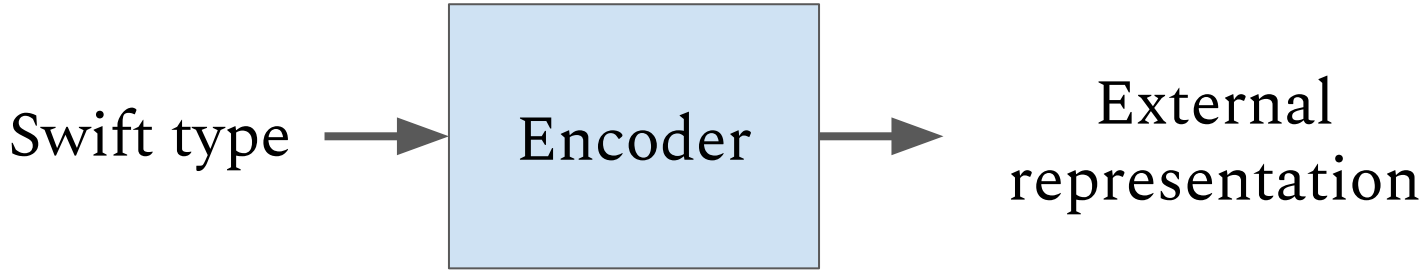
Swift's Encoder and Decoder Protocols

Kaitlin Mahar
Software Engineer @ MongoDB

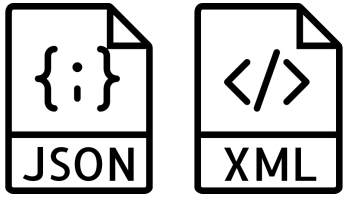
@k__mahar

@kmahar

What is encoding?

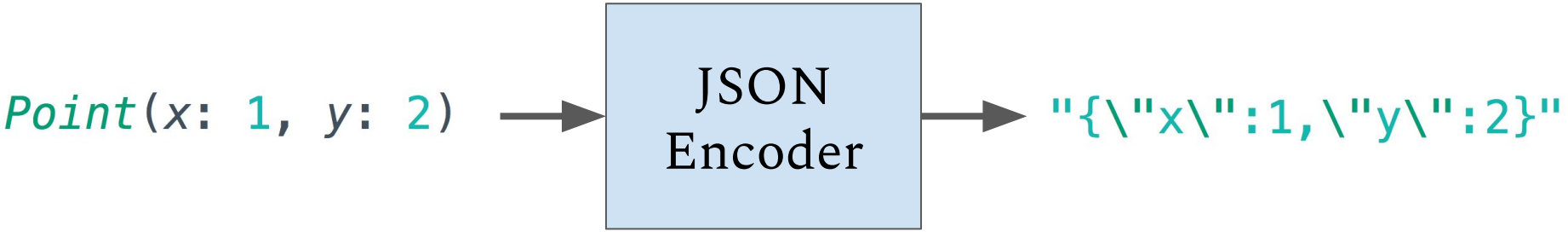


Scalar (e.g. `Int`)
`struct`
`class`
`enum`



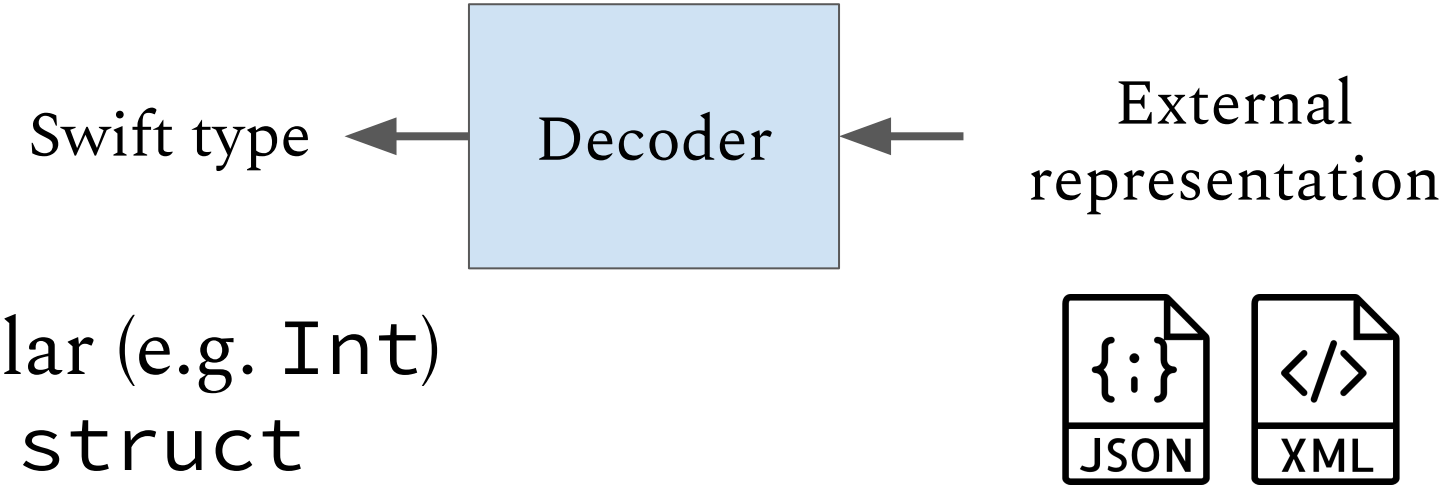
etc...

What is encoding?

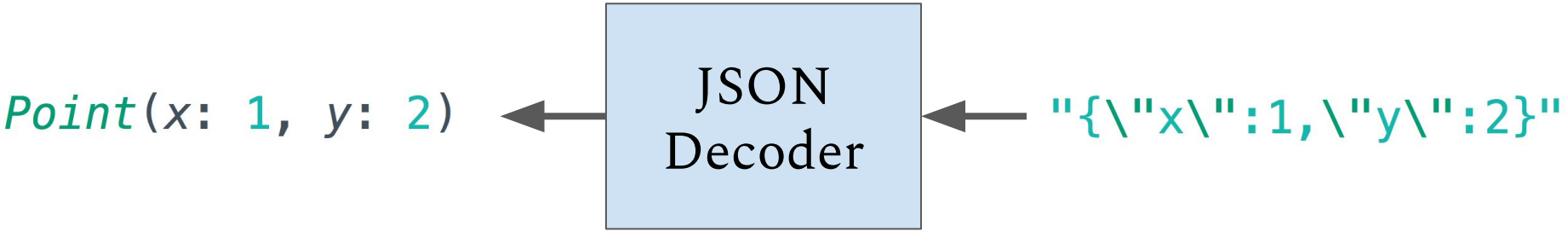



```
struct Point {
  let x: Int
  let y: Int
}
```

What is decoding?



What is decoding?




```
struct Point {
  let x: Int
  let y: Int
}
```

Swift 4 introduced a standardized approach to encoding and decoding.

How does it actually work?

Why does it matter?

