Functional Programming in Swift

Manuel M T Chakravarty
Applicative

mchakravarty
@TacticalGrace
@tacticalgrace.bsky.social

cross-platform

open source

cross-platform

open source

cross-platform

multi-paradigm

open source

cross-platform

multi-paradigm

high-performance

strong functional core

open source

cross-platform

multi-paradigm

high-performance

”Are you a functional programmer?”

What is ing

”~~Are you a~~ functional programm~~er~~?”

Functions?

What is ing

"~~Are you a~~ functional programm~~er~~?"

Functions?

Closures?

What is ing

”~~Are you a~~ functional programm~~er~~?”

Functions?

Closures?

What is ing

"~~Are you a~~ functional programm~~er~~?"

Higher-order functions?

Language Features

Functional idioms

Functions Rule

Functions Rule

```
let arr = [1, 2, 3, 4, 5, 6]
```

Functions Rule

```
let arr: [Int]
```

```
let arr = [1, 2, 3, 4, 5, 6]
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map({ x in x + 1 })        // map (\x -> x + 1) arr
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }            // map (\x -> x + 1) arr
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }          // map (\x -> x + 1) arr
arr.map{ $0 + 1 }              // map (+ 1) arr
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }          // map (\x -> x + 1) arr
arr.map{ $0 + 1 }              // map (+ 1) arr
arr.map(-)                     // map negate arr
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }            // map (\x -> x + 1) arr
arr.map{ $0 + 1 }                // map (+ 1) arr
arr.map(-)                       // map negate arr
arr.reduce(0, +)                 // foldl (+) 0 arr
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }          // map (\x -> x + 1) arr
arr.map{ $0 + 1 }              // map (+ 1) arr
arr.map(-)                     // map negate arr
arr.reduce(0, +)               // foldl (+) 0 arr
```

```
struct Array<Element> {        // [Element]
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }            // map (\x -> x + 1) arr
arr.map{ $0 + 1 }                // map (+ 1) arr
arr.map(-)                       // map negate arr
arr.reduce(0, +)                 // foldl (+) 0 arr
```

```
struct Array<Element> {          // [Element]
  func map<T>((Element) -> T) -> [T]
}
```

```
// map :: (element -> t) -> [element] -> [t]
```

Functions Rule

```
let arr: [Int]
```

Haskell version

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ x in x + 1 }           // map (\x -> x + 1) arr
arr.map{ $0 + 1 }               // map (+ 1) arr
arr.map(-)                      // map negate arr
arr.reduce(0, +)                // foldl (+) 0 arr
```

```
struct Array<Element> {         // [Element]
  func map<T>((Element) throws -> T) rethrows -> [T]
}
```

Algebraic Data Types

Algebraic Data Types

Products

```
struct Vector {
  let x: Float
  let y: Float
}
```

```
// data Vector = Vector Float Float
```

Algebraic Data Types

Products

```
struct Vector {
  let x: Float
  let y: Float
}
```

```
// data Vector = Vector Float Float
```

Sums

```
enum Bool {
  case false
  case true
}
```

```
// data Bool = False | True
```

Algebraic Data Types

Products

```
struct Vector {
  let x: Float
  let y: Float
}
```

```
// data Vector = Vector Float Float
```

Sums

```
enum Optional<T> {
  case none
  case some(T)
}
```

```
// data Optional t
//   = None
//   | Some t
```

Algebraic Data Types

Products

```
struct Vector {
  let x: Float
  let y: Float
}
```

Sums

```
enum Optional<T> {
  case none
  case some(T)
}
```

```
enum Tree<T> {
  case leaf(T)
  case node(left: Tree<T>, right: Tree<T>)
}
```

```
// data Tree t
//   = Leaf t
//   | Node (Tree t) (Tree t)
```

Algebraic Data Types

Products

```
struct Vector {
  let x: Float
  let y: Float
}
```

Sums

```
enum Optional<T> {
  case none
  case some(T)
}
```

```
indirect enum Tree<T> {
  case leaf(T)
  case node(left: Tree<T>, right: Tree<T>)
}
```

