Encoding and Decoding in Swift

Kaitlin Mahar
Software Engineer @ MongoDB

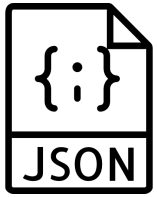
@k__mahar

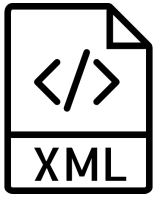
@kmahar

What is encoding?

Swift type → Encoder → External representation

Scalar (e.g. `Int`)
`struct`
`class`
`enum`



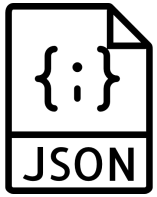


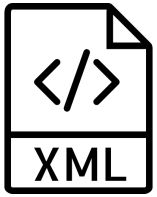
etc...

What is decoding?

Swift type ← Decoder ← External representation

Scalar (e.g. `Int`)
`struct`
`class`
`enum`





etc...

Why would I want to encode and decode data?

- Allows data transfer in and out of your application
  - Communicating with a REST API via JSON
  - Reading from and writing to a database
  - Importing and exporting data from files

Swift 4 introduced a standardized approach to encoding and decoding.

How does it actually work?

Basic Usage

```swift
public protocol Encodable {
  func encode(to encoder: Encoder) throws
}
```

An `Encodable` type knows how to write itself to an `Encoder`.

```swift
public protocol Encodable {
  func encode(to encoder: Encoder) throws
}
```

- Automatic conformance if all properties are `Encodable`
- Types can provide custom implementations
- Format agnostic: write it once, works with any `Encoder`!

```swift
public protocol Decodable {
  init(from decoder: Decoder) throws
}
```

A `Decodable` type knows how to initialize by reading from a `Decoder`.

```swift
public protocol Decodable {
  init(from decoder: Decoder) throws
}
```

- Automatic conformance if all properties are `Decodable`
- Types can provide custom implementations
- Write it once, works with any `Decoder`

```
public typealias Codable =
    Encodable & Decodable
```

Types With Built-In `Codable` Support

- Numeric types
- `Bool`
- `String`
- If the values they contain are `Encodable` / `Decodable`:
  - `Array`
  - `Set`
  - `Dictionary`
  - `Optional`
- Common Foundation types: `URL`, `Data`, `Date`, etc.

Making Types Codable

```
struct Cat {
    let name: String
    let color: String
}
```

```
struct Cat: Codable {
    let name: String
    let color: String
}
```

… and that's it!

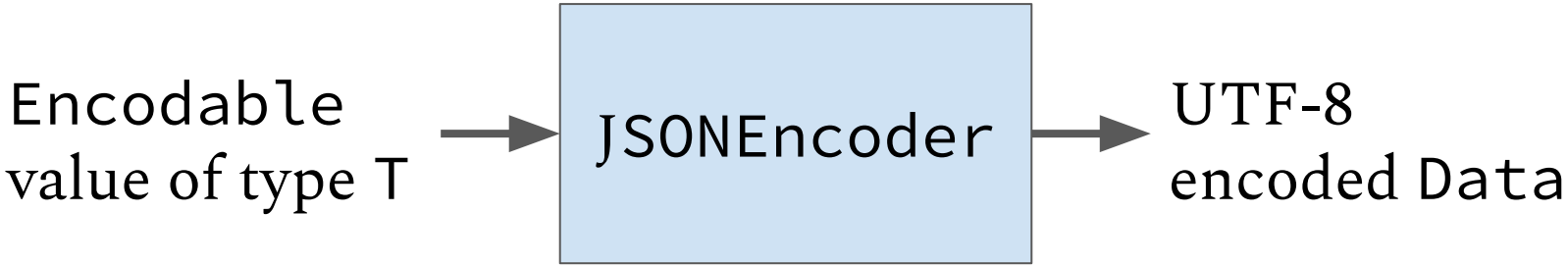
Using Encoders and Decoders

```
struct Cat: Codable {
    let name: String
    let color: String
}
```

```
let roscoe = Cat(name: "Roscoe", color: "orange")
```

Using An Encoder

```
class JSONEncoder {
    func encode<T: Encodable>(_ value: T) throws -> Data
}
```



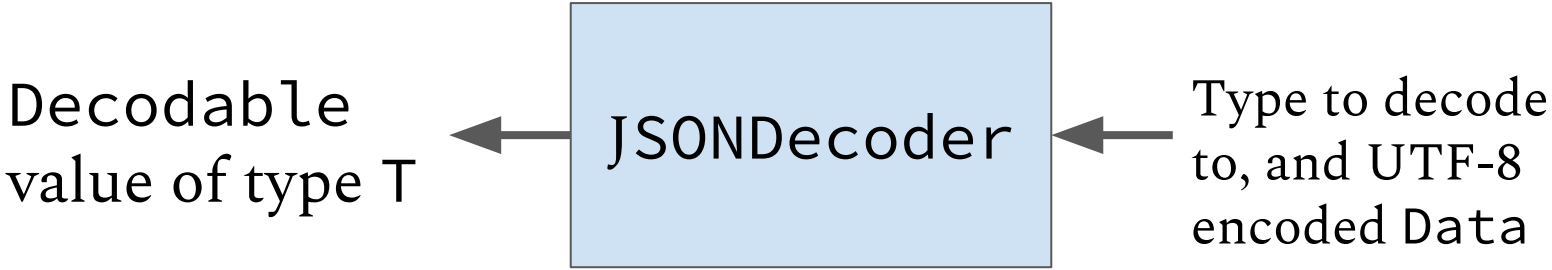
Using An Encoder

```
class JSONEncoder {
    func encode<T: Encodable>(_ value: T) throws -> Data
}

let encoder = JSONEncoder()
```

```
{
    "name":"Roscoe",
    "color":"orange"
}
```

Using A Decoder

```
class JSONDecoder {
    func decode<T: Decodable>(_ type: T.Type,
                              from data: Data) throws -> T
}
```



Using A Decoder

```
class JSONDecoder {
    func decode<T: Decodable>(_ type: T.Type,
                              from data: Data) throws -> T
}
```

Data we got
from encoding

```
let decoder = JSONDecoder()
let roscoe = try decoder.decode(Cat.self, from: roscoeData)

print(roscoe)
>> Cat(name: "Roscoe", color: "orange")
```

Advanced Usage: Customizing How Your Types are Encoded/Decoded

Q: What if I want to omit a property?

```
struct Cat: Encodable {
    let name: String
    let color: String
}
```

```
{
    "name":"Roscoe",
    "color":"orange"
}
```

A: Use CodingKeys

```
struct Cat: Encodable {
    let name: String
    let color: String

    enum CodingKeys: CodingKey {
        case name, color
    }
}
```

Compiler-generated default

- Nested type that specifies the keys that will be used for encoding
- Compiler generated, but custom implementation can be provided

Omitting a property

```
struct Cat: Encodable {
    let name: String
    let color: String

    enum CodingKeys: CodingKey {
        case name
    }
}
```

```
{"name":"Roscoe"}
```

Q: What if I want to rename a key?

```swift
struct Cat: Encodable {
    let name: String
    let color: String
}
```

```
{
    "name":"Roscoe",
    "color":"orange"
}
```

Renaming a key

```
struct Cat: Encodable {
    let name: String
    let color: String

    enum CodingKeys: String, CodingKey {
        case name = "firstName", color
    }
}
```

```
{
    "firstName":"Roscoe",
    "color":"orange"
}
```

Q: What if I want to modify properties as I encode them?

```
struct Cat: Encodable {
    let name: String
    let color: String
}
```

