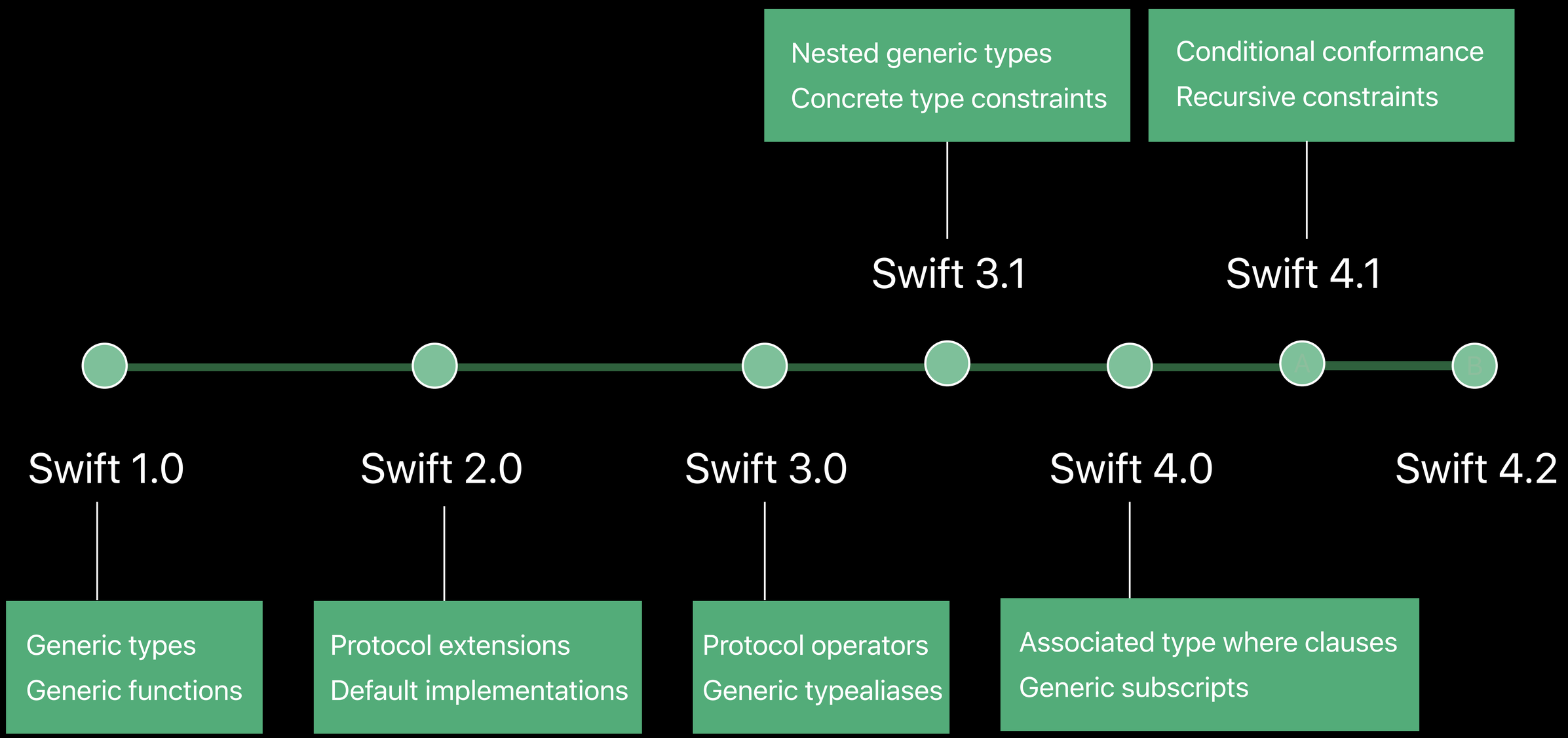
#WWDC18

Swift Generics

Session 406

Ben Cohen, Swift Standard Library
Doug Gregor, Swift Compiler


Nested generic types
Concrete type constraints
Conditional conformance
Recursive constraints
Swift 3.1
Swift 4.1
Swift 1.0
Swift 2.0
Swift 3.0
Swift 4.0
Swift 4.2
Generic types
Generic functions
Protocol extensions
Default implementations
Protocol operators
Generic typealiases
Associated type where clauses
Generic subscripts

What are generics?

Protocol design

Protocol inheritance

Conditional conformance

Classes and generics

Why Generics?

A Simple Buffer Type

```
struct Buffer  {
  var count: Int

  subscript(at: Int) -> ??? {
   // get/set from storage
  }
}
```

A Simple Buffer Type

```
struct Buffer  {
  var count: Int

  subscript(at: Int) -> Any {
   // get/set from storage
  }
}
```

Untyped Storage

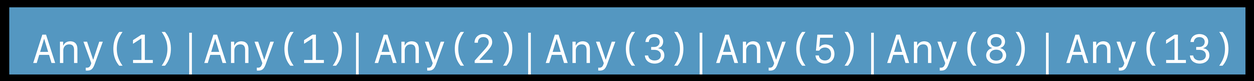
```
var words: Buffer = ["subtyping","ftw"]

// I know this array contains strings
words[0] as! String

// Uh-oh, now it doesn't!
words[0] = 42
```