```
// data Tree t
//   = Leaf t
//   | Node (Tree t) (Tree t)
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float
}
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)

func length(vector: Vector) -> Float {
  sqrt(vector.x * vector.x + vector.y * vector.y)
}
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)

func length(vector: Vector) -> Float {
  sqrt(vector.x * vector.x + vector.y * vector.y)
}

length(vector: vec)
```

Algebraic Data Types

```swift
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)

func length(vector: Vector) -> Float {
  sqrt(vector.x * vector.x + vector.y * vector.y)
}

length(vector: vec)
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float

  var length: Float { sqrt(x * x + y * y) }
}
```

Algebraic Data Types

```
struct Vector {
  let x: Float
  let y: Float

  var length: Float { sqrt(x * x + y * y) }
}

let vec = Vector(x: 10, y: 20)

vec.length
```

Immutable Data Structures

```
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)
```

Immutable Data Structures

```swift
struct Vector {
  let x: Float
  let y: Float
}

let vec = Vector(x: 10, y: 20)

vec = Vector(x: 0, y: 0)
```

Cannot assign to value: 'vec' is a 'let' constant

Immutable Data Structures

```
struct Vector {
  let x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec = Vector(x: 0, y: 0)
```

Immutable Data Structures

```
struct Vector {
  let x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)
```

Immutable Data Structures

```
struct Vector {
  let x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec.x = 0
```

Cannot assign to property: 'x' is a 'let' constant

Immutable Data Structures

```swift
struct Vector {
  var x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec.x = 0
```

Immutable Data Structures

```
struct Vector {
  var x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec.x = 0
```

Cannot assign to property: 'x' is a 'let' constant

We'll come back to this!

Strong Typing

Strong Typing

```
enum Optional<T> {
  case none
  case some(T)
}
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String = nil
```

'nil' cannot initialize specified type 'String'

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: Optional<String> = .none
```

> 'nil' cannot initialize specified type 'String'

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: Optional<String> = .none
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil        // nil == .none
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil        // nil == .none
var city: String? = "Berlin"   // implicit .some
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil        // nil == .none
var city: String? = ”Berlin"   // implicit .some
switch city {
case .none:                    break
case .some(let cityName): print(cityName)
}
```

Strong Typing

generics

```
enum Optional<T> {
  case none
  case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil          // nil == .none
var city: String? = "Berlin"   // implicit .some

if let cityName = city {
  print(cityName)
}
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil         // nil == .none
var city: String? = ”Berlin"    // implicit .some

if let city {
  print(city)
}
```

Strong Typing

generics

```
enum Optional<T> {
   case none
   case some(T)
}
```

(local) type inference

no null pointers

```
var name: String? = nil        // nil == .none
var city: String? = ”Berlin"   // implicit .some
if let city { print(city) }
```

We’ll come back to this!

Structured Concurrency

Structured Concurrency

async/await

Structured Concurrency

async/await

async is
statically tracked

Structured Concurrency

async/await

async is statically tracked

actors for isolation

Structured Concurrency

async/await

async is statically tracked

actors for isolation

`Sendable` to mark safe types

Controlling Mutability

Value types

Controlling Mutability

```
struct Vector {
  var x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec.x = 0
```

Controlling Mutability

```
struct Vector {
  var x: Float
  let y: Float
}

var vec = Vector(x: 10, y: 20)

vec.x = 0
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
```

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1

v2.x = 0
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

var v1 = VectorC(x: 15,
                 y: 25)
var v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}

let v1 = VectorC(x: 15,
                 y: 25)
let v2 = v1

v2.x = 0
print(v1)
] VectorC(x: 0, y: 25)
```

reference gets copied

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
let v1 = VectorC(x: 15,
                 y: 25)
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
let v1 = VectorC(x: 15,
                 y: 25)
func zero(vec: VectorC) {
  vec.x = 0
}
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
let v1 = VectorC(x: 15,
                 y: 25)
func zero(vec: VectorC) {
  vec.x = 0
}
zero(vec: v1)
```

Controlling Mutability

Value type

```
struct Vector {
  var x: Float
  var y: Float
}

var v1 = Vector(x: 15,
                y: 25)
var v2 = v1

v2.x = 0
print(v1)
] Vector(x: 15, y: 25)
```

value gets copied

Reference type

```
class VectorC {
  var x: Float
  var y: Float
}
let v1 = VectorC(x: 15,
                 y: 25)
func zero(vec: VectorC) {
  vec.x = 0
}
zero(vec: v1)
print(v1)
] VectorC(x: 0, y: 25)
```

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  var length: Float { sqrt(x * x + y * y) }
}
```

