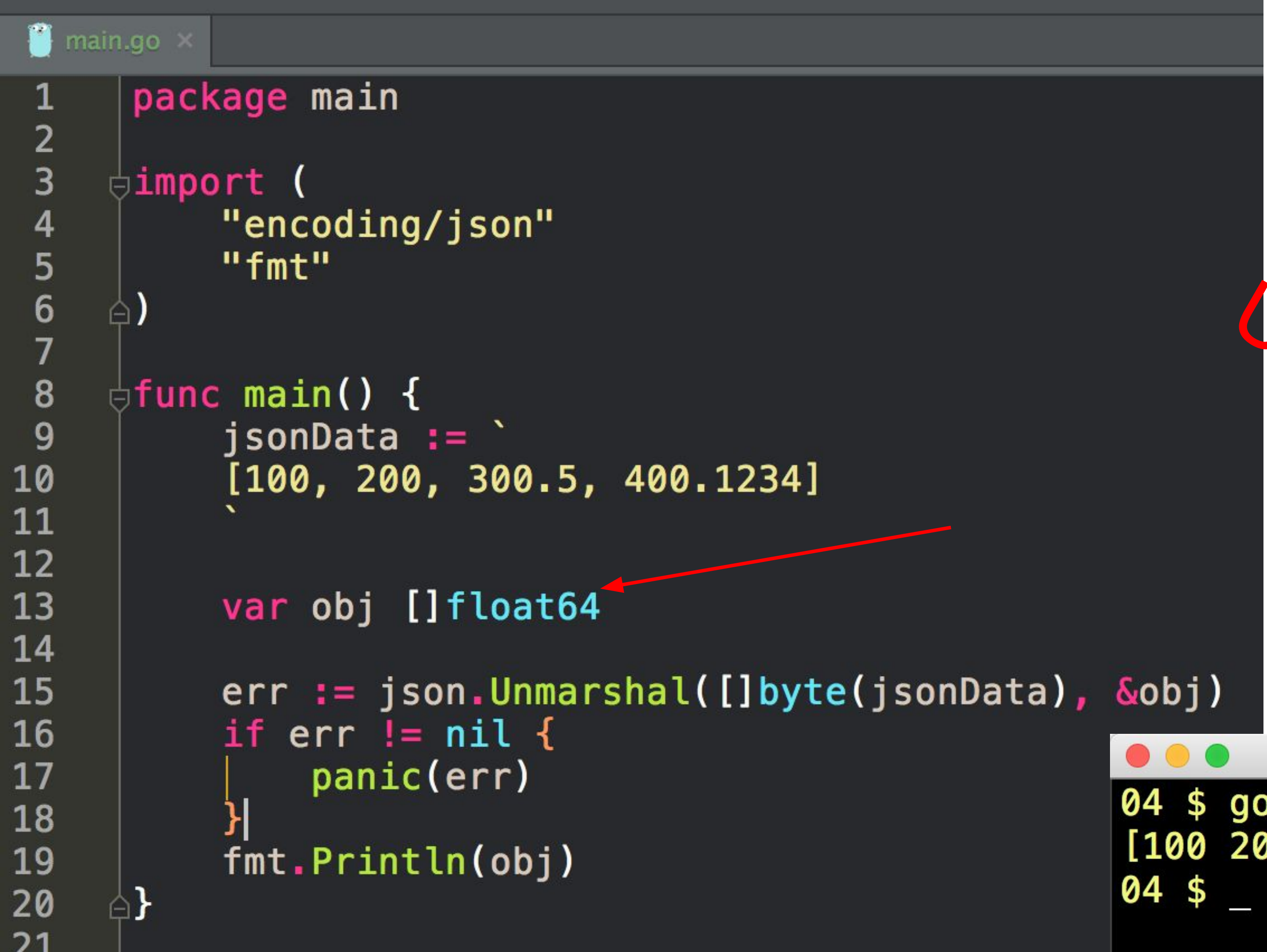


```go
package main

import (
	"encoding/json"
	"fmt"
)

func main() {
	jsonData := `
	{
	"name": "Todd McLeod",
	"age": 44
	}
	`

	var obj map[string]interface{}

	err := json.Unmarshal([]byte(jsonData), &obj)
	if err != nil {
		panic(err)
	}
	fmt.Println(obj)
}
```