- Customize how your types are encoded and decoded
  - Omit or rename properties, flatten nested structs...
- Write your own encoder and/or decoder
  - MongoDB driver
    - `BSONEncoder`: Swift type → MongoDB document
    - `BSONDecoder`: MongoDB document → Swift type

What we'll talk about

- The public API
- Internals
- Limitations

The Public API

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

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

```
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`!

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

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

```
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!

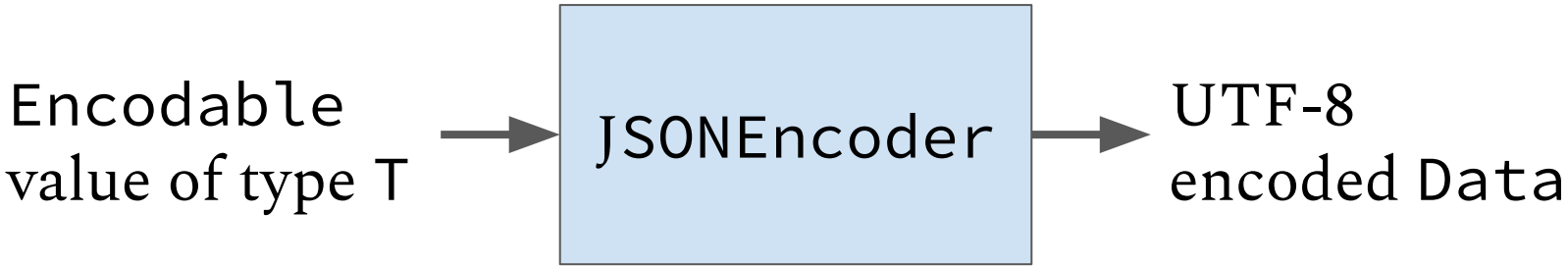
Using Encoders and Decoders

```
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

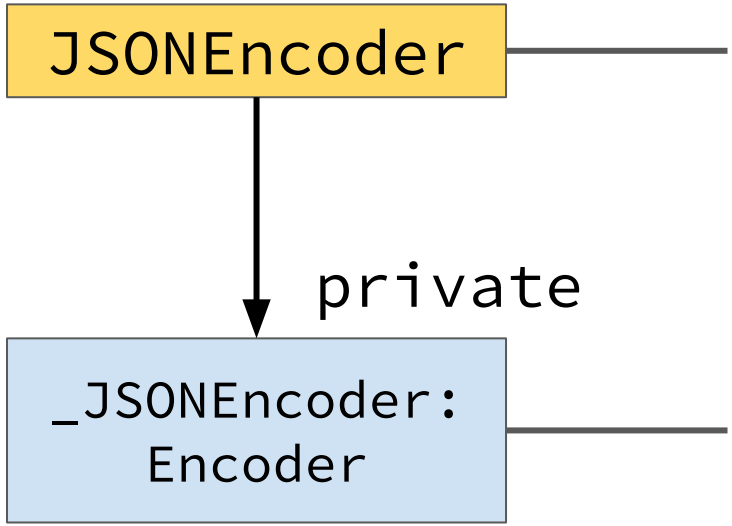
```
class JSONEncoder {
    func encode<T : Encodable>(_ value: T) throws -> Data
}


let encoder = JSONEncoder()
let roscoeData = try encoder.encode(roscoe)


let roscoeStr = String(decoding: roscoeData, as: UTF8.self)
print(roscoeStr)
>> "{\"name\":\"roscoe\",\"color\":\"orange\"}"
```

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
}
```

Why doesn't the API match the Encodable protocol?

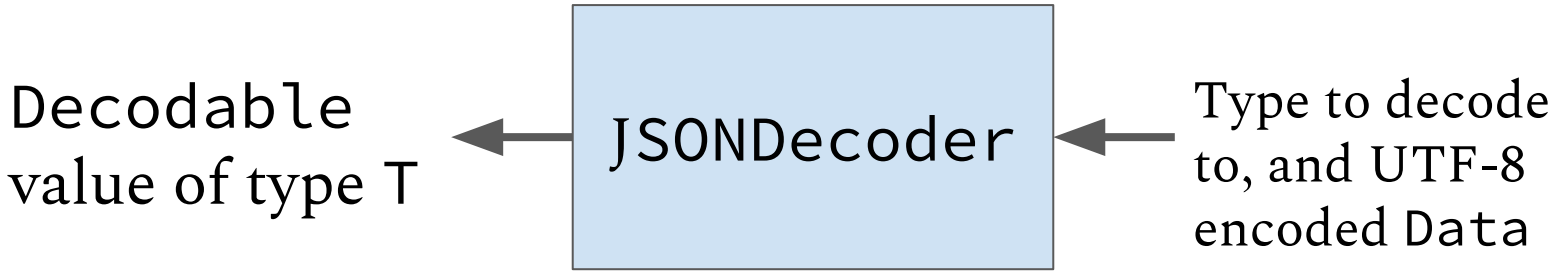


- Not dictated by any protocol
- Allows setting top-level options (e.g. `DateEncodingStrategy`)
- Simple API, just one method

- Implements Encoder protocol
- Actually does the encoding work
- More complex API (stay tuned)