In-Memory Representation

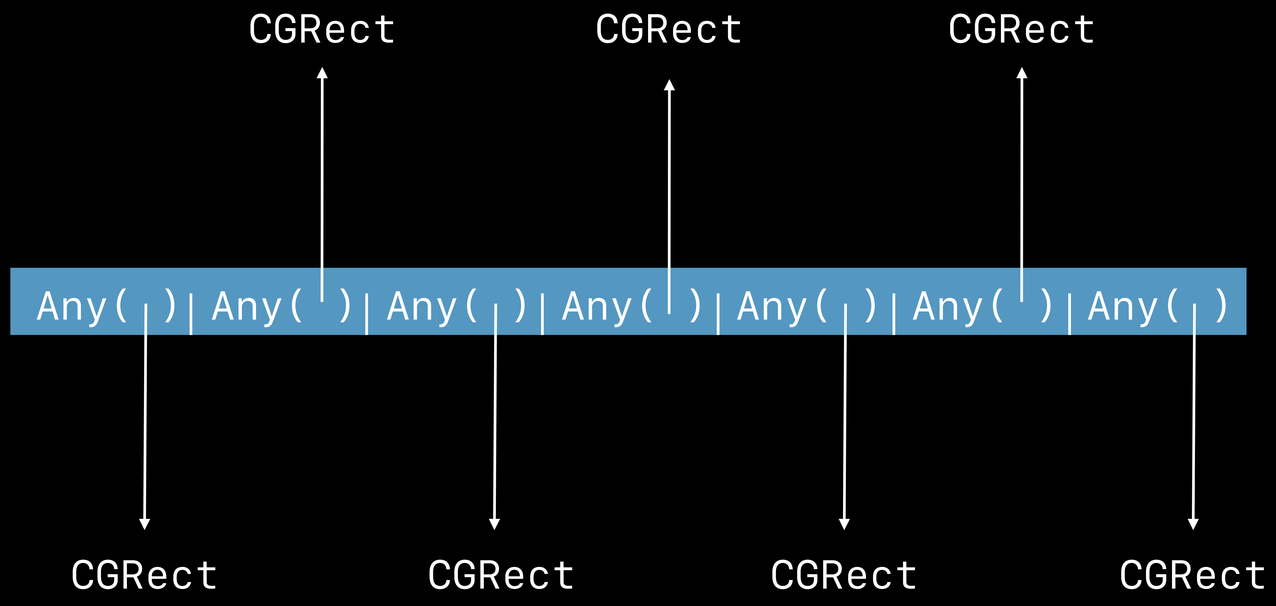
```
let numbers: Buffer = [1,1,2,3,5,8,13]
```

In-Memory Representation

```
let numbers: Buffer = [1,1,2,3,5,8,13]
```



Any and Indirection



Parametric Polymorphism

Generics

```
// Generic Buffer Type

struct Buffer          {
  let count: Int

  subscript(at: Int) -> Any    {
    // fetch from storage
  }
}
```

```
// Generic Buffer Type

struct Buffer<Element> {
  let count: Int

  subscript(at: Int) -> Element {
    // fetch from storage
  }
}
```

```
// Generic Type Safety

var words: Buffer<String> = ["generics","ftw"]

words[0] as! String
```

Forced cast of 'String' to same type has no effect

```
// Generic Type Safety

var words: Buffer<String> = ["generics","ftw"]


words[0]

words[0] = 42

var boxes: Buffer<CGRect> = words


var boxes: Buffer
```

Cannot assign value of type 'Int' to type 'String'

Cannot convert value of type 'Buffer<String>' to specified type 'Buffer<CGRect>'

Reference to generic type 'Buffer' requires arguments in <...>

Implied Generic Parameters

```
let words: Buffer          = ["generics","ftw"]
```

Implied Generic Parameters

```
let words: Buffer<String> = ["generics","ftw"]
```

Swift’s In-Memory Representation

```
let numbers: Buffer = [1,1,2,3,5,8,13]
```

Swift’s In-Memory Representation

```
// Optimization Opportunities

  let numbers: Buffer = [1,1,2,3,5,8,13]

  var total = 0
  for i in 0..<numbers.count {
    total += numbers[i]
  }
```

```
// Wrap Common Algorithms in Methods



    var total = 0
    for i in 0..<self.count {
        total += self[i]
      }
    }




let total = numbers.sum()
```

```
// Wrap Common Algorithms in Methods

extension Buffer                        {
  func sum() -> Element {
     var total = 0
     for i in 0..<self.count {
         total += self[i]
       }
     }
    return total
  }
}


let total = numbers.sum()
```

```
// Wrap Common Algorithms in Methods

extension Buffer                          {
  func sum() -> Element {
     var total = 0
     for i in 0..<self.count {
         total += self[i]
       }
     }
    return total
  }
}


let total = numbers.sum()
```

Cannot convert value of type 'Element' to expected argument type 'Int'

Cannot convert return expression of type 'Int' to return type 'Element'

```
// Constrain Element to an Int

extension Buffer where Element == Int {
  func sum() -> Element {
     var total = 0
     for i in 0..<self.count {
         total += self[i]
       }
     }
    return total
  }
}


let total = numbers.sum()
```

```
// Wrap Common Algorithms in Methods

extension Buffer where Element: Numeric {
  func sum() -> Element {
    var total: Element = 0
    for i in 0..<self.count {
        total += self[i]
      }
    }
    return total
  }
}


let total = numbers.sum()
```

