

LEARN THE GO PROGRAMMING LANGUAGE

For experienced developers or
those of an adventurous nature

gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

LESSON 09

JSON, JSON, JSON

v0.1 draft

THE BASICS

READS

- In-depth blog post:
 - <http://blog.golang.org/json-and-go>
- Package Documentation:
 - <http://golang.org/pkg/encoding/json/>

TWO FLAVORS

```
// serializing
func Marshal(v interface{}) ([]byte, error)
func (enc *Encoder) Encode(v interface{}) error

// deserializing
func Unmarshal(data []byte, v interface{}) error
func (dec *Decoder) Decode(v interface{}) error
```

TWO FLAVORS

- Marshall serializes to a byte slice
- Unmarshall deserializes a byte slice
- Encode serializes and writes bytes to an `io.Writer`
- Decoder deserialized bytes read from an `io.Reader`

THREE SCENARIOS

```
// serializing  
interface{} -> JSONBytes
```

```
// deserializing  
JSONBytes -> struct  
JSONBytes -> interface{}
```

SERIALIZATION

THREE SCENARIOS: SERIALIZING

- Serializing is relatively straight forward. Pass in an `interface{}` and get out some bytes.
- Normally, you pass in structs or maps or slices/arrays.

SERIALIZING: MAPS

- maps keys must be string
- map values must:
 - be a type supported by the json package
 - bools, floats, ints, numbers
 - strings
 - or structs that implement the Marshaller interface

SERIALIZING: MAPS

```
// http://play.golang.org/p/sUYgLd6WRq
```

```
jsonBytes, err := json.Marshal(map[string][]string{"1":  
[]string{"x","y"}})  
fmt.Println(string(jsonBytes))  
fmt.Println(err)
```

```
_, err = json.Marshal(map[int][]string{1:  
[]string{"x","y"}})  
fmt.Println(err)
```

```
// outputs:  
// {"1":["x","y"]}  
// <nil>  
// json: unsupported type: map[int][]string
```


SERIALIZING: SLICES & ARRAYS

- slices or arrays must also be a type supported by the json package or structs that implement the Marshaller interface

```
// http://play.golang.org/p/s0ghfITGJF  
jsonBytes, err := json.Marshal([]string{"a","b","c"})  
fmt.Println(string(jsonBytes))  
fmt.Println(err)
```

```
jsonBytes, err = json.Marshal([]complex128{complex128(1)})  
fmt.Println(string(jsonBytes))  
fmt.Println(err)
```

```
// outputs  
// ["a","b","c"]  
// <nil>  
//  
// json: unsupported type: complex128
```


SERIALIZING: SCALARS

- Passing in scalars works, but isn't very useful
- <http://play.golang.org/p/xeqNclLQTj>

```
jsonBytes, err := json.Marshal(1.1)
fmt.Println(string(jsonBytes))
fmt.Println(err)
jsonBytes, err = json.Marshal("why?")
fmt.Println(string(jsonBytes))
fmt.Println(err)
```

```
1.1
<nil>
"why?"
<nil>
```

SERIALIZING: STRUCTS

- Only Exported fields are serialized.
- <http://play.golang.org/p/sS3wL60X7v>

```
type Shape struct {  
    Name  string  
    sides int  
}  
  
func main() {  
    jsonBytes, err := json.Marshal(Shape{"square", 4})  
    fmt.Println(string(jsonBytes))  
    fmt.Println(err)  
}  
  
// outputs  
// {"Name":"square"}  
// <nil>
```

SERIALIZING STRUCTS: TAGS

From the `encoding/json` documentation:

```
// Field is ignored by this package.  
Field int `json:"-`
```

```
// Field appears in JSON as key "myName".  
Field int `json:"myName"`
```

```
// Field appears in JSON as key "myName" and  
// the field is omitted from the object if its value is empty,  
// as defined above.  
Field int `json:"myName,omitempty"`
```

```
// Field appears in JSON as key "Field" (the default), but  
// the field is skipped if empty.  
// Note the leading comma.  
Field int `json:",omitempty"`
```


