Chapter 24: Reading & Writing JSON

Section 24.1: JSON Serialization, Encoding, and Decoding with Apple Foundation and the Swift Standard Library

The JSONSerialization class is built into Apple's Foundation framework.

`Version = 2.2`

Read JSON

The `JSONObjectWithData` function takes `NSData`, and returns `AnyObject`. You can use `as?` to convert the result to your expected type.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else {
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}
```

You can pass `options: .AllowFragments` instead of `options: []` to allow reading JSON when the top-level object isn't an array or dictionary.

Write JSON

Calling `dataWithJSONObject` converts a JSON-compatible object (nested arrays or dictionaries with strings, numbers, and `NSNull`) to raw `NSData` encoded as UTF-8.

```
do {
    // Convert object to JSON as NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

You can pass `options: .PrettyPrinted` instead of `options: []` for pretty-printing.

`Version = 3.0`

Same behavior in Swift 3 but with a different syntax.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
```

```
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}

do {
    // Convert object to JSON as NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

> Note: The Following is currently available only in **Swift 4.0** and later.

As of Swift 4.0, the Swift standard library includes the protocols `Encodable` and `Decodable` to define a standardized approach to data encoding and decoding. Adopting these protocols will allow implementations of the `Encoder` and `Decoder` protocols take your data and encode or decode it to and from an external representation such as JSON. Conformance to the `Codable` protocol combines both the `Encodable` and `Decodable` protocols. This is now the recommended means to handle JSON in your program.

Encode and Decode Automatically

The easiest way to make a type codable is to declare its properties as types that are already `Codable`. These types include standard library types such as `String`, `Int`, and `Double`; and Foundation types such as `Date`, `Data`, and `URL`. If a type's properties are codable, the type itself will automatically conform to `Codable` by simply declaring the conformance.

Consider the following example, in which the `Book` structure conforms to `Codable`.

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Note that standard collections such as `Array` and `Dictionary` conform to `Codable` if they contain codable types.

By adopting `Codable`, the `Book` structure can now be encoded to and decoded from JSON using the Apple Foundation classes `JSONEncoder` and `JSONDecoder`, even though `Book` itself contains no code to specifically handle

JSON. Custom encoders and decoders can be written, as well, by conforming to the `Encoder` and `Decoder` protocols, respectively.

Encode to JSON data

```
// Create an instance of Book called book
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // Do not use try! in production code
print(data)
```

> Set `encoder.outputFormatting = .prettyPrinted` for easier reading. ## Decode from JSON data

Decode from JSON data

```
// Retrieve JSON string from some source
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // Do not use try! in production code
print(book)
```

> In the above example, `Book.self` informs the decoder of the type to which the JSON should be decoded.

Encoding or Decoding Exclusively

Sometimes you may not need data to be both encodable and decodable, such as when you need only read JSON data from an API, or if your program only submits JSON data to an API.

If you intend only to write JSON data, conform your type to `Encodable`.

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

If you intend only to read JSON data, conform your type to `Decodable`.

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Using Custom Key Names

APIs frequently use naming conventions other than the Swift-standard camel case, such as snake case. This can become an issue when it comes to decoding JSON, since by default the JSON keys must align exactly with your type's property names. To handle these scenarios you can create custom keys for your type using the `CodingKey` protocol.

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
        case authors
```

```
        case publicationDate = "publication_date"
    }
}
```

`CodingKeys` are generated automatically for types which adopt the `Codable` protocol, but by creating our own implementation in the example above we're allow our decoder to match the local camel case `publicationDate` with the snake case `publication_date` as it's delivered by the API.

Section 24.2: SwiftyJSON

SwiftyJSON is a Swift framework built to remove the need for optional chaining in normal JSON serialization.

You can download it here: https://github.com/SwiftyJSON/SwiftyJSON

Without SwiftyJSON, your code would look like this to find the name of the first book in a JSON object:

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments) as?
[[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //We can now use the book name
}
```

In SwiftyJSON, this is hugely simplified:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

It removes the need to check every field, as it will return nil if any of them are invalid.