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  func translate(offset: Vector) -> Vector {
    Vector(x: x + offset.x, y: y + offset.y)
  }
```

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  func translate(offset: Vector) -> Vector {
    Vector(x: x + offset.x, y: y + offset.y)
  }

  mutating func move(offset: Vector) {
    x += offset.x
    y += offset.y
  }
}
```

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  func translate(offset: Vector) -> Vector {
    Vector(x: x + offset.x, y: y + offset.y)
  }

  mutating func move(offset: Vector) {
    x += offset.x
    y += offset.y
  }
}
let vec = Vector(x: 10, y: 20)
vec.move(offset: Vector(x: 5, y: 5))
```

Cannot use mutating member on immutable value: 'vec' is a 'let' constant

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  func translate(offset: Vector) -> Vector {
    Vector(x: x + offset.x, y: y + offset.y)
  }

  mutating func move(offset: Vector) {
    x += offset.x
    y += offset.y
  }
}
var vec = Vector(x: 10, y: 20)
vec.move(offset: Vector(x: 5, y: 5))
```

Cannot use mutating member on immutable value: 'vec' is a 'let' constant

Explicit Mutability

```
struct Vector {
  var x: Float
  var y: Float

  func translate(offset: Vector) -> Vector {
    Vector(x: x + offset.x, y: y + offset.y)
  }

  mutating func move(offset: Vector) {
    x += offset.x
    y += offset.y
  }
}
var vec = Vector(x: 10, y: 20)
vec.move(offset: Vector(x: 5, y: 5))
```

Compound Value Types

Compound Value Types

`String`

`Dictionary`

`Array`

`Set`

`…and so on`

Compound Value Types

`String`

`Dictionary`

`Array`

`Set`

`…and so on`

Build your own!

Compound Value Types

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
```

Compound Value Types

```
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                          // ⇒ 3
```

Compound Value Types

```swift
let arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                          // ⇒ 3
arr[2] = 10
```

Cannot assign through subscript: 'arr' is a 'let' constant

Compound Value Types

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                         // ⇒ 3
arr[2] = 10
```

Compound Value Types

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                          // ⇒ 3
arr[2] = 10

func printShuffled(arr: [Int]) {
  var localArr = arr
  localArr.shuffle()
  print(localArr)
}
```

Compound Value Types

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                        // ⇒ 3
arr[2] = 10

func printShuffled(arr: [Int]) {
  var localArr = arr
  localArr.shuffle()
  print(localArr)
}
```

changes local copy only

Compound Value Types

```
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                         // ⇒ 3
arr[2] = 10

func printShuffled(arr: [Int]) {
  var localArr = arr
  localArr.shuffle()
  print(localArr)
}

arr.shuffle()
```

Compound Value Types

```swift
var arr = [1, 2, 3, 4, 5, 6]
arr.map{ 2 + $0 }
arr[2]                          // ⇒ 3
arr[2] = 10

func printShuffled(arr: [Int]) {
  var localArr = arr
  localArr.shuffle()
  print(localArr)
}

arr.shuffle()
```

now we have changed `arr`

```swift
struct Array<Element> {   // [Element]
  mutating func shuffle()
}
```

Key Paths

```
struct Path {
  var vectors: [Vector]
}
```

Key Paths

```
struct Path {
  var vectors: [Vector]
}
var paths: [Path] = …
```

Key Paths

```
struct Path {
  var vectors: [Vector]
}
var paths: [Path] = …

paths[2].vectors[10].x = 0
```

Key Paths

```
struct Path {
  var vectors: [Vector]
}
var paths: [Path] = …

paths[2].vectors[10].x = 0

let keyPath = \[Path][2].vectors[10].x
```

the type that the key path is defined on

Key Paths

```
struct Path {
  var vectors: [Vector]
}
var paths: [Path] = …

paths[2].vectors[10].x = 0

let keyPath = \[Path][2].vectors[10].x
```

the type that the key path is defined on

```
WriteableKeyPath<[Path], Int>
```

Key Paths

```
struct Path {
  var vectors: [Vector]
}
var paths: [Path] = …

paths[2].vectors[10].x = 0

let keyPath = \[Path][2].vectors[10].x

              WriteableKeyPath<[Path], Int>
```

the type that the key path is defined on

the projected type

Strong Types

Protocols & associated types

Protocols

```
protocol Equatable {
```

Protocols

```
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
```

Protocols

```
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
  static func != (lhs: Self, rhs: Self) -> Bool {
```