SERIALIZING STRUCTS: TAGS

// <http://play.golang.org/p/6IzFngGX8k>

```
type thing struct {
    IgnoreMe int    `json:"-"`
    Name      string `json:"name"`
    Age       int    `json:"age_of_thing,omitempty"`
    Comment   string `json:",omitempty"`
}

func main() {
    // please don't ignore errors.
    jsonBytes, _ := json.Marshal(thing{1, "Bob", 0, ""})
    fmt.Println(string(jsonBytes))
    jsonBytes, _ = json.Marshal(thing{1, "Bill", 1, ""})
    fmt.Println(string(jsonBytes))
    jsonBytes, _ = json.Marshal(thing{1, "Waldo", 1, "Where?"})
    fmt.Println(string(jsonBytes))
}

// {"name":"Bob"}
// {"name":"Bill","age_of_thing":1}
// {"name":"Waldo","age_of_thing":1,"Comment":"Where?"}
```


SERIALIZING: OTHERS

- complex, channels and functions will panic
- cyclic structs will likely infinitely recurse
- pointer values usually dereference
- interface values usually unbox

DESERIALIZATION

THREE SCENARIOS

```
// serializing  
interface{} -> JSONBytes
```

```
// deserializing  
JSONBytes -> struct  
JSONBytes -> interface{}
```

THREE SCENARIOS: DESERIALIZING

- deserializing to a struct:
 - encoding/json will attempt to populate a struct with any fields it finds in a JSON object.
- deserializing to interface{}
 - encoding/json will create a new type based on maps, slices & scalars
 - map[string]interface{},
 - []interface{}

THREE SCENARIOS: DESERIALIZING

```
type Shape struct {  
  Names []string  
  Sides int  
}
```

THREE SCENARIOS: DESERIALIZING

```
// http://play.golang.org/p/iC8a0VMDbq

jsonBytes := []byte(`{"Names":["square","rectangle","trapezoid"],"Sides":4}`)
shapePtr := new(Shape)
_ = json.Unmarshal(jsonBytes, shapePtr)
fmt.Println("deserialized:", *shapePtr)
fmt.Printf("type: %T\n", shapePtr)

var v interface{}
_ = json.Unmarshal(jsonBytes, &v)
fmt.Println("deserialized:", v)
fmt.Printf("type: %T\n", v)
fmt.Printf("type: %T\n", v.(map[string]interface{})["Names"])

// deserialized: {[square rectangle trapezoid] 4}
// type: *main.Shape
// deserialized: map[Names:[square rectangle trapezoid] Sides:4]
// type: map[string]interface {}
// type: []interface {}
```

DESERIALIZING: MIXED TYPE MAPS

```
// http://play.golang.org/p/AL0CggwBmB
```

```
jsonBytes := []byte(`{"Names":["square","rectangle",1],"Sides":4}`)
shapePtr := new(Shape)
_ = json.Unmarshal(jsonBytes, shapePtr)
fmt.Println("deserialized:", *shapePtr)
fmt.Println("len(shapePtr.Names):", len(shapePtr.Names))
```

mixed types ok
in JSON!

```
var v interface{}
_ = json.Unmarshal(jsonBytes, &v)
fmt.Println("deserialized:", v)
```

third Name is the
empty string

```
// deserialized: {[square rectangle ] 4}
// len(shapePtr.Names): 3
// deserialized: map[Names:[square rectangle 1] Sides:4]
// type: map[string]interface {}
```


DESERIALIZING: EXTRA FIELDS

```
// http://play.golang.org/p/WHxRB-Kg\_4
```

```
jsonBytes := []byte(`{"Names":["square","rectangle",1],"Sides":  
4, "Color":"red"}`)  
shapePtr := new(Shape)  
_ = json.Unmarshal(jsonBytes, shapePtr)  
fmt.Println("deserialized:", *shapePtr)
```

extra fields...

```
var v interface{}  
_ = json.Unmarshal(jsonBytes, &v)  
fmt.Println("deserialized:", v)
```

... are ignored when
unmarshalling into struts

```
// deserialized: {[square rectangle ] 4}  
// len(shapePtr.Names): 3  
// deserialized: map[Names:[square rectangle 1] Sides:4  
Color:red]
```


DESERIALIZING STRUCTS: TAGS

```
// http://play.golang.org/p/I-yadK-CLy
```

```
type thing struct {  
    IgnoreMe int    `json:"-"`  
    Name      string `json:"name"`  
    Age       int     `json:"age_of_thing,omitempty"`  
    Comment   string `json:",omitempty"`  
}
```

```
var t1, t2, t3 thing  
_ = json.Unmarshal([]byte(`{"name":"Bill","Age":2}`), &t1)  
fmt.Println(t1)  
_ = json.Unmarshal([]byte(`{"Name":"Bill","Age_of_thing":1, "Age":2}`), &t2)  
fmt.Println(t2)  
_ = json.Unmarshal([]byte(`{"IgnoreMe":1,"name":"Waldo","age_of_thing":  
1,"Comment":"Where?"}`), &t3)  
fmt.Println(t3)
```

```
// {0 Bill 0 }  
// {0 Bill 1 }  
// {0 Waldo 1 Where?}
```