To use SwiftyJSON, download the correct version from the Git repository - there is a branch for Swift 3. Simply drag the "SwiftyJSON.swift" into your project and import into your class:

```
import SwiftyJSON
```

You can create your JSON object using the following two initializers:

```
let jsonObject = JSON(data: dataObject)
```

or

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

To access your data, use subscripts:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

You can then parse your value to a certain data type, which will return an optional value:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string

let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

You can also compile your paths into a swift Array:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Is the same as:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON also has functionality to print its own errors:

```
if let name = json[1337].string {
    //You can use the value - it is valid
} else {
    print(json[1337].error) // "Array[1337] is out of bounds" - You cant use the value
}
```

If you need to write to your JSON object, you can use subscripts again:

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //This changes the age to 25
originalJSON["surname"] = "Smith" //This creates a new field called "surname" and adds the value to
it
```

Should you need the original String for the JSON, for example if you need to write it to a file, you can get the raw value out:

```
if let string = json.rawString() { //This is a String object
    //Write the string to a file if you like
}

if let data = json.rawData() { //This is an NSData object
    //Send the data to your server if you like
}
```

Section 24.3: Freddy

Freddy is a JSON parsing library maintained by Big Nerd Ranch. It has three principal benefits:

1. Type Safety: Helps you work with sending and receiving JSON in a way that prevents runtime crashes.
2. Idiomatic: Takes advantage of Swift's generics, enumerations, and functional features, without complicated documentation or magical custom operators.
3. Error Handling: Provides informative error information for commonly occurring JSON errors.

Example JSON Data

Let's define some example JSON data for use with these examples.

```
{
  "success": true,
  "people": [
    {
      "name": "Matt Mathias",
      "age": 32,
      "spouse": true
    },
```

```
    {
      "name": "Sergeant Pepper",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "teacher",
    "judge"
  ],
  "states": {
    "Georgia": [
      30301,
      30302,
      30303
    ],
    "Wisconsin": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"Matt Mathias\",\"age\":
32,\"spouse\": true},{\"name\": \"Sergeant Pepper\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"teacher\",\"judge\"],\"states\": {\"Georgia\": [30301,30302,30303],\"Wisconsin\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

Deserializing Raw Data

To deserialize the data, we initialize a `JSON` object then access a particular key.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

We `try` here because accessing the `json` for the key `"success"` could fail--it might not exist, or the value might not be a boolean.

We can also specify a path to access elements nested in the JSON structure. The path is a comma-separated list of keys and indices that describe the path to a value of interest.

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // do something with the error
}
```

Deserializing Models Directly

JSON can be directly parsed to a model class that implements the `JSONDecodable` protocol.

```
public struct Person {
    public let name: String
    public let age: Int
```

```
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // do something with the error
}
```

Serializing Raw Data

Any `JSON` value can be serialized directly to `NSData`.

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Serializing Models Directly

Any model class that implements the `JSONEncodable` protocol can be serialized directly to `NSData`.

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

Section 24.4: JSON Parsing Swift 3

Here is the JSON File we will be using called `animals.json`

```
{
    "Sea Animals": [
    {
     "name": "Fish",
     "question": "How many species of fish are there?"    },
                {
                "name": "Sharks",
                "question": "How long do sharks live?"
                },
                {
                "name": "Squid",
                "question": "Do squids have brains?"
                },
                {
                "name": "Octopus",
```

```
                "question": "How big do octopus get?"
                },
                {
                "name": "Star Fish",
                "question": "How long do star fish live?"
                }
                ],
    "mammals": [
                {
      "name": "Dog",
      "question": "How long do dogs live?"
    },
    {
      "name": "Elephant",
      "question": "How much do baby elephants weigh?"
    },
    {
      "name": "Cats",
      "question": "Do cats really have 9 lives?"
    },
    {
      "name": "Tigers",
      "question": "Where do tigers live?"
    },
    {
      "name": "Pandas",
      "question": "WHat do pandas eat?"
    }   ] }
```

Import your JSON File in your project

You can perform this simple function to print out your JSON file