Designing a Protocol

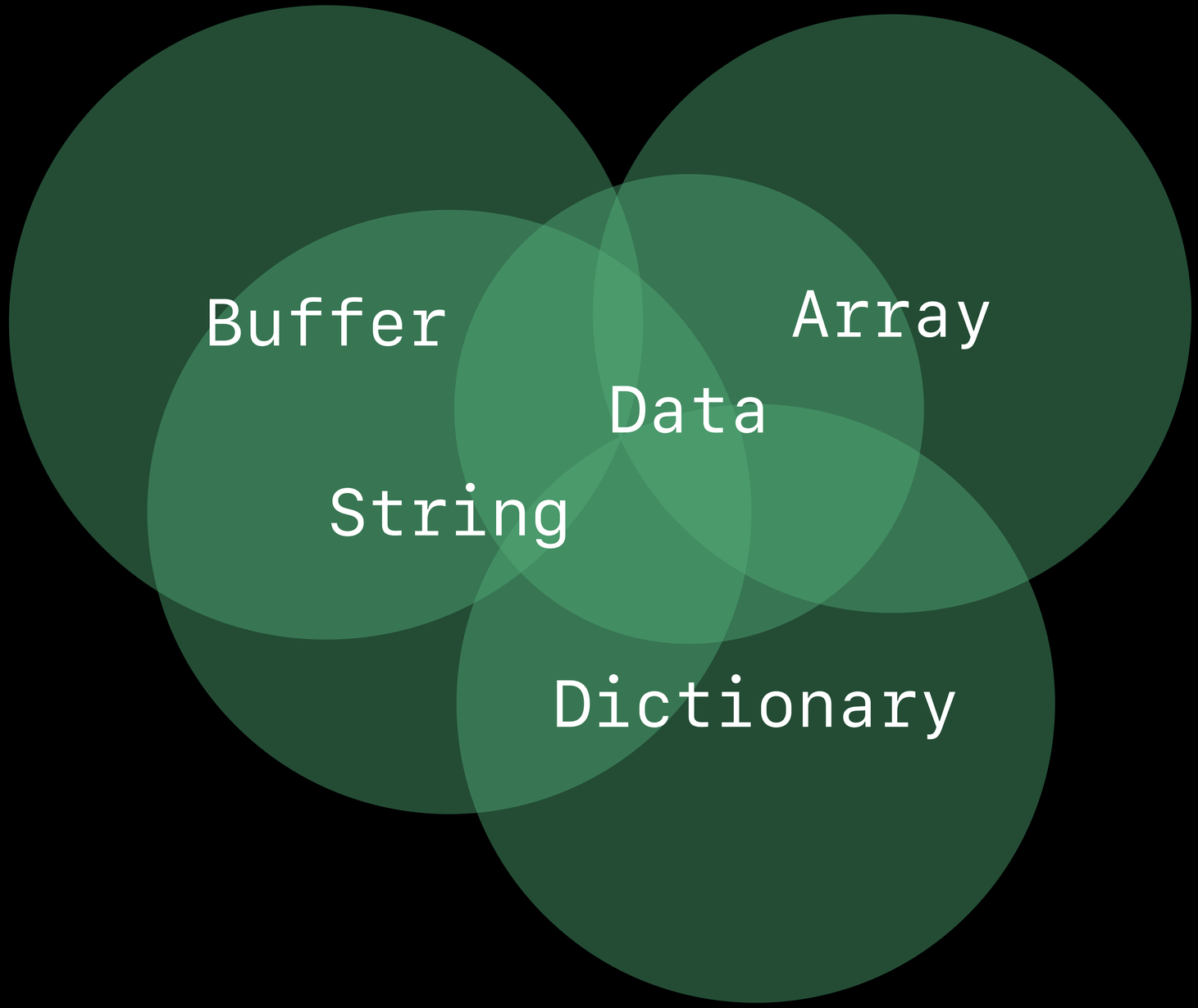
```
// Different Kinds of Collections

struct Buffer<Element> {
  let count: Int

  subscript(at: Int) -> Element


}
```

```
// Array

struct Array<Element> {
  let count: Int

  subscript(at: Int) -> Element


}
```

```
// Dictionary

struct Dictionary <Key: Hashable,Value> {
  let count: Int

  subscript(at: Index) -> (Key,Value)


}
```

```
// Non-Generic Collections

struct Data {
  let count: Int

  subscript(at: Int) -> UInt8


}

struct String {
  let count: Int

  subscript(at: Index) -> Character


}
```

```
// Collection Protocol

protocol Collection {


}
```


Buffer
Array
Data
String
Dictionary

```
// Collection Protocol


protocol Collection {


}
```

```
// Setting the Element Type


protocol Collection {
  associatedtype Element

}
```

```
// Setting the Element Type
```

```
protocol Collection {
  associatedtype Element

}
```

```
struct Buffer<Element> { }
extension Buffer: Collection { }

struct Array<Element> { }
extension Array: Collection { }
```

```
struct Dictionary<Key,Value> { }

extension Dictionary: Collection {
  typealias Element = (Key,Value)
}
```

// Defining Subscript

```
protocol Collection {
  associatedtype Element

  var count: Int { get }
  subscript(at: Int) -> Element
}
```

```
extension Collection {
  func dump() {
    for i in 0..<count {
      print(self[i])
    }
  }
}
```

```
// Defining Subscript for Dictionary


extension Dictionary:  Collection {
  private var _storage: HashBuffer<Element>
  struct Index {
    private let _offset: Int
  }
  subscript(at: Index) -> Element {
    return _storage[at._offset]
  }









}
```

```
// Dictionary Index Operations


                extension Dictionary:  Collection {
                  private var _storage: HashBuffer<Element>
                  struct Index {
                    private let _offset: Int
                  }
                  subscript(at: Index) -> Element {
                    return _storage[at._offset]
                  }


                  func index(after: Index ) ->  Index
                  var startIndex:  Index
                  var endIndex:  Index



                }
```

```
// Dictionary Index Operations



protocol Collection {
   associatedtype Element


   subscript(at: Index) -> Element

   func index(after: Index ) ->  Index

   var startIndex:  Index { get }

   var endIndex:  Index { get }

}
```

```
// Dictionary Index Operations



protocol Collection {
   associatedtype Element
   associatedtype Index

   subscript(at: Index) -> Element

   func index(after: Index ) ->  Index

   var startIndex:  Index { get }

   var endIndex:  Index { get }

}
```

```
// Adding Back Count



extension Collection {
  var count: Int {
    var i = 0
    var position = startIndex
    while position != endIndex {
      position = index(after: position)
      i += 1
    }
    return i
  }
}
```

```
// Adding Back Count


extension Collection {
  var count: Int {
    var i = 0
    var position = startIndex
    while position != endIndex {
      position = index(after: position)
      i += 1
    }
    return i
  }
}
```