Unmarshal



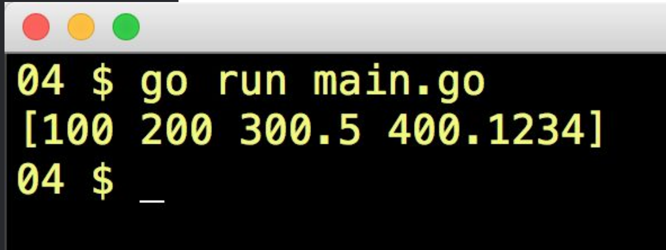
```
02 $ go run main.go
map[name:Todd McLeod age:44]
02 $ _
```



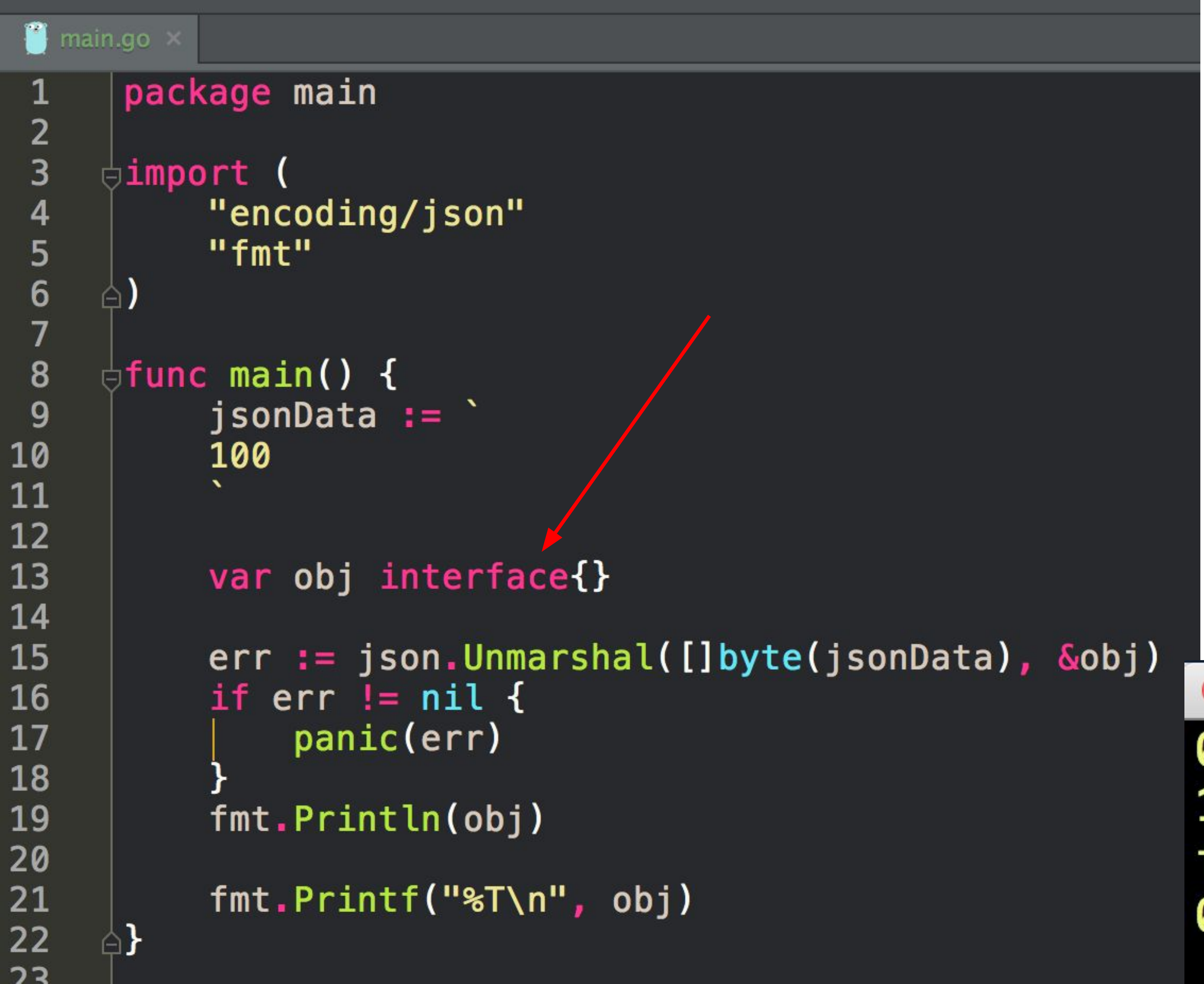
main.go

```go
package main

import (
	"encoding/json"
	"fmt"
)

func main() {
	jsonData := `
	[100, 200, 300.5, 400.1234]
	`

	var obj []float64

	err := json.Unmarshal([]byte(jsonData), &obj)
	if err != nil {
		panic(err)
	}
	fmt.Println(obj)
}
```