DESERIALIZING STRUCTS: TAGS

// <http://play.golang.org/p/I-yadK-CLy>

```
type thing struct {  
    IgnoreMe int    `json:"-";omitempty`  
    Name     string  `json:"name"`  
    Age      int     `json:"age_of_thing,omitemp`  
    Comment  string  `json:"comment,omitemp`  
}
```

case-insensitive

respects the tagged
name

```
var t1, t2, t3 thing  
_ = json.Unmarshal([]byte(`{"name":"Bill","Age":2}`), &t1)  
fmt.Println(t1)  
_ = json.Unmarshal([]byte(`{"Name":"Bill","Age_of_thing":1, "Age":2}`), &t2)  
fmt.Println(t2)  
_ = json.Unmarshal([]byte(`{"IgnoreMe":1,"name":"Waldo","age_of_thing":  
1,"Comment":"Where?"}`), &t3)  
fmt.Println(t3)
```

```
// {0 Bill 0 }  
// {0 Bill 1 }  
// {0 Waldo 1 Where?}
```

"-" tagged is always
ignored.

omitempty doesn't
mean much for
deserialization

ANONYMOUS FIELDS

ANONYMOUS FIELDS

- Review:

```
type Person struct {  
    Name string  
    Age  int  
}  
type Employee struct {  
    Person  
    StartDate time.Time  
    Salary    int  
}  
type Manager struct {  
    Employee  
    Reports []Employee  
}
```

ANONYMOUS FIELDS

- Anonymous Fields are “flattened”

```
pointy := new(Manager)
pointy.Name = "Fred"
pointy.Age = 38
pointy.StartDate = time.Now()
rupert := new(Employee)
rupert.Name = "Rupert"
pointy.Reports = []Employee{rupert}

jsonBytes, err = json.Marshal(pointy)
```

ANONYMOUS FIELDS

```
fmt.Println(jsonBytes)
```

```
// outputs (pretty printed for legibility)
```

```
{  
  "Name": "name",  
  "Age": 38,  
  "StartDate": "2015-03-05T09:58:26.575833992-08:00",  
  "Salary": 0,  
  "Reports": [  
    {  
      "Name": "Rupert",  
      "Age": 0,  
      "StartDate": "0001-01-01T00:00:00Z",  
      "Salary": 0  
    }  
  ]  
}
```


ANONYMOUS FIELDS

- Deserialization works the similarly:

```
fmt.Println(jsonBytes)
```

```
jsonBytes = []byte(`{"Name":"Harry","Age":  
38,"StartDate":"2015-03-05T09:58:26.575833992-08:00",  
"Salary":0,"Reports":[{"Name":"Rupert","Age":  
0,"StartDate":"0001-01-01T00:00:00Z","Salary":0}]}`)
```

```
hairy := new(Manager)  
_ = json.Unmarshal(jsonBytes, hairy)
```

ANONYMOUS FIELDS

- Deserialization will “drill down” and will best effort populate anonymous fields

```
fmt.Printf("%+v\n", hairy)
```

```
// outputs (pretty printed for legibility)
```

```
&{
  Employee: {
    Person:{Name:Harry Age:38}
    StartDate:2015-03-05 09:58:26.575833992 -0800 PST
    Salary:0
  }
  Reports:[{
    Person:{Name:Rupert Age:0}
    StartDate:0001-01-01 00:00:00 +0000 UTC Salary:0
  }]
}
```

OVERLAPPING ANONYMOUS FIELDS

- Anonymous fields with overlap are a point of caution:

```
type Eye struct {  
    Count int  
    Color string  
}  
type Hair struct {  
    Texture string  
    Color    string  
}  
type Face struct {  
    Eye  
    Hair  
}
```

```
face := new(Face)  
face.Eye.Count = 2  
face.Eye.Color = "Brown"  
face.Hair.Color = "Blonde"  
jsonBytes, _ = json.Marshal(face)  
// outputs:  
// {"Count":2,"Texture":""}
```