Using A Decoder

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



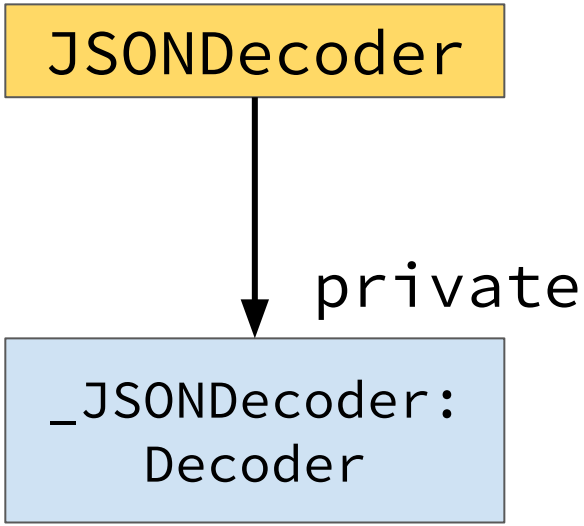
Using A Decoder

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

Data we got from encoding

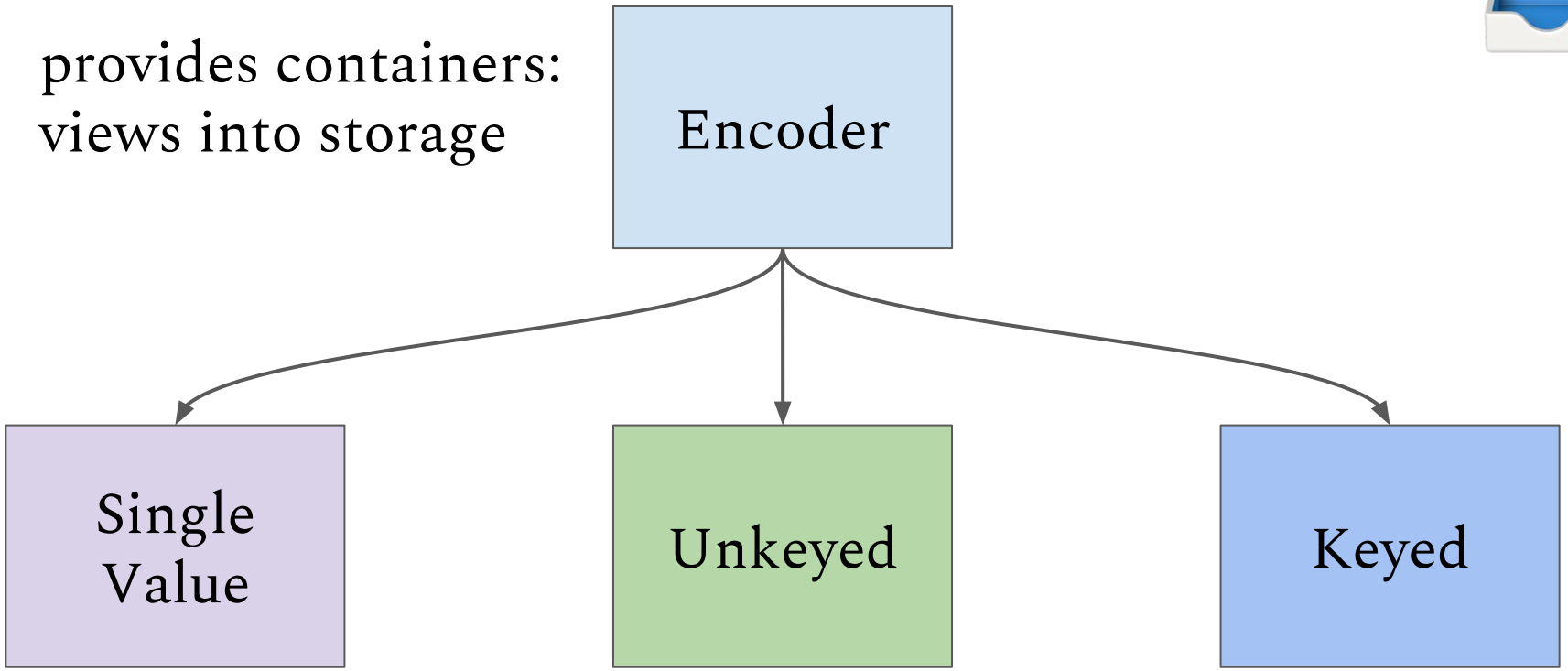
```
let decoder = JSONDecoder()
let roscoe = try decoder.decode(Cat.self, from: roscoeData)

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

Why doesn't the API match the Decodable protocol?



Ok, so... what do these protocols actually require?

provides containers:
views into storage



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

```
public protocol KeyedEncodingContainerProtocol {
  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

CodingKey

```
public protocol Encoder {
  // ...
  func container<Key: CodingKey>(keyedBy type: Key.Type)
                                 throws -> KeyedEncodingContainer<Key>
  // ...
}

public protocol KeyedEncodingContainerProtocol {
  associatedtype Key : CodingKey
  // ...
}
```

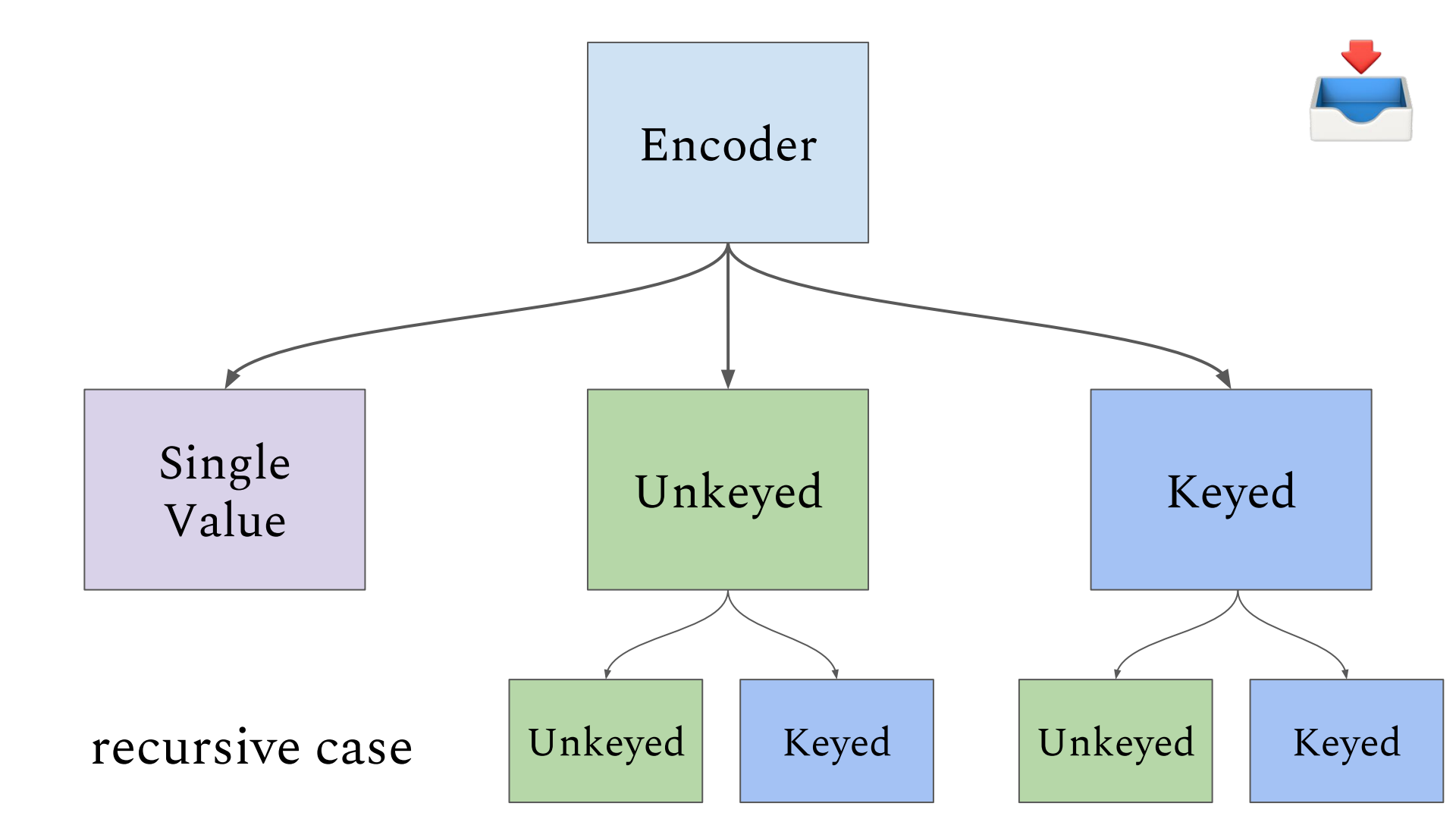
CodingKey

```
public protocol CodingKey {

  init?(stringValue: String)
  init?(intValue: Int)

  var intValue: Int? { get }
  var stringValue: String { get }
}
```

Requirements:

- Initializable by `String` and/or `Int`
- Must have a `String` representation
- May have an `Int` representation

Synthesized conformance for `enum`s!