Binary operator '!=' cannot be applied to two ‘Self.Index' operands

```
// Constraining Index


extension Collection where Index: Equatable {
  var count: Int {
    var i = 0
    var position = startIndex
    while position != endIndex {
      position = index(after: position)
      i += 1
    }
    return i
  }
}
```

```
// Constraining Index

protocol Collection {
  associatedtype Element
  associatedtype Index
}
```

```
// Constraining Index

protocol Collection {
  associatedtype Element
  associatedtype Index: Equatable
}
```

```
// Constraining Index

protocol Collection {
  associatedtype Element
  associatedtype Index: Equatable
}

extension Dictionary.Index: Equatable { }
```

Customization Points

```
// Slow Count

extension Collection {
  /// The number of elements in the collection
  var count: Int {
    var i = 0
    var position = startIndex
    while position != endIndex {
      i += 1
      position = index(after: position)
    }
    return i
  }
}
```

```
// Speedy Count
```

```
extension Dictionary {
  /// The number of elements in the collection
  var count: Int {
    return _storage.entryCount
  }
}
```

```
let d: Dictionary = ...

// Runs in O(1) not O(n)
let n = d.count
```

```
// Using Count in a Generic Context


extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {
    var result: [Element] = []


    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result
  }
}
```

```
// Using Count in a Generic Context


extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {
    var result: [Element] = []
    result.reserveCapacity(self.count)
    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result
  }
}
```

```
// Using Count in a Generic Context


extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {
    var result: [Element] = []
    result.reserveCapacity(self.count)
    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result
  }
}
```

```swift
// Using Count in a Generic Context

extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {
    var result: [Element] = []
    result.reserveCapacity(self.count)

    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result
  }
}
```

```swift
extension Collection {
  /// The number of elements in the collection
  var count: Int {
    // ...
  }
}
```

```
// Using Count in a Generic Context


protocol Collection {
  associatedtype Element
  associatedtype Index: Equatable

  subscript(at: Index) -> Element

  func index(after: Index ) ->  Index

  var startIndex:  Index { get }

  var endIndex:  Index { get }

  var count:  Int { get }
}
```

```
// Using .count in a Generic Context

extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {

    var result: [Element] = []

    result.reserveCapacity(self.count)

    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result

  }

}
```

```
// Using .count in a Generic Context

extension Collection {
  func map<T>(_ transform: (Element)->T) -> [Element] {

    var result: [Element] = []

    result.reserveCapacity(self.count)

    var position = startIndex
    while position != endIndex {
      result.append(transform(self[position]))
      position = index(after: position)
    }
    return result
  }
}
```

```
protocol Collection {

  var count:  Index { get }

}
```

```
extension Dictionary {
  /// The number of elements in the collection
  var count: Int {
    return _buffer.entryCount
  }
}
```

// Choosing When to Define Customization Points

```
protocol Collection {

  var count:  Index { get }

}

extension Collection {

  func map<T>(_ transform: (Element)->T) -> [Element] {
    // ...
  }

}
```

Protocol Inheritance

The Collection Protocol Is Not Enough

Some collection algorithms need more than `Collection` provides:

- `lastIndex(where:)` needs to walk backwards to be efficient
- `shuffle()` needs to swap elements to work at all

Some conforming types have these capabilities

Protocol Inheritance: BidirectionalCollection

Inheritance describes additional requirements for a subset of conforming types

- `SinglyLinkedList` cannot conform to `BidirectionalCollection`

```
protocol BidirectionalCollection: Collection {
  func index(before idx: Index) -> Index
}
```