Protocols

```
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
  static func != (lhs: Self, rhs: Self) -> Bool {
    !(lhs == rhs)
  }
}
```

Protocols

```swift
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
  static func != (lhs: Self, rhs: Self) -> Bool {
    !(lhs == rhs)
  }
}
```

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  lhs != rhs = not (lhs == rhs)
```

Protocols

```swift
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
  static func != (lhs: Self, rhs: Self) -> Bool {
    !(lhs == rhs)
  }
}
```

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  lhs != rhs = not (lhs == rhs)
```

(interfaces in Java and traits in Rust)

Protocols

```
 protocol Identifiable<ID> {

   associatedtype ID : Hashable

   var id: ID { get }
}
```

Protocols

```swift
protocol Identifiable<ID> {

  associatedtype ID : Hashable

  var id: ID { get }
}
```

```haskell
class Identifiable a where
  type ID a
  id :: Hashable (ID a) => ID a
```

Protocols

```
protocol Identifiable<ID> {

  associatedtype ID : Hashable

  var id: ID { get }
}
```

```
class Identifiable a where
  type ID a
  id :: Hashable (ID a) => ID a
```

```
struct MyData: Identifiable {   // protocol conformance
```

Protocols

```
protocol Identifiable<ID> {

  associatedtype ID : Hashable

  var id: ID { get }
}
```

```
class Identifiable a where
  type ID a
  id :: Hashable (ID a) => ID a
```

```
struct MyData: Identifiable {   // protocol conformance

  let id = UUID()        // ⇒ ID == UUID

  …
}
```

Associated Types

```
 protocol Sequence<Element> {

   associatedtype Element
}
```

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element
}
```

Which operations do we want?

Associated Types

```
protocol Sequence<Element> {

   associatedtype Element
}
```

Which operations do we want?

containment check

Associated Types

```
protocol Sequence<Element> {

   associatedtype Element
}
```

Which operations do we want?

containment check

filter

Associated Types

```
protocol Sequence<Element> {

   associatedtype Element
}
```

Which operations do we want?

containment check

map

filter

Associated Types

```
protocol Sequence<Element> {

   associatedtype Element
}
```

Which operations do we want?

containment check

map

filter

many more…

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

Associated Types

```swift
protocol Sequence<Element> {

  associatedtype Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

```swift
protocol IteratorProtocol<Element> {

  associatedtype Element

  mutating func next() -> Element?
}
```

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element where Element == Iterator.Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

```
protocol IteratorProtocol<Element> {

  associatedtype Element

  mutating func next() -> Element?
}
```

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element where Element == Iterator.Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

```
protocol IteratorProtocol<Element> {

  associatedtype Element

  mutating func next() -> Element?
}
```

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element where Element == Iterator.Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

Default implementation for `contains(:)`, `map(:)`, `filter(:)`, ...

```
protocol IteratorProtocol<Element> {
  associatedtype Element
  mutating func next() -> Element?
}
```

Associated Types

```
protocol Sequence<Element> {

  associatedtype Element where Element == Iterator.Element
  associatedtype Iterator: IteratorProtocol

  func makeIterator() -> Iterator
}
```

A sequence provides sequential and possibly destructive access to its elements.

Associated Types

```
protocol Collection<Element> : Sequence {
```

Associated Types

```
protocol Collection<Element> : Sequence {

  associatedtype Element
  associatedtype Index : Comparable where …
```

safe indexing

Associated Types

```
protocol Collection<Element> : Sequence {

  associatedtype Element
  associatedtype Index : Comparable where …

  var startIndex: Index { get }
  var endIndex:   Index { get }
  func index(after i: Index) -> Index
```

