Patterns in Swift's Standard Library

Nicholas Lindley

Motivation

Optional

Optional

```
let imagePath: String?
  = imageEnd != nil
  ? "https://example.com/v1/Images?imagePath=\(imageEnd!)"
  : nil
```

Optional

```
@frozen
public enum Optional<Wrapped>: ExpressibleByNilLiteral {
  case none
  case some(Wrapped)
}
```

Optional

Syntactic sugar causes cancer of the semicolon.
— Alan Perlis[1]

[1] A Brief, Incomplete, and Mostly Wrong History of Programming Languages, http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

Maybe

Haskell

```
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)
```

Maybe

Elm

```
type Maybe a
    = Just a
    | Nothing
```

Option

Rust

```
pub enum Option<T> {
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

Option

OCaml

```
type 'a t = 'a option = None | Some of 'a
```

Words

> ***Often times you'll see this presented as the* `Maybe` *monad (there it is). The reason I mention the "M" word is because a monad is also a functor, and a functor is a map between categories. This is a long way of saying optional types are mappable, like so:***
>
> ***— Me***

Monads

Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"[1]

[1] A Brief, Incomplete, and Mostly Wrong History of Programming Languages, http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

Functor

```
let doubles = [1, 2, 3].map { $0 * 2 }
```

Functor

```
@inlinable
public func map<U>(
  _ transform: (Wrapped) throws -> U
) rethrows -> U? {
  switch self {
  case .some(let y):
    return .some(try transform(y))
  case .none:
    return .none
  }
}
```

Refactor

```
let imagePath: String?
  = imageEnd != nil
  ? "https://example.com/v1/Images?imagePath=\(imageEnd!)"
  : nil
```

Refactor

```swift
let imagePath = imageEnd.map { "https://example.com/v1/Images?imagePath=\($0)" }
```

Collection

```swift
extension Collection {
  @inlinable
  public func map<T>(
    _ transform: (Element) throws -> T
  ) rethrows -> [T] {
    let n = self.count
    if n == 0 {
      return []
    }

    var result = ContiguousArray<T>()
    result.reserveCapacity(n)

    var i = self.startIndex

    for _ in 0..<n {
      result.append(try transform(self[i]))
      formIndex(after: &i)
    }

    _expectEnd(of: self, is: i)
    return Array(result)
  }
}
```

Sequence

```swift
extension Sequence {
  @inlinable
  public func map<T>(
    _ transform: (Element) throws -> T
  ) rethrows -> [T] {
    let initialCapacity = underestimatedCount
    var result = ContiguousArray<T>()
    result.reserveCapacity(initialCapacity)

    var iterator = self.makeIterator()

    for _ in 0..<initialCapacity {
      result.append(try transform(iterator.next()!))
    }

    while let element = iterator.next() {
      result.append(try transform(element))
    }
    return Array(result)
  }
}
```

Functor?

```
// Optional
func map<U>(_ transform: (T) -> U) ->  Optional<U>
// Collection
func map<U>(_ transform: (T) -> U) ->     Array<U>
// Sequence
func map<U>(_ transform: (T) -> U) ->     Array<U>
// Result
func map<U>(_ transform: (T) -> U) -> Result<U, V>
```

Functor?

Functor

Higher-Kinded Type

flatMap

JavaScript

```
Array.prototype.flatMap ( mapperFunction [ , thisArg ] ) // ES2019
_.flatMap(collection, [iteratee=_.identity]) // lodash
R.chain(fn, list) // Ramda
```

andThen/concatMap

Elm

```
andThen : Maybe a -> (a -> Maybe b) -> Maybe
concatMap : (a -> List b) -> List a -> List b b
```

merged

Rust

```
let words = ["alpha", "beta", "gamma"];
let merged: String = words.iter()
                          .flat_map(|s| s.chars())
                          .collect();
```

>>=

Haskell

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

flatMap

Swift

```
["abc", "def", "ghi"].map { Array($0) }
// [["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]]

