Swift Archival & Serialization

Introduction

Foundation's current archival and serialization APIs (NSCoding, NSJSONSerialization, NSPropertyListSerialization, etc.), while fitting for the dynamism of Objective-C, do not always map optimally into Swift. This document lays out the design of an updated API that improves the developer experience of performing archival and serialization in Swift.

Specifically:

It aims to provide a solution for the archival of Swift struct and enum types
It aims to provide a more type-safe solution for serializing to external formats, such as JSON and plist
Motivation
The primary motivation for this proposal is the inclusion of native Swift enum and struct types in archival and serialization. Currently, developers targeting Swift cannot participate in NSCoding without being willing to abandon enum and struct types — NSCoding is an @objc protocol, conformance to which excludes non-class types. This can be limiting in Swift because small enums and structs can be an idiomatic approach to model representation; developers who wish to perform archival have to either forgo the Swift niceties that constructs like enums provide, or provide an additional compatibility layer between their "real" types and their archivable types.

Secondarily, we would like to refine Foundation's existing serialization APIs (NSJSONSerialization and NSPropertyListSerialization) to better match Swift's strong type safety. From experience, we find that the conversion from the unstructured, untyped data of these formats into strongly-typed data structures is a good fit for archival mechanisms, rather than taking the less safe approach that 3rd-party JSON conversion approaches have taken (described further in an appendix below).

We would like to offer a solution to these problems without sacrificing ease of use or type safety.

Agenda
This proposal is the first stage of three that introduce different facets of a whole Swift archival and serialization API:

This proposal describes the basis for this API, focusing on the protocols that users adopt and interface with
The next stage will propose specific API for new encoders
The final stage will discuss how this new API will interop with NSCoding as it is today
SE-0167 provides stages 2 and 3.

Proposed solution

We will be introducing the Encodable and Decodable protocols, adoption of which will allow end user types to participate in encoding and decoding:

```
// Codable implies Encodable and Decodable
// If all properties are Codable, protocol implementation is automatically generated by the compiler:
public struct Location : Codable {
    public let latitude: Double
    public let longitude: Double
}

public enum Animal : Int, Codable {
    case chicken = 1
    case dog
    case turkey
    case cow
}

public struct Farm : Codable {
    public let name: String
    public let location: Location
    public let animals: [Animal]
}
```

With developer participation, we will offer encoders and decoders (described in SE-0167, not here) that take advantage of this conformance to offer type-safe serialization of user models:

```
let farm = Farm(name: "Old MacDonald's Farm",
                location: Location(latitude: 51.621648, longitude: 0.269273),
                animals: [.chicken, .dog, .cow, .turkey, .dog, .chicken, .cow, .turkey, .dog])
let payload: Data = try JSONEncoder().encode(farm)

do {
    let farm = try JSONDecoder().decode(Farm.self, from: payload)

    // Extracted as user types:
    let coordinates = "\(farm.location.latitude, farm.location.longitude)"
} catch {
    // Encountered error during deserialization
}
```

This gives developers access to their data in a type-safe manner and a recognizable interface.

Detailed design

We will be introducing the following new types to the Swift standard library:

protocol Encodable & protocol Decodable: Adopted by types to opt into archival. Implementation can be synthesized by the compiler in cases where all properties are also Encodable or Decodable

protocol CodingKey: Adopted by types used as keys for keyed containers, replacing String keys with semantic types. Implementation can be synthesized by the compiler in most cases
protocol Encoder: Adopted by types which can take Encodable values and encode them into a native format
protocol KeyedEncodingContainerProtocol: Adopted by types which provide a concrete way to store encoded values by CodingKey. Types adopting Encoder should provide types conforming to KeyedEncodingContainerProtocol to vend
struct KeyedEncodingContainer<Key : CodingKey>: A concrete type-erased box for exposing KeyedEncodingContainerProtocol types; this is a type consumers of the API interact with directly
protocol UnkeyedEncodingContainer: Adopted by types which provide a concrete way to stored encoded values with no keys. Types adopting Encoder should provide types conforming to UnkeyedEncodingContainer to vend
protocol SingleValueEncodingContainer: Adopted by types which provide a concrete way to store a single encoded value. Types adopting Encoder should provide types conforming to SingleValueEncodingContainer to vend
protocol Decoder: Adopted by types which can take payloads in a native format and decode Decodable values out of them
protocol KeyedDecodingContainerProtocol: Adopted by types which provide a concrete way to retrieve encoded values from storage by CodingKey. Types adopting Decoder should provide types conforming to KeyedDecodingContainerProtocol to vend
struct KeyedDecodingContainer<Key : CodingKey>: A concrete type-erased box for exposing KeyedDecodingContainerProtocol types; this is a type consumers of the API interact with directly
protocol UnkeyedDecodingContainer: Adopted by types which provide a concrete way to retrieve encoded values from storage with no keys. Types adopting Decoder should provide types conforming to UnkeyedDecodingContainer to vend
protocol SingleValueDecodingContainer: Adopted by types which provide a concrete way to retrieve a single encoded value from storage. Types adopting Decoder should provide types conforming to SingleValueDecodingContainer to vend
struct CodingUserInfoKey: A String RawRepresentable struct for representing keys to use in Encoders' and Decoders' userInfo dictionaries
To support user types, we expose the Encodable and Decodable protocols:

```
/// Conformance to `Encodable` indicates that a type can encode itself to an external representation.
public protocol Encodable {
    /// Encodes `self` into the given encoder.
    ///
     /// If `self` fails to encode anything, `encoder` will encode an empty keyed container in its place.
    ///
    /// - parameter encoder: The encoder to write data to.
    /// - throws: An error if any values are invalid for `encoder`'s format.
    func encode(to encoder: Encoder) throws
}
```

/// Conformance to `Decodable` indicates that a type can decode itself from an external representation.
public protocol Decodable {
  /// Initializes `self` by decoding from `decoder`.
  ///
  /// - parameter decoder: The decoder to read data from.
  /// - throws: An error if reading from the decoder fails, or if read data is corrupted or otherwise invalid.
  init(from decoder: Decoder) throws
}

/// Conformance to `Codable` indicates that a type can convert itself into and out of an external representation.
public typealias Codable = Encodable & Decodable
By adopting these protocols, user types opt in to this system.

Structured types (i.e. types which encode as a collection of properties) encode and decode their properties in a keyed manner. Keys are semantic String-convertible enums which map properties to encoded names. Keys must conform to the CodingKey protocol:

/// Conformance to `CodingKey` indicates that a type can be used as a key for encoding and decoding.
public protocol CodingKey {
  /// The string to use in a named collection (e.g. a string-keyed dictionary).
  var stringValue: String { get }

  /// Initializes `self` from a string.
  ///
  /// - parameter stringValue: The string value of the desired key.
  /// - returns: An instance of `Self` from the given string, or `nil` if the given string does not correspond to any instance of `Self`.
  init?(stringValue: String)

  /// The int to use in an indexed collection (e.g. an int-keyed dictionary).
  var intValue: Int? { get }

  /// Initializes `self` from an integer.
  ///
  /// - parameter intValue: The integer value of the desired key.
  /// - returns: An instance of `Self` from the given integer, or `nil` if the given integer does not correspond to any instance of `Self`.
  init?(intValue: Int)
}
For performance, where relevant, keys may be Int-convertible, and Encoders may choose to make use of Ints over Strings as appropriate. Framework types should provide keys which

have both for flexibility and performance across different types of Encoders.

By default, CodingKey conformance can be derived for enums which have no raw type and no associated values, or String or Int backing:

```
enum Keys1 : CodingKey {
    case a // (stringValue: "a", intValue: nil)
    case b // (stringValue: "b", intValue: nil)

    // The compiler automatically generates the following:
    var stringValue: String {
        switch self {
        case .a: return "a"
        case .b: return "b"
        }
    }

    init?(stringValue: String) {
        switch stringValue {
        case "a": self = .a
        case "b": self = .b
        default: return nil
        }
    }

    var intValue: Int? {
        return nil
    }

    init?(intValue: Int) {
        return nil
    }
}

enum Keys2 : String, CodingKey {
    case c = "foo" // (stringValue: "foo", intValue: nil)
    case d         // (stringValue: "d", intValue: nil)

        // stringValue, init?(stringValue:), intValue, and init?(intValue:) are generated by the compiler as well
}

enum Keys3 : Int, CodingKey {
    case e = 4 // (stringValue: "e", intValue: 4)
    case f     // (stringValue: "f", intValue: 5)
    case g = 9 // (stringValue: "g", intValue: 9)
```

// stringValue, init?(stringValue:), intValue, and init?(intValue:) are generated by the compiler as well
}
Coding keys which are not enums, have associated values, or have other raw representations must implement these methods manually.

In addition to automatic CodingKey requirement synthesis for enums, Encodable & Decodable requirements can be automatically synthesized for certain types as well:

Types conforming to Encodable whose properties are all Encodable get an automatically generated String-backed CodingKey enum mapping properties to case names. Similarly for Decodable types whose properties are all Decodable
Types falling into (1) — and types which manually provide a CodingKey enum (named CodingKeys, directly, or via a typealias) whose cases map 1-to-1 to Encodable/Decodable properties by name — get automatic synthesis of init(from:) and encode(to:) as appropriate, using those properties and keys
Types which fall into neither (1) nor (2) will have to provide a custom key type if needed and provide their own init(from:) and encode(to:), as appropriate
This synthesis can always be overridden by a manual implementation of any protocol requirements. Many types will either allow for automatic synthesis of all of codability (1), or provide a custom key subset and take advantage of automatic method synthesis (2).