Use Inheritance to Provide More-Specialized Algorithms

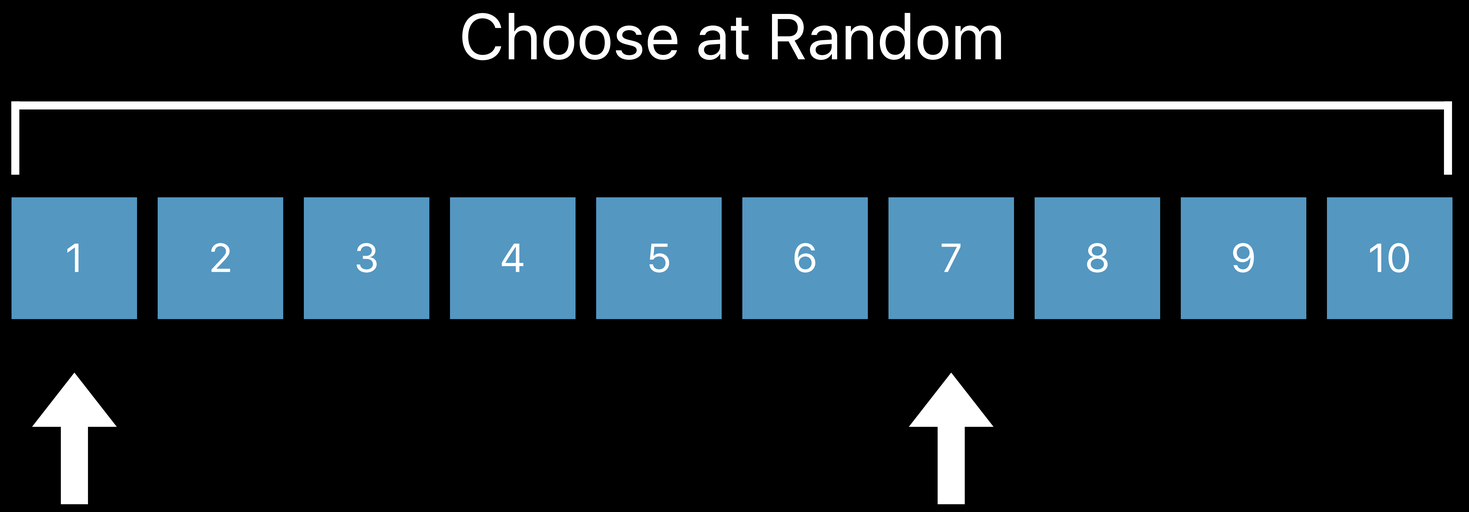
```
extension BidirectionalCollection {
  func lastIndex(where predicate: (Element) -> Bool) -> Index? {
    var position = endIndex
    while position != startIndex {
      position = index(before: position)
      if predicate(self[position]) { return position }
    }
    return nil
  }
}
```

Fisher-Yates Shuffle

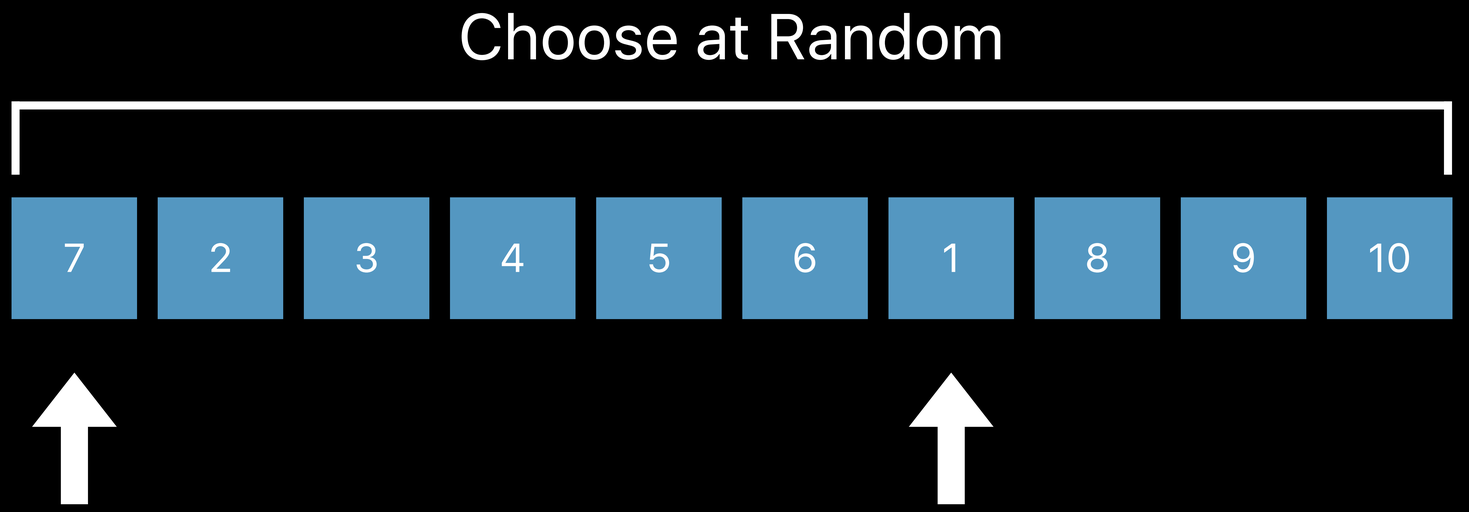
```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
array.shuffle()
print(array) // [7, 5, 9, 6, 10, 8, 3, 1, 2, 4]
```