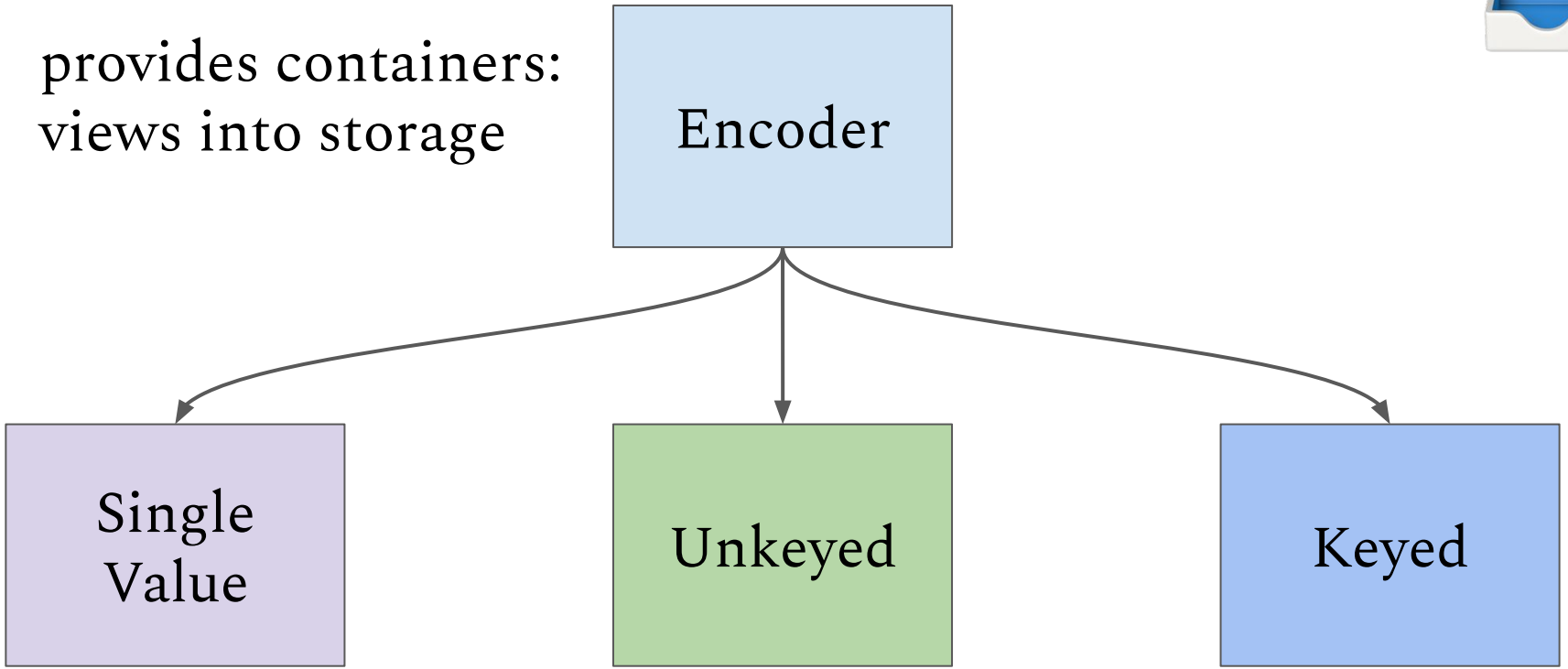
```
{
    "name":"Roscoe",
    "color":"orange"
}
```

e.g. Convert a string to lowercase?

A: the `Encodable.encode` method

```
public protocol Encodable {
  func encode(to encoder: Encoder) throws
}
```

provides containers:
views into storage



one value

sequence of values

key/value pairs

In code...

```
public protocol Encoder {

  func singleValueContainer() -> SingleValueEncodingContainer

  func unkeyedContainer() throws -> UnkeyedEncodingContainer

  func container<Key: CodingKey>(keyedBy type: Key.Type)
                                 throws -> KeyedEncodingContainer<Key>

  // ...
}
```

Encoding containers support storing three types of values.

base case 1: nil	`nil`
base case 2: primitives	`Bool`, `String`, `Double`, `Float` all `Int` and `UInt` types
recursive case	`Encodable` type

```
public protocol SingleValueEncodingContainer {

  mutating func encodeNil() throws

% for type in primitives:
  mutating func encode(_ value: ${type}) throws
% end

  mutating func encode<T: Encodable>(_  value: T) throws
}
```

nil

primitive

Encodable

SingleValueEncodingContainer

UnkeyedEncodingContainer

```
public protocol                                    {

  mutating func encodeNil() throws

% for type in primitives:
  mutating func encode(_ value: ${type}) throws
% end

  mutating func encode<T: Encodable>(_  value: T) throws
}
```

nil

primitive

Encodable

KeyedEncodingContainerProtocol

```
public protocol {
  associatedtype Key: CodingKey

  mutating func encodeNil(forKey key: Key) throws

% for type in primitives:
  mutating func encode(_ value: ${type},
                       forKey key: Key) throws
% end

  mutating func encode<T: Encodable>(_ value: T,
                                     forKey key: Key) throws
}
```

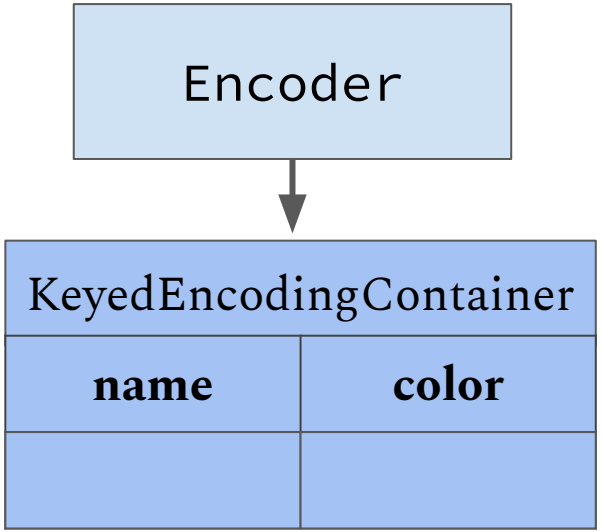
nil

primitive

Encodable

So how are these containers used?

```swift
struct Cat: Encodable {
    let name: String
    let color: String
}
```




```swift
let roscoe = Cat(name: "Roscoe", color: "orange")
```

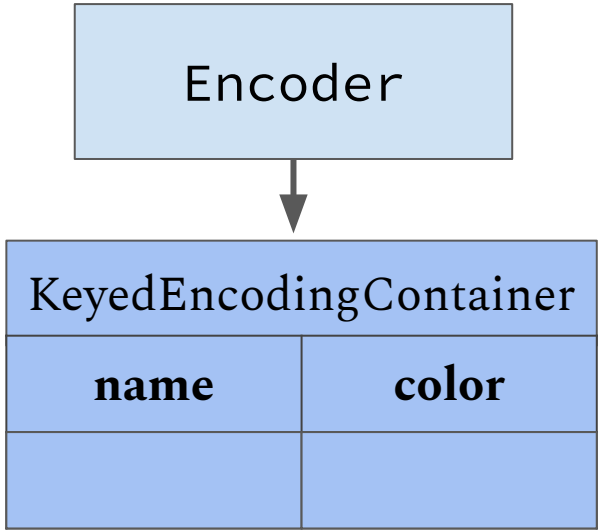
Q: What if I want to modify properties as I encode them?

```
struct Cat: Encodable {
    let name: String
    let color: String
}
```