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

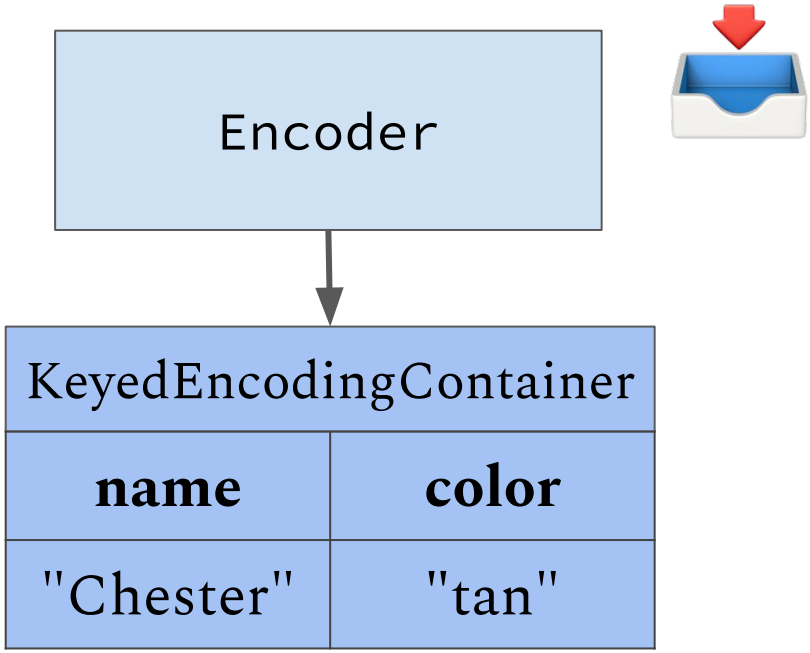
Unkeyed

So how are these containers used?

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

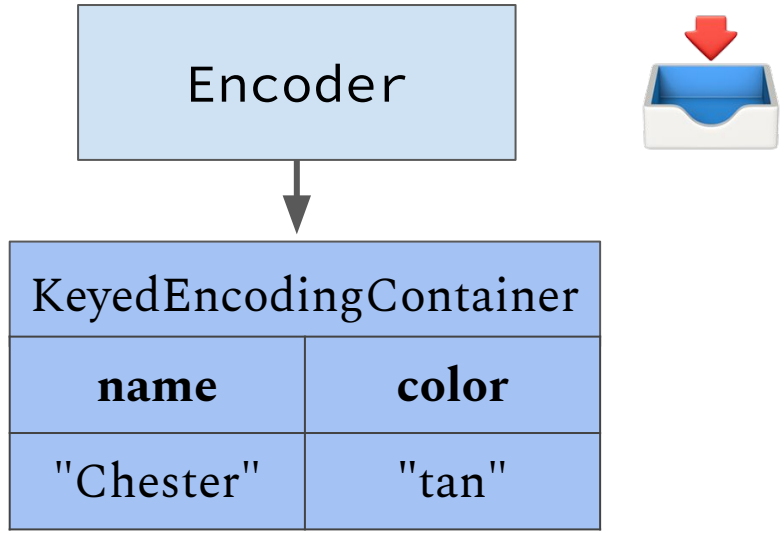
```
let chester = Cat(name: "Chester", color: "tan")
```

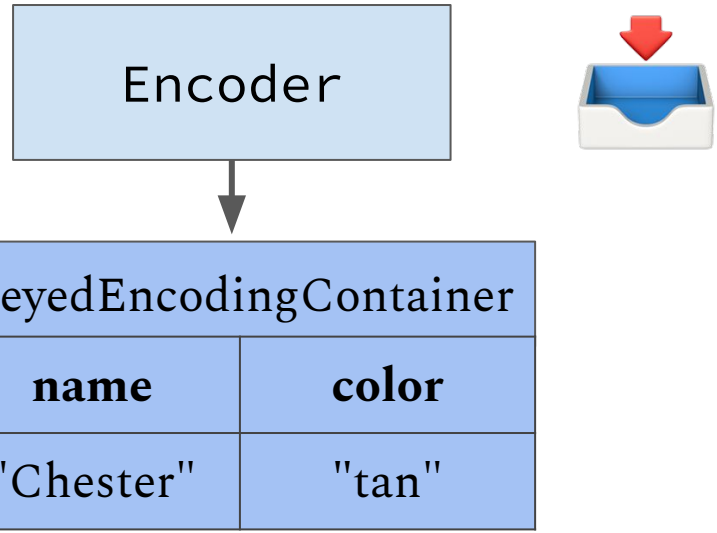
```swift
struct Cat: Codable {
    let name: String
    let color: String
}
```




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

```
struct Cat: Encodable {
    // ...

    enum CodingKeys: CodingKey {
        case name, color
    }


}
```



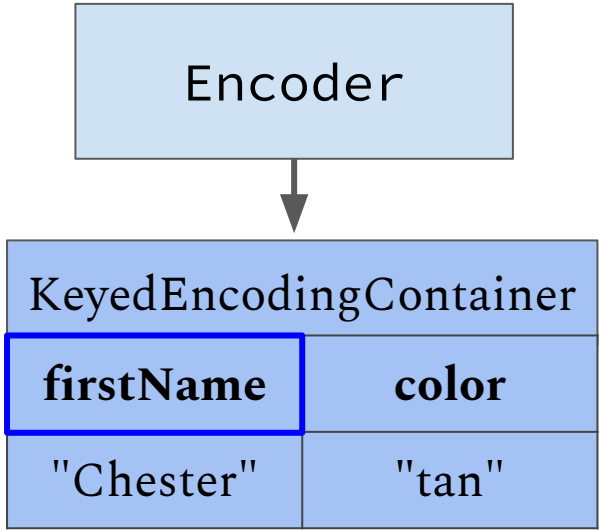



```swift
struct Cat: Encodable {
    // ...

    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)
    }
}
```

Compiler generated!

But what if we don't want the defaults?

```
struct Cat: Encodable {

    let name: String
    let color: String

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

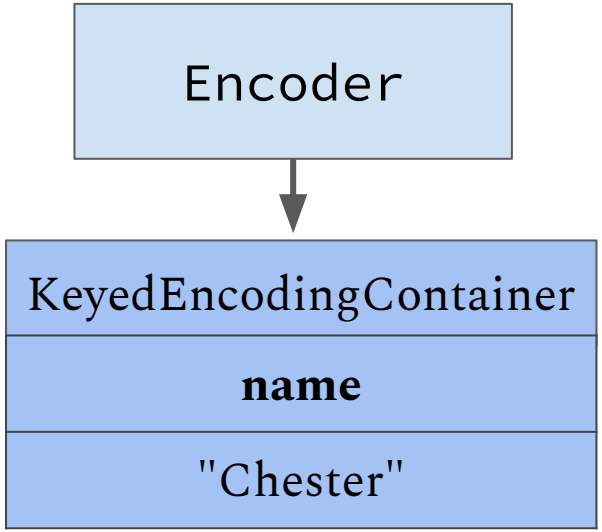


We can override just `CodingKeys` to rename keys but keep the default `encode(to:)` implementation.

But what if we don't want the defaults?

