
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

2: Swift 4 Serialization

Swift 4 introduced the `Codable` system, enabling format-independent serialization for all model types with minimal code.

In this hands-on tutorial, you will learn how to bend the `Codable` system to your will and master best practices though a series of bite-sized Swift playground missions.

4 Swift 4 Serialization: Demo 1

By Ray Fix

In this demo, you will gain experience with the fundamentals of `Codable` by building a simple Mars Rover model. A series of short missions will get you up to speed on encoding and decoding, how to get the compiler to generate code for you, how to customize the code it generates, and how to make your own implementations. You will also gain XP (experience points) on future-proofing your code.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo1\starter**.

1) Mission 1: Basic Codable Conformance

Each project has a number of `Codable` unit test missions for you to complete. The test fails and you will modify the code so that it passes. Each unit test bundle has some extra details in the file **Extras.swift**. These are details you have already covered or are not important to your understanding of the concept at hand.

Your first mission is to make `Rover` conform to `Codable` and encode and decode the `curiosity` instance with both a `JSONEncoder` and `JSONDecoder` as well as `PropertyListEncoder` and `PropertyListDecoder`.

Add the `Codable` conformance to each of the models so you have:

```
struct Sols: Codable {
  // ....
}

enum Camera: Codable {
  // ....
}
```

```
struct Photo: Codable {
  // ....
}

struct Rover: Codable {
  // ....
}
```

This almost works but the compiler cannot automatically generate code for the `Camera` enumeration. Make `Camera` a `RawRepresentable` with `String` by adding `: String` to its inheritance declaration like this:

```
enum Camera: String, Codable {
  // ....
}
```

You are done writing serialization code. Thanks Swift 4.

Next, kick the tires by encoding and decoding to JSON data. At the bottom, implement `testEncodeDecodeJSON()`

```
func testEncodeDecodeJSON() throws {
  let encoder = JSONEncoder()
  let data = try encoder.encode(curiosity)
  print(String(data: data, encoding: .utf8)!)

  let decoder = JSONDecoder()
  let restored = try decoder.decode(Rover.self, from: data)
  dump(restored)
}
```

Finally, try out property lists. Implement `testEncodeDecodePlist()`:

```
func testEncodeDecodePlist() throws {
  let encoder = PropertyListEncoder()
  encoder.outputFormat = .xml
  let data = try encoder.encode(curiosity)
  print(String(data: data, encoding: .utf8)!)

  let decoder = PropertyListDecoder()
  let restored = try decoder.decode(Rover.self, from: data)
  dump(restored)
}
```

Run tests for Mission 1 noting that they now pass.

2) Mission 2: Testing

Proceed to Mission 2 to add some proper testing. Notice that the models now conform to `Equatable`. This is trivial to do in Swift 4.1.

Next uncomment out the function for round tripping a model though JSON.

```
enum TestError: Error {
  case notEqual
  case dataCorrupt
}

func roundTripTest<T: Codable & Equatable>(item: T) throws {
  let encoder = JSONEncoder()
  let data = try encoder.encode(item)
  let decoder = JSONDecoder()
  let restored = try decoder.decode(T.self, from: data)

  if item != restored {
    NSLog("Expected")
    dump(item)
    NSLog("Actual")
    dump(restored)
    throw TestError.notEqual
  }
}
```

This ensures the encoding and decoding don't suffer from information loss. However, it doesn't protect you against breaking old archives. You need another utility function to test that. Uncomment out:

```
func archiveTest<T: Codable & Equatable>(json: String, expected: T)
throws {
  guard let data = json.data(using: .utf8) else {
    throw TestError.dataCorrupt
  }
  let decoder = JSONDecoder()
  let restored = try decoder.decode(T.self, from: data)
  if expected != restored {
    NSLog("Expected")
    dump(expected)
    NSLog("Actual")
    dump(restored)
    throw TestError.notEqual
  }
}
```

Finally test the curiosity instance by updating the failing test at the end:

```
func testCodable() throws {
  try roundTripTest(item: curiosity)
  try archiveTest(json: curiosityJSON, expected: curiosity)
}
```

3) Mission 3: Customization

In this mission you will handle some changes to the file format. Namely, "name" changes to "rover_name" and "value" changes to "sols".