Ambiguous fields are
just skipped

OVERLAPPING ANONYMOUS FIELDS

- Anonymous fields with overlap are a point of caution:

```
type Eye struct {  
    Count int  
    Color string  
}  
type Hair struct {  
    Texture string  
    Color string  
}  
type Face struct {  
    Eye  
    Hair  
    Color string  
}
```

```
face := new(Face)  
face.Color = "Fair"  
face.Eye.Count = 2  
face.Eye.Color = "Brown"  
face.Hair.Color = "Blonde"  
jsonBytes, _ = json.Marshal(face)  
// outputs:  
// {"Count":2,"Texture":"","Color":"Fair"}
```



Parent or Outer
structs have priority

OVERLAPPING ANONYMOUS FIELDS

- Anonymous fields with overlap are a point of caution:

```
type Eye struct {  
    Count int  
    Color string  
}  
type Hair struct {  
    Texture string  
    Color    string  
}  
type Face struct {  
    MyEye Eye  
    MyHair Hair  
}  
  
face := new(Face)  
face.Eye.Count = 2  
face.Eye.Color = "Brown"  
face.Hair.Color = "Blonde"  
jsonBytes, _ = json.Marshal(face)  
// outputs:
```

MARSHALER/UNMARSHALER INTERFACE

THE INTERFACES

```
type Marshaler interface {  
    MarshalJSON() ([]byte, error)  
}
```

```
type Unmarshaler interface {  
    UnmarshalJSON([]byte) error  
}
```

WHY?

- implemented by some StdLib types:
 - RawMessage, Time, big.Int
- Design so YOU can use them
 - your own custom structs
 - customize the serialization
 - example: you want UUIDs to be in the readable form:
 - de305d54-75b4-431b-adb2-eb6b9e546013
 - instead of [1 2 3 4 5 ...]

EXAMPLE: TIME.TIME

```
// MarshalJSON implements the json.Marshaler interface.
// The time is a quoted string in RFC 3339 format,
// with sub-second precision added if present.
func (t Time) MarshalJSON() ([]byte, error) {
    if y := t.Year(); y < 0 || y >= 10000 {
        // RFC 3339 is clear that years are 4 digits exactly.
        // See golang.org/issue/4556#c15 for more discussion.
        return nil, errors.New("Time.MarshalJSON: year out<snip>")
    }
    return []byte(t.Format(`"` + RFC3339Nano + `"`)), nil
}

// UnmarshalJSON implements the json.Unmarshaler interface.
// The time is expected to be a quoted string in RFC 3339 format.
func (t *Time) UnmarshalJSON(data []byte) (err error) {
    // Fractional seconds are handled implicitly by Parse.
    *t, err = Parse(`"` + RFC3339 + `"` , string(data))
    return
}
```


EXAMPLE: TIME.TIME

```
type Meal struct {  
    Food string  
    Timestamp time.Time  
}
```

```
lunch := Meal{"sushi", time.Now()}  
jsonBytes, _ = json.Marshal(lunch)  
fmt.Println(string(jsonBytes))
```

// outputs:

```
{"Food": "Sushi", "Timestamp": "2015-03-05T10:54:58.6384  
73751-08:00"}
```

Quotes manually inserted by
MarshalJSON() implementation

EXAMPLE: HUID

```
type HUID [8]byte

func (huid HUID) String() string {
    return fmt.Sprintf(
        "%s-%s",
        hex.EncodeToString(huid[0:4]),
        hex.EncodeToString(huid[4:8]),
    )
}

type Clown struct {
    Huid      HUID
    Name      string
    NoseColor string
}

huid := HUID{1, 2, 3, 4, 10, 11, 12, 13}
fmt.Println(huid)
// outputs
010203040-a0b0c0d
```

EXAMPLE: HUID

```
huid := HUID{1, 2, 3, 4, 10, 11, 12, 13}
wally := Clown{huid, "Wally", "Red"}
jsonBytes, _ = json.Marshal(wally)
fmt.Println(string(jsonBytes))

// outputs (pretty printed for legibility)
{
    "Huid": [1,2,3,4,10,11,12,13],
    "Name": "Wally",
    "NoseColor": "Red"
}
```


HUID: MARSHALJSON()

```
func (huid HUID) MarshalJSON() ([]byte, error) {  
    return []byte(huid.String()), nil  
}
```

```
huid := HUID{1, 2, 3, 4, 10, 11, 12, 13}  
wally := Clown{huid, "Wally", "Red"}  
_, err = json.Marshal(wally)  
fmt.Println(err)
```