```
struct Cat: Encodable {

    let name: String
    let color: String

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

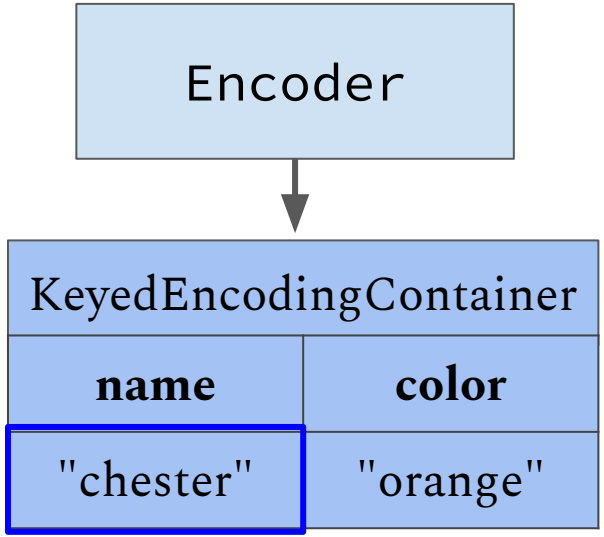
We can override just `CodingKeys` to omit keys altogether.



But what if we don't want the defaults?

```
struct Cat: Encodable {

    let name: String
    let color: String

    private 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)
    }
}
```

Or we can override `encode(to:)` to further customize behavior.



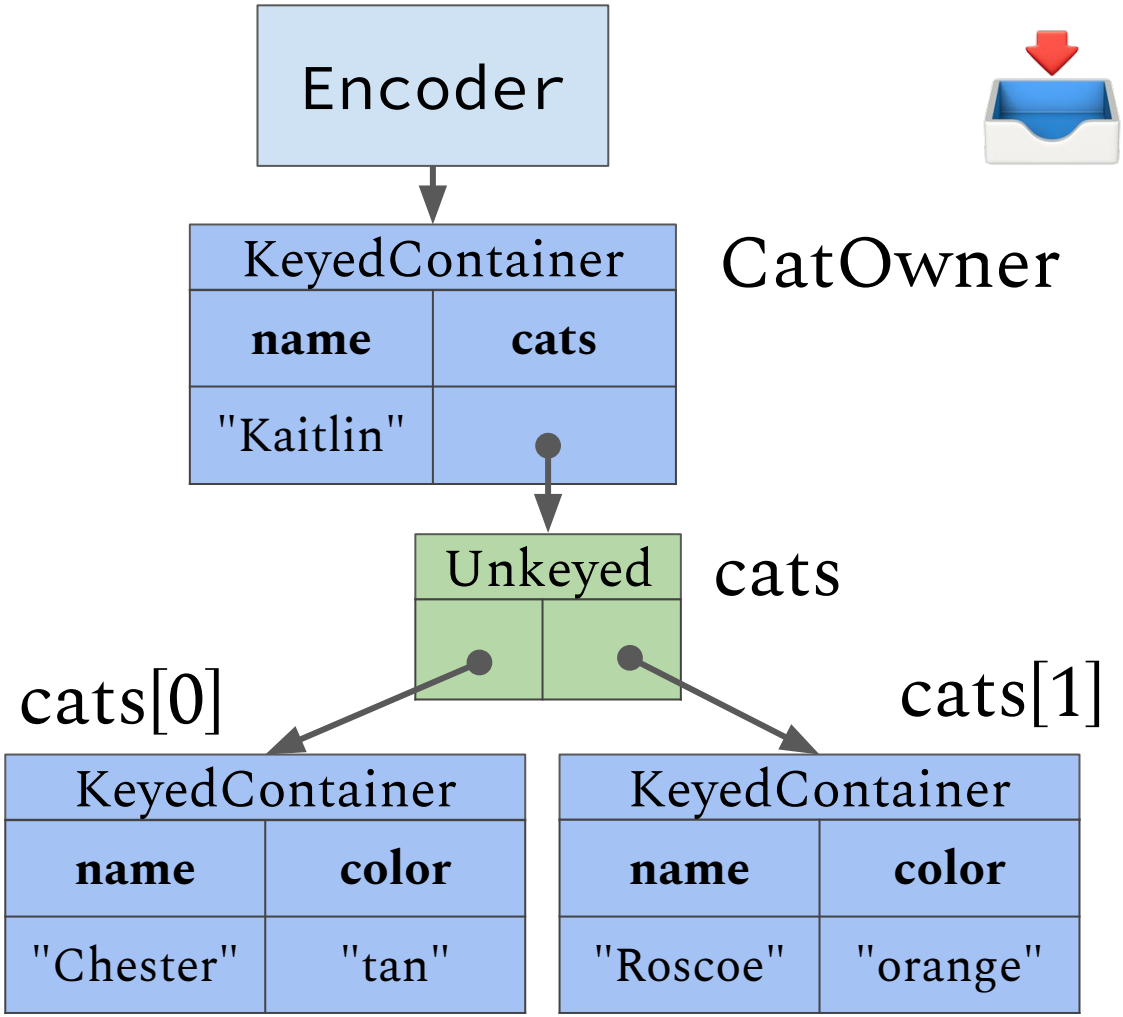
Let's make things more complicated...

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

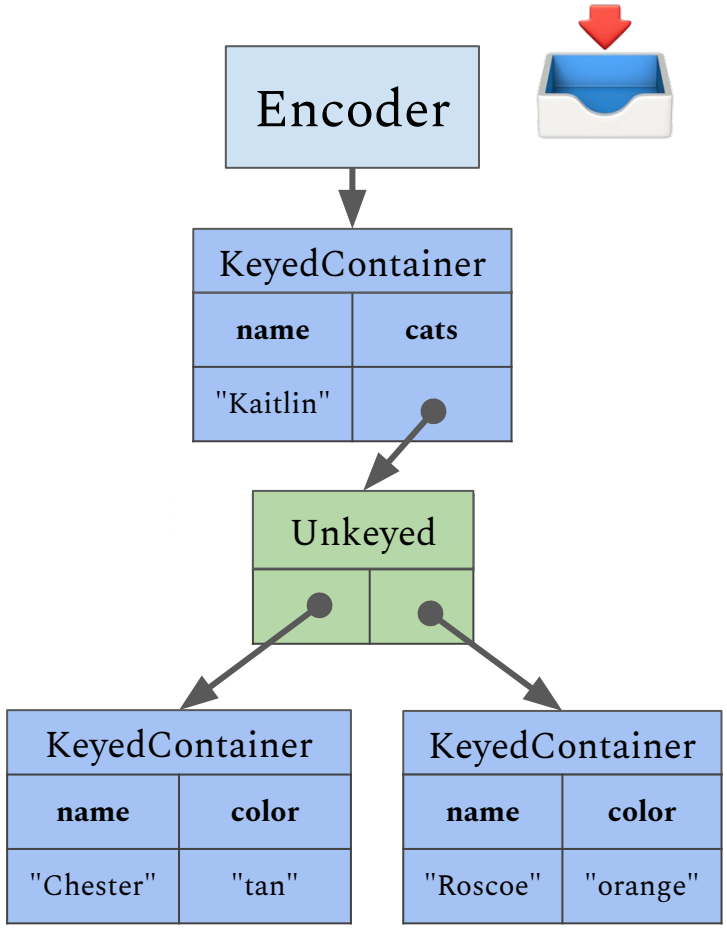
```
let chester = Cat(name: "Chester", color: "tan")
let roscoe = Cat(name: "Roscoe", color: "orange")
let kaitlin = CatOwner(name: "Kaitlin", cats: [chester, roscoe])
```