To handle this you will use your own custom version of `CodingKeys`. Inside the `Sols`

model add:

```
enum CodingKeys: String, CodingKey {
  case value = "sols"
}
```

Next, inside the Rover model add:

```
enum CodingKeys: String, CodingKey {
  case name = "rover_name"
  case photos
}
```

That solves the problem. And the test now passes.

Note that `Rover` has a custom implementation of `Codable`. This doesn't do anything that the compiler generated version does differently.

5) Mission 4: Adding a Nested Container (I)

The key "photos" has been wrapped inside "mission_data" and made optional. It may or may not appear. This mission shows how to change the model to handle this and use the compiler to generate the serialization code.

Study the code and then enable the test at the end to see it in action.

```
func testCodable() throws {
  try roundTripTest(item: curiosity)
  try archiveTest(json: curiosityJSON, expected: curiosity)
}
```

5) Mission 5: Adding a Nested Container (II)

You will support exactly the same format as in the last mission except this time you won't change the `Rover` model. Make a `Codable` extension for `Rover`. You can start with the version provided for you in **Mission 3**.

```
extension Rover: Codable {
  enum CodingKeys: String, CodingKey {
    case name = "rover_name"
    case photos
  }

  init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    name = try container.decode(String.self, forKey: .name)
    photos = try container.decode([Photo].self, forKey: .photos)
  }
```

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

Update the `CodingKeys` with `missionData` and create `MissionDataCodingKeys` with `photos`:

```
enum CodingKeys: String, CodingKey {
  case name = "rover_name"
  case missionData = "mission_data"
}

enum MissionDataCodingKeys: String, CodingKey {
  case photos
}
```

Next update the `init(from:)` method:

```
init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: CodingKeys.self)
  name = try container.decode(String.self, forKey: .name)
  let missionDataContainer = try container.nestedContainer(keyedBy:
  MissionDataCodingKeys.self, forKey: .missionData)
  photos = try missionDataContainer.decodeIfPresent([Photo].self, forKey:
.photos) ?? []
}
```

Next update the `encode(to:)` method:

```
func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)
  try container.encode(name, forKey: .name)
  let optionalPhotos: [Photo]? = photos.isEmpty ? nil : photos
  var missionDataContainer = container.nestedContainer(keyedBy:
MissionDataCodingKeys.self, forKey: .missionData)
  try missionDataContainer.encodeIfPresent(optionalPhotos,
forKey: .photos)
}
```

Finally, run the tests and watch them pass.

6) That's it!

Congrats, at this time you should have a good understanding of the basics of `Codable` and serialization! It's time to move onto error handling and dealing with more advanced situations like polymorphism.

5 Swift 4 Serialization: Demo 2

By Ray Fix

In this demo, you will see how things can go wrong with missing values, incorrect polymorphic types and colliding keys. You will learn about serialization errors in general. More importantly, though, you will learn how to fix the errors or avoid them entirely building upon what you have learned in Demo 1.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter playground in **Demo2\starter**.

1) Mission 1: Show the keys of nil values

The project has an implementation of `Codable` similar to what the compiler produces automatically for you. Because `url` is optional, it uses the `encodeIfPresent` form. This causes the test to fail. Change it to `encode`:

```
try container.encode(url, forKey: .url)
```

Notice the output includes `"url":null` with your change. Also, round tripping the object works correctly. You don't need to change the decodable implementation. So you can delete it if you would like!

2) Mission 2: Custom coding keys

Click on **Mission2** in the file navigator. The `Distance<Units>` type currently serializes as `{"value":3.25}`. This has grave consequences if you get the units mixed up. In this mission you will make it use the `Units` phantom type to make it serialize as `{"feet" = 3}` or `{"meters" = 1}` depending on the type. If you do get

the units mixed up, you will make sure you generate a `keyNotFound` error.