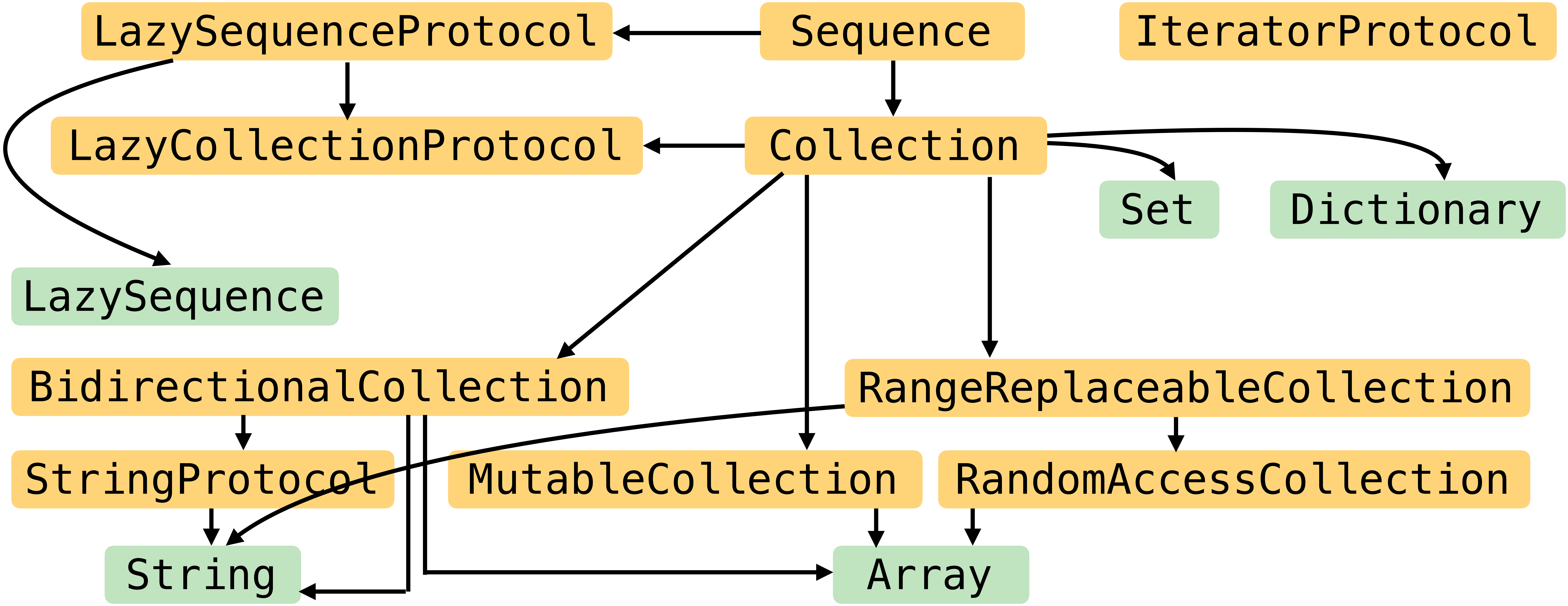
safe indexing

Associated Types

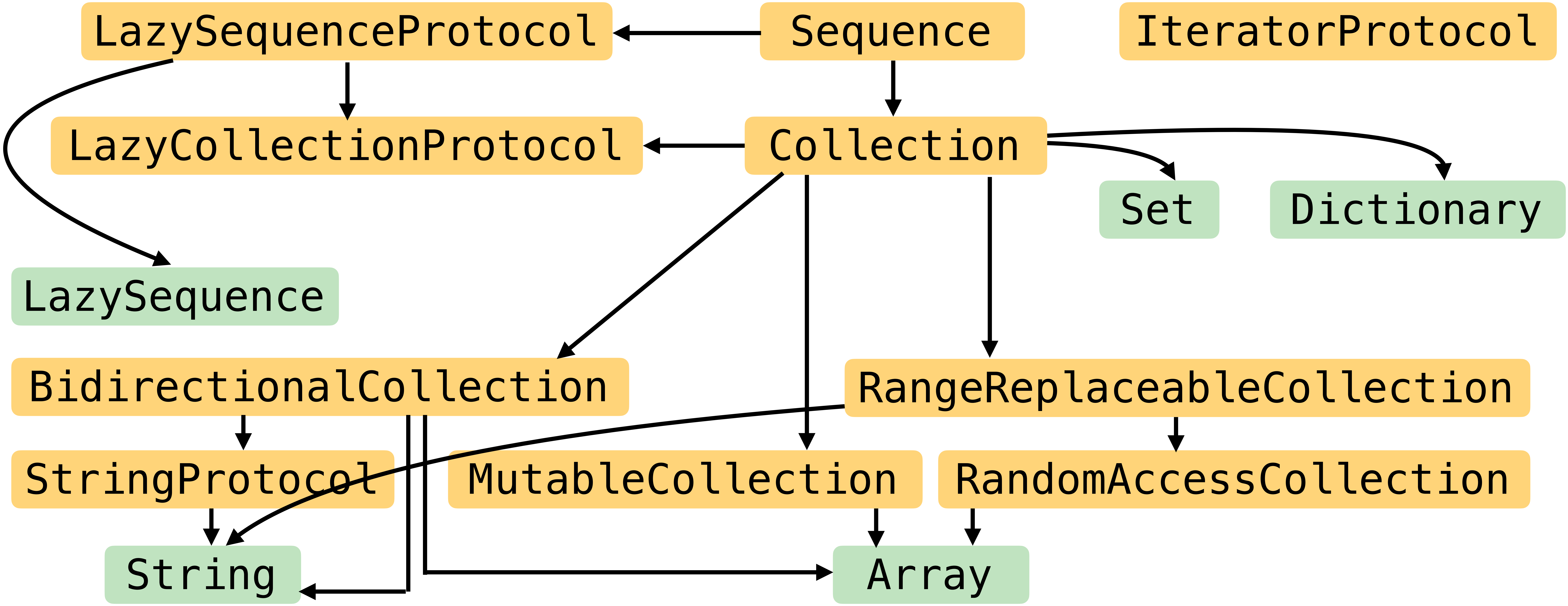
```
protocol Collection<Element> : Sequence {

  associatedtype Element
  associatedtype Index : Comparable where …

  var startIndex: Index { get }
  var endIndex:   Index { get }
  func index(after i: Index) -> Index

  subscript(position: Index) -> Element { get }
}
```

safe indexing

Collections in Foundation



Collections in Foundation



Protocol-oriented programming

Safety

Very flexible, but still safe!

Safety

Very flexible, but still safe!

Java AbstractCollection<E> – add(E e)

Throws:

UnsupportedOperationException - if the add operation is not supported by this collection

ClassCastException - if the class of the specified element prevents it from being added to this collection

NullPointerException - if the specified element is null and this collection does not permit null elements