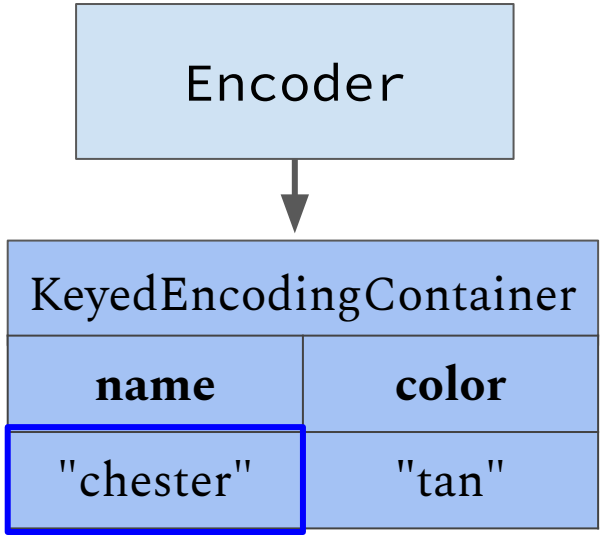
```
{
    "name":"Roscoe",
    "color":"orange"
}
```

e.g. Convert a string to lowercase?

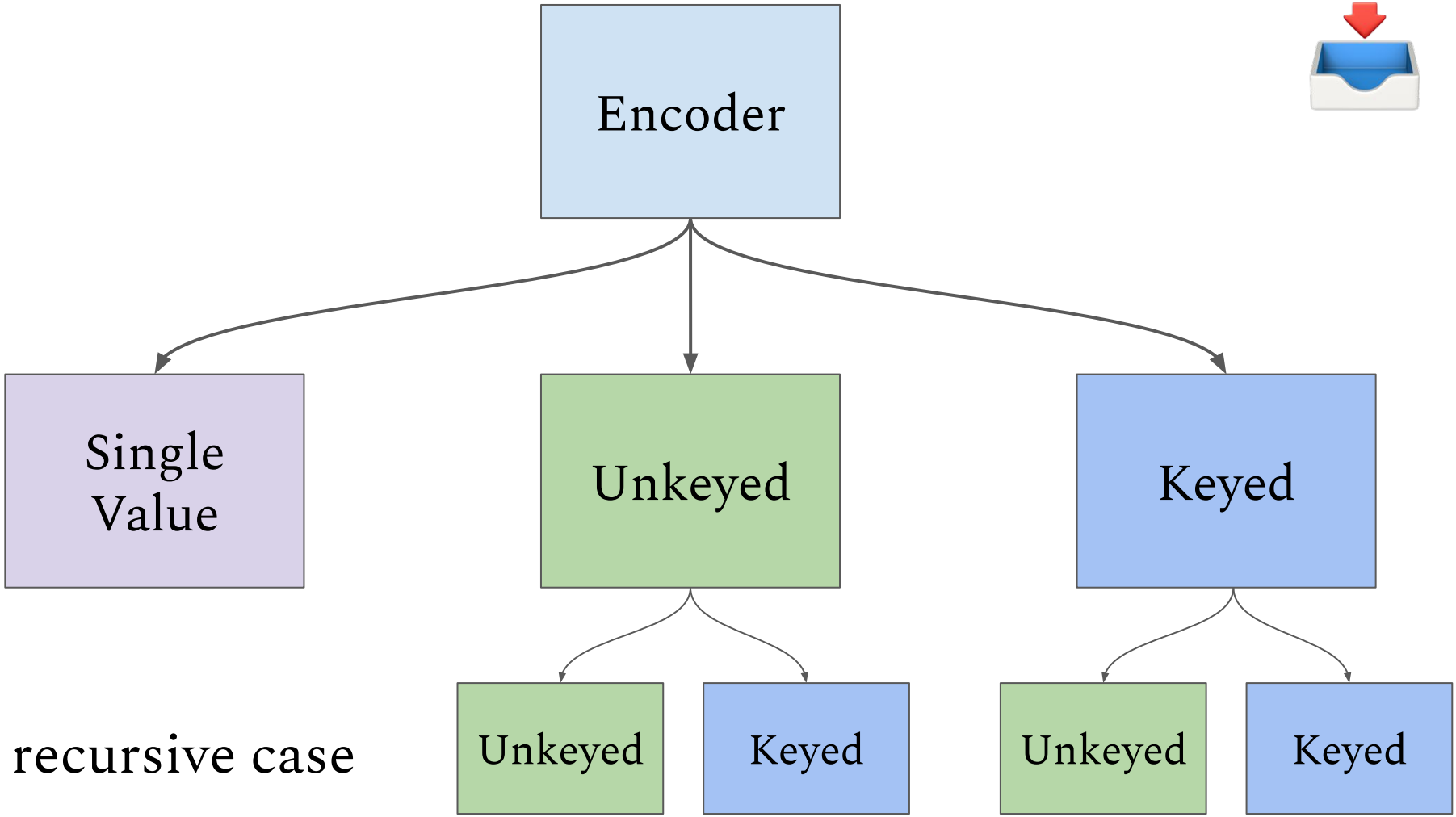
```swift
struct Cat: Encodable {
    let name: String
    let color: String

    enum CodingKeys: CodingKey {
        case name, color
    }

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



```swift
struct Cat: Encodable {
    let name: String
    let color: String

    enum CodingKeys: CodingKey {
        case name, color
    }

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



What if I have types nested within other types?


Encoder
Single Value
Unkeyed
Keyed
recursive case
Unkeyed
Keyed
Unkeyed
Keyed

```
public protocol UnkeyedEncodingContainer {
  // ...

  mutating func nestedContainer<NestedKey: CodingKey>(
    keyedBy keyType: NestedKey.Type)
     -> KeyedEncodingContainer<NestedKey>

  mutating func nestedUnkeyedContainer()
    -> UnkeyedEncodingContainer
}
```

Keyed

Unkeyed

```
public protocol KeyedEncodingContainerProtocol {
  associatedtype Key : CodingKey

  // ...
  mutating func nestedContainer<NestedKey: CodingKey>(
    keyedBy keyType: NestedKey.Type, forKey key: Key
  ) -> KeyedEncodingContainer<NestedKey>

  mutating func nestedUnkeyedContainer(
    forKey key: Key) -> UnkeyedEncodingContainer
}
```

Keyed

Unkeyed

Let's make things more complicated...