Fisher-Yates Shuffle

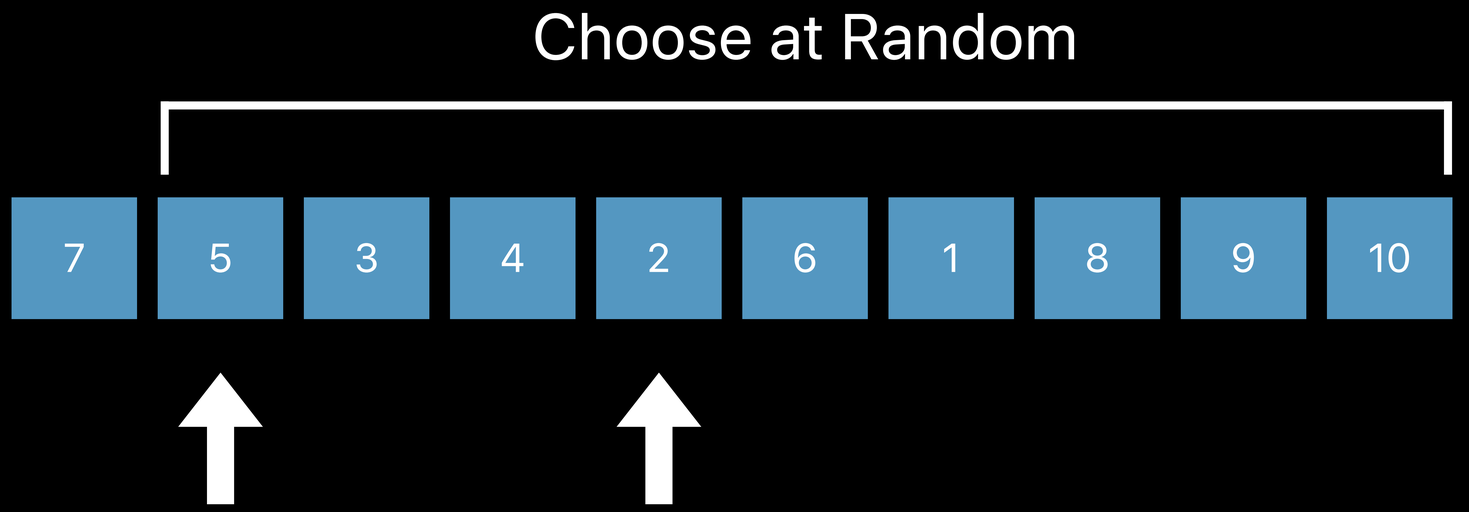
Fisher-Yates Shuffle



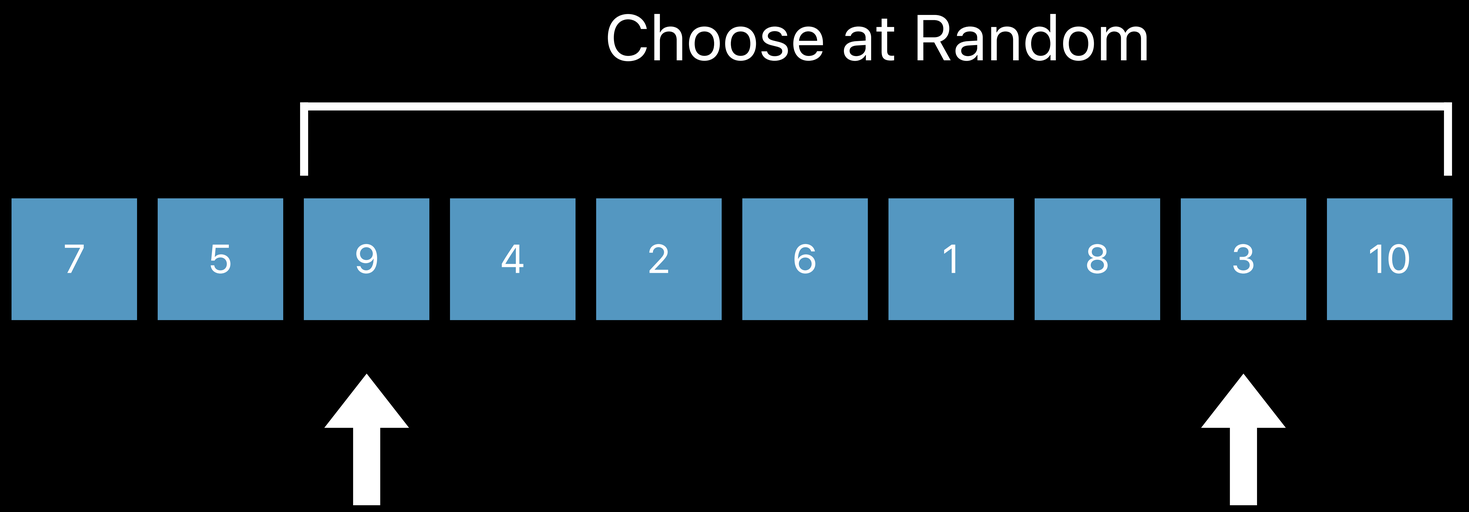
Fisher-Yates Shuffle