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

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



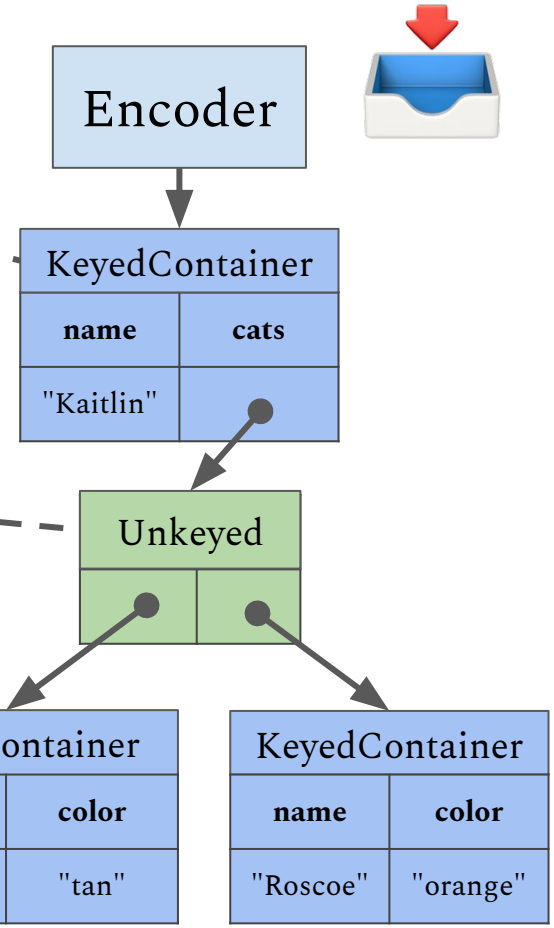
```
struct CatOwner: Encodable {
    // ...

    enum CodingKeys: CodingKey {
        case name, cats
    }

    }
}
```



```swift
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)
}
```

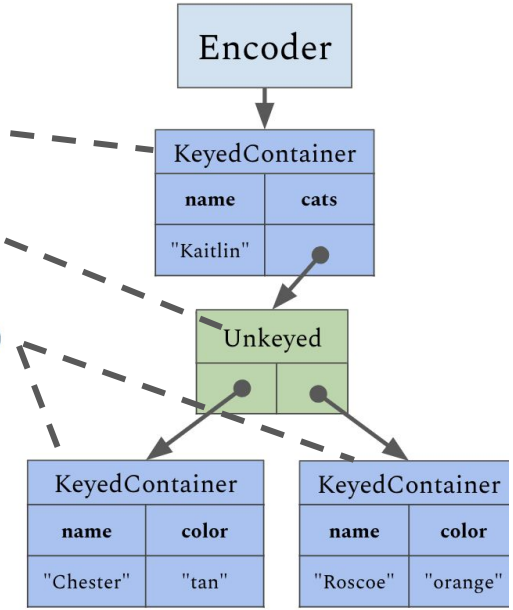
```swift
func encode(to encoder: Encoder) throws {
    var container = encoder.unkeyedContainer()
    for elt in self {
        try container.encode(elt)
    }
}
```

```swift
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)
}
```

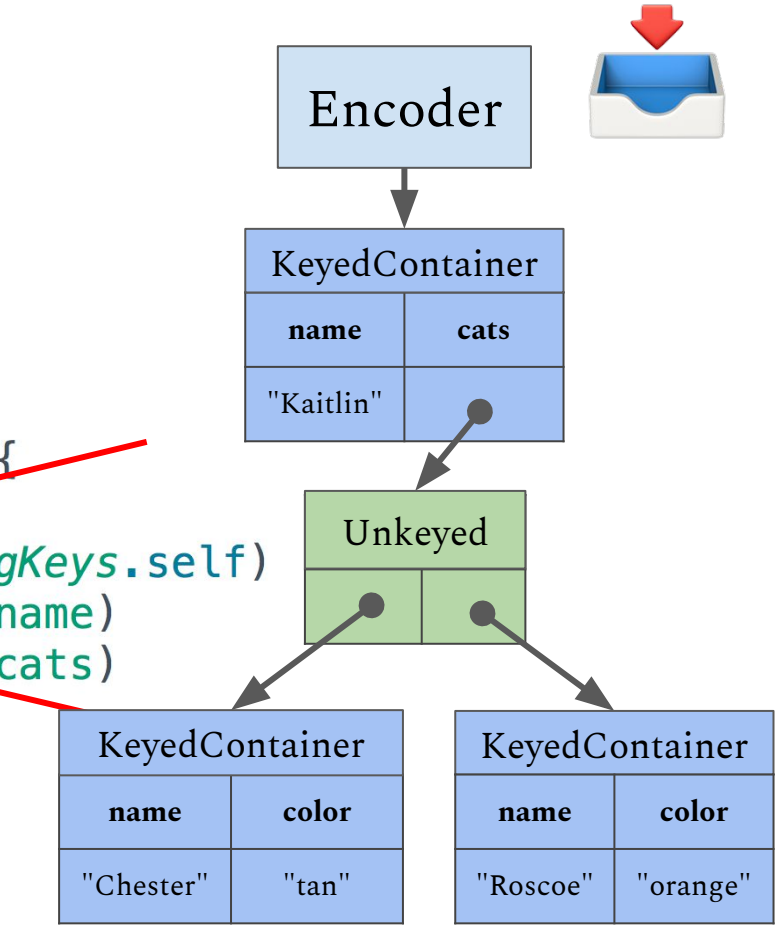


Alternatively...

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

    var catsContainer = encoder.nestedUnkeyedContainer(forKey: .cats)

    for cat in cats {
        var aCat = catsContainer.nestedContainer(keyedBy: CatKeys.self)
        try aCat.encode(cat.name, forKey: .name)
        try aCat.encode(cat.color, forKey: .color)
    }
}
```



```
struct CatOwner: Encodable {
    // ...