In the definition of `Distance` add:

```
static var unitName: String {
  return String(describing: Units.self).lowercased()
}
```

This turns the generic parameter `Units` into a lowercased string. Next, inside of `Distance` add a custom coding key type and use it instead of `CodingKeys`:

```
struct UnitsKey: CodingKey {
  var stringValue: String
  init?(stringValue: String) {
    self.stringValue = stringValue
  }
  var intValue: Int? { return nil }
  init?(intValue: Int) { fatalError() }
}
```

Use the `UnitsKey` in your custom encoding and decoding methods. To `Distance` add:

```
init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: UnitsKey.self)
  self.value = try container.decode(Double.self, forKey:
UnitsKey(stringValue: Distance.unitName)!)
}

func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: UnitsKey.self)
  try container.encode(value, forKey: UnitsKey(stringValue:
Distance.unitName)!)
}
```

Finally replace the test failure with:

```
XCTAssertThrowsError(try decoder.decode([Distance<Meters>].self, from:
data))
```

With this coded added, notice that an error is now thrown when you encode with feet and try to decode with meters. In this case, getting an error is better than crashing on the surface of Mars!

3) Mission 3: Handle Polymorphism

Click on **Mission3** in the file navigator. In this mission you will handle distance measurements with different units such as `[{"feet":3.25},{"meters":4.25}, {"feet":0.25}]`.To do this, you will use an algebraic sum-type (aka: Swift enum!) called `AnyDistance`.

Most of the code is written for you. However the decoder isn't quite done. Replace

the current implementation with this:

```
init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: CodingKeys.self)
  guard container.allKeys.count == 1 else {
    let context = DecodingError.Context(codingPath: decoder.codingPath,
debugDescription: "wrong number of keys")
    throw DecodingError.dataCorrupted(context)
  }

  let key = container.allKeys[0]

  switch key {
  case .meters:
    self = .meters(try Distance<Meters>(from: decoder))
  case .feet:
    self = .feet(try Distance<Feet>(from: decoder))
  }
}
```

Update the test code so that it uses the `AnyDistance` type:

```
let restored = try decoder.decode([AnyDistance].self, from: jsonData)
```

Finally, run the test to see it in action.

4) Mission 4: Work with Inheritance

Click on **Mission4.swift** in the file navigator. In this mission you will see how to use `Codable` with subclasses.

The `RockSample` doesn't save properties of the base class. First fix the initializer:

```
required init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: CodingKeys.self)
  self.mass = try container.decode(Double.self, forKey: .mass)
  self.rockType = try container.decode(RockType.self, forKey: .rockType)
  try super.init(from: container.superDecoder(forKey: .sample))
}
```

Next, fix encoding:

```
override func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)
  try container.encode(mass, forKey: .mass)
  try container.encode(rockType, forKey: .rockType)
  try super.encode(to: container.superEncoder(forKey: .sample))
}
```

At this point the tests run and produce the correct output.

5) That's it!

Congrats, at this time you should have a good understanding of how to really customize codable conformance! Take a break, and then it's time to move onto date handling, relationships, and geting a deeper look into encoders and decoders.

6 Swift 4 Serialization: Demo 3

By Ray Fix

In this demo, you will learn how to handle dates and object relationships. You will also learn how to customize coding behavior based on outside user info parameters supplied to your encoder or decoder.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter playground in **Demo3\starter**.

1) Mission 1: Handle multiple date formats

In this mission you will make it possible for a single date field to handle multiple date formats. In particular, you want to support ISO 8601 dates both with and without milliseconds.

Under **Mission1** open **DateHandling.swift**. Add another static date formatter to the end of the `DateFormatter` extension:

```
// Handles dates of the form "2018-02-22T23:35:48-0800"
public static let iso8601: DateFormatter = {
  let formatter = DateFormatter()
  formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
  formatter.locale = Locale(identifier: "en_US_POSIX")
  return formatter
}()
```

Next, in the same file, update your `dateStringDecode` helper function so that it accepts multiple formatters by adding:

```
public
func dateStringDecode<C>(forKey key: C.Key, from container: C, with
```

```
  formatters: DateFormatter...) throws -> Date
    where C: KeyedDecodingContainerProtocol {
      let dateString = try container.decode(String.self, forKey: key)

      for formatter in formatters {
        if let date = formatter.date(from: dateString) {
          return date
        }
      }
      throw DecodingError.dataCorruptedError(forKey: key, in: container,
  debugDescription: dateString)
  }
```

Now go back to the **Mission1.swift** main page and update decode init for `Sample` to use `.iso8601`:

```
  required init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: SampleCodingKeys.self)
    self.date = try dateStringDecode(forKey: .date, from: container,
  with: .iso8601Milliseconds, .iso8601)
  }
```

You did it. Run the tests to make sure you did it right.