Encoding and Decoding
Types which are Encodable encode their data into a container provided by their Encoder:

/// An `Encoder` is a type which can encode values into a native format for external representation.
public protocol Encoder {
/// Returns an encoding container appropriate for holding multiple values keyed by the given key type.
///
/// - parameter type: The key type to use for the container.
/// - returns: A new keyed encoding container.
/// - precondition: May not be called after a prior `self.unkeyedContainer()` call.
/// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.
func container<Key : CodingKey>(keyedBy type: Key.Type) -> KeyedEncodingContainer<Key>

/// Returns an encoding container appropriate for holding multiple unkeyed values.
///
/// - returns: A new empty unkeyed container.
/// - precondition: May not be called after a prior `self.container(keyedBy:)` call.
/// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.

```
    func unkeyedContainer() -> UnkeyedEncodingContainer

    /// Returns an encoding container appropriate for holding a single primitive value.
    ///
    /// - returns: A new empty single value container.
    /// - precondition: May not be called after a prior `self.container(keyedBy:)` call.
    /// - precondition: May not be called after a prior `self.unkeyedContainer()` call.
     /// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.
    func singleValueContainer() -> SingleValueEncodingContainer

    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }
}

// Continuing examples from before; below is automatically generated by the compiler if no customization is needed.
public struct Location : Codable {
    private enum CodingKeys : CodingKey {
        case latitude
        case longitude
    }

    public func encode(to encoder: Encoder) throws {
        // Generic keyed encoder gives type-safe key access: cannot encode with keys of the wrong type.
        var container = encoder.container(keyedBy: CodingKeys.self)

        // The encoder is generic on the key -- free key autocompletion here.
        try container.encode(latitude, forKey: .latitude)
        try container.encode(longitude, forKey: .longitude)
    }
}

public struct Farm : Codable {
    private enum CodingKeys : CodingKey {
        case name
        case location
        case animals
    }

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

```
        try container.encode(animals, forKey: .animals)
    }
}
```

Similarly, Decodable types initialize from data read from their Decoder's container:

```
/// A `Decoder` is a type which can decode values from a native format into in-memory representations.
public protocol Decoder {
    /// Returns the data stored in `self` as represented in a container keyed by the given key type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - returns: A keyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
    func container<Key : CodingKey>(keyedBy type: Key.Type) throws -> KeyedDecodingContainer<Key>

    /// Returns the data stored in `self` as represented in a container appropriate for holding values with no keys.
    ///
    /// - returns: An unkeyed container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    func unkeyedContainer() throws -> UnkeyedDecodingContainer

    /// Returns the data stored in `self` as represented in a container appropriate for holding a single primitive value.
    ///
    /// - returns: A single value container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a single value container.
    func singleValueContainer() throws -> SingleValueDecodingContainer

    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }
}

// Continuing examples from before; below is automatically generated by the compiler if no customization is needed.
public struct Location : Codable {
    public init(from decoder: Decoder) throws {
        var container = try decoder.container(keyedBy: CodingKeys.self)
        latitude = try container.decode(Double.self, forKey: .latitude)
        longitude = try container.decode(Double.self, forKey: .longitude)
```

```
    }
}

public struct Farm : Codable {
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        location = try container.decode(Location.self, forKey: .location)
        animals = try container.decode([Animal].self, forKey: .animals)
    }
}
```

Keyed Containers

Keyed containers are the primary interface that most Codable types interact with for encoding and decoding. Through these, Codable types have strongly-keyed access to encoded data by using keys that are semantically correct for the operations they want to express.

Since semantically incompatible keys will rarely (if ever) share the same key type, it is impossible to mix up key types within the same container (as is possible with String keys), and since the type is known statically, keys get autocompletion by the compiler.

```
/// Conformance to `KeyedEncodingContainerProtocol` indicates that a type provides a view into an `Encoder`'s storage and is used to hold the encoded properties of an `Encodable` type in a keyed manner.
///
/// Encoders should provide types conforming to `KeyedEncodingContainerProtocol` for their format.
public protocol KeyedEncodingContainerProtocol {
    associatedtype Key : CodingKey

    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode<T : Encodable>(_ value: T?, forKey key: Key) throws

    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode(_ value: Bool?,   forKey key: Key) throws
    mutating func encode(_ value: Int?,    forKey key: Key) throws
```

```
    mutating func encode(_ value: Int8?,   forKey key: Key) throws
    mutating func encode(_ value: Int16?,  forKey key: Key) throws
    mutating func encode(_ value: Int32?,  forKey key: Key) throws
    mutating func encode(_ value: Int64?,  forKey key: Key) throws
    mutating func encode(_ value: UInt?,   forKey key: Key) throws
    mutating func encode(_ value: UInt8?,  forKey key: Key) throws
    mutating func encode(_ value: UInt16?, forKey key: Key) throws
    mutating func encode(_ value: UInt32?, forKey key: Key) throws
    mutating func encode(_ value: UInt64?, forKey key: Key) throws
    mutating func encode(_ value: Float?,  forKey key: Key) throws
    mutating func encode(_ value: Double?, forKey key: Key) throws
    mutating func encode(_ value: String?, forKey key: Key) throws

    /// Encodes the given object weakly for the given key.
    ///
     /// For `Encoder`s that implement this functionality, this will only encode the given object and associate it with the given key if it is encoded unconditionally elsewhere in the payload (either previously or in the future).
    ///
      /// For formats which don't support this feature, the default implementation encodes the given object unconditionally.
    ///
    /// - parameter object: The object to encode.
    /// - parameter key: The key to associate the object with.
       /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
     mutating func encodeWeak<T : AnyObject & Encodable>(_ object: T?, forKey key: Key) throws

    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }
}

/// `KeyedEncodingContainer` is a type-erased box for `KeyedEncodingContainerProtocol` types, similar to `AnyCollection` and `AnyHashable`. This is the type which consumers of the API interact with directly.
public struct KeyedEncodingContainer<K : CodingKey> : KeyedEncodingContainerProtocol {
    associatedtype Key = K

    /// Initializes `self` with the given container.
    ///
    /// - parameter container: The container to hold.
         init<Container : KeyedEncodingContainerProtocol>(_ container: Container) where Container.Key == Key
```

```
    // + methods from KeyedEncodingContainerProtocol
}

/// Conformance to `KeyedDecodingContainerProtocol` indicates that a type provides a view into a `Decoder`'s storage and is used to hold the encoded properties of a `Decodable` type in a keyed manner.
///
/// Decoders should provide types conforming to `KeyedDecodingContainerProtocol` for their format.
public protocol KeyedDecodingContainerProtocol {
    associatedtype Key : CodingKey

    /// All the keys the `Decoder` has for this container.
    ///
    /// Different keyed containers from the same `Decoder` may return different keys here; it is possible to encode with multiple key types which are not convertible to one another. This should report all keys present which are convertible to the requested type.
    var allKeys: [Key] { get }

    /// Returns whether the `Decoder` contains a value associated with the given key.
    ///
    /// The value associated with the given key may be a null value as appropriate for the data format.
    ///
    /// - parameter key: The key to search for.
    /// - returns: Whether the `Decoder` has an entry for the given key.
    func contains(_ key: Key) -> Bool

    /// Decodes a value of the given type for the given key.
    ///
    /// A default implementation is given for these types which calls into the `decodeIfPresent` implementations below.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
    /// - returns: A value of the requested type, if present for the given key and convertible to the requested type.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key or if the value is null.
    func decode(_ type: Bool.Type,   forKey key: Key) throws -> Bool
    func decode(_ type: Int.Type,    forKey key: Key) throws -> Int
    func decode(_ type: Int8.Type,   forKey key: Key) throws -> Int8
    func decode(_ type: Int16.Type,  forKey key: Key) throws -> Int16
    func decode(_ type: Int32.Type,  forKey key: Key) throws -> Int32
```

```
    func decode(_ type: Int64.Type,  forKey key: Key) throws -> Int64
    func decode(_ type: UInt.Type,   forKey key: Key) throws -> UInt
    func decode(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8
    func decode(_ type: UInt16.Type, forKey key: Key) throws -> UInt16
    func decode(_ type: UInt32.Type, forKey key: Key) throws -> UInt32
    func decode(_ type: UInt64.Type, forKey key: Key) throws -> UInt64
    func decode(_ type: Float.Type,  forKey key: Key) throws -> Float
    func decode(_ type: Double.Type, forKey key: Key) throws -> Double
    func decode(_ type: String.Type, forKey key: Key) throws -> String
    func decode<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T

    /// Decodes a value of the given type for the given key, if present.
    ///
    /// This method returns `nil` if the container does not have a value associated with `key`, or if the value is null. The difference between these states can be distinguished with a `contains(_:)` call.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
    /// - returns: A decoded value of the requested type, or `nil` if the `Decoder` does not have an entry associated with the given key, or if the value is a null value.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    func decodeIfPresent(_ type: Bool.Type,   forKey key: Key) throws -> Bool?
    func decodeIfPresent(_ type: Int.Type,    forKey key: Key) throws -> Int?
    func decodeIfPresent(_ type: Int8.Type,   forKey key: Key) throws -> Int8?
    func decodeIfPresent(_ type: Int16.Type,  forKey key: Key) throws -> Int16?
    func decodeIfPresent(_ type: Int32.Type,  forKey key: Key) throws -> Int32?
    func decodeIfPresent(_ type: Int64.Type,  forKey key: Key) throws -> Int64?
    func decodeIfPresent(_ type: UInt.Type,   forKey key: Key) throws -> UInt?
    func decodeIfPresent(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8?
    func decodeIfPresent(_ type: UInt16.Type, forKey key: Key) throws -> UInt16?
    func decodeIfPresent(_ type: UInt32.Type, forKey key: Key) throws -> UInt32?
    func decodeIfPresent(_ type: UInt64.Type, forKey key: Key) throws -> UInt64?
    func decodeIfPresent(_ type: Float.Type,  forKey key: Key) throws -> Float?
    func decodeIfPresent(_ type: Double.Type, forKey key: Key) throws -> Double?
    func decodeIfPresent(_ type: String.Type, forKey key: Key) throws -> String?
    func decodeIfPresent<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T?

    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }
}

/// `KeyedDecodingContainer` is a type-erased box for `KeyedDecodingContainerProtocol` types, similar to `AnyCollection` and `AnyHashable`. This is the type which consumers of the
```

API interact with directly.

```
public struct KeyedDecodingContainer<K : CodingKey> : KeyedDecodingContainerProtocol
{
    associatedtype Key = K

    /// Initializes `self` with the given container.
    ///
    /// - parameter container: The container to hold.
        init<Container : KeyedDecodingContainerProtocol>(_ container: Container) where
Container.Key == Key

    // + methods from KeyedDecodingContainerProtocol
}
```

These encode(_:forKey:) and decode(_:forKey:) overloads give strong, static type guarantees about what is encodable (preventing accidental attempts to encode an invalid type), and provide a list of primitive types which are common to all encoders and decoders that users can rely on.

When the conditional conformance feature lands in Swift, the ability to express that "a collection of things which are Codable is Codable" will allow collections (Array, Dictionary, etc.) to be extended and fall into these overloads as well.

Unkeyed Containers

For some types, when the source and destination of a payload can be guaranteed to agree on the payload layout and format (e.g. in cross-process communication, where both sides agree on the payload format), it may be appropriate to eschew the encoding of keys and encode sequentially, without keys. In this case, a type may choose to make use of an unkeyed container for its properties:

```
/// Conformance to `UnkeyedEncodingContainer` indicates that a type provides a view into
an `Encoder`'s storage and is used to hold the encoded properties of an `Encodable` type
sequentially, without keys.
///
/// Encoders should provide types conforming to `UnkeyedEncodingContainer` for their
format.
public protocol UnkeyedEncodingContainer {
    /// Encodes the given value.
    ///
    /// - parameter value: The value to encode.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current
context for this format.
    mutating func encode<T : Encodable>(_ value: T?) throws

    /// Encodes the given value.
    ///
    /// - parameter value: The value to encode.
```

```
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode(_ value: Bool?) throws
    mutating func encode(_ value: Int?) throws
    mutating func encode(_ value: Int8?) throws
    mutating func encode(_ value: Int16?) throws
    mutating func encode(_ value: Int32?) throws
    mutating func encode(_ value: Int64?) throws
    mutating func encode(_ value: UInt?) throws
    mutating func encode(_ value: UInt8?) throws
    mutating func encode(_ value: UInt16?) throws
    mutating func encode(_ value: UInt32?) throws
    mutating func encode(_ value: UInt64?) throws
    mutating func encode(_ value: Float?) throws
    mutating func encode(_ value: Double?) throws
    mutating func encode(_ value: String?) throws