    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)
    }
}
```

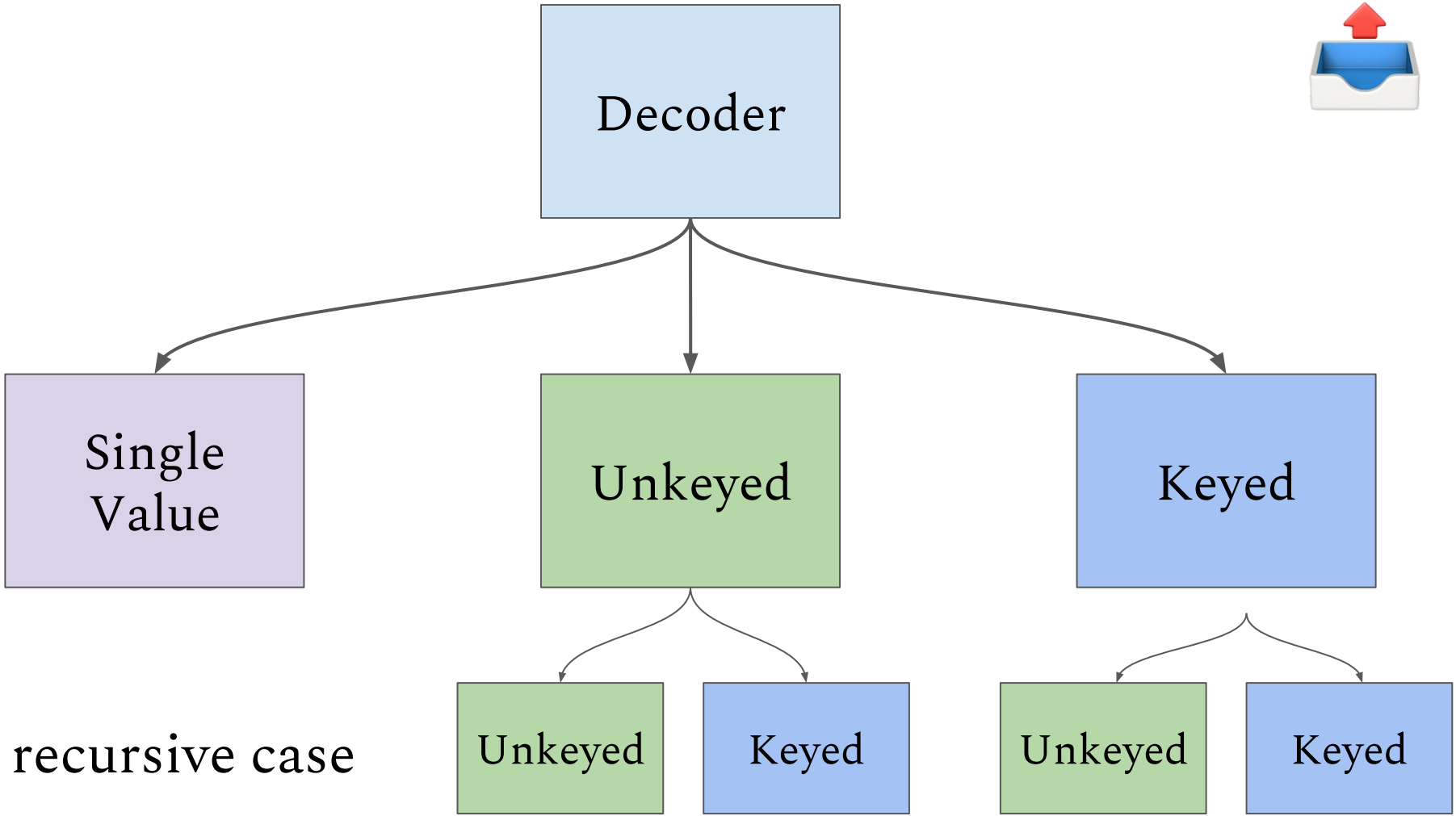


Again, compiler generated!

Weren't we also talking about decoding?

Encoding: putting values into containers

Decoding: taking values out of containers


Decoder
Single Value
Unkeyed
Keyed
Unkeyed
Keyed
Unkeyed
Keyed
recursive case

In code...

```
public protocol Decoder

  func singleValueContainer() throws -> SingleValueDecodingContainer

  func unkeyedContainer() throws -> UnkeyedDecodingContainer

  func container<Key>(keyedBy type: Key.Type) throws
                                  -> KeyedDecodingContainer<Key>
  // ...
}
```

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

```
public protocol SingleValueDecodingContainer {

  func decodeNil() -> Bool

% for type in primitives:
  func decode(_ type: ${type}.Type) throws -> ${type}
% end

  func decode<T : Decodable>(_ type: T.Type) throws -> T
}
```

nil

primitive

Decodable

```
public protocol UnkeyedDecodingContainer

  mutating func decodeNil() throws -> Bool

  % for type in primitives:
    mutating func decode(_ type: ${type}.Type) throws -> ${type}
  % end

  mutating func decode<T : Decodable>(_ type: T.Type) th
}
```

nil

primitive

Decodable

KeyedDecodingContainerProtocol

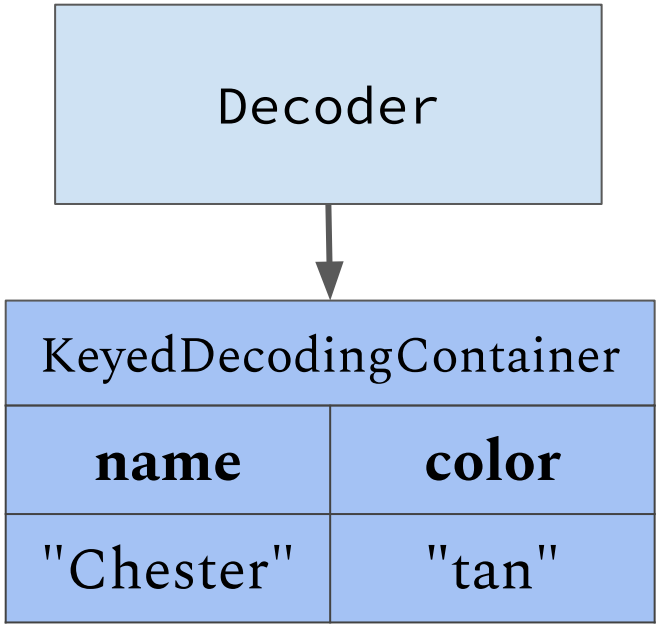
```
public protocol                                    {
  associatedtype Key : CodingKey

  func decodeNil(forKey key: Key) throws -> Bool

% for type in primitives:
  func decode(_ type: ${type}.Type,
              forKey key: Key) throws -> ${type}
% end