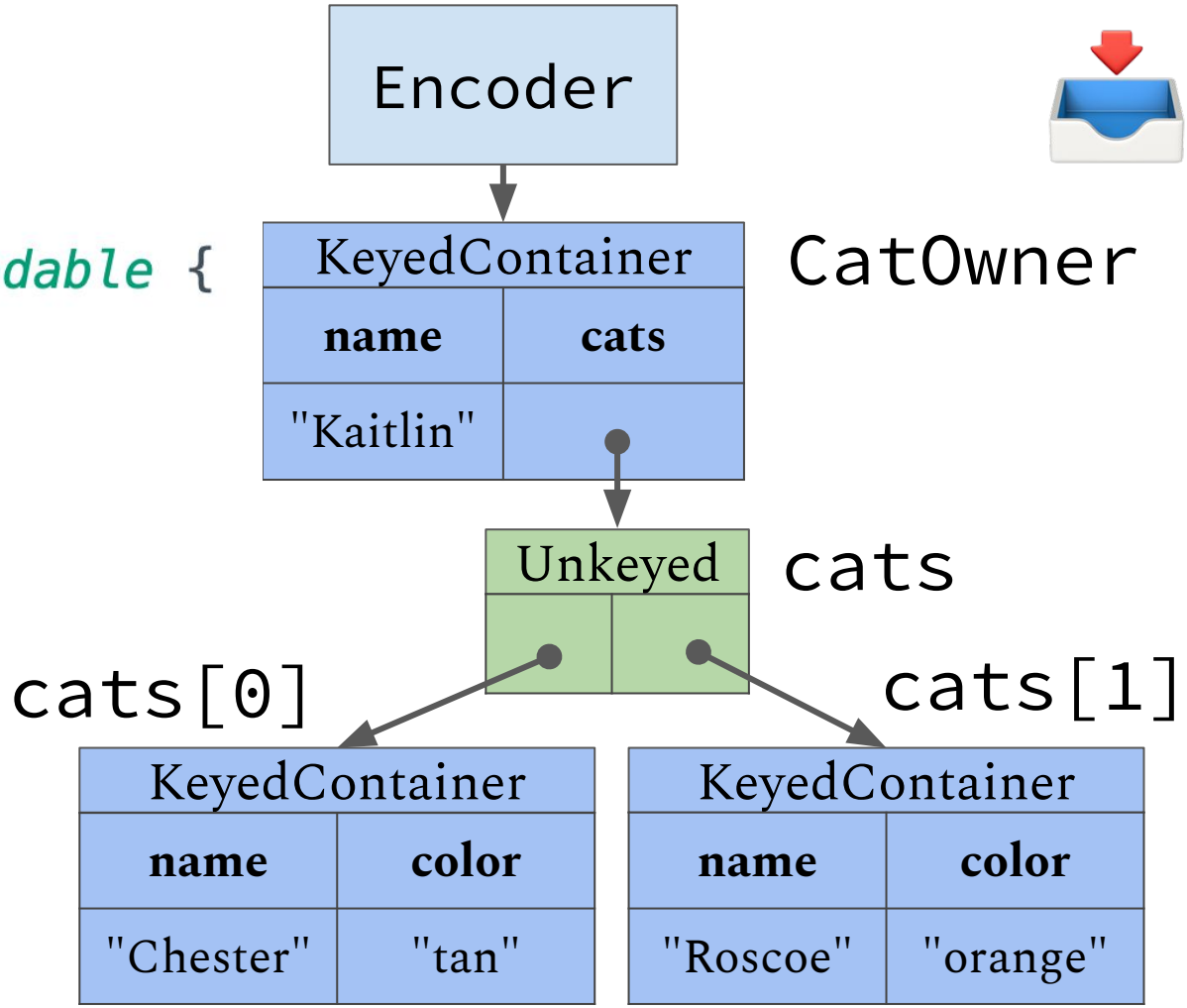
```
struct CatOwner: Encodable {
    let name: String
    let cats: [Cat]
}
```

```
let chester = Cat(name: "Chester", color: "tan")
let roscoe = Cat(name: "Roscoe", color: "orange")
let kaitlin = CatOwner(name: "Kaitlin", cats: [chester, roscoe])
```

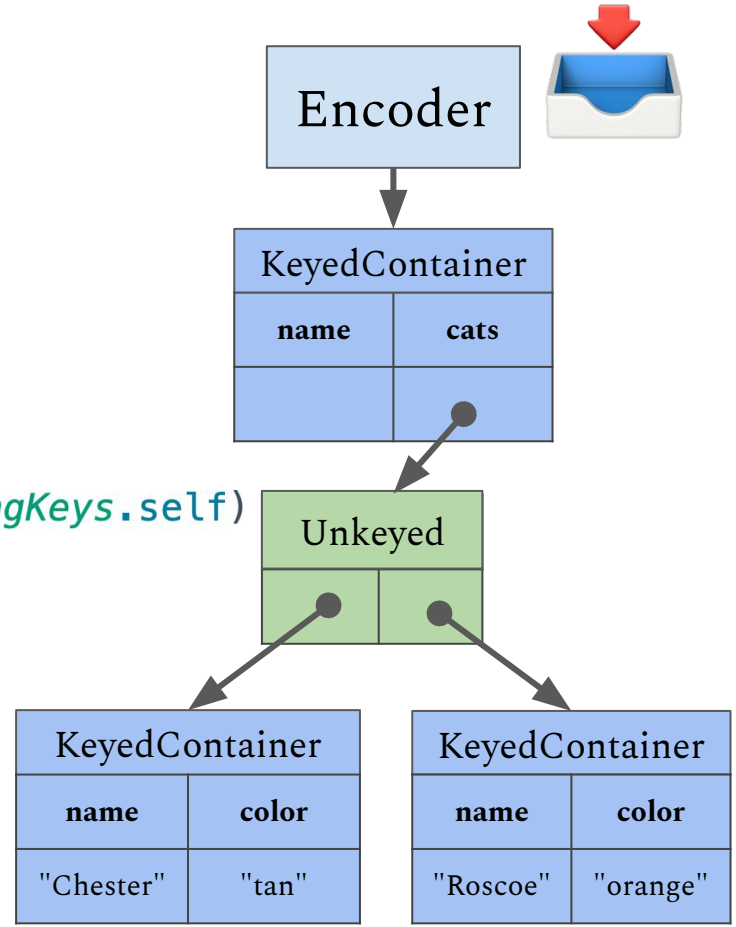
```swift
struct CatOwner: Encodable {
    let name: String
    let cats: [Cat]
}

struct Cat: Encodable {
    let name: String
    let color: String
}
```

```
{
    "name":"Kaitlin",
    "cats":[
        {
            "name":"Chester",
            "color":"tan"
        },
        {
            "name":"Roscoe",
            "color":"orange"
        }
    ]
}
```

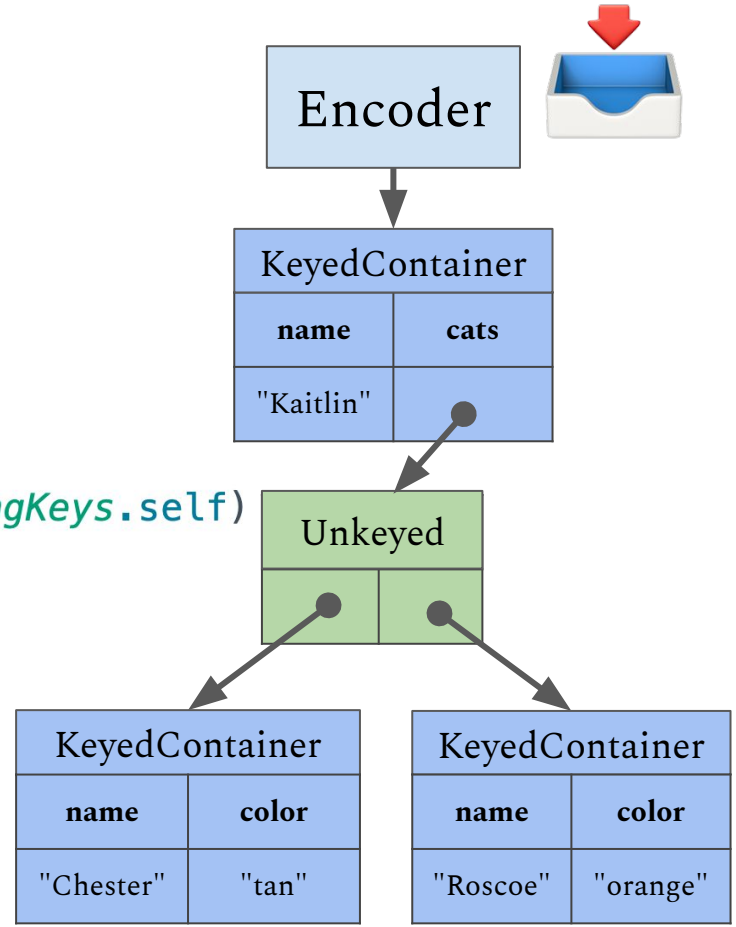
```swift
struct CatOwner: Encodable {
    let name: String
    let cats: [Cat]
}
```



```swift
struct CatOwner: Encodable {
  let name: String
  let cats: [Cat]

  enum CodingKeys: CodingKey {
    case name, cats
  }

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