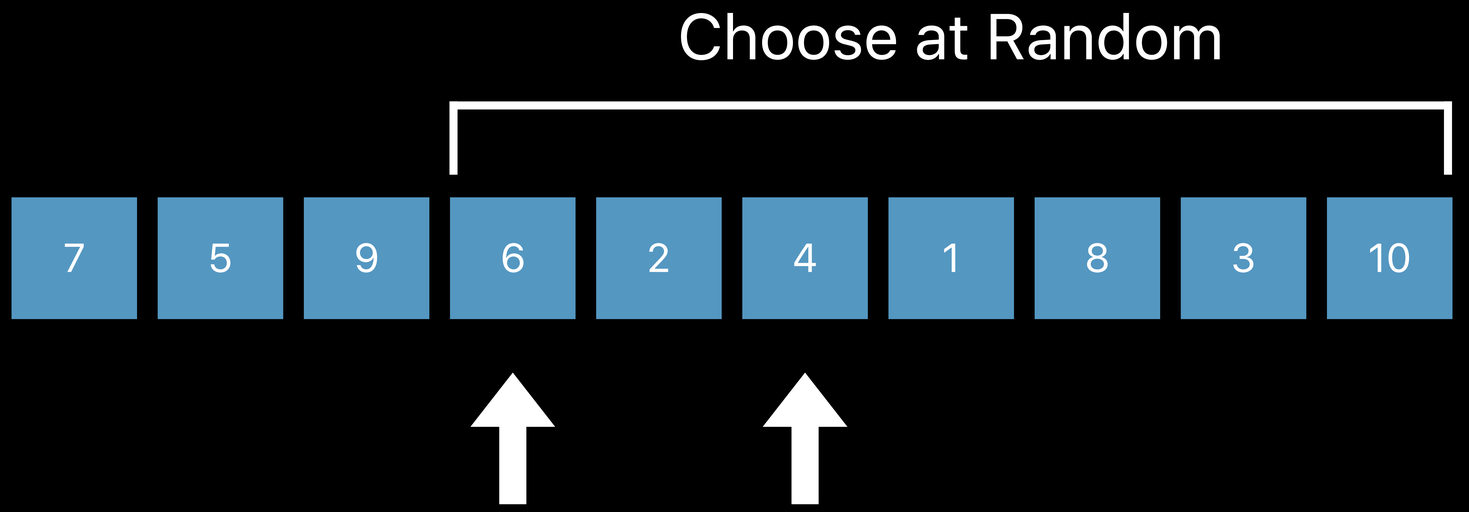
Fisher-Yates Shuffle



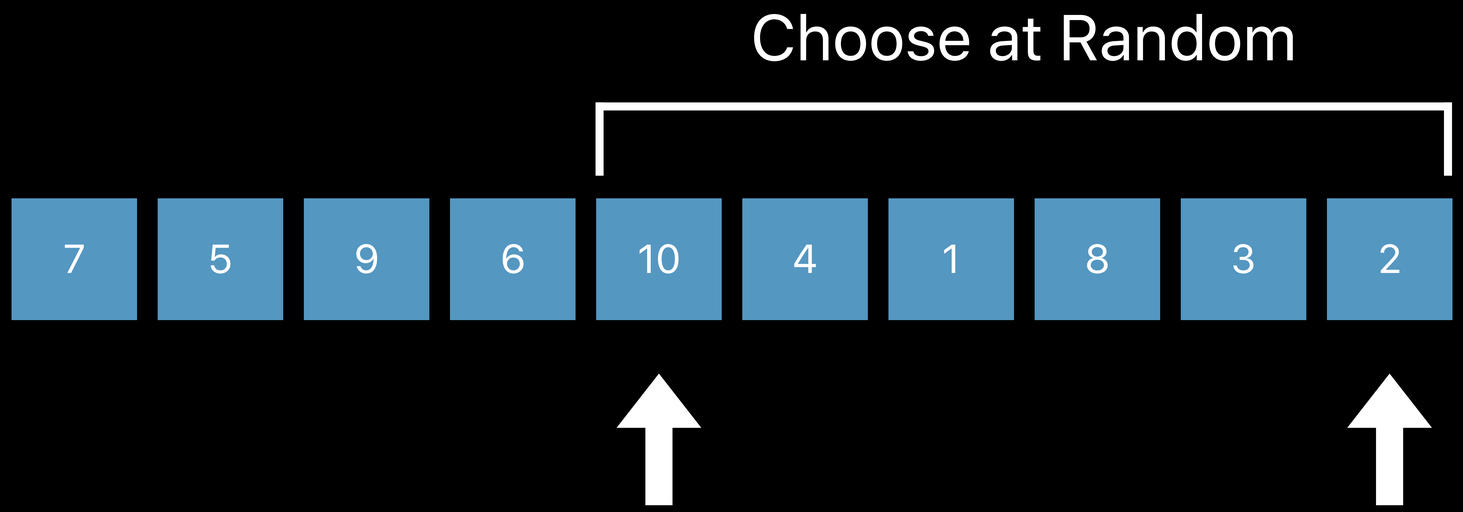
Fisher-Yates Shuffle



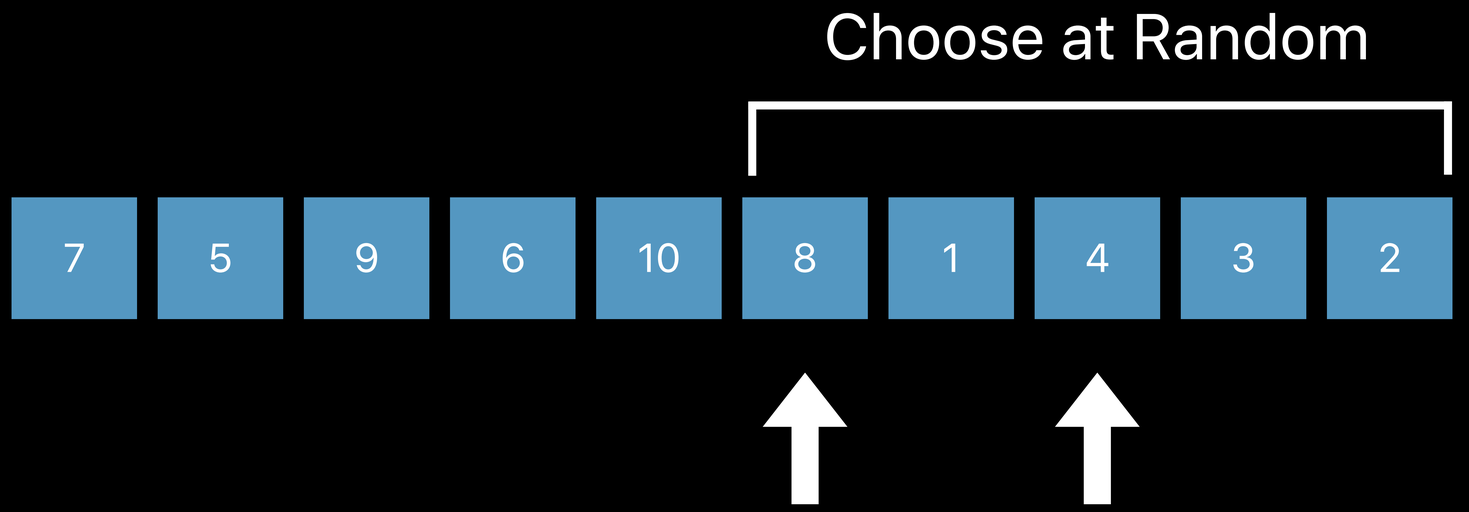