    /// Encodes the given object weakly.
    ///
    /// For `Encoder`s that implement this functionality, this will only encode the given object if it is encoded unconditionally elsewhere in the payload (either previously or in the future).
    ///
    /// For formats which don't support this feature, the default implementation encodes the given object unconditionally.
    ///
    /// - parameter object: The object to encode.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encodeWeak<T : AnyObject & Encodable>(_ object: T?) throws

    /// Encodes the elements of the given sequence.
    ///
    /// A default implementation of these is given in an extension.
    ///
    /// - parameter sequence: The sequences whose contents to encode.
    /// - throws: An error if any of the contained values throws an error.
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Bool
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int
    // ...
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element : Encodable

    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
```

```
    var codingPath: [CodingKey?] { get }
}

/// Conformance to `UnkeyedDecodingContainer` indicates that a type provides a view into a `Decoder`'s storage and is used to hold the encoded properties of a `Decodable` type sequentially, without keys.
///
/// Decoders should provide types conforming to `UnkeyedDecodingContainer` for their format.
public protocol UnkeyedDecodingContainer {
    /// Returns the number of elements (if known) contained within this container.
    var count: Int? { get }

    /// Returns whether there are no more elements left to be decoded in the container.
    var isAtEnd: Bool { get }

    /// Decodes a value of the given type.
    ///
     /// A default implementation is given for these types which calls into the `decodeIfPresent` implementations below.
    ///
    /// - parameter type: The type of value to decode.
     /// - returns: A value of the requested type, if present for the given key and convertible to the requested type.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
     /// - throws: `CocoaError.coderValueNotFound` if the encountered encoded value is null, or of there are no more values to decode.
    mutating func decode(_ type: Bool.Type) throws -> Bool
    mutating func decode(_ type: Int.Type) throws -> Int
    mutating func decode(_ type: Int8.Type) throws -> Int8
    mutating func decode(_ type: Int16.Type) throws -> Int16
    mutating func decode(_ type: Int32.Type) throws -> Int32
    mutating func decode(_ type: Int64.Type) throws -> Int64
    mutating func decode(_ type: UInt.Type) throws -> UInt
    mutating func decode(_ type: UInt8.Type) throws -> UInt8
    mutating func decode(_ type: UInt16.Type) throws -> UInt16
    mutating func decode(_ type: UInt32.Type) throws -> UInt32
    mutating func decode(_ type: UInt64.Type) throws -> UInt64
    mutating func decode(_ type: Float.Type) throws -> Float
    mutating func decode(_ type: Double.Type) throws -> Double
    mutating func decode(_ type: String.Type) throws -> String
    mutating func decode<T : Decodable>(_ type: T.Type) throws -> T

    /// Decodes a value of the given type, if present.
    ///
```

```
    /// This method returns `nil` if the container has no elements left to decode, or if the value is null. The difference between these states can be distinguished by checking `isAtEnd`.
    ///
    /// - parameter type: The type of value to decode.
    /// - returns: A decoded value of the requested type, or `nil` if the value is a null value, or if there are no more elements to decode.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    mutating func decodeIfPresent(_ type: Bool.Type) throws -> Bool?
    mutating func decodeIfPresent(_ type: Int.Type) throws -> Int?
    mutating func decodeIfPresent(_ type: Int8.Type) throws -> Int8?
    mutating func decodeIfPresent(_ type: Int16.Type) throws -> Int16?
    mutating func decodeIfPresent(_ type: Int32.Type) throws -> Int32?
    mutating func decodeIfPresent(_ type: Int64.Type) throws -> Int64?
    mutating func decodeIfPresent(_ type: UInt.Type) throws -> UInt?
    mutating func decodeIfPresent(_ type: UInt8.Type) throws -> UInt8?
    mutating func decodeIfPresent(_ type: UInt16.Type) throws -> UInt16?
    mutating func decodeIfPresent(_ type: UInt32.Type) throws -> UInt32?
    mutating func decodeIfPresent(_ type: UInt64.Type) throws -> UInt64?
    mutating func decodeIfPresent(_ type: Float.Type) throws -> Float?
    mutating func decodeIfPresent(_ type: Double.Type) throws -> Double?
    mutating func decodeIfPresent(_ type: String.Type) throws -> String?
    mutating func decodeIfPresent<T : Decodable>(_ type: T.Type) throws -> T?

    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }
}
```

Unkeyed encoding is fragile and generally not appropriate for archival without specific format guarantees, so keyed encoding remains the recommended approach (and is why CodingKey enums are synthesized by default unless otherwise declined).

Single Value Containers

For other types, an array or dictionary container may not even make sense (e.g. values which are RawRepresentable as a single primitive value). Those types may encode and decode directly as a single value, instead of requesting an outer container:

```
/// A `SingleValueEncodingContainer` is a container which can support the storage and direct encoding of a single non-keyed value.
public protocol SingleValueEncodingContainer {
    /// Encodes a single value of the given type.
    ///
    /// - parameter value: The value to encode.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    /// - precondition: May not be called after a previous `self.encode(_:)` call.
```

```
    mutating func encode(_ value: Bool) throws
    mutating func encode(_ value: Int) throws
    mutating func encode(_ value: Int8) throws
    mutating func encode(_ value: Int16) throws
    mutating func encode(_ value: Int32) throws
    mutating func encode(_ value: Int64) throws
    mutating func encode(_ value: UInt) throws
    mutating func encode(_ value: UInt8) throws
    mutating func encode(_ value: UInt16) throws
    mutating func encode(_ value: UInt32) throws
    mutating func encode(_ value: UInt64) throws
    mutating func encode(_ value: Float) throws
    mutating func encode(_ value: Double) throws
    mutating func encode(_ value: String) throws
}

/// A `SingleValueDecodingContainer` is a container which can support the storage and direct decoding of a single non-keyed value.
public protocol SingleValueDecodingContainer {
    /// Decodes a single value of the given type.
    ///
    /// - parameter type: The type to decode as.
    /// - returns: A value of the requested type.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value cannot be converted to the requested type.
    func decode(_ type: Bool.Type) throws -> Bool
    func decode(_ type: Int.Type) throws -> Int
    func decode(_ type: Int8.Type) throws -> Int8
    func decode(_ type: Int16.Type) throws -> Int16
    func decode(_ type: Int32.Type) throws -> Int32
    func decode(_ type: Int64.Type) throws -> Int64
    func decode(_ type: UInt.Type) throws -> UInt
    func decode(_ type: UInt8.Type) throws -> UInt8
    func decode(_ type: UInt16.Type) throws -> UInt16
    func decode(_ type: UInt32.Type) throws -> UInt32
    func decode(_ type: UInt64.Type) throws -> UInt64
    func decode(_ type: Float.Type) throws -> Float
    func decode(_ type: Double.Type) throws -> Double
    func decode(_ type: String.Type) throws -> String
}

// Continuing example from before; below is automatically generated by the compiler if no customization is needed.
public enum Animal : Int, Codable {
    public func encode(to encoder: Encoder) throws {
        // Encode as a single value; no keys.
```

```
        try encoder.singleValueContainer().encode(self.rawValue)
    }

    public init(from decoder: Decoder) throws {
        // Decodes as a single value; no keys.
        let intValue = try decoder.singleValueContainer().decode(Int.self)
        if let value = Self(rawValue: intValue) {
            self = value
        } else {
            throw CocoaError.error(.coderReadCorrupt)
        }
    }
}
```

In the example given above, since Animal uses a single value container, [.chicken, .dog, .cow, .turkey, .dog, .chicken, .cow, .turkey, .dog] would encode directly as [1, 2, 4, 3, 2, 1, 4, 3, 2].

Nesting

In practice, some types may also need to control how data is nested within their container, or potentially nest other containers within their container. Keyed containers allow this by returning nested containers of differing types:

```
// Continuing from before
public protocol KeyedEncodingContainerProtocol {
    /// Stores a keyed encoding container for the given key and returns it.
    ///
    /// - parameter keyType: The key type to use for the container.
    /// - parameter key: The key to encode the container for.
    /// - returns: A new keyed encoding container.
        mutating func nestedContainer<NestedKey : CodingKey>(keyedBy keyType: NestedKey.Type, forKey key: Key) -> KeyedEncodingContainer<NestedKey>

    /// Stores an unkeyed encoding container for the given key and returns it.
    ///
    /// - parameter key: The key to encode the container for.
    /// - returns: A new unkeyed encoding container.
    mutating func nestedUnkeyedContainer(forKey key: Key) -> UnkeyedEncodingContainer
}

public protocol KeyedDecodingContainerProtocol {
    /// Returns the data stored for the given key as represented in a container keyed by the given key type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - parameter key: The key that the nested container is associated with.
    /// - returns: A keyed decoding container view into `self`.
```

```
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
    func nestedContainer<NestedKey : CodingKey>(keyedBy type: NestedKey.Type, forKey key: Key) throws -> KeyedDecodingContainer<NestedKey>

  /// Returns the data stored for the given key as represented in an unkeyed container.
  ///
  /// - parameter key: The key that the nested container is associated with.
  /// - returns: An unkeyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
  func nestedUnkeyedContainer(forKey key: Key) throws -> UnkeyedDecodingContainer
}
```

This can be common when coding against specific external data representations:

```
// User type for interfacing with a specific JSON API. JSON API expects encoding as {"id": ..., "properties": {"name": ..., "timestamp": ...}}. Swift type differs from encoded type, and encoding needs to match a spec:
struct Record : Codable {
  // We care only about these values from the JSON payload
  let id: Int
  let name: String
  let timestamp: Double

  // ...

  private enum Keys : CodingKey {
    case id
    case properties
  }

  private enum PropertiesKeys : CodingKey {
    case name
    case timestamp
  }

  public func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: Keys.self, type: .dictionary)
    try container.encode(id, forKey: .id)

    // Set a dictionary for the "properties" key
        let nested = container.nestedContainer(keyedBy: PropertiesKeys.self, forKey: .properties)
    try nested.encode(name, forKey: .name)
    try nested.encode(timestamp, forKey: .timestamp)
  }
```

```
    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: Keys.self)
        id = try container.decode(Int.self, forKey: .id)

            let nested = try container.nestedContainer(keyedBy: PropertiesKeys.self, forKey: .properties)
        name = try nested.decode(String.self, forKey: .name)
        timestamp = try nested.decode(Double.self, forKey: .timestamp)
    }
}
```

Unkeyed containers allow for the same types of nesting:

```
// Continuing from before
public protocol UnkeyedEncodingContainer {
    /// Encodes a nested container keyed by the given type and returns it.
    ///
    /// - parameter keyType: The key type to use for the container.
    /// - returns: A new keyed encoding container.
        mutating func nestedContainer<NestedKey : CodingKey>(keyedBy keyType: NestedKey.Type) -> KeyedEncodingContainer<NestedKey>

    /// Encodes an unkeyed encoding container and returns it.
    ///
    /// - returns: A new unkeyed encoding container.
    mutating func nestedUnkeyedContainer() -> UnkeyedEncodingContainer
}

public protocol UnkeyedDecodingContainer {
    /// Decodes a nested container keyed by the given type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - returns: A keyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
        mutating func nestedContainer<NestedKey : CodingKey>(keyedBy type: NestedKey.Type) throws -> KeyedDecodingContainer<NestedKey>