// outputs

json: error calling MarshalJSON for type main.HUID: invalid character '1' after top-level value

// This code is attempting to generate:

// {"Huid":01020304-0a0b0c0d,"Name":"Wally","NoseColor":"Red"}

// which is not valid json

HUID: MARSHALJSON()

```
func (huid HUID) MarshalJSON() ([]byte, error) {  
    return []byte(`" + huid.String() + "`), nil  
}
```

```
huid := HUID{1, 2, 3, 4, 10, 11, 12, 13}  
wally := Clown{huid, "Wally", "Red"}  
jsonBytes, _ = json.Marshal(wally)  
fmt.Println(jsonBytes)
```

```
// outputs
```

```
{"Huid":"01020304-0a0b0c0d","Name":"Wally","NoseColor  
":"Red"}
```

HUID: UNMARSHALJSON()

```
func (huid *HUID) UnmarshalJSON(data []byte) (err error) {
    // not a truly robust solution...
    // in practice check err, len, etc.
    unquoted := strings.Trim(string(data), `"` )
    first_half, _ := hex.DecodeString(unquoted[0:8])
    second_half, _ := hex.DecodeString(unquoted[9:17])
    for i := 0; i < 4; i++ {
        huid[i] = first_half[i]
        huid[i+4] = second_half[i]
    }
    return nil
}
```

```
jsonBytes = []byte(`{"Huid":"11223344-aabbccdd","Name":"Squally","NoseColor":"Blue"}`)
squally := new(Clown)
json.Unmarshal(jsonBytes, squally)
fmt.Println(squally)
// outputs
&{11223344-aabbccdd Squally Blue}
```


WHEN?

- Custom types (MyInt, etc)
- Enums (iota) or fixed sets
 - Especially when the internal representation is a number, but the logical representation is a word.
- When you want more human readable representation (eg. time.Time instead of a unix timestamp)
- When you have existing parsers/formatters or encoders/decoders

RAWMESSAGE

JSON.RawMessage

- Sometimes you don't want to deserialize the whole thing.

```
// hypothetical json pattern:  
{"type":"shape","raw":{"sides":4}}  
{"type":"person","raw":{"name":"lucy"}}  
{"type":"list","raw":[1,2,"abc"]}
```

```
type Generic struct {  
    Type string  
    Raw  *json.RawMessage  
}
```


JSON.RawMessage INTERNALS

- It's just bytes

```
// RawMessage is a raw encoded JSON object.  
// It implements Marshaler and Unmarshaler and can  
// be used to delay JSON decoding or precompute a JSON encoding.  
type RawMessage []byte
```

```
// MarshalJSON returns *m as the JSON encoding of m.  
func (m *RawMessage) MarshalJSON() ([]byte, error) {  
    return *m, nil  
}
```

```
// UnmarshalJSON sets *m to a copy of data.  
func (m *RawMessage) UnmarshalJSON(data []byte) error {  
    if m == nil {  
        return errors.New("json.RawMessage: UnmarshalJSON on nil pointer")  
    }  
    *m = append((*m)[0:0], data...)  
    return nil  
}
```

JSON.RawMessage DESERIALIZE

```
type Shape struct {  
    Sides int  
}  
  
func decode(g Generic) interface{} {  
    switch g.Type {  
    case "shape":  
        jsonBytes, _ := g.Raw.MarshalJSON()  
        var s Shape  
        _ = json.Unmarshal(jsonBytes, &s)  
        return s  
    case "list":  
        var l interface{}  
        _ = json.Unmarshal(jsonBytes, &l)  
        return l  
    default:  
        return nil  
    }  
}
```

JSON.RawMessage DESERIALIZE

```
func decode(g Generic) interface{} {
    switch g.Type {
    case "shape":
        jsonBytes, _ := g.Raw.MarshalJSON()
        var s Shape
        _ = json.Unmarshal(jsonBytes, &s)
        return s
    case "list":
        var l interface{}
        _ = json.Unmarshal(jsonBytes, &l)
        return l
    default:
        return nil
    }
}

_ = json.Unmarshal([]byte(`{"type":"shape","raw":{"sides":4}}`), &g1)
decoded_raw := decode(g1)
fmt.Println("Generic struct: g1", g1)
fmt.Printf("decoded_raw type: %T, value: %+v\n", decoded_raw, decoded_raw)

// Generic struct: g1 {shape 0x20820e840}
// decoded_raw type: main.Shape, value: {Sides:4}
```


