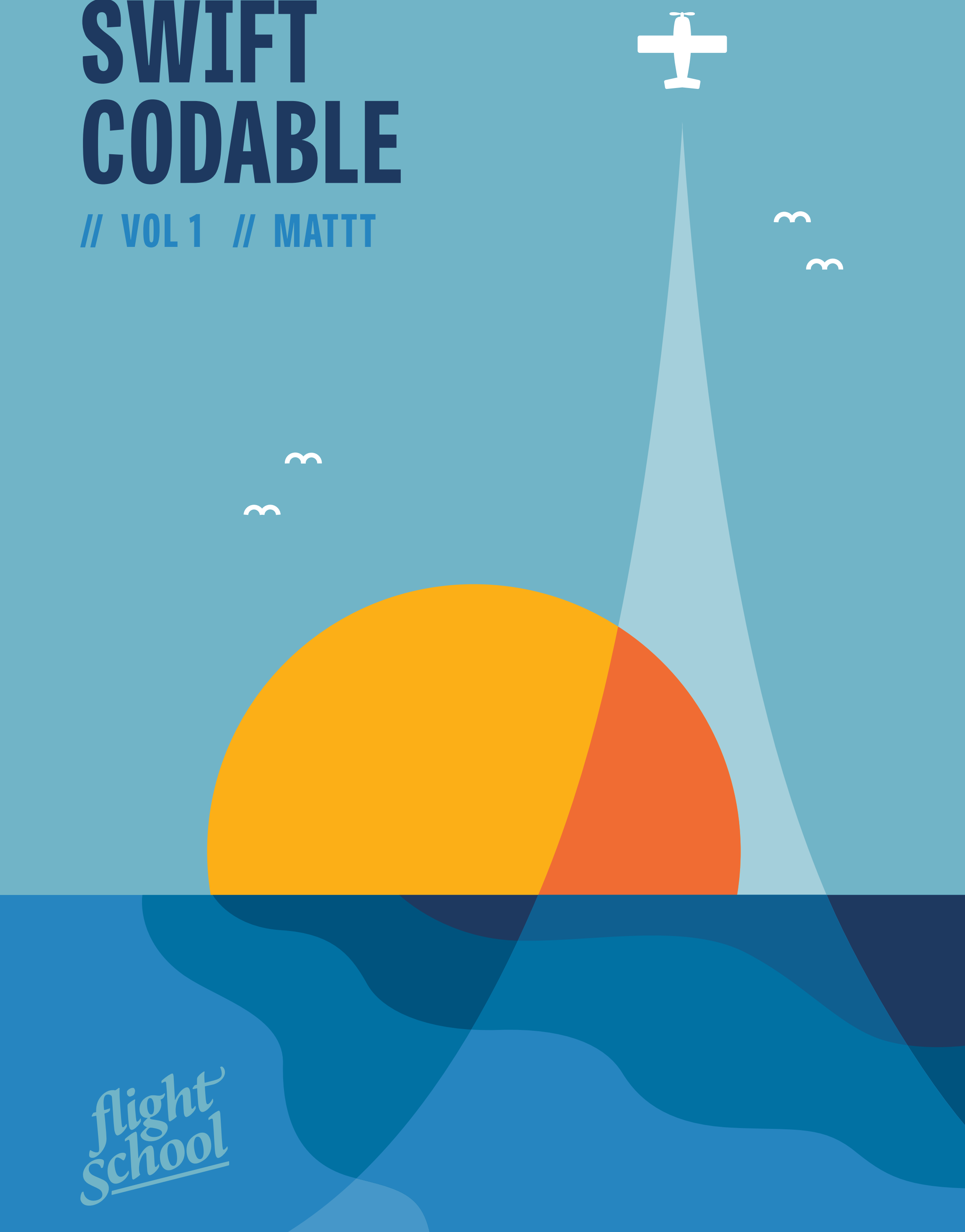
GUIDE TO SWIFT CODABLE

// VOL 1 // MATTT



flight school

Flight School
An imprint of Read Evaluate Press, LLC
Portland, Oregon
https://readeval.press

Publisher	Mattt Zmuda
Illustrator	Lauren Mendez
Cover Designer	Mette Hornung Rankin
Editor	Morgan Tarling

ISBN	978-1-949080-00-1

Printed in IBM Plex, designed by Mike Abbink at IBM in collaboration with Bold Monday.
Additional glyphs provided by Google Noto Sans.
Twemoji graphics made by Twitter and other contributors.
Cover type set in Tablet Gothic, designed by José Scaglione & Veronika Burian.