```swift
struct CatOwner: Encodable {
  let name: String
  let cats: [Cat]

  enum CodingKeys: CodingKey {
    case name, cats
  }

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



Encoding containers support storing three types of values.

base case 1: nil

`nil`

base case 2: primitives

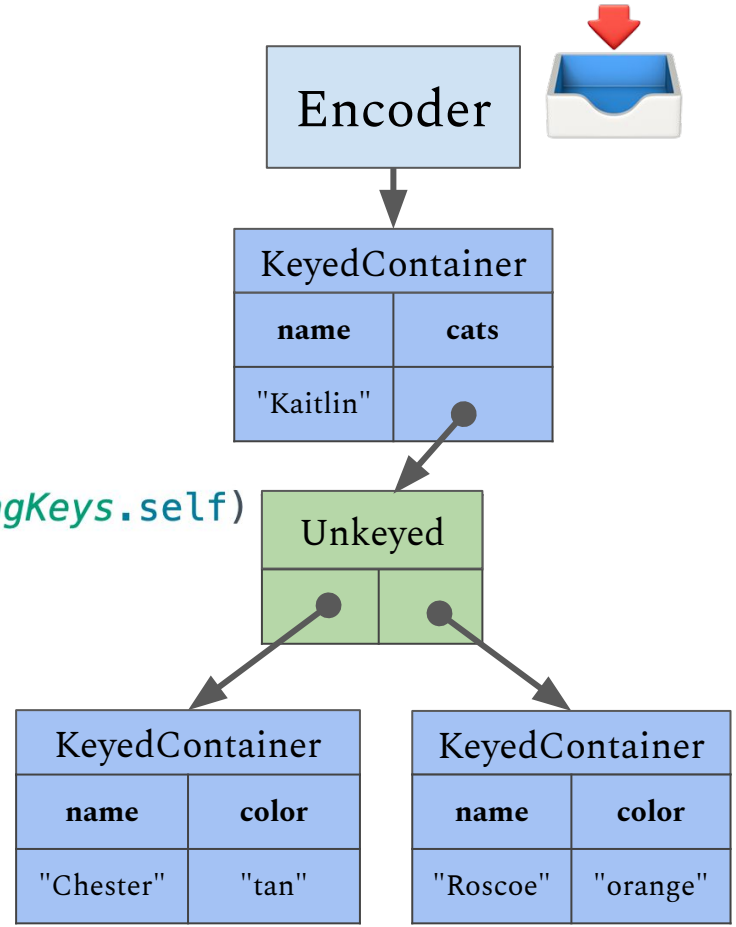
`Bool`, `String`, `Double`, `Float`
all `Int` and `UInt` types

recursive case

`Encodable` type

```swift
struct CatOwner: Encodable {
  let name: String
  let cats: [Cat]

  enum CodingKeys: CodingKey {
    case name, cats
  }

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

Calls `Array<Cat>.encode(to: self)`

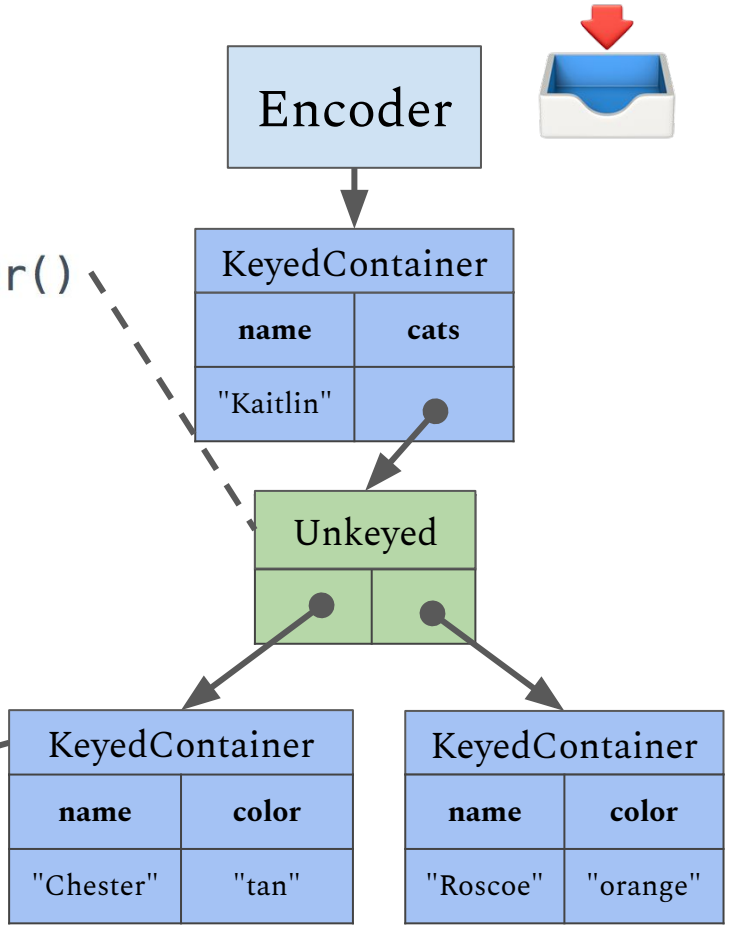


Array<Cat>

```
func encode(to encoder: Encoder) throws {
    var container = encoder.unkeyedContainer()
    for elt in self {
        try container.encode(elt)
    }
}
```

Cat

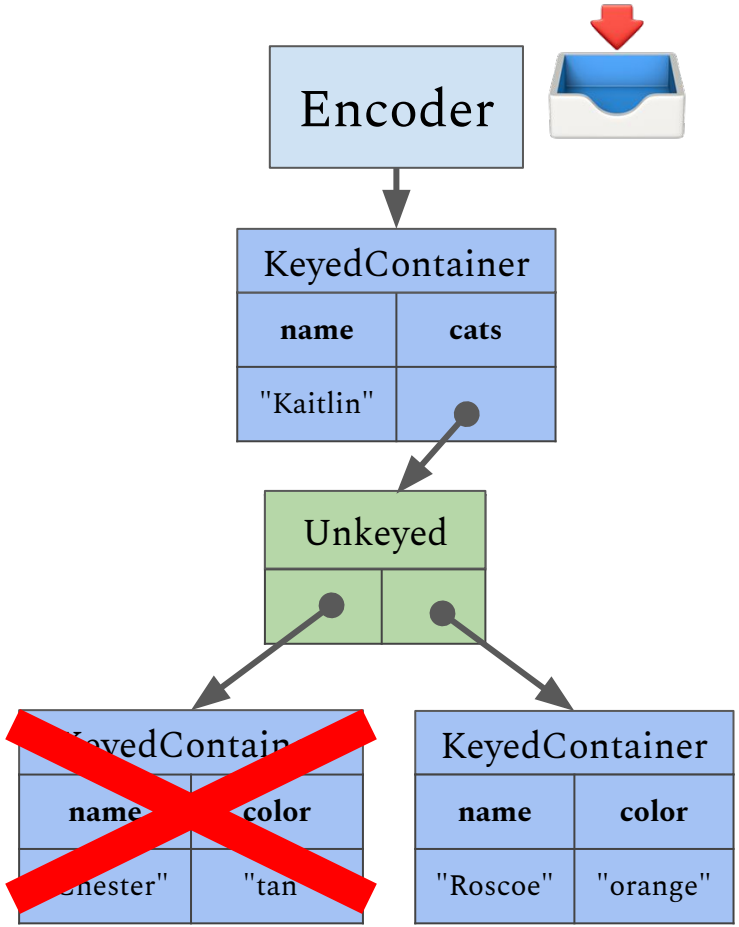
```
func encode(to encoder: Encoder) throws {
    var container =
        encoder.container(keyedBy: CodingKeys.self)
    try container.encode(name, forKey: .name)
    try container.encode(color, forKey: .color)
}
```



Again, compiler and encoder do this for you!

Orange cats only