Unmarshal

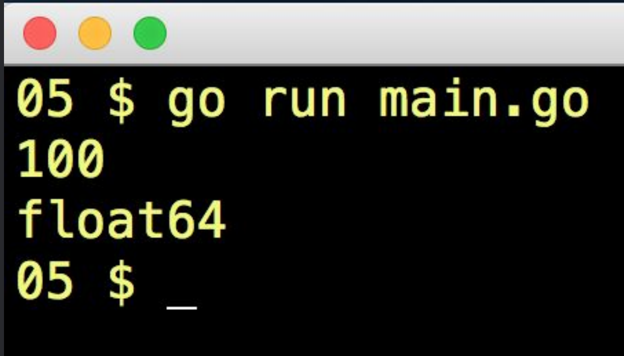


```
04 $ go run main.go
[100 200 300.5 400.1234]
04 $ _
```

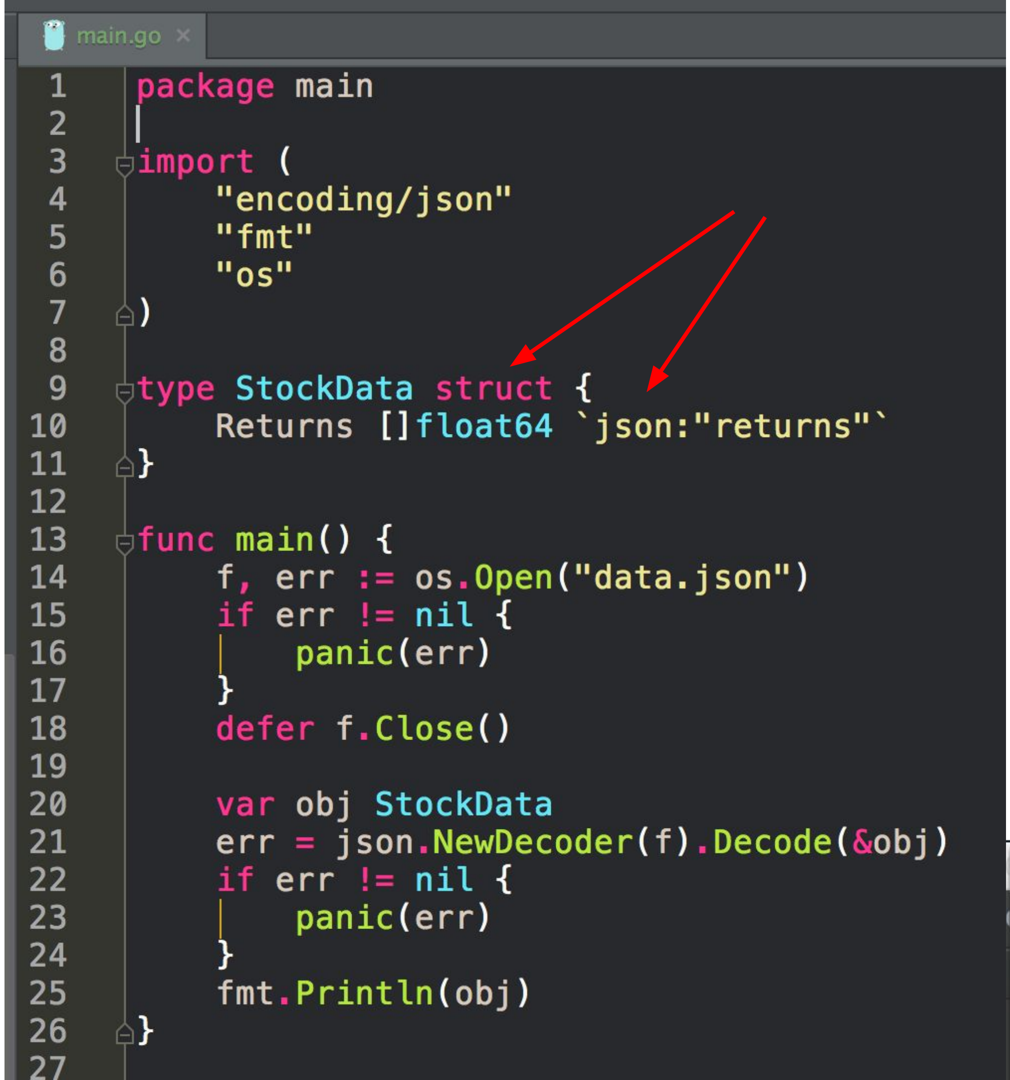


```go
package main

import (
	"encoding/json"
	"fmt"
)

func main() {
	jsonData := `
	100
	`

	var obj interface{}

	err := json.Unmarshal([]byte(jsonData), &obj)
	if err != nil {
		panic(err)
	}
	fmt.Println(obj)

	fmt.Printf("%T\n", obj)
}
```