Contents


flight
School

Chapter 1:
First Flight

For your first flight with `Codable`, let's take a round-trip journey by decoding a JSON payload into a model object and then encoding that model object back into JSON. We'll then circle around and try out these maneuvers in different array formations. Finally, we'll do a quick run-through of our emergency procedures to get a better idea of how to handle errors, should they arise.

By performing these simple drills, you'll get a 10,000-foot view of how to use `Codable`, and be ready to take on more advanced exercises in the next chapters.

So without further ado, let's head over to the ramp and check out our aircraft!

Inspecting the Airplane

Prior to engaging in any operation, it's customary to perform a visual inspection of the plane, starting at the left wingtip and moving around clockwise.

```json
{
  "manufacturer": "Cessna",
  "model": "172 Skyhawk",
  "seats": 4
}
```

This is a Cessna 172 Skyhawk, represented as JSON. The opening and closing curly braces (`{}`), aptly reminiscent of a fuselage, denote the start and end of the aircraft. Inside this top-level object, there are three attributes listed as key/value pairs:

Key	Value
`"manufacturer"`	“Cessna”
`"model"`	“172 Skyhawk”
`"seats"`	4

Note:

JSON, or **J**ava**S**cript **O**bject **N**otation, is a text-based format for representing information. It’s easy for both humans and computers to read and write, which has helped make it a ubiquitous standard on the web.

Building Out a Model

To interact programmatically with this aircraft (and others like it), we create a Swift type that fits the JSON representation, matching up each JSON value to its counterpart in Swift’s type system.

JSON can represent ordered structures (*arrays*) and unordered structures (*objects*), each of which may contain any combination of strings, numbers, or the values `true`, `false`, and `null`. Fortunately for us, each of these JSON types maps rather nicely onto something in Swift.

JSON	Swift Types
object	`Dictionary`[1]
array	`Array`
`true` / `false`	`Bool`
`null`	`Optional`[2]
string	`String`
number	`Int`, `Double`, et. al.[3]

1. Objects are unordered key-value pairs. They’re analogous to a Swift `Dictionary` with `String` key type and can be converted to and from types that conform to `Codable`.
2. `Codable` can automatically map `null` values to `nil` for properties with an `Optional` type.
3. JSON represents numbers as a sequence of digits, agnostic of any semantics; no distinction is made between integer and floating-point, fixed and variable-width, or binary and decimal formats.

   Each implementation chooses how to interpret number values. `JSONDecoder` (and its underlying

We start by defining a new structure, `Plane`, to correspond to the top-level object in the payload. In this example, we use a structure, but a class would work just as well. Each JSON attribute becomes a Swift property with the key as the property name and the value type as the property type. Both the `manufacturer` and `model` attributes have string values and map onto `String` properties with the same name. The `seats` attribute has a number value and maps onto the `seats` property of type `Int`.

```json
{
  "manufacturer": "Cessna",
  "model": "172 Skyhawk",
  "seats": 4
}
```

```swift
struct Plane {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

Now that we've defined a `Plane` structure, the next objective is to decode one from our JSON payload.

Introducing the Codable Protocol

Swift 4.0 introduces a new language feature called `Codable`, which vastly improves the experience of converting objects to and from a representation.

The best way to understand `Codable` is to look at its declaration:

```swift
typealias Codable = Decodable & Encodable
```

`Codable` is a *composite type* consisting of the `Decodable` and `Encodable` protocols.

---

processor, `JSONSerialization`) interprets number values using `NSNumber`, and provides conversions into most number types in the Swift Standard Library.

The Decodable protocol defines a single initializer:

```
init(from decoder: Decoder) throws
```

Types that conform to the Decodable protocol can be initialized by any Decoder type.

The Encodable protocol defines a single method:

```
func encode(to encoder: Encoder) throws
```

If a type conforms to the Encodable protocol, any Encoder type can create a representation for a value of that type.

Let's return to our Plane model and take Decodable out for a spin.

Adopting Decodable in the Model

You *adopt* a protocol by adding the protocol's name after the type's name, separated by a colon.