  func decode<T : Decodable>(_ type: T.Type,
                             forKey key: Key) throws -> T
}
```

nil

primitive

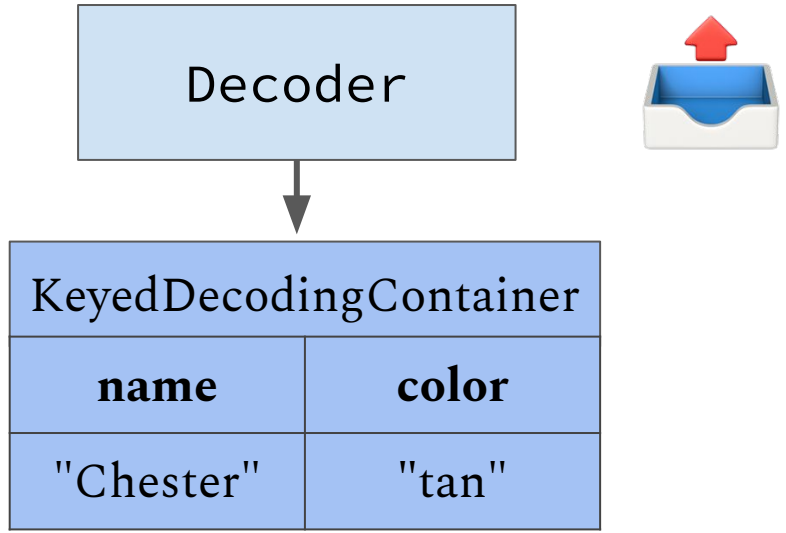
Decodable

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

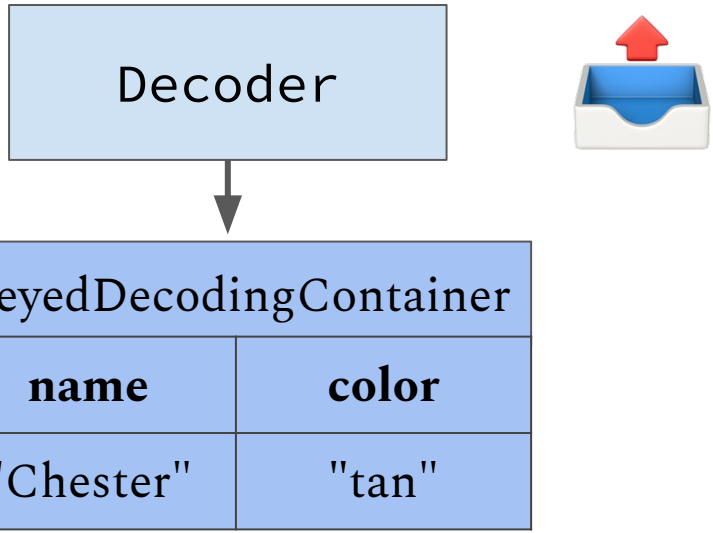



```swift
let chester = Cat(name: "Chester", color: "tan")
```

```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)
  }
}
```



```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)
  }
}
```



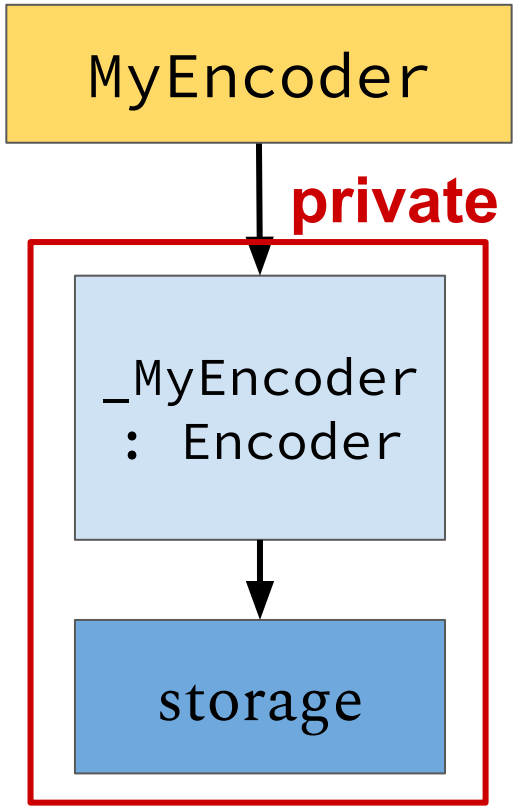
Compiler generated!

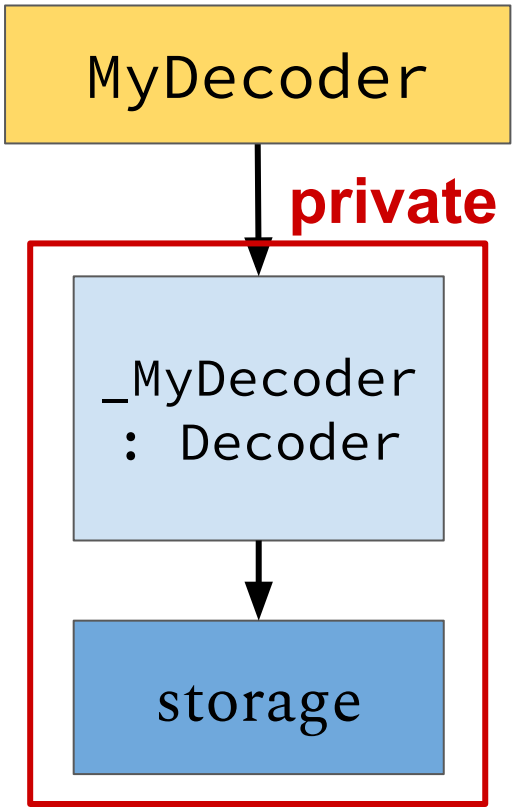
Public API Takeaways

- Types opt in by conforming to `Encodable` and/or `Decodable`
- Many types get conformance for free, but can customize when needed
- Public types (e.g. `JSONEncoder`) do *not* implement the corresponding protocols!
- `Encoders` and `Decoders` used container-based APIs for reading from/writing to storage
- The `Encoder` and `Decoder` APIs are very similar!

Underneath the Hood

Typical Structure





Storage

In what format do I store values while I am still in the middle of encoding/decoding?

- `JSON`: `NS*` types that work with `JSONSerialization`.
- `PropertyList`: `NS*` types that work with `PropertyListSerialization`.
- `BSON`: types conforming to `BsonValue` that work with MongoDB documents.

How do I track where I am in the maze of nested containers?

- `JSON`, `BSON`, `PropertyList`: stack with whatever is backing the current container on top

4 more small but important things to note:

1. `SingleValueXContainer` is often just implemented via an extension of the private `Encoder` type.
2. The `Encoder` must enforce that only one value is written to a `SingleValueEncodingContainer`.
3. There are various bookkeeping requirements on the protocols for tracking the path of keys taken so far.
4. There are also `superEncoder` and `superDecoder` requirements for use with classes.

Limitations

- Lots of boilerplate required for all of the containers, and the different encode and decode methods that they support
- Depending on design, you may end up duplicating a lot of code from `JSONEncoder` etc.
- Types cannot be extended to become `Decodable` (but should it be possible?)

In conclusion...

- The API makes Codable conformance trivial in many cases, but also allows for very advanced customization when needed.
- The API makes it possible to write Encoders and Decoders that don't know what the types using them look like, and makes it possible for types to write encode methods without caring about the actual output format!
- Investigating the standard library source code and writing your own encoder/decoder reveals some hidden requirements.

Thank you!

Kaitlin Mahar
Software Engineer @ MongoDB


@k__mahar


@kmahar