["abc", "def", "ghi"].flatMap { Array($0) }
// ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
```

Result

```
public enum Result<Success, Failure: Error> {
  /// A success, storing a `Success` value.
  case success(Success)

  /// A failure, storing a `Failure` value.
  case failure(Failure)
}
```

Result

map

```
public func map<NewSuccess>(
  _ transform: (Success) -> NewSuccess
) -> Result<NewSuccess, Failure> {
  switch self {
  case let .success(success):
    return .success(transform(success))
  case let .failure(failure):
    return .failure(failure)
  }
}
```

Result

mapError

```
public func mapError<NewFailure>(
  _ transform: (Failure) -> NewFailure
) -> Result<Success, NewFailure> {
  switch self {
  case let .success(success):
    return .success(success)
  case let .failure(failure):
    return .failure(transform(failure))
  }
}
```

Result

flatMap

```
public func flatMap<NewSuccess>(
  _ transform: (Success) -> Result<NewSuccess, Failure>
) -> Result<NewSuccess, Failure> {
  switch self {
  case let .success(success):
    return transform(success)
  case let .failure(failure):
    return .failure(failure)
  }
}
```

Result

catching body

```
extension Result where Failure == Swift.Error {
  @_transparent
  public init(catching body: () throws -> Success) {
    do {
      self = .success(try body())
    } catch {
      self = .failure(error)
    }
  }
}
```

Result

Example

```swift
struct JsonIpResponse: Decodable {
    let ip: String
}
let url = URL(string: "https://jsonip.com")!
let responseData = Result { try Data(contentsOf: url) }
func decodeJsonIp(_ data: Data) -> Result<JsonIpResponse, Error> {
    let decoder = JSONDecoder()
    return Result { try decoder.decode(JsonIpResponse.self, from: data) }
}
let jsonIpResult = responseData.flatMap(decodeJsonIp)
let ip = jsonIpResult.map { $0.ip }
```

Other Categories?

→ Monoid

→ Semigroup

→ Applicative

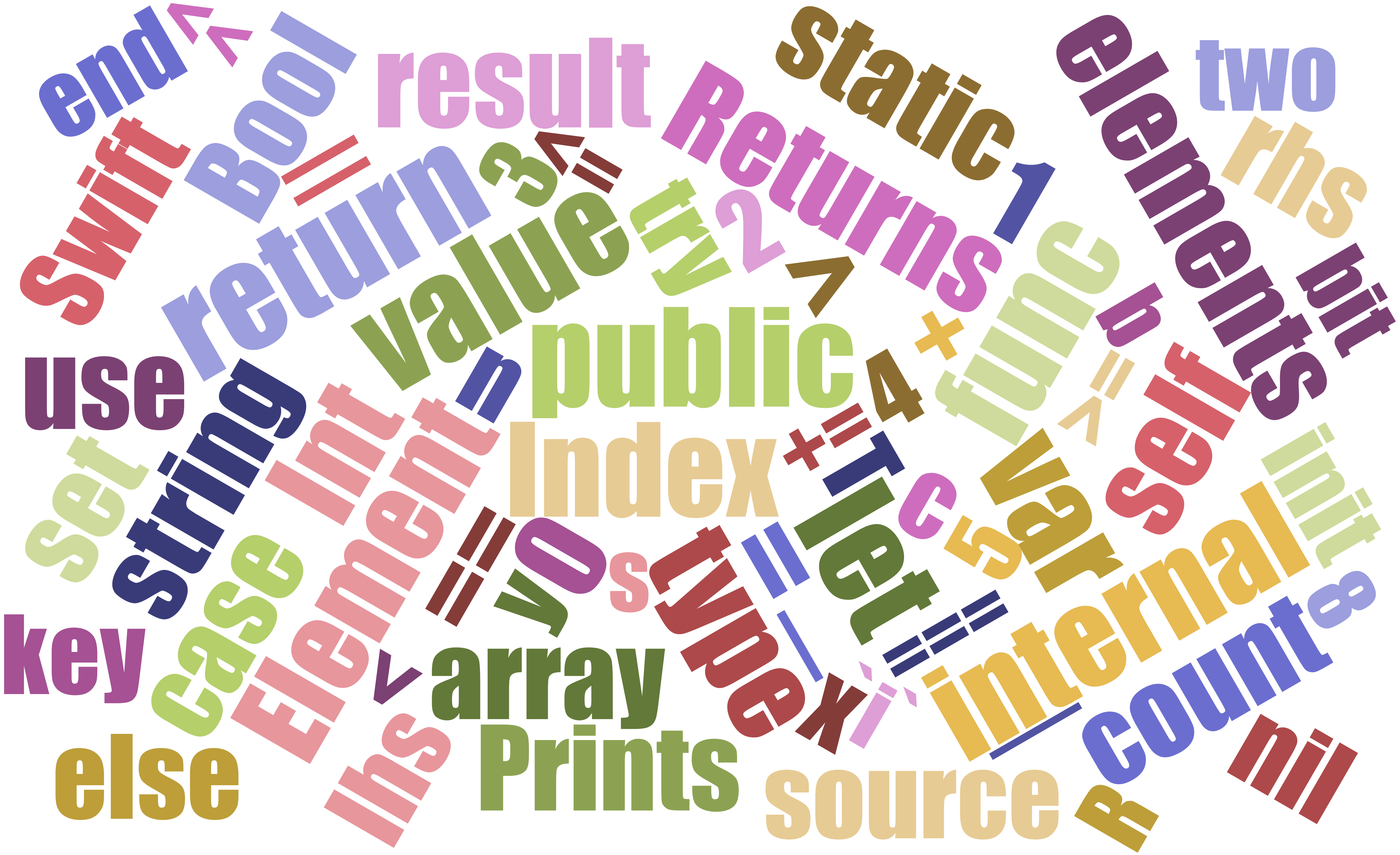
Monoid?