```
struct Plane: Decodable {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

A type is said to *conform* to a protocol if it satisfies all the requirements of that protocol. For Decodable, the only requirement is for the type to have an implementation of `init(from:)`.

The `init(from:)` initializer takes a single Decoder argument. Decoder is a protocol that specifies the requirements for decoding a Decodable object from a representation. In order to accommodate a variety of data interchange formats, including JSON and property lists, both decoders and encoders use an abstraction called *containers*. A container is something that holds a value. It can hold a single value, or it can hold multiple values — either keyed, like a dictionary, or unkeyed, like an array.

Because the JSON payload has an object at the top level, we'll create a keyed container and then decode each property value by its respective key.

`Codable` expects a type called `CodingKeys` that conforms to the `CodingKey` protocol, which defines a mapping between Swift property names and container keys. This type is typically an enumeration with a `String` raw value, because keys are unique and represented by string values.

Let's create a `CodingKeys` enumeration for `Plane`:

```
struct Plane: Decodable {
    // ...

    private enum CodingKeys: String, CodingKey {
        case manufacturer
        case model
        case seats
    }
}
```

We don't need to provide an explicit raw value for any of the enumeration cases. because the names of each property are the same as the corresponding JSON key.

Next, in the `init(from:)` initializer, we create a keyed container by calling the `container(keyedBy:)` method on `decoder` and passing our `CodingKeys` type as an argument.[4]

4. The postfix `self` expression allows types to be used as values.

Finally, we initialize each property value by calling the `decode(_:,forKey:)` method on `container`:

```
init(from decoder: Decoder) throws {
  let container =
    try decoder.container(keyedBy: CodingKeys.self)

  self.manufacturer =
    try container.decode(String.self,
                         forKey: .manufacturer)
  self.model =
    try container.decode(String.self,
                         forKey: .model)
  self.seats =
    try container.decode(Int.self,
                         forKey: .seats)
}
```

We have a `Plane` model, and by conforming to the `Decodable` protocol, we can now create a `Plane` object from a `JSON` representation. We're now ready for take-off.

Decoding JSON Into a Model Object

Start by importing Foundation. We'll need it for `JSONDecoder` and `JSONEncoder`.

```
import Foundation
```

Apps typically load JSON from a network request or a local file. For simplicity, we can define the JSON directly in source code using another feature added in Swift 4.0: *multi-line string literals*.

```
let json = """
{
    "manufacturer": "Cessna",
    "model": "172 Skyhawk",
    "seats": 4,
}
""".data(using: .utf8)!
```

Multi-line string literals are delimited by triple quotation marks (`"""`). Unlike conventional string literals, the multi-line variant allows newline returns as well as unescaped quotation marks ("). This makes it perfectly suited for representing JSON in source code.

The `data(using:)` method converts a string into a `Data` value that can be passed to a decoder. This method returns an optional, and in most cases, a `guard` statement with a conditional assignment (`if-let`) would be preferred. However, because we're calling this method on a literal string value that we know will always produce valid UTF-8 data, we can use the forced unwrapping postfix operator (`!`) to get a nonoptional `Data` value.

Next, we create a `JSONDecoder` object and use it to call the `decode(_:from:)` method[5] . This method can throw an error. For expediency, we're using `try!` for now instead of doing proper error handling (we'll do it the right way on the next go around).

```
let decoder = JSONDecoder()

let plane = try! decoder.decode(Plane.self, from: json)
```

We have liftoff! Go ahead and check it out for yourself.

```
print(plane.manufacturer)
// Prints "Cessna"

print(plane.model)
// Prints "172 Skyhawk"

print(plan.seats)
// Prints "4"
```

5. Swift could infer the type of the decode method without supplying the type as the first argument, but a design decision was made to make this explicit to reduce ambiguity about what's happening.