```
{
    "name":"Kaitlin",
    "cats":[
        {
            "name":"Roscoe",
            "color":"orange"
        }
    ]
}
```

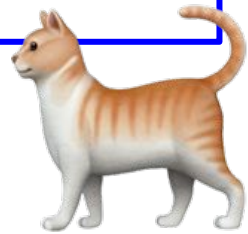


Orange cats only

```
enum CodingKeys: CodingKey {
  case name, cats
}

func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)
  try container.encode(name, forKey: .name)
  var catContainer = container.nestedUnkeyedContainer(forKey: .cats)
  for cat in cats where cat.color == "orange" {
      try catContainer.encode(cat)
  }
}
```

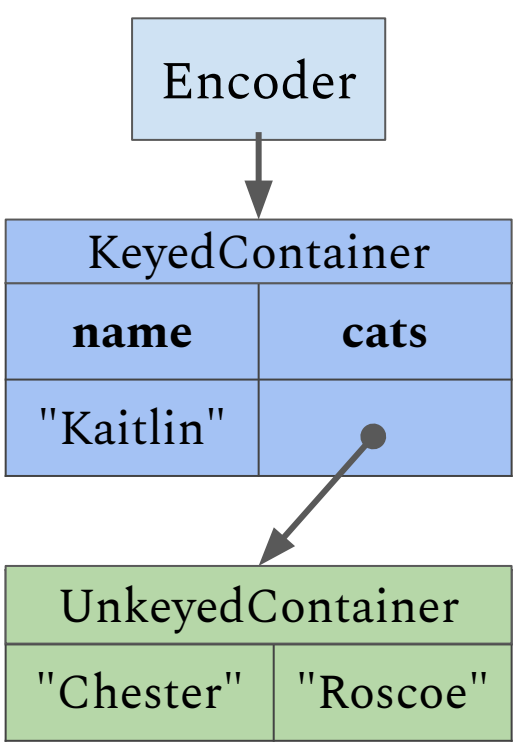
bypass

`Array<Cat>.encode(to:)`



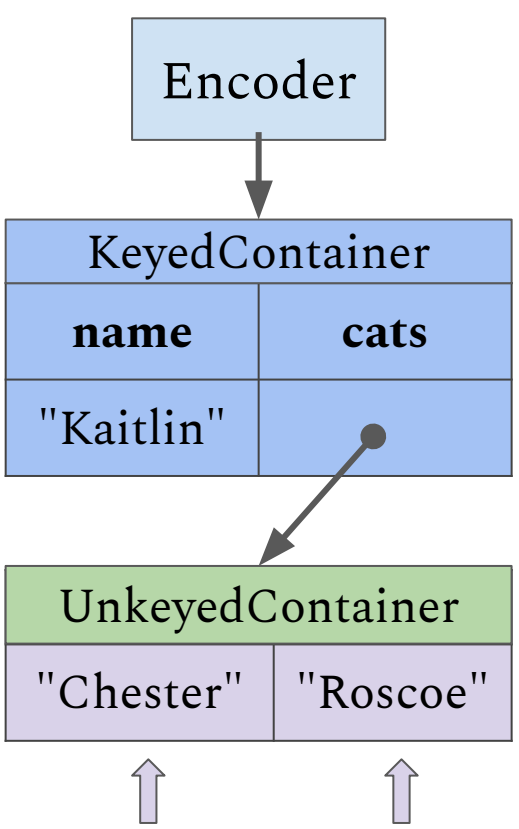
Flattening data

```
{
    "name":"Kaitlin",
    "cats":[
        "Chester",
        "Roscoe"
    ]
}
```



Flattening data

```
{
    "name":"Kaitlin",
    "cats":[
        "Chester",
        "Roscoe"
    ]
}
```



Single value containers!

Flattening data

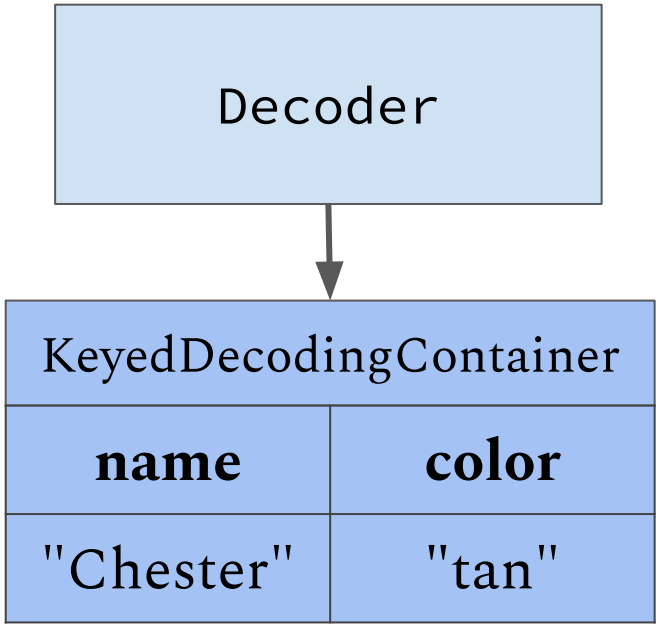
```
struct Cat: Encodable {
    let name: String
    let color: String

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        try container.encode(name)
    }
}
```

no `CodingKeys` needed!

+ compiler generated `CatOwner.encode`

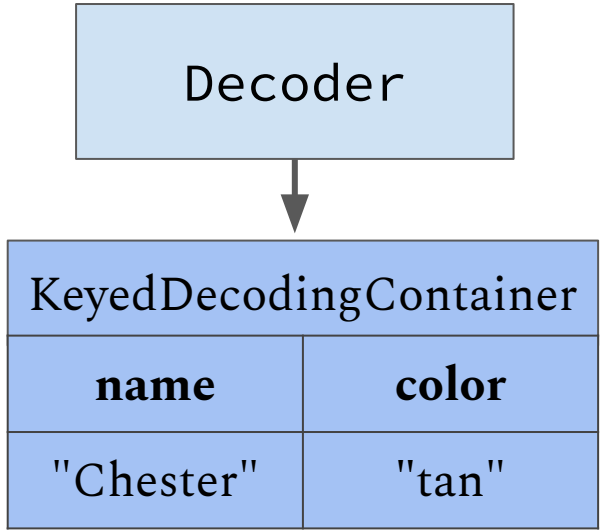
Weren't we also talking about decoding?