IllegalArgumentException - if some property of the element prevents it from being added to this collection

IllegalStateException - if the element cannot be added at this time due to insertion restrictions

Ecosystem

Tools, packages & evolution

Memory Management

Memory Management

Swift cares about resource use

Memory Management

Swift cares about resource use

deallocation latency

Memory Management

Swift cares about resource use

deallocation latency

stop-the-world pauses

Memory Management

Swift cares about resource use

deallocation latency

stop-the-world pauses

”Swift Godot:
Fixing the Multi-million Dollar Mistake”

https://www.youtube.com/watch?v=tzt36EGKEZo

Memory Management

Swift cares about resource use

Memory Management

Swift cares about resource use

Automatic Reference Counting

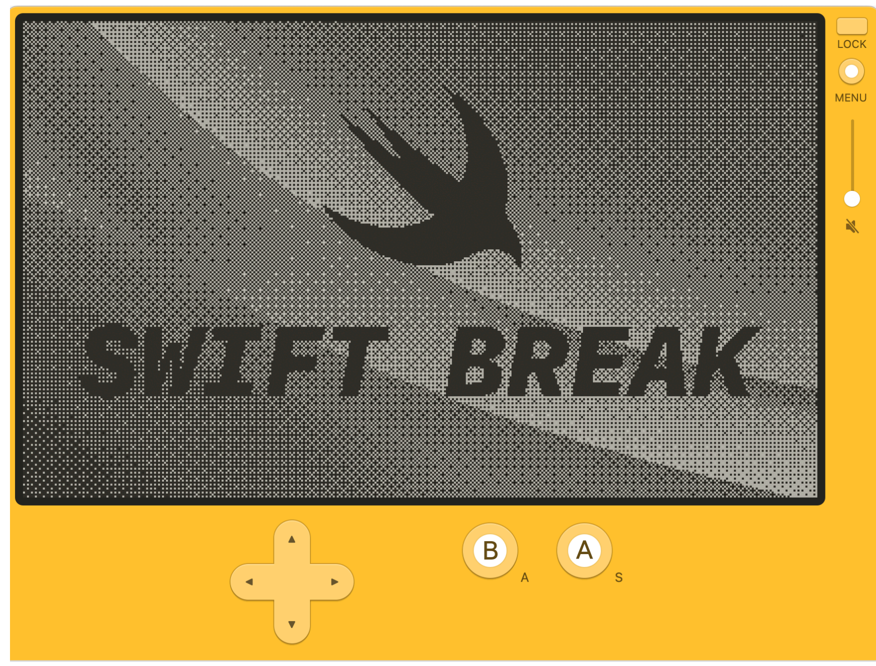
Non-copyable Types (Ownership)