Encoding a Model Object Into JSON

In order to complete the loop, we need to update `Plane` to adopt `Encodable`.

```
struct Plane: Decodable, Encodable { }
```

As we saw earlier, `Codable` is defined as a typealias for the composition of `Decodable` and `Encodable`. Therefore, we can simplify this further:

```
struct Plane: Codable { }
```

Next, we implement the `encode(to:)` method required by `Encodable`. As you might expect, `encode(to:)` reads much like a mirror image of `init(from:)`. Start by creating a container by calling the `container(keyedBy:)` method on `encoder` and passing `CodingKeys.self` as we did before. Here, `container` is a variable (`var` instead of `let`), because this method populates the `encoder` argument, which requires modification. For each property, we call `encode(_:forKey:)`, passing the property's value and its corresponding key.

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(self.manufacturer,
                         forKey: .manufacturer)
    try container.encode(self.model,
                         forKey: .model)
    try container.encode(self.seats,
                         forKey: .seats)
}
```

We've done all of the necessary setup, and all systems are go. Let's make for our final approach and complete our loop:

```
let encoder = JSONEncoder()
let reencodedJSON = try! encoder.encode(plane)

print(String(data: reencodedJSON, encoding: .utf8)!)
// Prints "{"manufacturer":"Cessna",
//          "model":"172 Skyhawk",
//          "seats":4}"
```

Aces! Other than a lack of whitespace, what we got back from `JSONEncoder` is exactly what we originally put into `JSONDecoder`.[6]

We could keep going around and around, converting back and forth between model and representation to our heart's content. If you think about the demands of a real app as it brokers information requests between client and server, this is exactly the kind of guarantee we need to ensure that nothing is lost in translation.

Deleting Unnecessary Code

Now we're going to do something a bit unexpected: delete all of the code we've written so far to conform to `Codable`. This should leave only the structure and property declarations.

```
struct Plane: Codable {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

Go ahead and try running the code to decode and encode JSON. What changed? *Absolutely nothing.* Which leads us to the true killer feature of `Codable`:

---

6. JSON isn't whitespace sensitive, but you might be. If you find compressed output to be aesthetically challenging, set the `outputFormatting` property of `JSONEncoder` to `.prettyPrinted`.

Swift automatically synthesizes conformance for `Decodable` and `Encodable`.

So long as a type adopts the protocol in its declaration — that is, not in an extension — and each of its properties has a type that conforms, everything is taken care of for you.

We can see for ourselves that `Plane` meets all of the criteria for synthesizing conformance to `Codable`[7] :

- ☑ `Plane` adopts `Codable` in its type declaration.
- ☑ Each of the properties in `Plane` has types conforming to `Codable`.

Most built-in types in the Swift Standard Library and many types in the Foundation framework conform to `Codable`. So most of the time, it's just up to you to keep the `Codable` party going.

> **Note:**
>
> Aspiring pilots may be alarmed when — halfway into their introductory flight, cruising at a few thousand feet — their flight instructor tells them to cut power to the engine to see what happens. (Just as here, absolutely nothing. You just glide.) Not all CFIs do this, but it's a visceral demonstration of the physics of flight.

If you're miffed about writing all of that code when it wasn't necessary, please be advised: there are occasions in which you do need to implement conformance manually. We'll be covering that in Chapter 3.

For now, let's collect ourselves and get ready for another loop.

7. To be more precise, Swift automatically synthesizes conformance `Decodable` and `Encodable`. Think of `Codable` synthesis as two separate passes: one for `Decodable` and another for `Encodable`.

Flying in Formation

Objects don't always fly solo. In fact, it's quite common for payloads to have arrays of objects in them.

For example, a JSON response may have an array as its top-level value:

```
[
  {
    "manufacturer": "Cessna",
    "model": "172 Skyhawk",
    "seats": 4
  },
  {
    "manufacturer": "Piper",
    "model": "PA-28 Cherokee",
    "seats": 4
  }
]
```

Decoding this into an array of `Plane` objects couldn't be simpler: take our call to `decode(_:from:)` from before, and replace `Plane.self` with `[Plane].self`. This changes the decoded type from a `Plane` object to an array of `Plane` objects.

```
let planes = try! decoder.decode([Plane].self, from: json)
```

`[Plane]` is syntactic shorthand for `Array<Plane>`, or an `Array` with the generic constraint that its elements are `Plane` objects. Why does this work? It's thanks to *conditional conformance* — another feature of Swift 4.

```
// swift/stdlib/public/core/Codable.swift.gyb
extension Array : Decodable where Element : Decodable {
    // ...
}
```