JSON.RawMessage DESERIALIZE

```
func decode(g Generic) interface{} {
    switch g.Type {
    case "shape":
        jsonBytes, _ := g.Raw.MarshalJSON()
        var s Shape
        _ = json.Unmarshal(jsonBytes, &s)
        return s
    case "list":
        var l interface{}
        _ = json.Unmarshal(jsonBytes, &l)
        return l
    default:
        return nil
    }
}

_ = json.Unmarshal([]byte(`{"type":"list","raw":[1,2,"abc"]}`), &g3)
decoded_raw = decode(g3)
fmt.Println("Generic struct: g3", g3)
fmt.Printf("decoded_raw type: %T, value: %+v\n", decoded_raw, decoded_raw)

// Generic struct: g3 {list 0x20820e940}
// decoded_raw type: []interface {}, value: [1 2 abc]
```

JSON.RawMessage SERIALIZE

- If you have []byte that you want to turn into a RawMessage, you have to use RawMessage.UnmarshalJSON(data)

```
triangle := Shape{3}
g4 := new(Generic)
g4.Type = "shape"
jsonBytes, _ = json.Marshal(triangle)
raw_message := new(json.RawMessage)
raw_message.UnmarshalJSON(jsonBytes)
g4.Raw = raw_message
```

```
jsonBytes, _ = json.Marshal(g4)
fmt.Println(string(jsonBytes))
// output
{"Type":"shape","Raw":{"Sides":3}}
```

JSON.RawMessage SERIALIZE

- Usually, you make a convenience method:

```
func (s Shape) Genericize() Generic {  
    g := Generic{}  
    g.Type = "shape"  
    jsonBytes, _ := json.Marshal(s)  
    raw_message := new(json.RawMessage)  
    raw_message.UnmarshalJSON(jsonBytes)  
    g.Raw = raw_message  
    return g  
}
```

```
pentagon := Shape{5}  
jsonBytes, _ = json.Marshal(pentagon.Genericize())  
fmt.Println(string(jsonBytes))  
// output  
{"Type":"shape","Raw":{"Sides":5}}
```


MISC

QUICK REST API

```
http.HandleFunc("/api/shape/", func(w http.ResponseWriter, r *http.Request) {
    path_components := strings.Split(r.URL.Path, "/")
    last_path_component := path_components[len(path_components)-1]
    sides, err := strconv.ParseInt(last_path_component, 10, 64)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("400 You messed up"))
    } else {
        shape := Shape{int(sides)}
        jsonBytes, _ := json.Marshal(shape)
        w.Header().Add("Content-Type", "application/json")
        w.Write(jsonBytes)
    }
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

PRETTY PRINTING

```
// http://play.golang.org/p/UZ4YyZg8Bd
```

```
jsonBytes := []byte(`{"Names":["square","rectangle",1],"Sides":4,
"Color":"red"}`)
var buf bytes.Buffer
json.Indent(&buf, jsonBytes, "", "\t")
pretty, _ := ioutil.ReadAll(&buf)
fmt.Println(string(pretty))
```

```
// Outputs:
```

```
{
    "Names": [
        "square",
        "rectangle",
        1
    ],
    "Sides": 4,
    "Color": "red"
}
```


COMPACTING

// http://play.golang.org/p/hIxM_nwqrq

```
jsonBytes := []byte(`{
    "Names": [
        "square",
        "rectangle",
        1
    ],
    "Sides": 4,
    "Color": "red"
}`)
var buf bytes.Buffer
json.Compact(&buf, jsonBytes)
pretty, _ := ioutil.ReadAll(&buf)
fmt.Println(string(pretty))

// Outputs
{"Names":["square","rectangle",1],"Sides":4,"Color":"red"}
```

THANK YOU, CREDITS & LICENSE

<http://gotutorial.net>
@GoTutorialNet

Matt Nunogawa
@amattn

- I owe many many, thanks to the many authors of Go and to Rob Pike in particular.
- These slides are Copyright 2013-2014 Matthew Nunogawa
- All content is licensed under the Creative Commons Attribution 4.0 License (<http://creativecommons.org/licenses/by/4.0/>)
 - attribution: Matt Nunogawa, Copyright 2013-2014 Matthew Nunogawa, <http://gotutorial.net>
- All code is licensed under a BSD License (<http://opensource.org/licenses/BSD-2-Clause>)