Memory Management

Swift cares about resource use

Automatic Reference Counting

Non-copyable Types (Ownership)



https://www.swift.org/blog/byte-sized-swift-tiny-games-playdate/

Cross-Platform

Install Swift

Follow the instructions below to install the latest version of Swift on a supported platform.

You can also download nightly snapshots and older releases.

Latest Release: Swift 5.10

---

macOS

Xcode

Download the current version of Xcode which contains the latest Swift release.

Download Xcode

Additional install options for macOS:

- Package Installer - *Package installers (.pkg) are available on download page.*

Linux

Docker

The offical Docker images for Swift.

Instructions

Additional install options for Linux:

- Tarball - *Tarball packages are available on download page.*
- RPM - *Swift 5.10 RPMs for Amazon Linux 2 and CentOS 7 are for experimental use only. Please provide your feedback.*

Windows

Package Manager

Install Swift via Windows Package Manager (aka WinGet).

Instructions

Additional install options for Windows:

- Scoop - *Install Swift via Scoop.*
- Package Installer - *Package installers (.exe) are available on download page.*

Cross-Platform

LLVM-based toolchain

Swift Server Workgroup

VSCode support via LSP

Cross-platform libraries

Install Swift

Follow the instructions below to install the latest version of Swift on a supported platform.

You can also download nightly snapshots and older releases.

Latest Release: Swift 5.10

macOS

Xcode

Download the current version of Xcode which contains the latest Swift release.

Download Xcode

Additional install options for macOS:

- Package Installer - *Package installers (.pkg) are available on download page.*

Linux

Docker

The offical Docker images for Swift.

Instructions

Additional install options for Linux:

- Tarball - *Tarball packages are available on download page.*
- RPM - *Swift 5.10 RPMs for Amazon Linux 2 and CentOS 7 are for experimental use only. Please provide your feedback.*

Windows

Package Manager

Install Swift via Windows Package Manager (aka WinGet).

Instructions

Additional install options for Windows:

- Scoop - *Install Swift via Scoop.*
- Package Installer - *Package installers (.exe) are available on download page.*

Swift Package Manager

Swift Package Manager

Uses Git repos as package sources

Swift Package Manager

Uses Git repos as package sources

Package manifests are Swift code (`Package.swift`)

Swift Package Manager

Uses Git repos as package sources

Package manifests are Swift code (`Package.swift`)

Dependency resolution & building

Swift Evolution

"How is Swift, the language, developed?"

Swift Evolution

”How is Swift, the language, developed?”

The Swift evolution process

Swift Evolution

”How is Swift, the language, developed?”

The Swift evolution process

Public proposals and discussion

Swift Evolution

”How is Swift, the language, developed?”

The Swift evolution process

Public proposals and discussion

Led by Language Steering Group
(two year term)

https://swift.org/

cross-platform

multi-paradigm

open source

high-performance

strong functional core

Tutorial: "SwiftUI: Declarative GUIs for Mobile and Desktop Applications"

mchakravarty
@TacticalGrace
@tacticalgrace.bsky.social

Thank you!

Image Attribution

pixabay

https://pixabay.com/photos/equipment-technology-retro-camera-3106852/

https://pixabay.com/photos/stones-rocks-stack-pebbles-2040340/

https://pixabay.com/photos/cable-steel-wire-metal-woven-metal-7771045/

https://pixabay.com/photos/christmastree-worms-close-up-coral-202320/

Icons licensed
from Noun Project