Unmarshal

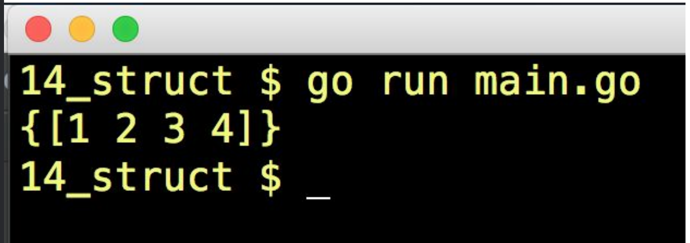


```
05 $ go run main.go
100
float64
05 $ _
```

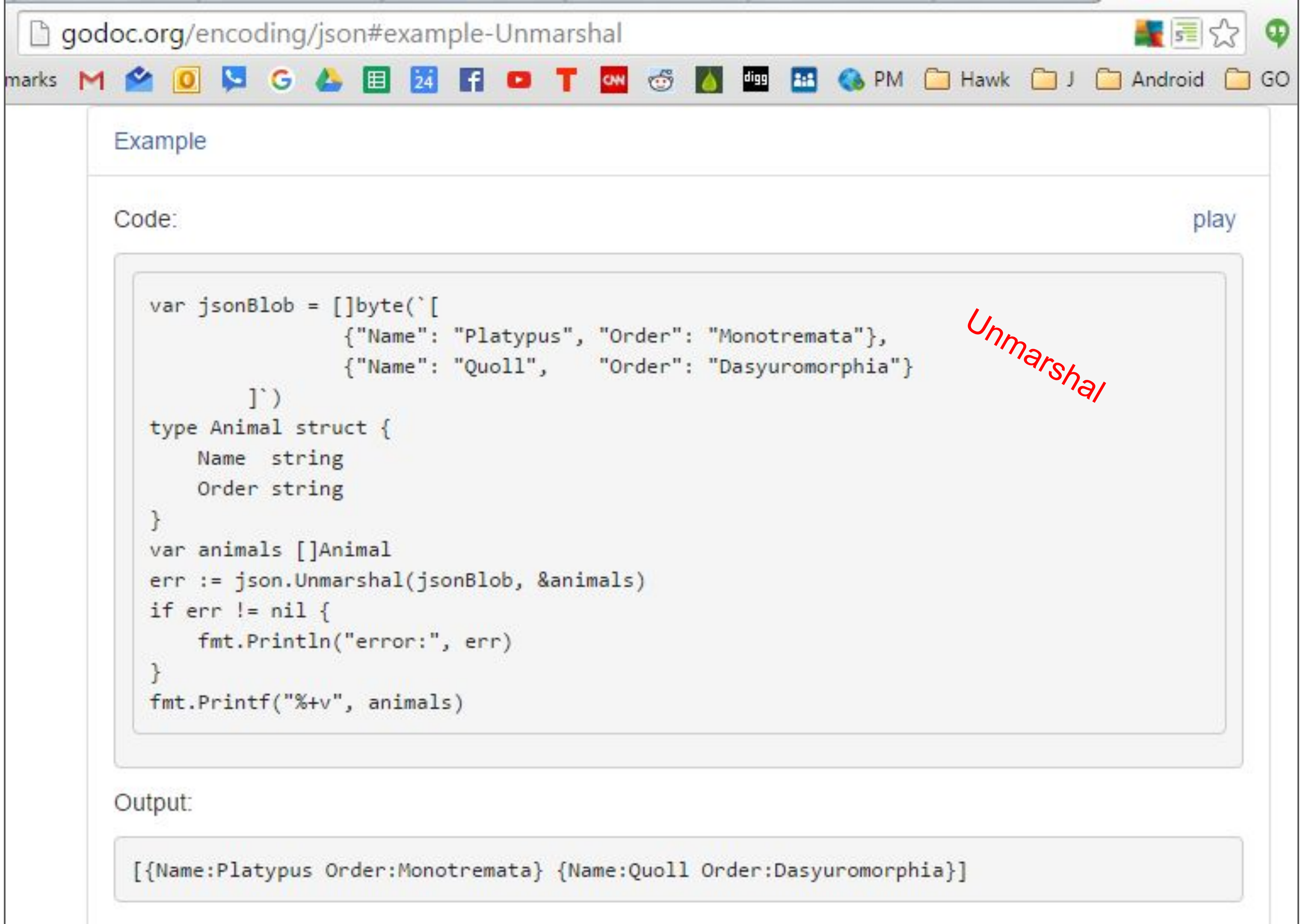


```go
package main

import (
	"encoding/json"
	"fmt"
	"os"
)

type StockData struct {
	Returns []float64 `json:"returns"`
}

func main() {
	f, err := os.Open("data.json")
	if err != nil {
		panic(err)
	}
	defer f.Close()

	var obj StockData
	err = json.NewDecoder(f).Decode(&obj)
	if err != nil {
		panic(err)
	}
	fmt.Println(obj)
}
```

Unmarshal



```
14_struct $ go run main.go
{[1 2 3 4]}
14_struct $ _
```



Example

Code:

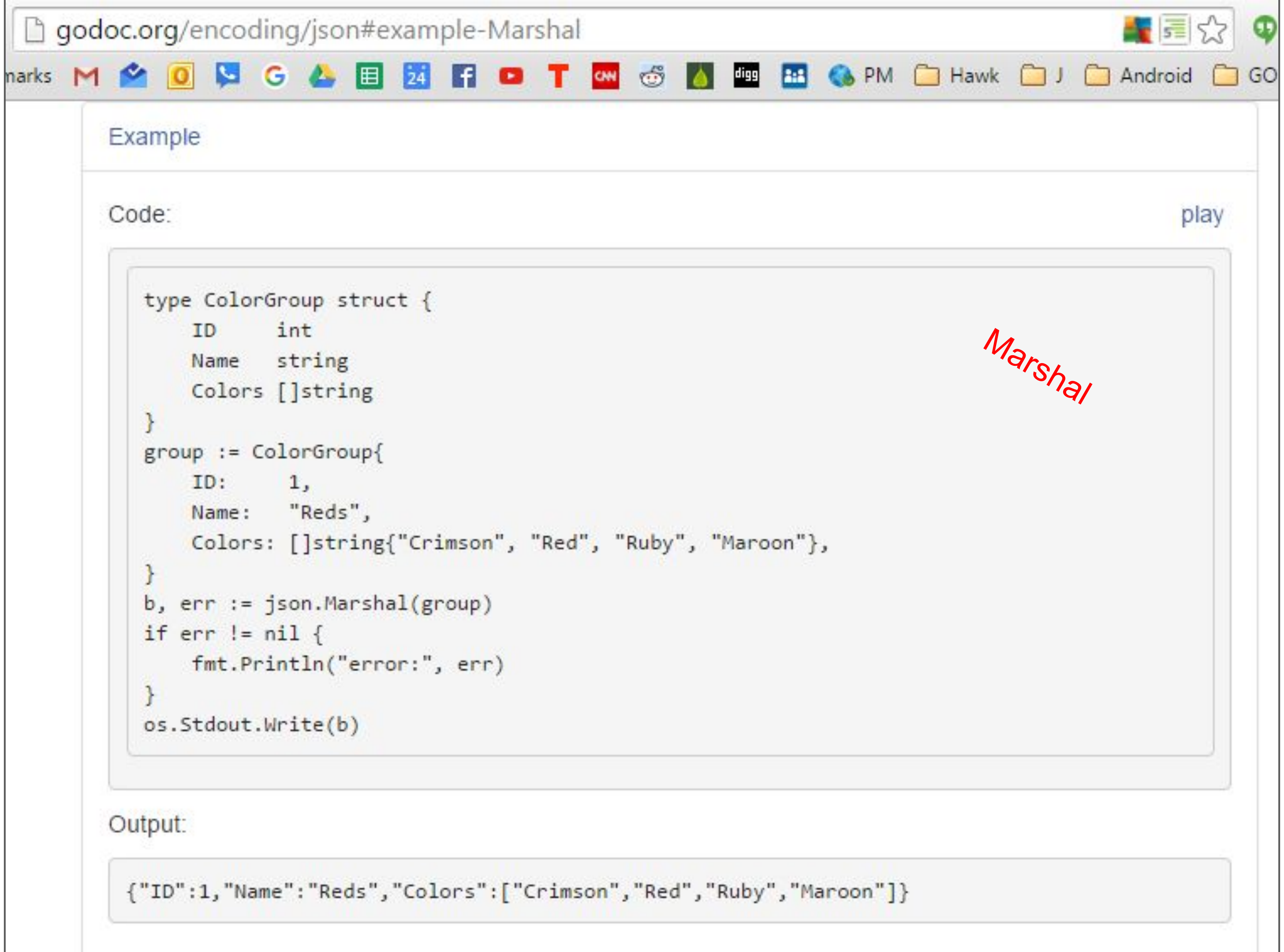
play

```
var jsonBlob = []byte(`[
                {"Name": "Platypus", "Order": "Monotremata"},
                {"Name": "Quoll",    "Order": "Dasyuromorphia"}
        ]`)
type Animal struct {
    Name  string
    Order string
}
var animals []Animal
err := json.Unmarshal(jsonBlob, &animals)
if err != nil {
    fmt.Println("error:", err)
}
fmt.Printf("%+v", animals)
```

Unmarshal

Output:

```
[{Name:Platypus Order:Monotremata} {Name:Quoll Order:Dasyuromorphia}]
```