    /// Decodes an unkeyed nested container.
    ///
    /// - returns: An unkeyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    mutating func nestedUnkeyedContainer() throws -> UnkeyedDecodingContainer
}
```

Dynamic Context-Based Behavior

In some cases, types may need context in order to decide on their external representation. Some types may choose a different representation based on the encoding format that they are being read from or written to, and others based on other runtime contextual information. To facilitate this, Encoders and Decoders expose user-supplied context for consumption:

```
/// Represents a user-defined key for providing context for encoding and decoding.
public struct CodingUserInfoKey : RawRepresentable, Hashable {
    typealias RawValue = String
    let rawValue: String
    init?(rawValue: String)
}

// Continuing from before:
public protocol Encoder {
    /// Any contextual information set by the user for encoding.
    var userInfo: [CodingUserInfoKey : Any] { get }
}

public protocol Decoder {
    /// Any contextual information set by the user for decoding.
    var userInfo: [CodingUserInfoKey : Any] { get }
}
```

Consuming types may then support setting contextual information to inform their encoding and decoding:

```
public struct Person : Encodable {
                                public static let codingUserInfoKey =
CodingUserInfoKey("com.foocorp.person.codingUserInfoKey")

    public struct UserInfo {
        let shouldEncodePrivateFields: Bool
        // ...
    }

    func encode(to encoder: Encoder) throws {
        if let context = encoder.userInfo[Person.codingUserInfoKey] as? Person.UserInfo {
            if context.shouldEncodePrivateFields {
                // Do something special.
            }
        }

        // Fall back to default.
    }
}

let encoder = ...
```

```
encoder.userInfo[Person.codingUserInfoKey] = Person.UserInfo(...)
let data = try encoder.encode(person)
```

Encoders and Decoders may choose to expose contextual information about their configuration as part of the context as well if necessary.

Inheritance

Inheritance in this system is supported much like it is with NSCoding — on encoding, objects which inherit from a type that is Encodable encode super using their encoder, and pass a decoder to super.init(from:) on decode if they inherit from a type that is Decodable. With the existing NSCoding API, this is most often done like so, by convention:

```
- (void)encodeWithCoder:(NSCoder *)encoder {
    [super encodeWithCoder:encoder];
    // ... encode properties
}

- (instancetype)initWithCoder:(NSCoder *)decoder {
    if ((self = [super initWithCoder:decoder])) {
        // ... decode properties
    }

    return self;
}
```

In practice, this approach means that the properties of self and the properties of super get encoded into the same container: if self encodes values for keys "a", "b", and "c", and super encodes "d", "e", and "f", the resulting object is encoded as {"a": ..., "b": ..., "c": ..., "d": ..., "e": ..., "f": ...}. This approach has two drawbacks:

Things which self encodes may overwrite super's (or vice versa, depending on when -[super encodeWithCoder:] is called

self and super may not encode into different container types (e.g. self in a sequential fashion, and super in a keyed fashion)

The second point is not an issue for NSKeyedArchiver, since all values encode with keys (sequentially coded elements get autogenerated keys). This proposed API, however, allows for self and super to explicitly request conflicting containers (.array and .dictionary, which may not be mixed, depending on the data format).

To remedy both of these points, we adopt a new convention for inheritance-based coding — encoding super as a sub-object of self:

```
public class MyCodable : SomethingCodable {
    public func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        // ... encode some properties

        // superEncoder() gives `super` a nested container to encode into (for
```

```
        // a predefined key).
        try super.encode(to: container.superEncoder())
    }

    public init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        // ... decode some properties

        // Allow `super` to decode from the nested container.
        try super.init(from: container.superDecoder())
    }
}
```

If a shared container is desired, it is still possible to call super.encode(to: encoder) and super.init(from: decoder), but we recommend the safer containerized option.

superEncoder() and superDecoder() are provided on containers to provide handles to nested containers for super to use.

```
// Continuing from before
public protocol KeyedEncodingContainerProtocol {
    /// Stores a new nested container for the default `super` key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
    /// Equivalent to calling `superEncoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder() -> Encoder

    /// Stores a new nested container for the given key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
    /// - parameter key: The key to encode `super` for.
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder(forKey key: Key) -> Encoder
}

public protocol KeyedDecodingContainerProtocol {
    /// Returns a `Decoder` instance for decoding `super` from the container associated with the default `super` key.
    ///
    /// Equivalent to calling `superDecoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
    /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the
```

```
default `super` key, or if the stored value is null.
    func superDecoder() throws -> Decoder

    /// Returns a `Decoder` instance for decoding `super` from the container associated with the given key.
    ///
    /// - parameter key: The key to decode `super` for.
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
    /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key, or if the stored value is null.
    func superDecoder(forKey key: Key) throws -> Decoder
}

public protocol UnkeyedEncodingContainer {
    /// Encodes a nested container and returns an `Encoder` instance for encoding `super` into that container.
    ///
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder() -> Encoder
}

public protocol UnkeyedDecodingContainer {
    /// Decodes a nested container and returns a `Decoder` instance for decoding `super` from that container.
    ///
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
    /// - throws: `CocoaError.coderValueNotFound` if the encountered encoded value is null, or of there are no more values to decode.
    mutating func superDecoder() throws -> Decoder
}
```

Primitive Codable Conformance

The encoding container types offer overloads for working with and processing the API's primitive types (String, Int, Double, etc.). However, for ease of implementation (both in this API and others), it can be helpful for these types to conform to Codable themselves. Thus, along with these overloads, we will offer Codable conformance on these types:

```
extension Bool : Codable {
    public init(from decoder: Decoder) throws {
        self = try decoder.singleValueContainer().decode(Bool.self)
    }

    public func encode(to encoder: Encoder) throws {
        try encoder.singleValueContainer().encode( self)
    }
}
```

// Repeat for others...
This conformance allows one to write functions which accept Codable types without needing specific overloads for the fifteen primitive types as well.

Since Swift's function overload rules prefer more specific functions over generic functions, the specific overloads are chosen where possible (e.g. encode("Hello, world!", forKey: .greeting) will choose encode(_: String, forKey: Key) over encode<T : Codable>(_: T, forKey: Key)).

Additional Extensions
Along with the primitive Codable conformance above, extensions on Codable RawRepresentable types whose RawValue is a primitive types will provide default implementations for encoding and decoding:

```
public extension RawRepresentable where RawValue == Bool, Self : Codable {
    public init(from decoder: Decoder) throws {
        let decoded = try decoder.singleValueContainer().decode(RawValue.self)
        guard let value = Self(rawValue: decoded) else {
            throw CocoaError.error(.coderReadCorrupt)
        }

        self = value
    }

    public func encode(to encoder: Encoder) throws {
        try encoder.singleValueContainer().encode(self.rawValue)
    }
}
```

// Repeat for others...
This allows for trivial Codable conformance of enum types (and manual RawRepresentable implementations) with primitive backing.

Source compatibility
This proposal is additive — existing code will not have to change due to this API addition. This implementation can be made available in both Swift 4 and the Swift 3 compatibility mode.

Effect on ABI stability
The addition of this API will not be an ABI-breaking change. However, this will add limitations for changes in future versions of Swift, as parts of the API will have to remain unchanged between versions of Swift (barring some additions, discussed below).

Effect on API resilience
Much like new API added to the standard library, once added, many changes to this API will be ABI- and source-breaking changes. In particular, changes which change the types or

names of methods or arguments, add required methods on protocols or classes, or remove supplied default implementations will break client behavior.

The following types may not have methods added to them without providing default implementations:

Encodable
Decodable
CodingKey
Encoder
KeyedEncodingContainerProtocol
KeyedEncodingContainer
UnkeyedEncodingContainer
SingleValueEncodingContainer
Decoder
KeyedDecodingContainerProtocol
KeyedDecodingContainer
UnkeyedEncodingContainer
SingleValueDecodingContainer
Various extensions to Swift primitive types (Bool, Int, Double, etc.) and to RawRepresentable types (where RawValue == Bool, == Int, == Double, etc.) may also not be removed.

In general, changes to the proposed types will be restricted as described in the library evolution document in the Swift repository.

Alternatives considered
The following are a few of the more notable approaches considered for the problem:

Leverage the existing NSCoding implementation by adding support for struct and enum types, either through NSCoding itself, or through a similar protocol.

Although technically feasible, this can feel like a "missed opportunity" for offering something better tuned for Swift. This approach would also not offer any additional integration with JSONSerialization and PropertyListSerialization, unless JSON and plist archivers were added to offer support.
The following type-erased, declarative approach:

```
// Similar hack to AnyHashable; these wrap values which have not yet been
// encoded, or not yet decoded.
struct AnyEncodable { ... }
struct AnyDecodable { ... }

protocol CodingPrimitive {}
protocol PrimitiveCodable { /* same as above */ }

protocol OrderedCodable {
```

```
    init(from: [AnyDecodable]) throws
    var encoded: [AnyEncodable] { get }
}

protocol KeyedCodable {
    init(from: [String: AnyDecodable]) throws
    var encoded: [String: AnyEncodable] { get }
}

// Same as above
protocol OrderedEncoder { ... }
protocol OrderedDecoder { ... }
protocol KeyedEncoder { ... }
protocol KeyedDecoder { ... }

// Sample:
struct Location: OrderedCodable {
    let latitude: Double
    let longitude: Double

    init(from array: [AnyDecodable]) throws {
        guard array.count == 2 else { /* throw */ }

        // These `.as()` calls perform the actual decoding, and fail by
        // throwing an error.
        let latitude = try array[0].as(Double.self)
        let longitude = try array[1].as(Double.self)
        try self.init(latitutde: latitude, longitude: longitude)
    }

    var encoded: [AnyEncodable] {
        // With compiler support, AnyEncodable() can be automatic.
        return [AnyEncodable(latitude), AnyEncodable(longitude)]
    }
}

struct Farm: KeyedCodable {
    let name: String
    let location: Location
    let animals: [Animal]

    init(from dictionary: [String: AnyDecodable]) throws {
        guard let name = dictionary["name"],
            let location = dictionary["location"],
            let animals = dictionary["animals"] else { /* throw */ }
```

```
        self.name = try name.as(String.self)
        self.location = try location.as(Location.self)
        self.animals = try animals.asArrayOf(Animal.self)
    }

    var encoded: [String: AnyEncodable] {
        // Similarly, AnyEncodable() should go away.
        return ["name": AnyEncodable(name),
                "location": AnyEncodable(location),
                "animals": AnyEncodable(animals)]
    }
}
```

Although the more declarative nature of this approach can be appealing, this suffers from the same problem that JSONSerialization currently does: as-casting. Getting an intermediate type-erased value requires casting to get a "real" value out. Doing this with an as?-cast requires compiler support to interject code to decode values of a given type out of their type-erased containers (similar to what happens today with AnyHashable). If the user requests a value of a different type than what is stored, however, the as?-cast will fail by returning nil — there is no meaningful way to report the failure. Getting the code to throw in cases like this requires methods on AnyDecodable (as shown above), but these can be confusing (when should you use .as() and when should you use as??).

Modifications can be made to improve this:

```
protocol OrderedCodable {
    // AnyDecodable can wrap anything, including [AnyDecodable]; unwrapping
    // these can be tedious, so we want to give default implementations
    // that do this.
    // Default implementations for these are given in terms of the
    // initializer below.
    init?(from: AnyDecodable?) throws
    init(from: AnyDecodable) throws

    init(from: [AnyDecodable]) throws
    var encoded: [AnyEncodable] { get }
}

protocol KeyedCodable {
    // AnyDecodable can wrap anything, including [String: AnyDecodable];
    // unwrapping these can be tedious, so we want to give default
    // implementations that do this.
    // Default implementations for these are given in terms of the
    // initializer below.
    init?(from: AnyDecodable?) throws
    init(from: AnyDecodable) throws
```

```
    init(from: [String: AnyDecodable]) throws
    var encoded: [String: AnyEncodable] { get }
}

// Sample:
struct Location: OrderedCodable {
    // ...
    init(from array: [AnyDecodable]) throws {
        guard array.count == 2 else { /* throw */ }
        let latitude = try Double(from: array[0])
        let longitude = try Double(from: array[1])
        try self.init(latitude: latitude, longitude: longitude)
    }
    // ...
}

struct Farm: KeyedCodable {
    // ...
    init(from dictionary: [String: AnyDecodable]) throws {
        guard let name = try String(from: dictionary["name"]),
            let Location = try Location(from: dictionary["location"])
            let animals = try [Animal](from: dictionary["animals"]) else {
            /* throw */
        }

        self.name = name
        self.location = location
        self.animals = animals
    }
    // ...
}
```

By providing the new initializer methods, we can perform type casting via initialization, rather than by explicit casts. This pushes the .as() calls into the Swift primitives (CodingPrimitives, Array, Dictionary), hiding them from end users. However, this has a different problem, namely that by offering the same type-erased initializers, OrderedCodable and KeyedCodable now conflict, and it is impossible to conform to both.

The declarative benefits here are not enough to outweigh the fact that this does not effectively remove the need to as?-cast.

The following approach, which relies on compiler code generation:

```
protocol Codable {
    /// `EncodedType` is an intermediate representation of `Self` -- it has
    /// the properties from `Self` that need to be archived and unarchived
    /// (and performs that archival work), but represents at type that is
```

```
    /// not yet domain-validated like `self` is.
    associatedtype EncodedType: CodingRepresentation
    init(from encoded: EncodedType)
    var encoded: EncodedType { get }
}

protocol CodingPrimitive {}
protocol CodingRepresentation {}
protocol PrimitiveCodingRepresentation: CodingRepresentation {
    /* Similar to PrimitiveCodable above */
}

protocol OrderedCodingRepresentation: CodingRepresentation {
    /* Similar to OrderedCodable above */
}

protocol KeyedCodingRepresentation: CodingRepresentation {
    /* Similar to KeyedCodable above */
}

// Sample:
struct Location : Codable {
    let latitude: Double
    let longitude: Double

    // ----------------------------------------------------------------
    // Ideally, the following could be generated by the compiler (in simple
    // cases; developers can choose to implement subsets of the following
    // code based on where they might need to perform customizations.
    init(from encoded: Encoded) throws {
        latitude = encoded.latitude
        longitude = encoded.longitude
    }

    var encoded: Encoded {
        return Encoded(self)
    }

    // Keyed coding is the default generated by the compiler; consumers who
    // want OrderedCodingRepresentation need to provide their own encoded
    // type.
    struct Encoded: OrderedCodingRepresentation {
        let latitude: String
        let longitude: String

        init(_ location: Location) {
```

```
            latitude = location.latitude
            longitude = location.longitude
        }

        init(from: KeyedDecoder) throws { ... }
        func encode(to: KeyedEncoder) { ... }
    }
    // ------------------------------------------------------------------
}
```

This approach separates encoding and decoding into constituent steps:

Converting self into a representation fit for encoding (EncodedType, particularly if EncodedType has different properties from Self)
Converting that representation into data (encode(into:))
Converting arbitrary bytes into validated types (EncodedType.init(from:))
Converting validated data and types into a domain-validated value (Self.init(from:)).
These steps can be generated by the compiler in simple cases, with gradations up to the developer providing implementations for all of these. With this approach, it would be possible to:

Have a type where all code generation is left to the compiler
Have a type where EncodedType is autogenerated, but the user implements init(from:) (allowing for custom domain validation on decode) or var encoded, or both
Have a type where the user supplies EncodedType, Self.init(from:), and var encoded, but the compiler generates EncodedType.init(from:) and EncodedType.encode(into:). This allows the user to control what properties EncodedType has (or control its conformance to one of the CodingRepresentation types) without having to perform the actual encode and decode calls
Have a type where the user supplies everything, giving them full control of encoding and decoding (for implementing archive versioning and other needs)
While cases 1 and 2 save on boilerplate, types which need to be customized have significantly more boilerplate to write by hand.

The following approach, which delineates between keyed encoding (with String keys) and ordered encoding (this is the approach proposed in v1 and v2 of this proposal):

```
protocol PrimitiveCodable {
    associatedtype Atom: CodingAtom
    var atomValue: Atom { get }
    init(atomValue value: Atom)
}

protocol OrderedCodable {
    init(from decoder: OrderedDecoder) throws
    func encode(into encoder: OrderedEncoder)
}
```

```
protocol KeyedCodable {
   init(from decoder: KeyedDecoder) throws
   func encode(into encoder: KeyedEncoder)
}

protocol OrderedEncoder {
   func encode<Value>(_ value: Value?) where Value: CodingAtom
   func encode<Value>(_ value: Value?) where Value: PrimitiveCodable
   func encode<Value>(_ value: Value?) where Value: OrderedCodable
   func encode<Value>(_ value: Value?) where Value: KeyedCodable
   func encode<Value>(_ value: Value?) where Value: OrderedCodable & KeyedCodable
}

protocol OrderedDecoder {
   var count: Int { get }
   func decode<Value>(_ type: Value.Type) throws -> Value? where Value: CodingAtom
   func decode<Value>(_ type: Value.Type) throws -> Value? where Value: PrimitiveCodable
   func decode<Value>(_ type: Value.Type) throws -> Value? where Value: OrderedCodable
   func decode<Value>(_ type: Value.Type) throws -> Value? where Value: KeyedCodable
    func decode<Value>(_ type: Value.Type) throws -> Value? where Value: OrderedCodable
& KeyedCodable
}

protocol KeyedEncoder {
   func encode<Value>(_ value: Value?, forKey key: String) where Value: CodingPrimitive
   func encode<Value>(_ value: Value?, forKey key: String) where Value: PrimitiveCodable
   func encode<Value>(_ value: Value?, forKey key: String) where Value: OrderedCodable
   func encode<Value>(_ value: Value?, forKey key: String) where Value: KeyedCodable
   func encode<Value>(_ value: Value?, forKey key: String) where Value: OrderedCodable &
KeyedCodable
}

protocol KeyedDecoder {
   var allKeys: [String] { get }
   func hasValue(forKey key: String) -> Bool
      func decode<Value>(_ type: Value.Type, forKey key: String) throws -> Value? where
Value: CodingPrimitive
      func decode<Value>(_ type: Value.Type, forKey key: String) throws -> Value? where
Value: PrimitiveCodable
      func decode<Value>(_ type: Value.Type, forKey key: String) throws -> Value? where
Value: OrderedCodable
      func decode<Value>(_ type: Value.Type, forKey key: String) throws -> Value? where
Value: KeyedCodable
      func decode<Value>(_ type: Value.Type, forKey key: String) throws -> Value? where
Value: OrderedCodable & KeyedCodable
```

```
}
```
Although this semantically separates between different types of encoding, the multiple protocols can be confusing, and it is not immediately apparent which to adopt and use. This also specifically calls out a difference between string-keyed and non-keyed coding, which is unnecessary.

A closure-based version of the current approach which scopes keyed encoders/decoders to call sites via closures:

```
protocol Encoder {
   func encode(as value: Bool) throws
   // ...

     func with<Key>(keys type: Key.Type, _ block: (KeyedEncoder<Key>) throws -> Void)
rethrows
   // ...
}

internal struct Record : Codable {
   let id: Int
   let name: String
   let timestamp: Double

   // ...

   public func encode(into encoder: Encoder) throws {
      try encoder.with(keys: Keys.self) { keyedEncode in
         try keyedEncode.encode(id, forKey: .id)

          try keyedEncode.encode(.dictionary, forKey: .properties, keys: PropertiesKeys.self) {
properties in
            try properties.encode(name, forKey: .name)
            try properties.encode(timestamp, forKey: .timestamp)
         }
      }
   }
}
```
However, this cannot currently be applied to decoding:

```
public init(from decoder: Decoder) throws {
   // This closure implicitly references self. Since Swift has no
   // guarantees that this closure will get called exactly once, self must
   // be fully initialized before this call.
   //
   // This would require all instance variables to be vars with default
   // values.
```

```
    try decoder.with(keys: Keys.self) { keyedDecoder in
        id = try keyedDecoder.decode(Int.self, forKey: .id)
        // ...
    }
}
```

Although it is not currently possible to initialize self within a closure in Swift, this may be added in the future as annotations make these guarantees possible.

A previous approach similar to the current approach with single value encode calls available directly on Encoder, and a KeyedEncoder type instead of KeyedEncodingContainer:

```
public protocol Encoder {
    func keyed<Key : CodingKey>(by: Key.Type) throws -> KeyedEncoder<Key>

    func encode(as: Bool) throws
    func encode(as: Int) throws
    func encode(as: Int8) throws
    func encode(as: Int16) throws
    func encode(as: Int32) throws
    // ...
}

public class KeyedEncoder<Key : CodingKey> {
    // Identical to KeyedEncodingContainer
}
```

Appendix

JSONSerialization Friction and Third-Party Solutions (Motivation)

The following example usage of JSONSerialization is taken from the README of SwiftyJSON, a third-party library that many developers use to interface with JSON models:

```
if let statusesArray = try? JSONSerialization.jsonObject(with: data, options: .allowFragments)
as? [[String: Any]],
    let user = statusesArray[0]["user"] as? [String: Any],
    let username = user["name"] as? String {
    // Finally we got the username
}
```

SwiftyJSON attempts to elide the verbosity of casting by offering the following solution instead:

```
let json = JSON(data: dataFromNetworking)
if let userName = json[0]["user"]["name"].string {
  // Now you got your value
}
```

This friction is not necessarily a design flaw in the API, simply a truth of interfacing between JavaScript and JSON's generally untyped, unstructured contents, and Swift's strict typing. Some libraries, like SwiftyJSON, do this at the cost of type safety; others, like ObjectMapper

and Argo below, maintain type safety by offering archival functionality for JSON types:

```
// Taken from https://github.com/Hearst-DD/ObjectMapper
class User: Mappable {
    var username: String?
    var age: Int?
    var weight: Double!
    var array: [AnyObject]?
    var dictionary: [String : AnyObject] = [:]
    var bestFriend: User?                    // Nested User object
    var friends: [User]?                     // Array of Users
    var birthday: NSDate?