```
Prelude> :info Monoid
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Monoid?

AdditiveArithmetic?

```
public protocol AdditiveArithmetic: Equatable {
  static var zero: Self { get }
  static func +(lhs: Self, rhs: Self) -> Self
  static func +=(lhs: inout Self, rhs: Self)
  static func -(lhs: Self, rhs: Self) -> Self
  static func -=(lhs: inout Self, rhs: Self)
}
```


end
<<
Bool
result
Returns
Static
elements
two
swift
||
return
3
>=
value
try
2
>
1
rhs
func
b
bit
use
public
n
4
+
<=
self
init
set
string
Int
Element
Index
+=
T
let
c
var
5
internal
==
0
s
type
key
case
===
8
count
<
array
\
x
i
_
nil
else
lhs
Prints
source
R

extension
public
Hashable
Equatable
BidirectionalCollection
Collection
Sequence
struct
Bound
Range
protocol
base
Decodable
Character
Elements
element
ObjectIdentifier
RangeReplaceableCollection
Strideable
func
return
operators
error
Array
Iterator
Value
type
self
Bool
Hasher
String
functions
used
literal
Note
FIXME
Success
body
instance
closed
represent
nil
get
split
integer
Extending
internal
structs
class
enum
case
errors
rhs
create
failure
enums
add
static
default
code
extensions
look
isnt
LazyFilterCollection
enumeration
sometimes
switch
associated
new
Result
see
using
catch
Use
values
upper
Value>
associatedtype
provided
separate
balks
Array<Element>
Slice<Self>
compatibility
typealias
inference
getting
considered
lower
empty
shorthand
Complexity
often
like
seem
usage
let
heavily
Therefore
`05`
TODO
=
==
var
prefix
import
cover
50
r
etc
classes
SwiftShims
Syntax
equal
`ClosedRange`
theyre
contains
without
common
Strideable>
lhs
examples
put
sugar
just
input
42
5
00
lot
Show
includes
Container
override
IndexingIterator
useful
CountableClosedRange<Bound
CollectionDifference
requires
zeroInclusive

Structs

→ Array

→ Bool

→ Dictionary

→ Range

→ Set

→ String

→ Zip2Sequence

Enums

- → Optional
- → Result

Classes

→ ???