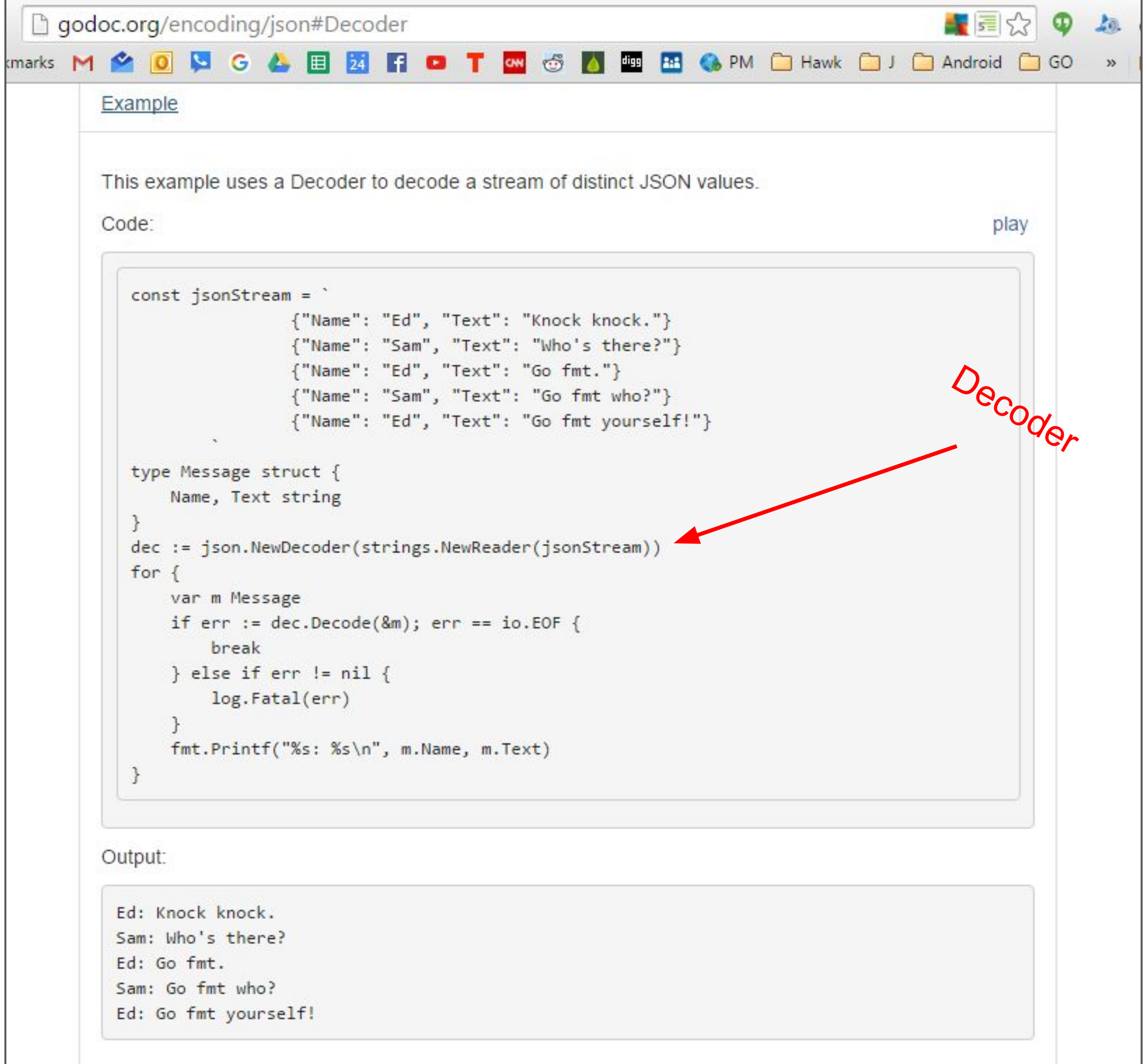
Code:

```
type ColorGroup struct {
    ID     int
    Name   string
    Colors []string
}
group := ColorGroup{
    ID:     1,
    Name:   "Reds",
    Colors: []string{"Crimson", "Red", "Ruby", "Maroon"},
}
b, err := json.Marshal(group)
if err != nil {
    fmt.Println("error:", err)
}
os.Stdout.Write(b)
```

Marshal

Output:

```
{"ID":1,"Name":"Reds","Colors":["Crimson","Red","Ruby","Maroon"]}
```



This example uses a Decoder to decode a stream of distinct JSON values.

Code:

```
const jsonStream = `
                {"Name": "Ed", "Text": "Knock knock."}
                {"Name": "Sam", "Text": "Who's there?"}
                {"Name": "Ed", "Text": "Go fmt."}
                {"Name": "Sam", "Text": "Go fmt who?"}
                {"Name": "Ed", "Text": "Go fmt yourself!"}
        `
type Message struct {
    Name, Text string
}
dec := json.NewDecoder(strings.NewReader(jsonStream))
for {
    var m Message
    if err := dec.Decode(&m); err == io.EOF {
        break
    } else if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s: %s\n", m.Name, m.Text)
}
```

Decoder

Output:

```
Ed: Knock knock.
Sam: Who's there?
Ed: Go fmt.
Sam: Go fmt who?
Ed: Go fmt yourself!
```

exercises

JSON

Create a struct, encode it as JSON, and write it out to stdout.

JSON

- Create a JSON file
- Create a program that reads that JSON file into a struct, then writes it to stdout.

JSON

Create a program which can read a csv file and write it as a JSON file.