3) Mission 2: Handle relationships correctly

In this mission you will create a relationship between `Sample` types and the `Rover` that they came from. You will correctly handle that relationship for encoding and decoding.

Go to **Mission2** in the file navigator. Notice some addition printing has been added to the decode and encode methods so you can visualize what is happening. Add a `rover` property to `Sample` right under the `date` property.

```
  weak var rover: Rover?
```

Next update `Rover`'s member-wise initializer to set the relation of `rockSamples` to `Rover`.

```
  init(name: String, launchDate: Date, rockSamples: [RockSample]) {
    self.name = name
    self.launchDate = launchDate
    self.rockSamples = rockSamples
    rockSamples.forEach { $0.rover = self }  // this is new
  }
```

So far so good. Update `Rover`'s == operator in Section 1 of the code:

```
  static func ==(lhs: Rover, rhs: Rover) -> Bool {
    return lhs.name == rhs.name &&
      lhs.launchDate == rhs.launchDate &&
```

```
    lhs.rockSamples == rhs.rockSamples &&
    !zip(lhs.rockSamples, rhs.rockSamples).contains { $0.rover?.name !=
$1.rover?.name }
}
```

You are avoiding another infinite recursion probelm by only checking the name.

Update the the last line of `init(from:)` of `Rover` in Section 1 to set the `rover` property back to `self`:

```
init(from decoder: Decoder) throws {
  print("<-- Rover")
  let container = try decoder.container(keyedBy: CodingKeys.self)
  name = try container.decode(String.self, forKey: .name)
  launchDate = try dateStringDecode(forKey: .launchDate, from: container,
with: .yearMonthDay)
  rockSamples = try container.decode([RockSample].self,
forKey: .rockSamples)
  rockSamples.forEach { $0.rover = self }  // this is new
}
```

Finally, in the test, remove the `XCTFail()` and comment in `XCTAssertEqual(curiosity.rockSamples.first?.rover?.name, "Curiosity")`. Run the test and watch it pass.

Time to move on to the last mission!

4) Mission 3: Customized coding using outside parameters

In this mission you will customize the decoding of rover's rock samples to be more lenient when it encounters unexpected JSON and the key `deleteBadSamplesKey` is set to true.

Click on **Mission3** to open it.

Go down to the Section 4 test and change "metamorphic" in the `jsonString` to be an illegal value such as "metamorphical". You should now see a `Swift.DecodingError.dataCorrupted` error being thrown. You will change the code so it will just throw away bad samples if an option is set.

All the way at the top, add a utility class that isolates failing decodes:

```
struct FailableDecodeBox<Model: Decodable>: Decodable {
  let model: Model?
  init(from decoder: Decoder) throws {
    let container = try decoder.singleValueContainer()
    model = try? container.decode(Model.self)
  }
```

```
}
```

Next, inside of the `Rover` class, add a user info key:

```
static let deleteBadSamplesKey = CodingUserInfoKey(rawValue:
"Rover.deleteBadSamples")!
```

Next, change `Rover`'s decoding method to use this key:

```
init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: CodingKeys.self)
  name = try container.decode(String.self, forKey: .name)
  launchDate = try dateStringDecode(forKey: .launchDate, from: container,
with: .yearMonthDay)
  if let deleteBadSamples = decoder.userInfo[Rover.deleteBadSamplesKey]
as? Bool,
    deleteBadSamples {
    rockSamples = try
container.decode([FailableDecodeBox<RockSample>].self,
forKey: .rockSamples)
      .compactMap { $0.model }
  } else {
    rockSamples = try container.decode([RockSample].self,
forKey: .rockSamples)
  }
  rockSamples.forEach { $0.rover = self }
}
```

Finally, go down to the test code and activate the feature. Just below creation of the `decoder` uncomment in the line:

```
decoder.userInfo = [Rover.deleteBadSamplesKey: true]
```

With that, the `Rover` throws or decodes and removes the bad "metamorphical" sample depending on the setting.

5) That's it!

Congrats, you made it to the end. You should have an excellent understanding of Swift Serialization!