Although top-level arrays are technically valid JSON, it's considered best practice to always have an object at the top level instead. For example, the following JSON has an array keyed off the top-level object at `"planes"`:

```
{
  "planes": [
    {
      "manufacturer": "Cessna",
      "model": "172 Skyhawk",
      "seats": 4
    },
    {
      "manufacturer": "Piper",
      "model": "PA-28 Cherokee",
      "seats": 4
    }
  ]
}
```

Like `Array`, the `Dictionary` type conditionally conforms to `Decodable` if its associated `KeyType` and `ValueType` conform to `Decodable`:

```
// swift/stdlib/public/core/Codable.swift.gyb
extension Dictionary : Decodable where Key : Decodable,
                                       Value : Decodable {
    // ...
}
```

Because of this, you can change the same call to `decode(_:from:)` from before and pass `[String: [Plane]].self` (shorthand for `Dictionary<String, Array<Plane>>`) instead:

```
let keyedPlanes = try! decoder.decode([String: [Plane]].self,
                                      from: json)
let planes = keyedPlanes["planes"]
```

Alternatively, you can create a new type that conforms to `Decodable` and has a property whose name matches the JSON key and has a value of type `[Plane]`, and pass that to `decode(_:from:)`:

```
struct Fleet: Decodable {
    var planes: [Plane]
}

let fleet = try! decoder.decode(Fleet.self, from: json)
let planes = fleet.planes
```

You typically do this when the structure of your payload is semantically meaningful. If objects are nested arbitrarily, such as to namespace keys, you should probably just use a standard collection class.

Trying Our Best with Error Handling

Our first couple go-arounds with `Codable` went well, but our technique was a bit sloppy with those `try!` keywords. For this last lap, let's take a moment to show how the pros do it in real applications.

Swift functions and initializers marked with the `throws` keyword may generate an error instead of returning a value. Both the `Decodable` initializer `init(from:)` and the `Encodable` method `encode(to:)` are marked with the `throws` keyword — which makes sense, because it may not always be possible to complete the operation successfully. When decoding a representation into an object, the data might be corrupted, a key might be missing, there might be a type mismatch, or a nonoptional value might be missing. It's less typical for there to be problems during encoding; it really depends on the format.

Swift requires all errors to be handled one way or another. So when you call an expression marked with the `throws` keyword, you need to prefix it with the `try` operator or either of its variants.

```
do {
    let plane = try decoder.decode(Plane.self, from: json)
} catch {
    print("Error decoding JSON: \(error)")
}
```

Note:

For information about how to communicate errors to users, see the [Human Interface Guidelines](https://developer.apple.com/design/human-interface-guidelines/).

You can use the optional-try operator (`try?`) to convert a throwing expression into an optional expression. That is, the expression returns `nil` if an error occurs, otherwise, it returns the expression value in an optional. If you aren't doing anything meaningful in your error handling, this can be a useful shorthand.

```
if let plane = try? decoder.decode(Plane.self, from: json) {
    print(plane.model)
}
```

Alternatively, there's the forced-try operator (`try!`), which we've been using up until this point. Following Swift's language conventions, the trailing *"bang"*, or exclamation mark (`!`), indicates unsafe behavior. If you use the forced-try operator on an expression that results in an error, the program exits immediately. Because sudden app termination makes for a lousy user experience, you generally avoid `try!` in app code. However, for the purposes of exploring a new concept — such as in a Playground — failing fast can actually be to your advantage.

Thus concludes our first flight with `Codable`. If you feel like your head is spinning, don't worry — you'll get more comfortable with practice.

Things get even more exciting in Chapter 2. We'll apply what we learned here to a more challenging payload, and learn strategies for maximizing how much the compiler does for us.

Recap

- `Codable` is a *composite type* consisting of the `Decodable` and `Encodable` protocols.
- Swift automatically synthesizes conformance for `Decodable` and `Encodable` if a type adopts conformance in its declaration, and each of its stored properties also conforms to that protocol.
- An `Array` or `Dictionary` conforms to `Codable` if its associated element type is `Codable`-conforming.