    required init?(map: Map) {

    }

    // Mappable
    func mapping(map: Map) {
        username    <- map["username"]
        age         <- map["age"]
        weight      <- map["weight"]
        array       <- map["arr"]
        dictionary  <- map["dict"]
        bestFriend  <- map["best_friend"]
        friends     <- map["friends"]
        birthday    <- (map["birthday"], DateTransform())
    }
}

struct Temperature: Mappable {
    var celsius: Double?
    var fahrenheit: Double?

    init?(map: Map) {

    }

    mutating func mapping(map: Map) {
        celsius     <- map["celsius"]
        fahrenheit  <- map["fahrenheit"]
    }
}
```

or the more functional

```
// Taken from https://github.com/thoughtbot/Argo
```

```
struct User {
  let id: Int
  let name: String
  let email: String?
  let role: Role
  let companyName: String
  let friends: [User]
}

extension User: Decodable {
  static func decode(j: JSON) -> Decoded<User> {
    return curry(User.init)
      <^> j <| "id"
      <*> j <| "name"
      <*> j <|? "email" // Use ? for parsing optional values
      <*> j <| "role" // Custom types that also conform to Decodable just work
      <*> j <| ["company", "name"] // Parse nested objects
      <*> j <|| "friends" // parse arrays of objects
  }
}

// Wherever you receive JSON data:
let json: Any? = try? NSJSONSerialization.JSONObjectWithData(data, options: [])
if let j: Any = json {
  let user: User? = decode(j)
}
```

There are tradeoffs made here as well. ObjectMapper requires that all of your properties be optional, while Argo relies on a vast collection of custom operators and custom curried initializer functions to do its work. (While not shown in the snippet above, User.init code in reality is effectively implemented as User.init(id)(name)(email)(role)(companyName)(friends).)

We would like to provide a solution that skirts neither type safety, nor ease-of-use and -implementation.

Unabridged API

```
/// Conformance to `Encodable` indicates that a type can encode itself to an external representation.
public protocol Encodable {
    /// Encodes `self` into the given encoder.
    ///
     /// If `self` fails to encode anything, `encoder` will encode an empty keyed container in its place.
    ///
    /// - parameter encoder: The encoder to write data to.
    /// - throws: An error if any values are invalid for `encoder`'s format.
```

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

/// Conformance to `Decodable` indicates that a type can decode itself from an external representation.
public protocol Decodable {
    /// Initializes `self` by decoding from `decoder`.
    ///
    /// - parameter decoder: The decoder to read data from.
    /// - throws: An error if reading from the decoder fails, or if read data is corrupted or otherwise invalid.
    init(from decoder: Decoder) throws
}

/// Conformance to `Codable` indicates that a type can convert itself into and out of an external representation.
public typealias Codable = Encodable & Decodable

/// Conformance to `CodingKey` indicates that a type can be used as a key for encoding and decoding.
public protocol CodingKey {
    /// The string to use in a named collection (e.g. a string-keyed dictionary).
    var stringValue: String { get }

    /// Initializes `self` from a string.
    ///
    /// - parameter stringValue: The string value of the desired key.
    /// - returns: An instance of `Self` from the given string, or `nil` if the given string does not correspond to any instance of `Self`.
    init?(stringValue: String)

    /// The int to use in an indexed collection (e.g. an int-keyed dictionary).
    var intValue: Int? { get }

    /// Initializes `self` from an integer.
    ///
    /// - parameter intValue: The integer value of the desired key.
    /// - returns: An instance of `Self` from the given integer, or `nil` if the given integer does not correspond to any instance of `Self`.
    init?(intValue: Int)
}

/// An `Encoder` is a type which can encode values into a native format for external representation.
public protocol Encoder {
    /// The path of coding keys taken to get to this point in encoding.
```

```
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Any contextual information set by the user for encoding.
    var userInfo: [CodingUserInfoKey : Any] { get }

     /// Returns an encoding container appropriate for holding multiple values keyed by the given key type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - returns: A new keyed encoding container.
    /// - precondition: May not be called after a prior `self.unkeyedContainer()` call.
     /// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.
                func container<Key : CodingKey>(keyedBy type: Key.Type) -> KeyedEncodingContainer<Key>

    /// Returns an encoding container appropriate for holding multiple unkeyed values.
    ///
    /// - returns: A new empty unkeyed container.
    /// - precondition: May not be called after a prior `self.container(keyedBy:)` call.
     /// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.
    func unkeyedContainer() -> UnkeyedEncodingContainer

    /// Returns an encoding container appropriate for holding a single primitive value.
    ///
    /// - returns: A new empty single value container.
    /// - precondition: May not be called after a prior `self.container(keyedBy:)` call.
    /// - precondition: May not be called after a prior `self.unkeyedContainer()` call.
     /// - precondition: May not be called after a value has been encoded through a previous `self.singleValueContainer()` call.
    func singleValueContainer() -> SingleValueEncodingContainer
}

/// A `Decoder` is a type which can decode values from a native format into in-memory representations.
public protocol Decoder {
    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Any contextual information set by the user for decoding.
    var userInfo: [CodingUserInfoKey : Any] { get }

     /// Returns the data stored in `self` as represented in a container keyed by the given key
```

```
type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - returns: A keyed decoding container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
            func container<Key : CodingKey>(keyedBy type: Key.Type) throws -> KeyedDecodingContainer<Key>

     /// Returns the data stored in `self` as represented in a container appropriate for holding values with no keys.
    ///
    /// - returns: An unkeyed container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    func unkeyedContainer() throws -> UnkeyedDecodingContainer

     /// Returns the data stored in `self` as represented in a container appropriate for holding a single primitive value.
    ///
    /// - returns: A single value container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a single value container.
    func singleValueContainer() throws -> SingleValueDecodingContainer
}

/// Conformance to `KeyedEncodingContainerProtocol` indicates that a type provides a view into an `Encoder`'s storage and is used to hold the encoded properties of an `Encodable` type in a keyed manner.
///
/// Encoders should provide types conforming to `KeyedEncodingContainerProtocol` for their format.
public protocol KeyedEncodingContainerProtocol {
    associatedtype Key : CodingKey

    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
     /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode<T : Encodable>(_ value: T?, forKey key: Key) throws
```

```
    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode(_ value: Bool?,   forKey key: Key) throws
    mutating func encode(_ value: Int?,    forKey key: Key) throws
    mutating func encode(_ value: Int8?,   forKey key: Key) throws
    mutating func encode(_ value: Int16?,  forKey key: Key) throws
    mutating func encode(_ value: Int32?,  forKey key: Key) throws
    mutating func encode(_ value: Int64?,  forKey key: Key) throws
    mutating func encode(_ value: UInt?,   forKey key: Key) throws
    mutating func encode(_ value: UInt8?,  forKey key: Key) throws
    mutating func encode(_ value: UInt16?, forKey key: Key) throws
    mutating func encode(_ value: UInt32?, forKey key: Key) throws
    mutating func encode(_ value: UInt64?, forKey key: Key) throws
    mutating func encode(_ value: Float?,  forKey key: Key) throws
    mutating func encode(_ value: Double?, forKey key: Key) throws
    mutating func encode(_ value: String?, forKey key: Key) throws

    /// Encodes the given object weakly for the given key.
    ///
    /// For `Encoder`s that implement this functionality, this will only encode the given object and associate it with the given key if it is encoded unconditionally elsewhere in the payload (either previously or in the future).
    ///
    /// For formats which don't support this feature, the default implementation encodes the given object unconditionally.
    ///
    /// - parameter object: The object to encode.
    /// - parameter key: The key to associate the object with.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encodeWeak<T : AnyObject & Encodable>(_ object: T?, forKey key: Key) throws

    /// Stores a keyed encoding container for the given key and returns it.
    ///
    /// - parameter keyType: The key type to use for the container.
    /// - parameter key: The key to encode the container for.
    /// - returns: A new keyed encoding container.
    mutating func nestedContainer<NestedKey : CodingKey>(keyedBy keyType: NestedKey.Type, forKey key: Key) -> KeyedEncodingContainer<NestedKey>
```

```
    /// Stores an unkeyed encoding container for the given key and returns it.
    ///
    /// - parameter key: The key to encode the container for.
    /// - returns: A new unkeyed encoding container.
    mutating func nestedUnkeyedContainer(forKey key: Key) -> UnkeyedEncodingContainer

     /// Stores a new nested container for the default `super` key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
     /// Equivalent to calling `superEncoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder() -> Encoder

     /// Stores a new nested container for the given key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
    /// - parameter key: The key to encode `super` for.
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder(forKey key: Key) -> Encoder
}

/// `KeyedEncodingContainer` is a type-erased box for `KeyedEncodingContainerProtocol` types, similar to `AnyCollection` and `AnyHashable`. This is the type which consumers of the API interact with directly.
public struct KeyedEncodingContainer<K : CodingKey> : KeyedEncodingContainerProtocol {
    associatedtype Key = K

    /// Initializes `self` with the given container.
    ///
    /// - parameter container: The container to hold.
        init<Container : KeyedEncodingContainerProtocol>(_ container: Container) where Container.Key == Key

    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode<T : Encodable>(_ value: T?, forKey key: Key) throws
```

```
    /// Encodes the given value for the given key.
    ///
    /// - parameter value: The value to encode.
    /// - parameter key: The key to associate the value with.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode(_ value: Bool?,   forKey key: Key) throws
    mutating func encode(_ value: Int?,    forKey key: Key) throws
    mutating func encode(_ value: Int8?,   forKey key: Key) throws
    mutating func encode(_ value: Int16?,  forKey key: Key) throws
    mutating func encode(_ value: Int32?,  forKey key: Key) throws
    mutating func encode(_ value: Int64?,  forKey key: Key) throws
    mutating func encode(_ value: UInt?,   forKey key: Key) throws
    mutating func encode(_ value: UInt8?,  forKey key: Key) throws
    mutating func encode(_ value: UInt16?, forKey key: Key) throws
    mutating func encode(_ value: UInt32?, forKey key: Key) throws
    mutating func encode(_ value: UInt64?, forKey key: Key) throws
    mutating func encode(_ value: Float?,  forKey key: Key) throws
    mutating func encode(_ value: Double?, forKey key: Key) throws
    mutating func encode(_ value: String?, forKey key: Key) throws

    /// Encodes the given object weakly for the given key.
    ///
    /// For `Encoder`s that implement this functionality, this will only encode the given object and associate it with the given key if it is encoded unconditionally elsewhere in the payload (either previously or in the future).
    ///
    /// For formats which don't support this feature, the default implementation encodes the given object unconditionally.
    ///
    /// - parameter object: The object to encode.
    /// - parameter key: The key to associate the object with.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encodeWeak<T : AnyObject & Encodable>(_ object: T?, forKey key: Key) throws

    /// Stores a keyed encoding container for the given key and returns it.
    ///
    /// - parameter keyType: The key type to use for the container.
    /// - parameter key: The key to encode the container for.
    /// - returns: A new keyed encoding container.
    mutating func nestedContainer<NestedKey : CodingKey>(keyedBy keyType: NestedKey.Type, forKey key: Key) -> KeyedEncodingContainer<NestedKey>
```

```
    /// Stores an unkeyed encoding container for the given key and returns it.
    ///
    /// - parameter key: The key to encode the container for.
    /// - returns: A new unkeyed encoding container.
    mutating func nestedUnkeyedContainer(forKey key: Key) -> UnkeyedEncodingContainer

     /// Stores a new nested container for the default `super` key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
     /// Equivalent to calling `superEncoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder() -> Encoder

     /// Stores a new nested container for the given key and returns a new `Encoder` instance for encoding `super` into that container.
    ///
    /// - parameter key: The key to encode `super` for.
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder(forKey key: Key) -> Encoder
}

/// Conformance to `KeyedDecodingContainerProtocol` indicates that a type provides a view into a `Decoder`'s storage and is used to hold the encoded properties of a `Decodable` type in a keyed manner.
///
/// Decoders should provide types conforming to `KeyedDecodingContainerProtocol` for their format.
public protocol KeyedDecodingContainerProtocol {
    associatedtype Key : CodingKey

    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// All the keys the `Decoder` has for this container.
    ///
     /// Different keyed containers from the same `Decoder` may return different keys here; it is possible to encode with multiple key types which are not convertible to one another. This should report all keys present which are convertible to the requested type.
    var allKeys: [Key] { get }

    /// Returns whether the `Decoder` contains a value associated with the given key.
    ///
     /// The value associated with the given key may be a null value as appropriate for the data
```

```
format.
    ///
    /// - parameter key: The key to search for.
    /// - returns: Whether the `Decoder` has an entry for the given key.
    func contains(_ key: Key) -> Bool

    /// Decodes a value of the given type for the given key.
    ///
     /// A default implementation is given for these types which calls into the `decodeIfPresent` implementations below.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
     /// - returns: A value of the requested type, if present for the given key and convertible to the requested type.
      /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
       /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key or if the value is null.
    func decode(_ type: Bool.Type,   forKey key: Key) throws -> Bool
    func decode(_ type: Int.Type,    forKey key: Key) throws -> Int
    func decode(_ type: Int8.Type,   forKey key: Key) throws -> Int8
    func decode(_ type: Int16.Type,  forKey key: Key) throws -> Int16
    func decode(_ type: Int32.Type,  forKey key: Key) throws -> Int32
    func decode(_ type: Int64.Type,  forKey key: Key) throws -> Int64
    func decode(_ type: UInt.Type,   forKey key: Key) throws -> UInt
    func decode(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8
    func decode(_ type: UInt16.Type, forKey key: Key) throws -> UInt16
    func decode(_ type: UInt32.Type, forKey key: Key) throws -> UInt32
    func decode(_ type: UInt64.Type, forKey key: Key) throws -> UInt64
    func decode(_ type: Float.Type,  forKey key: Key) throws -> Float
    func decode(_ type: Double.Type, forKey key: Key) throws -> Double
    func decode(_ type: String.Type, forKey key: Key) throws -> String
    func decode<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T

    /// Decodes a value of the given type for the given key, if present.
    ///
    /// This method returns `nil` if the container does not have a value associated with `key`, or if the value is null. The difference between these states can be distinguished with a `contains(_:)` call.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
     /// - returns: A decoded value of the requested type, or `nil` if the `Decoder` does not have an entry associated with the given key, or if the value is a null value.
      /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not
```