```
   func jsonParsingMethod() {
        //get the file
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//Call which part of the file you'd like to pare
        if let results = json["mammals"] as? [[String: AnyObject]] {

            for res in results {
                //this will print out the names of the mammals from out file.
                if let rates = res["name"] as? String {
                    print(rates)
                }
            }
       }
    }
```

If you want to put it in a table view, I would create a dictionary first with an NSObject.

Create a new swift file called `ParsingObject` and create your string variables.

> Make sure that the variable name is the same as the JSON File

. For example, in our project we have `name` and `question` so in our new swift file, we will use

```
var name: String?
var question: String?
```

Initialize the NSObject we made back into our ViewController.swift var array = ParsingObject Then we would perform the same method we had before with a minor modification.

```
     func jsonParsingMethod() {
            //get the file
            let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
            let content = try! String(contentsOfFile: filePath!)

            let data: Data = content.data(using: String.Encoding.utf8)!
            let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//This time let's get Sea Animals
            let results = json["Sea Animals"] as? [[String: AnyObject]]

//Get all the stuff using a for-loop
            for i in 0 ..< results!.count {

//get the value
                let dict = results?[i]
                let resultsArray = ParsingObject()

//append the value to our NSObject file
                resultsArray.setValuesForKeys(dict!)
                array.append(resultsArray)

            }

        }
```

Then we show it in our tableview by doing this,

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
            return array.count
        }

        func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
            let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//This is where our values are stored
            let object = array[indexPath.row]
            cell.textLabel?.text = object.name
            cell.detailTextLabel?.text = object.question
            return cell
        }
```

Section 24.5: Simple JSON parsing into custom objects

Even if third-party libraries are good, a simple way to parse the JSON is provided by protocols You can imagine you have got an object `Todo` as

```
struct Todo {
    let comment: String
```

```
}
```

Whenever you receive the JSON, you can handle the plain `NSData` as shown in the other example using `NSJSONSerialization` object.

After that, using a simple protocol `JSONDecodable`

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

And making your `Todo` struct conforming to `JSONDecodable` does the trick

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

You can try it with this json code:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

When you got it from the API, you can serialize it as the previous examples shown in an `AnyObject` instance. After that, you can check if the instance is a `JSONDictionary` instance

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

The other thing to check, specific for this case because you have an array of `Todo` in the JSON, is the `todos` dictionary

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Now that you got the array of dictionaries, you can convert each of them in a `Todo` object by using `flatMap` (it will automatically delete the `nil` values from the array)

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

Section 24.6: Arrow

Arrow is an elegant JSON parsing library in Swift.

It allows to parse JSON and map it to custom model classes with help of an

<--

operator:

```
identifier <-- json["id"]
name <-- json["name"]
stats <-- json["stats"]
```

Example:

Swift model

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

JSON file

```
{
    "id": 15678,
    "name": "John Doe",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                    "label": "house",
                    "number": "9809876545"
                    }, {
                    "label": "cell",
                    "number": "0908070656"
                    }, {
                    "label": "work",
                    "number": "0916570656"
    }]
}
```

Mapping

```
extension Profile: ArrowParsable {
    mutating func deserialize(json: JSON) {
        identifier <-- json["id"]
        link <-- json["link"]
        name <-- json["name"]
        weekday <-- json["weekdayInt"]
        stats <- json["stats"]
        phoneNumbers <-- json["phoneNumbers"]
    }
}
```

Usage

```
let profile = Profile()
profile.deserialize(json)
```

Installation:

Carthage

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

Manually

Simply Copy and Paste Arrow.swift in your Xcode Project

https://github.com/s4cha/Arrow

As A Framework

Download Arrow from the GitHub repository and build the Framework target on the example project. Then Link against this framework.