Algebraic Data Types

→ sum type - enums

→ product type - structs, tuples, classes

extension
public
Hashable
Equatable
BidirectionalCollection
Collection
struct
Sequence
Bound
protocol
Range
base
element
Decodable
Character
Strideable
RangeReplaceableCollection
ObjectIdentifier
Iterator
Value
Hasher
Bool
Array
type
func
return
error
operators
case
enum
Elements
self
LazyFilterCollection
enumeration
sometimes
Result
functions
used
literal
instance
body
Success
FIXME
Note
represent
closed
String
Value>
values
upper
input
catch
Use
using
see
new
associated
static
default
extensions
enums
add
code
create
failure
rhs
class
structs
internal
Extending
integer
split
nil
get
errors
like
let
var
==
=
isnt
etc
look
switch
just
lhs
Slice<Self>
compatibility
typealias
Array<Element>
associatedtype
separate
provided
balks
getting
inference
considered
shorthand
Complexity
empty
lower
often
seem
usage
Therefore
heavily
put
sugar
examples
TODO
prefix
import
classes
SwiftShims
Syntax
equal
`ClosedRange`
theyre
without
Strideable>
common
contains
Container
includes
lot
00
5
50
42
r
cover
Show
override
IndexingIterator
useful
CountableClosedRange<Bound
CollectionDifference
requires
zeroInclusive
`05`

Collection

```swift
// Set.swift
extension Set: Collection {
  @inlinable
  public var startIndex: Index {
    return _variant.startIndex
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  public var endIndex: Index {
    return _variant.endIndex
  }
}
```

Collections

```
extension Set: Collection {
  @inlinable
  public var count: Int {
    return _variant.count
  }

  /// A Boolean value that indicates whether the set is empty.
  @inlinable
  public var isEmpty: Bool {
    return count == 0
  }
}
```

Collection

```swift
extension Set: Collection {
  @inlinable
  public subscript(position: Index) -> Element {
    get {
      return _variant.element(at: position)
    }
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  public func index(after i: Index) -> Index {
    return _variant.index(after: i)
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  public func formIndex(after i: inout Index) {
    _variant.formIndex(after: &i)
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  public func firstIndex(of member: Element) -> Index? {
    return _variant.index(for: member)
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  @inline(__always)
  public func _customIndexOfEquatableElement(
     _ member: Element
    ) -> Index?? {
    return Optional(firstIndex(of: member))
  }
}
```

Collection

```
extension Set: Collection {
  @inlinable
  @inline(__always)
  public func _customLastIndexOfEquatableElement(
     _ member: Element
    ) -> Index?? {
    return _customIndexOfEquatableElement(member)
  }
}
```

Complexity

```
/// - Complexity: O(*n*), where *n* is the length of the collection.
/// - Complexity: O(1)
```

@inlinable

Apply this attribute to a function, method, computed property, subscript, convenience initializer, or deinitializer declaration to expose that declaration’s implementation as part of the module’s public interface. The compiler is allowed to replace calls to an inlinable symbol with a copy of the symbol’s implementation at the call site.

@inlinable

→ 2422 @inlinable

→ 852 @inline

→ 621 @usableFromInline

→ 527 @_transparent

→ 232 @_effects

→ 213 @available

→ 195 @frozen

@inlinable

https://github.com/apple/swift-evolution/blob/master/proposals/0193-cross-module-inlining-and-specialization.md

Other

- → extension
- → Hashable
- → Equatable
- → Codable
- → typealias
- → GYB - *Generate Your Boilerplate*
- → FIXME/TODO

Questions?

Thank You