```swift
struct Cat: Decodable {
    let name: String
    let color: String
}
```

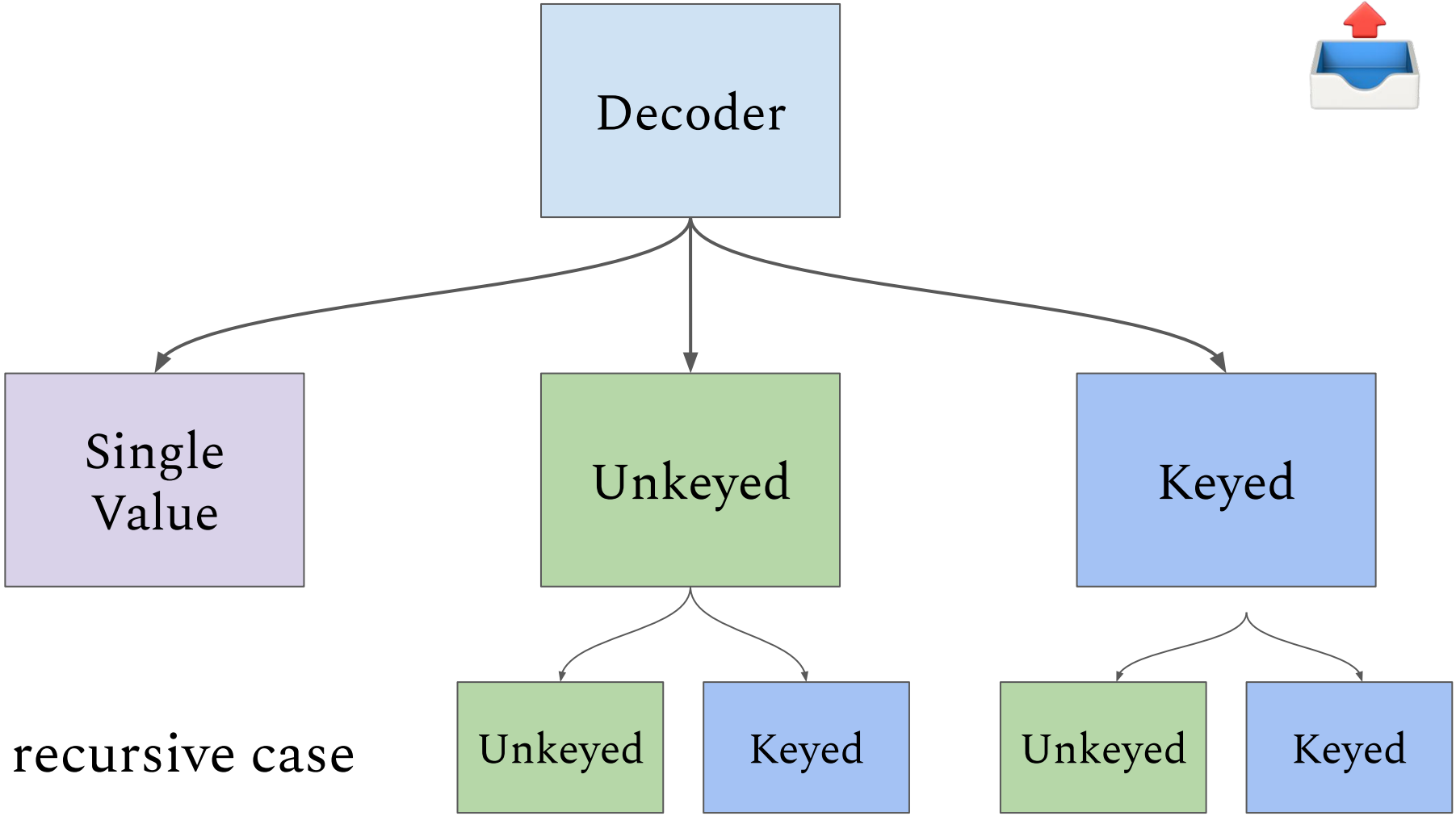


```swift
let chester = Cat(name: "Chester", color: "tan")
```

```swift
struct Cat: Decodable {
  let name: String
  let color: String

  enum CodingKeys: CodingKey {
    case name, color
  }

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



Compiler generated defaults


Decoder
Single Value
Unkeyed
Keyed
recursive case
Unkeyed
Keyed
Unkeyed
Keyed

Decoding containers support retrieving three types of values.

base case 1: nil	`nil`
base case 2: primitives	`Bool`, `String`, `Double`, `Float` all `Int` and `UInt` types
recursive case	`Decodable` type

Customization Takeaways

- Use `CodingKeys` to customize which properties are encoded/decoded, and what names they are encoded under and decoded from
- Use custom `encode(to:)` and `init(from:)` implementations to:
  - Transform data as you encode/decode it
  - Restructure your data

Super Advanced Usage: Writing Your Own Encoders and Decoders

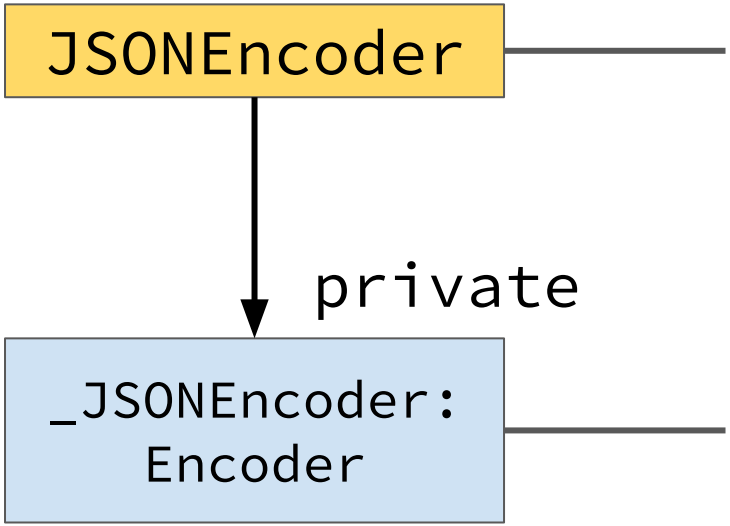
Why doesn't the API match the Encodable protocol?

```
class JSONEncoder {
    func encode<T: Encodable>(_ value: T) throws -> Data
}
```

```
public protocol Encodable {
  func encode(to encoder: Encoder) throws
}
```

Encoder != Encoder

Why doesn't the API match the Encodable protocol?