convertible to the requested type.
```
    func decodeIfPresent(_ type: Bool.Type,   forKey key: Key) throws -> Bool?
    func decodeIfPresent(_ type: Int.Type,    forKey key: Key) throws -> Int?
    func decodeIfPresent(_ type: Int8.Type,   forKey key: Key) throws -> Int8?
    func decodeIfPresent(_ type: Int16.Type,  forKey key: Key) throws -> Int16?
    func decodeIfPresent(_ type: Int32.Type,  forKey key: Key) throws -> Int32?
    func decodeIfPresent(_ type: Int64.Type,  forKey key: Key) throws -> Int64?
    func decodeIfPresent(_ type: UInt.Type,   forKey key: Key) throws -> UInt?
    func decodeIfPresent(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8?
    func decodeIfPresent(_ type: UInt16.Type, forKey key: Key) throws -> UInt16?
    func decodeIfPresent(_ type: UInt32.Type, forKey key: Key) throws -> UInt32?
    func decodeIfPresent(_ type: UInt64.Type, forKey key: Key) throws -> UInt64?
    func decodeIfPresent(_ type: Float.Type,  forKey key: Key) throws -> Float?
    func decodeIfPresent(_ type: Double.Type, forKey key: Key) throws -> Double?
    func decodeIfPresent(_ type: String.Type, forKey key: Key) throws -> String?
    func decodeIfPresent<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T?

     /// Returns the data stored for the given key as represented in a container keyed by the given key type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - parameter key: The key that the nested container is associated with.
    /// - returns: A keyed decoding container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
     func nestedContainer<NestedKey : CodingKey>(keyedBy type: NestedKey.Type, forKey key: Key) throws -> KeyedDecodingContainer<NestedKey>

    /// Returns the data stored for the given key as represented in an unkeyed container.
    ///
    /// - parameter key: The key that the nested container is associated with.
    /// - returns: An unkeyed decoding container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    func nestedUnkeyedContainer(forKey key: Key) throws -> UnkeyedDecodingContainer

     /// Returns a `Decoder` instance for decoding `super` from the container associated with the default `super` key.
    ///
     /// Equivalent to calling `superDecoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
      /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the default `super` key, or if the stored value is null.
    func superDecoder() throws -> Decoder
```

```
    /// Returns a `Decoder` instance for decoding `super` from the container associated with the given key.
    ///
    /// - parameter key: The key to decode `super` for.
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
    /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key, or if the stored value is null.
    func superDecoder(forKey key: Key) throws -> Decoder
}

/// `KeyedDecodingContainer` is a type-erased box for `KeyedDecodingContainerProtocol` types, similar to `AnyCollection` and `AnyHashable`. This is the type which consumers of the API interact with directly.
public struct KeyedDecodingContainer<K : CodingKey> : KeyedDecodingContainerProtocol {
    associatedtype Key = K

    /// Initializes `self` with the given container.
    ///
    /// - parameter container: The container to hold.
        init<Container : KeyedDecodingContainerProtocol>(_ container: Container) where Container.Key == Key

    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// All the keys the `Decoder` has for this container.
    ///
    /// Different keyed containers from the same `Decoder` may return different keys here; it is possible to encode with multiple key types which are not convertible to one another. This should report all keys present which are convertible to the requested type.
    var allKeys: [Key] { get }

    /// Returns whether the `Decoder` contains a value associated with the given key.
    ///
    /// The value associated with the given key may be a null value as appropriate for the data format.
    ///
    /// - parameter key: The key to search for.
    /// - returns: Whether the `Decoder` has an entry for the given key.
    func contains(_ key: Key) -> Bool

    /// Decodes a value of the given type for the given key.
    ///
```

```
    /// A default implementation is given for these types which calls into the `decodeIfPresent` implementations below.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
    /// - returns: A value of the requested type, if present for the given key and convertible to the requested type.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key or if the value is null.
    func decode(_ type: Bool.Type,   forKey key: Key) throws -> Bool
    func decode(_ type: Int.Type,    forKey key: Key) throws -> Int
    func decode(_ type: Int8.Type,   forKey key: Key) throws -> Int8
    func decode(_ type: Int16.Type,  forKey key: Key) throws -> Int16
    func decode(_ type: Int32.Type,  forKey key: Key) throws -> Int32
    func decode(_ type: Int64.Type,  forKey key: Key) throws -> Int64
    func decode(_ type: UInt.Type,   forKey key: Key) throws -> UInt
    func decode(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8
    func decode(_ type: UInt16.Type, forKey key: Key) throws -> UInt16
    func decode(_ type: UInt32.Type, forKey key: Key) throws -> UInt32
    func decode(_ type: UInt64.Type, forKey key: Key) throws -> UInt64
    func decode(_ type: Float.Type,  forKey key: Key) throws -> Float
    func decode(_ type: Double.Type, forKey key: Key) throws -> Double
    func decode(_ type: String.Type, forKey key: Key) throws -> String
    func decode<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T

    /// Decodes a value of the given type for the given key, if present.
    ///
    /// This method returns `nil` if the container does not have a value associated with `key`, or if the value is null. The difference between these states can be distinguished with a `contains(_:)` call.
    ///
    /// - parameter type: The type of value to decode.
    /// - parameter key: The key that the decoded value is associated with.
    /// - returns: A decoded value of the requested type, or `nil` if the `Decoder` does not have an entry associated with the given key, or if the value is a null value.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    func decodeIfPresent(_ type: Bool.Type,   forKey key: Key) throws -> Bool?
    func decodeIfPresent(_ type: Int.Type,    forKey key: Key) throws -> Int?
    func decodeIfPresent(_ type: Int8.Type,   forKey key: Key) throws -> Int8?
    func decodeIfPresent(_ type: Int16.Type,  forKey key: Key) throws -> Int16?
    func decodeIfPresent(_ type: Int32.Type,  forKey key: Key) throws -> Int32?
    func decodeIfPresent(_ type: Int64.Type,  forKey key: Key) throws -> Int64?
    func decodeIfPresent(_ type: UInt.Type,   forKey key: Key) throws -> UInt?
```

```
    func decodeIfPresent(_ type: UInt8.Type,  forKey key: Key) throws -> UInt8?
    func decodeIfPresent(_ type: UInt16.Type, forKey key: Key) throws -> UInt16?
    func decodeIfPresent(_ type: UInt32.Type, forKey key: Key) throws -> UInt32?
    func decodeIfPresent(_ type: UInt64.Type, forKey key: Key) throws -> UInt64?
    func decodeIfPresent(_ type: Float.Type,  forKey key: Key) throws -> Float?
    func decodeIfPresent(_ type: Double.Type, forKey key: Key) throws -> Double?
    func decodeIfPresent(_ type: String.Type, forKey key: Key) throws -> String?
    func decodeIfPresent<T : Decodable>(_ type: T.Type, forKey key: Key) throws -> T?

     /// Returns the data stored for the given key as represented in a container keyed by the given key type.
    ///
    /// - parameter type: The key type to use for the container.
    /// - parameter key: The key that the nested container is associated with.
    /// - returns: A keyed decoding container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
     func nestedContainer<NestedKey : CodingKey>(keyedBy type: NestedKey.Type, forKey key: Key) throws -> KeyedDecodingContainer<NestedKey>

    /// Returns the data stored for the given key as represented in an unkeyed container.
    ///
    /// - parameter key: The key that the nested container is associated with.
    /// - returns: An unkeyed decoding container view into `self`.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    func nestedUnkeyedContainer(forKey key: Key) throws -> UnkeyedDecodingContainer

     /// Returns a `Decoder` instance for decoding `super` from the container associated with the default `super` key.
    ///
     /// Equivalent to calling `superDecoder(forKey:)` with `Key(stringValue: "super", intValue: 0)`.
    ///
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
     /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the default `super` key, or if the stored value is null.
    func superDecoder() throws -> Decoder

     /// Returns a `Decoder` instance for decoding `super` from the container associated with the given key.
    ///
    /// - parameter key: The key to decode `super` for.
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
     /// - throws: `CocoaError.coderValueNotFound` if `self` does not have an entry for the given key, or if the stored value is null.
```

```
    func superDecoder(forKey key: Key) throws -> Decoder
}

/// Conformance to `UnkeyedEncodingContainer` indicates that a type provides a view into an `Encoder`'s storage and is used to hold the encoded properties of an `Encodable` type sequentially, without keys.
///
/// Encoders should provide types conforming to `UnkeyedEncodingContainer` for their format.
public protocol UnkeyedEncodingContainer {
    /// The path of coding keys taken to get to this point in encoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Encodes the given value.
    ///
    /// - parameter value: The value to encode.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode<T : Encodable>(_ value: T?) throws

    /// Encodes the given value.
    ///
    /// - parameter value: The value to encode.
      /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encode(_ value: Bool?) throws
    mutating func encode(_ value: Int?) throws
    mutating func encode(_ value: Int8?) throws
    mutating func encode(_ value: Int16?) throws
    mutating func encode(_ value: Int32?) throws
    mutating func encode(_ value: Int64?) throws
    mutating func encode(_ value: UInt?) throws
    mutating func encode(_ value: UInt8?) throws
    mutating func encode(_ value: UInt16?) throws
    mutating func encode(_ value: UInt32?) throws
    mutating func encode(_ value: UInt64?) throws
    mutating func encode(_ value: Float?) throws
    mutating func encode(_ value: Double?) throws
    mutating func encode(_ value: String?) throws

    /// Encodes the given object weakly.
    ///
