Article

Encoding and Decoding Custom Types

Make your data types encodable and decodable for compatibility with external representations such as JSON.

Overview

Many programming tasks involve sending data over a network connection, saving data to disk, or submitting data to APIs and services. These tasks often require data to be encoded and decoded to and from an intermediate format while the data is being transferred.

The Swift standard library defines a standardized approach to data encoding and decoding. You adopt this approach by implementing the `Encodable` and `Decodable` protocols on your custom types. Adopting these protocols lets implementations of the `Encoder` and `Decoder` protocols take your data and encode or decode it to and from an external representation such as JSON or property list. To support both encoding and decoding, declare conformance to `Codable`, which combines the `Encodable` and `Decodable` protocols. This process is known as making your types *codable*.

Encode and Decode Automatically

The simplest way to make a type codable is to declare its properties using types that are already `Codable`. These types include standard library types like `String`, `Int`, and `Double`; and Foundation types like `Date`, `Data`, and `URL`. Any type whose properties are codable automatically conforms to `Codable` just by declaring that conformance.

Consider a `Landmark` structure that stores the name and founding year of a landmark:

```
struct Landmark {
    var name: String
    var foundingYear: Int
```

```
}
```

Adding `Codable` to the inheritance list for `Landmark` triggers an automatic conformance that satisfies all of the protocol requirements from `Encodable` and `Decodable`:

```
struct Landmark: Codable {
    var name: String
    var foundingYear: Int

    // Landmark now supports the Codable methods init(from:) and encode(to:),
    // even though they aren't written as part of its declaration.
}
```

Adopting `Codable` on your own types enables you to serialize them to and from any of the built-in data formats, and any formats provided by custom encoders and decoders. For example, the `Landmark` structure can be encoded using both the `PropertyListEncoder` and `JSONEncoder` classes, even though `Landmark` itself contains no code to specifically handle property lists or JSON.

The same principle applies to custom types made up of other custom types that are codable. As long as all of its properties are `Codable`, any custom type can also be `Codable`.

The example below shows how automatic `Codable` conformance applies when a `location` property is added to the `Landmark` structure:

```
struct Coordinate: Codable {
    var latitude: Double
    var longitude: Double
}

struct Landmark: Codable {
    // Double, String, and Int all conform to Codable.
    var name: String
    var foundingYear: Int

    // Adding a property of a custom Codable type maintains overall Codable conforma
    var location: Coordinate
}
```

Built-in types such as `Array`, `Dictionary`, and `Optional` also conform to `Codable` whenever they contain codable types. You can add an array of `Coordinate` instances to `Landmark`, and the entire structure will still satisfy `Codable`.

The example below shows how automatic conformance still applies when adding multiple properties using built-in codable types within `Landmark`:

```
struct Landmark: Codable {
    var name: String
    var foundingYear: Int
    var location: Coordinate

    // Landmark is still codable after adding these properties.
    var vantagePoints: [Coordinate]
    var metadata: [String: String]
    var website: URL?
}
```

Encode or Decode Exclusively

In some cases, you may not need `Codable`'s support for bidirectional encoding and decoding. For example, some apps only need to make calls to a remote network API and do not need to decode a response containing the same type. Declare conformance to `Encodable` if you only need to support the encoding of data. Conversely, declare conformance to `Decodable` if you only need to read data of a given type.

The examples below show alternative declarations of the `Landmark` structure that only encode or decode data:

```
struct Landmark: Encodable {
    var name: String
    var foundingYear: Int
}
```

```
struct Landmark: Decodable {
    var name: String
    var foundingYear: Int
}
```

Choose Properties to Encode and Decode Using Coding Keys

Codable types can declare a special nested enumeration named `CodingKeys` that conforms to the `CodingKey` protocol. When this enumeration is present, its cases serve as the authoritative

list of properties that must be included when instances of a codable type are encoded or decoded. The names of the enumeration cases should match the names you've given to the corresponding properties in your type.

Omit properties from the `CodingKeys` enumeration if they won't be present when decoding instances, or if certain properties shouldn't be included in an encoded representation. A property omitted from `CodingKeys` needs a default value in order for its containing type to receive automatic conformance to `Decodable` or `Codable`.

If the keys used in your serialized data format don't match the property names from your data type, provide alternative keys by specifying `String` as the raw-value type for the `CodingKeys` enumeration. The string you use as a raw value for each enumeration case is the key name used during encoding and decoding. The association between the case name and its raw value lets you name your data structures according to the Swift API Design Guidelines rather than having to match the names, punctuation, and capitalization of the serialization format you're modeling.

The example below uses alternative keys for the `name` and `foundingYear` properties of the `Landmark` structure when encoding and decoding:

```
struct Landmark: Codable {
    var name: String
    var foundingYear: Int
    var location: Coordinate
    var vantagePoints: [Coordinate]

    enum CodingKeys: String, CodingKey {
        case name = "title"
        case foundingYear = "founding_date"

        case location
        case vantagePoints
    }
}
```

Encode and Decode Manually

If the structure of your Swift type differs from the structure of its encoded form, you can provide a custom implementation of `Encodable` and `Decodable` to define your own encoding and decoding logic.

In the examples below, the `Coordinate` structure is expanded to support an `elevation` property that's nested inside of an `additionalInfo` container:

```
struct Coordinate {
    var latitude: Double
    var longitude: Double
    var elevation: Double

    enum CodingKeys: String, CodingKey {
        case latitude
        case longitude
        case additionalInfo
    }

    enum AdditionalInfoKeys: String, CodingKey {
        case elevation
    }
}
```

Because the encoded form of the `Coordinate` type contains a second level of nested information, the type's adoption of the `Encodable` and `Decodable` protocols uses two enumerations that each list the complete set of coding keys used on a particular level.

In the example below, the `Coordinate` structure is extended to conform to the `Decodable` protocol by implementing its required initializer, `init(from:)`:

```
extension Coordinate: Decodable {
    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)
        latitude = try values.decode(Double.self, forKey: .latitude)
        longitude = try values.decode(Double.self, forKey: .longitude)

        let additionalInfo = try values.nestedContainer(keyedBy: AdditionalInfoKeys.
        elevation = try additionalInfo.decode(Double.self, forKey: .elevation)
    }
}
```

The initializer populates a `Coordinate` instance by using methods on the `Decoder` instance it receives as a parameter. The `Coordinate` instance's two properties are initialized using the keyed container APIs provided by the Swift standard library.

The example below shows how the `Coordinate` structure can be extended to conform to the `Encodable` protocol by implementing its required method, `encode(to:)`:

```
extension Coordinate: Encodable {
    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(latitude, forKey: .latitude)
        try container.encode(longitude, forKey: .longitude)

        var additionalInfo = container.nestedContainer(keyedBy: AdditionalInfoKeys.s
        try additionalInfo.encode(elevation, forKey: .elevation)
    }
}
```

This implementation of the `encode(to:)` method reverses the decoding operation from the previous example.

For more information about the container types used when customizing the encoding and decoding process, see `KeyedEncodingContainerProtocol` and `UnkeyedEncodingContainer`.

See Also

Adopting Codability

`typealias Codable = Decodable & Encodable`

A type that can convert itself into and out of an external representation.

`protocol NSCoding`

A protocol that enables an object to be encoded and decoded for archiving and distribution.

`protocol NSSecureCoding`

A protocol that enables encoding and decoding in a manner that is robust against object substitution attacks.