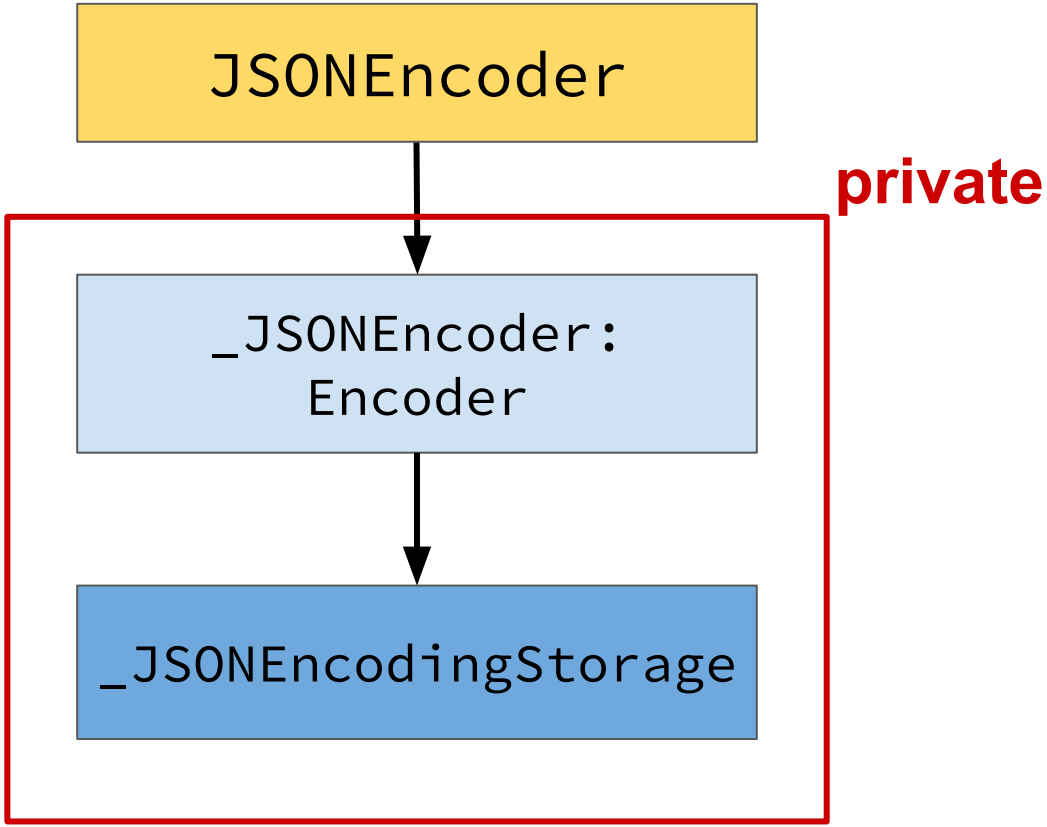


- Not dictated by any protocol
- Allows setting top-level options (e.g. `DateEncodingStrategy`)
- Simple API, just one method that calls through to `_JSONEncoder` methods

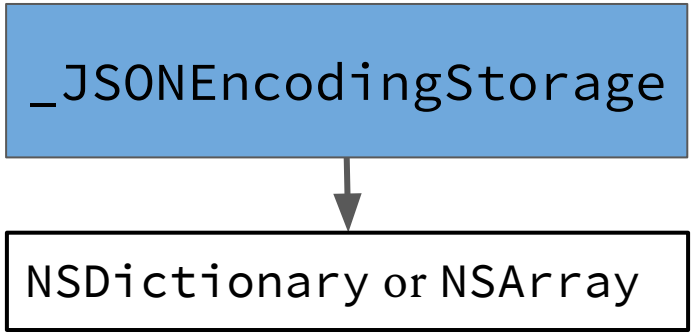
- Implements `Encoder` protocol
- Actually does the encoding work
- More complex container API

https://tinyurl.com/encoder-protocol

JSONEncoder Structure



JSONEncoder Structure



- NSArray if top-level object being encoded is an Array
- NSDictionary otherwise
- Container API is used to construct it
- Why use NS*?
  - JSONSerialization requires it

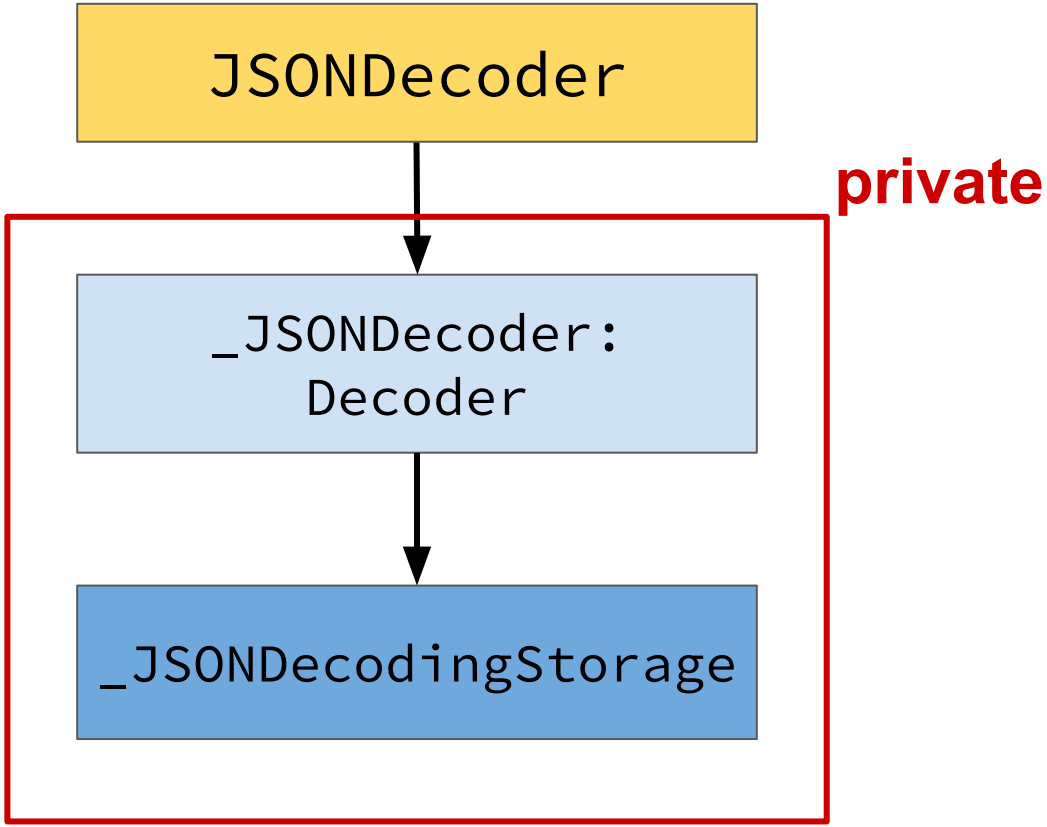
```
class JSONEncoder {
  func encode<T: Encodable>(_ value: T) throws -> Data {
    let privateEncoder = _JSONEncoder()
    try value.encode(to: privateEncoder)
    // ...
```

Get top-level object from `privateEncoder` and pass it to `JSONSerialization`

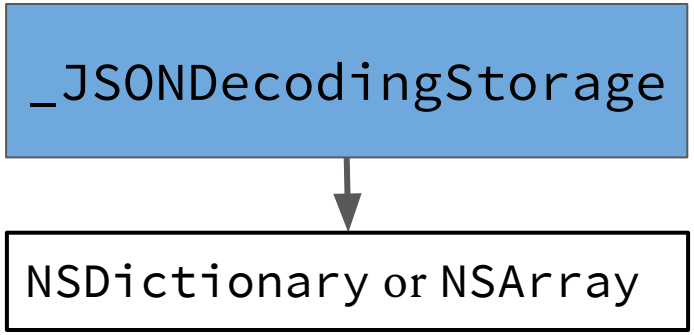
```
  }
}
```

Decoder != Decoder

JSONDecoder Structure



JSONDecoder Structure



- NSArray if JSON array was provided
- NSDictionary if JSON object was provided
- Container API is used to read from it
- Why use NS*?
  - JSONSerialization requires it

```
class JSONDecoder {
  func decode<T: Decodable>(_ type: T.Type, from data: Data) throws -> T {
```

Use `JSONSerialization` to create object from `data`

```
    let privateDecoder = _JSONDecoder(referencing: object)
    return try T(from: privateDecoder)
  }
}
```

Limitations

- Not very performant
  - See https://tinyurl.com/benchmark-codable
- Lots of boilerplate/error prone in some cases
- You can't make someone else's class conform to `Decodable`. (but should you be able to?)
  - See https://tinyurl.com/decodable-class

Advantages

- The API makes `Codable` conformance trivial in many cases, but also allows for very advanced customization when needed.
- The standardized approach makes it so any `Encodable` type can be used with any `Encoder`, and any `Decodable` type can be used with any `Decoder`.

Thank you!

Kaitlin Mahar
Software Engineer @ MongoDB

@k__mahar

@kmahar