     /// For `Encoder`s that implement this functionality, this will only encode the given object if it is encoded unconditionally elsewhere in the payload (either previously or in the future).
    ///
```

```
    /// For formats which don't support this feature, the default implementation encodes the given object unconditionally.
    ///
    /// - parameter object: The object to encode.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    mutating func encodeWeak<T : AnyObject & Encodable>(_ object: T?) throws

    /// Encodes the elements of the given sequence.
    ///
    /// A default implementation of these is given in an extension.
    ///
    /// - parameter sequence: The sequences whose contents to encode.
    /// - throws: An error if any of the contained values throws an error.
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Bool
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int8
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int16
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int32
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Int64
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == UInt
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == UInt8
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == UInt16
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == UInt32
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == UInt64
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Float
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == Double
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element == String
    mutating func encode<Sequence : Swift.Sequence>(contentsOf sequence: Sequence) throws where Sequence.Iterator.Element : Encodable

    /// Encodes a nested container keyed by the given type and returns it.
```

```
    ///
    /// - parameter keyType: The key type to use for the container.
    /// - returns: A new keyed encoding container.
        mutating func nestedContainer<NestedKey : CodingKey>(keyedBy keyType: NestedKey.Type) -> KeyedEncodingContainer<NestedKey>

    /// Encodes an unkeyed encoding container and returns it.
    ///
    /// - returns: A new unkeyed encoding container.
    mutating func nestedUnkeyedContainer() -> UnkeyedEncodingContainer

     /// Encodes a nested container and returns an `Encoder` instance for encoding `super` into that container.
    ///
    /// - returns: A new `Encoder` to pass to `super.encode(to:)`.
    mutating func superEncoder() -> Encoder
}

/// Conformance to `UnkeyedDecodingContainer` indicates that a type provides a view into a `Decoder`'s storage and is used to hold the encoded properties of a `Decodable` type sequentially, without keys.
///
/// Decoders should provide types conforming to `UnkeyedDecodingContainer` for their format.
public protocol UnkeyedDecodingContainer {
    /// The path of coding keys taken to get to this point in decoding.
    /// A `nil` value indicates an unkeyed container.
    var codingPath: [CodingKey?] { get }

    /// Returns the number of elements (if known) contained within this container.
    var count: Int? { get }

    /// Returns whether there are no more elements left to be decoded in the container.
    var isAtEnd: Bool { get }

    /// Decodes a value of the given type.
    ///
     /// A default implementation is given for these types which calls into the `decodeIfPresent` implementations below.
    ///
    /// - parameter type: The type of value to decode.
     /// - returns: A value of the requested type, if present for the given key and convertible to the requested type.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
     /// - throws: `CocoaError.coderValueNotFound` if the encountered encoded value is null,
```

```
or of there are no more values to decode.
    mutating func decode(_ type: Bool.Type) throws -> Bool
    mutating func decode(_ type: Int.Type) throws -> Int
    mutating func decode(_ type: Int8.Type) throws -> Int8
    mutating func decode(_ type: Int16.Type) throws -> Int16
    mutating func decode(_ type: Int32.Type) throws -> Int32
    mutating func decode(_ type: Int64.Type) throws -> Int64
    mutating func decode(_ type: UInt.Type) throws -> UInt
    mutating func decode(_ type: UInt8.Type) throws -> UInt8
    mutating func decode(_ type: UInt16.Type) throws -> UInt16
    mutating func decode(_ type: UInt32.Type) throws -> UInt32
    mutating func decode(_ type: UInt64.Type) throws -> UInt64
    mutating func decode(_ type: Float.Type) throws -> Float
    mutating func decode(_ type: Double.Type) throws -> Double
    mutating func decode(_ type: String.Type) throws -> String
    mutating func decode<T : Decodable>(_ type: T.Type) throws -> T

    /// Decodes a value of the given type, if present.
    ///
     /// This method returns `nil` if the container has no elements left to decode, or if the value is null. The difference between these states can be distinguished by checking `isAtEnd`.
    ///
    /// - parameter type: The type of value to decode.
     /// - returns: A decoded value of the requested type, or `nil` if the value is a null value, or if there are no more elements to decode.
      /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value is not convertible to the requested type.
    mutating func decodeIfPresent(_ type: Bool.Type) throws -> Bool?
    mutating func decodeIfPresent(_ type: Int.Type) throws -> Int?
    mutating func decodeIfPresent(_ type: Int8.Type) throws -> Int8?
    mutating func decodeIfPresent(_ type: Int16.Type) throws -> Int16?
    mutating func decodeIfPresent(_ type: Int32.Type) throws -> Int32?
    mutating func decodeIfPresent(_ type: Int64.Type) throws -> Int64?
    mutating func decodeIfPresent(_ type: UInt.Type) throws -> UInt?
    mutating func decodeIfPresent(_ type: UInt8.Type) throws -> UInt8?
    mutating func decodeIfPresent(_ type: UInt16.Type) throws -> UInt16?
    mutating func decodeIfPresent(_ type: UInt32.Type) throws -> UInt32?
    mutating func decodeIfPresent(_ type: UInt64.Type) throws -> UInt64?
    mutating func decodeIfPresent(_ type: Float.Type) throws -> Float?
    mutating func decodeIfPresent(_ type: Double.Type) throws -> Double?
    mutating func decodeIfPresent(_ type: String.Type) throws -> String?
    mutating func decodeIfPresent<T : Decodable>(_ type: T.Type) throws -> T?

    /// Decodes a nested container keyed by the given type.
    ///
    /// - parameter type: The key type to use for the container.
```

```
    /// - returns: A keyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not a keyed container.
    mutating func nestedContainer<NestedKey : CodingKey>(keyedBy type: NestedKey.Type) throws -> KeyedDecodingContainer<NestedKey>

    /// Decodes an unkeyed nested container.
    ///
    /// - returns: An unkeyed decoding container view into `self`.
    /// - throws: `CocoaError.coderTypeMismatch` if the encountered stored value is not an unkeyed container.
    mutating func nestedUnkeyedContainer() throws -> UnkeyedDecodingContainer

    /// Decodes a nested container and returns a `Decoder` instance for decoding `super` from that container.
    ///
    /// - returns: A new `Decoder` to pass to `super.init(from:)`.
    /// - throws: `CocoaError.coderValueNotFound` if the encountered encoded value is null, or of there are no more values to decode.
    mutating func superDecoder() throws -> Decoder
}

/// A `SingleValueEncodingContainer` is a container which can support the storage and direct encoding of a single non-keyed value.
public protocol SingleValueEncodingContainer {
    /// Encodes a single value of the given type.
    ///
    /// - parameter value: The value to encode.
    /// - throws: `CocoaError.coderInvalidValue` if the given value is invalid in the current context for this format.
    /// - precondition: May not be called after a previous `self.encode(_:)` call.
    mutating func encode(_ value: Bool) throws
    mutating func encode(_ value: Int) throws
    mutating func encode(_ value: Int8) throws
    mutating func encode(_ value: Int16) throws
    mutating func encode(_ value: Int32) throws
    mutating func encode(_ value: Int64) throws
    mutating func encode(_ value: UInt) throws
    mutating func encode(_ value: UInt8) throws
    mutating func encode(_ value: UInt16) throws
    mutating func encode(_ value: UInt32) throws
    mutating func encode(_ value: UInt64) throws
    mutating func encode(_ value: Float) throws
    mutating func encode(_ value: Double) throws
    mutating func encode(_ value: String) throws
}
```

```
/// A `SingleValueDecodingContainer` is a container which can support the storage and direct decoding of a single non-keyed value.
public protocol SingleValueDecodingContainer {
    /// Decodes a single value of the given type.
    ///
    /// - parameter type: The type to decode as.
    /// - returns: A value of the requested type.
     /// - throws: `CocoaError.coderTypeMismatch` if the encountered encoded value cannot be converted to the requested type.
    func decode(_ type: Bool.Type) throws -> Bool
    func decode(_ type: Int.Type) throws -> Int
    func decode(_ type: Int8.Type) throws -> Int8
    func decode(_ type: Int16.Type) throws -> Int16
    func decode(_ type: Int32.Type) throws -> Int32
    func decode(_ type: Int64.Type) throws -> Int64
    func decode(_ type: UInt.Type) throws -> UInt
    func decode(_ type: UInt8.Type) throws -> UInt8
    func decode(_ type: UInt16.Type) throws -> UInt16
    func decode(_ type: UInt32.Type) throws -> UInt32
    func decode(_ type: UInt64.Type) throws -> UInt64
    func decode(_ type: Float.Type) throws -> Float
    func decode(_ type: Double.Type) throws -> Double
    func decode(_ type: String.Type) throws -> String
}

/// Represents a user-defined key for providing context for encoding and decoding.
public struct CodingUserInfoKey : RawRepresentable, Hashable {
    typealias RawValue = String
    let rawValue: String
    init?(rawValue: String)
    init(_ value: String)
}

// Repeat for all primitive types...
extension Bool : Codable {
    public init(from decoder: Decoder) throws {
        self = try decoder.singleValueContainer().decode(Bool.self)
    }

    public func encode(to encoder: Encoder) throws {
        try encoder.singleValueContainer().encode( self)
    }
}

// Repeat for all primitive types...
```

```
public extension RawRepresentable where RawValue == Bool, Self : Codable {
    public init(from decoder: Decoder) throws {
        let decoded = try decoder.singleValueContainer().decode(RawValue.self)
        guard let value = Self(rawValue: decoded) else {
            throw CocoaError.error(.coderReadCorrupt)
        }

        self = value
    }

    public func encode(to encoder: Encoder) throws {
        try encoder.singleValueContainer().encode(self.rawValue)
    }
}
```