Fisher-Yates Shuffle



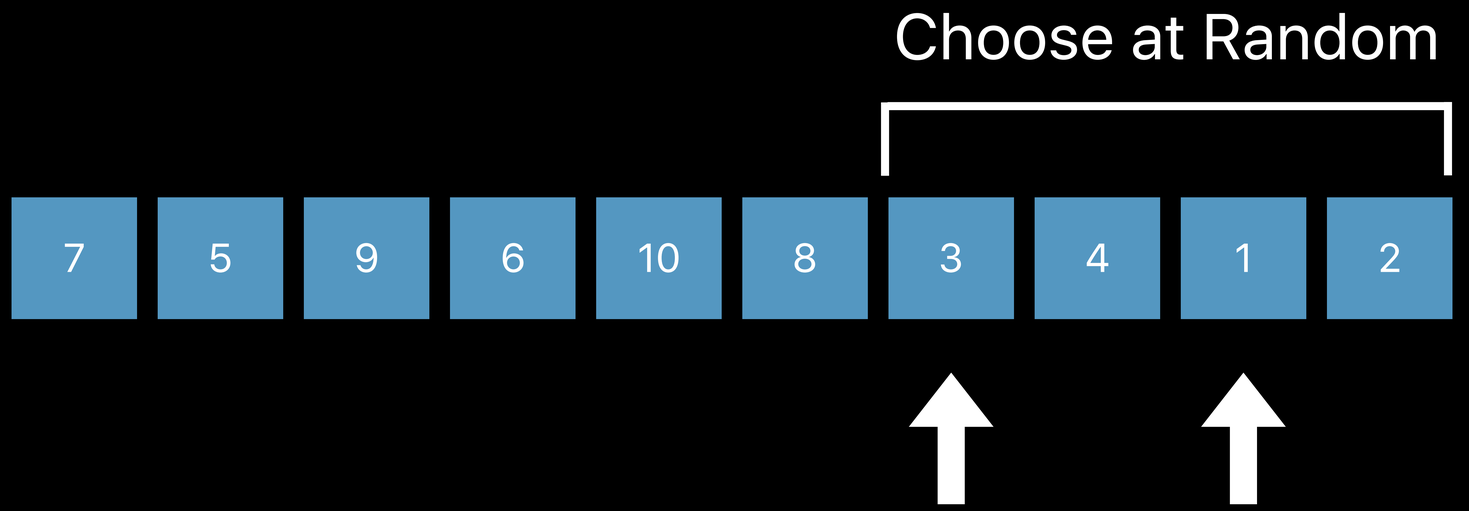
Fisher-Yates Shuffle



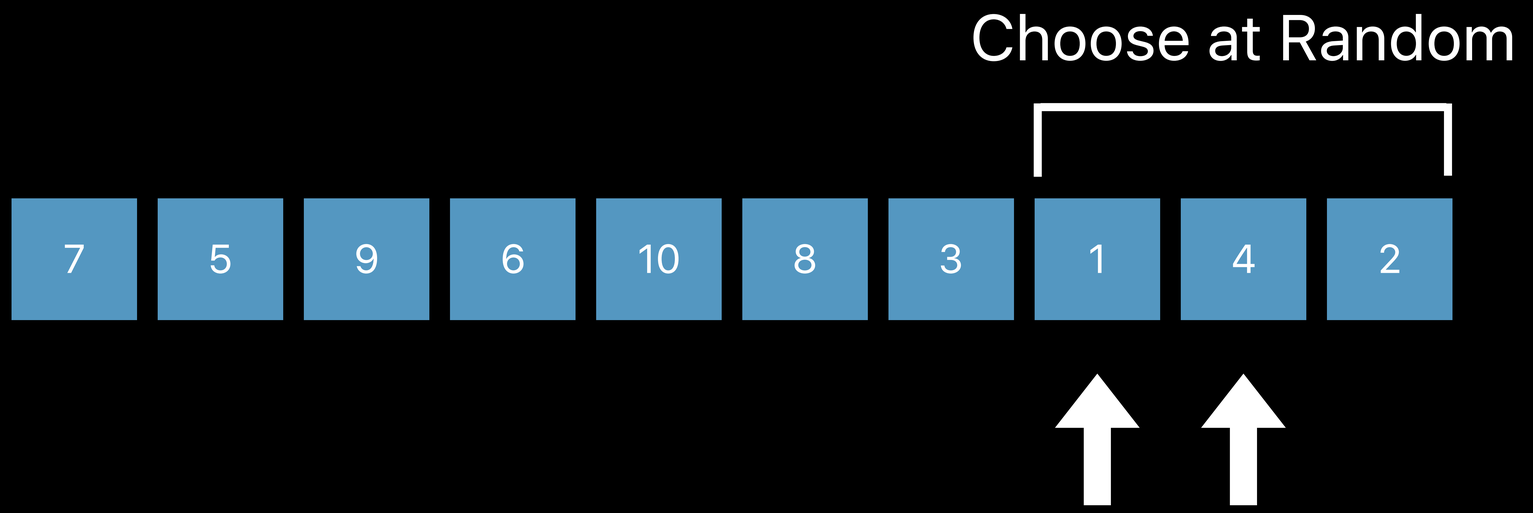
Fisher-Yates Shuffle



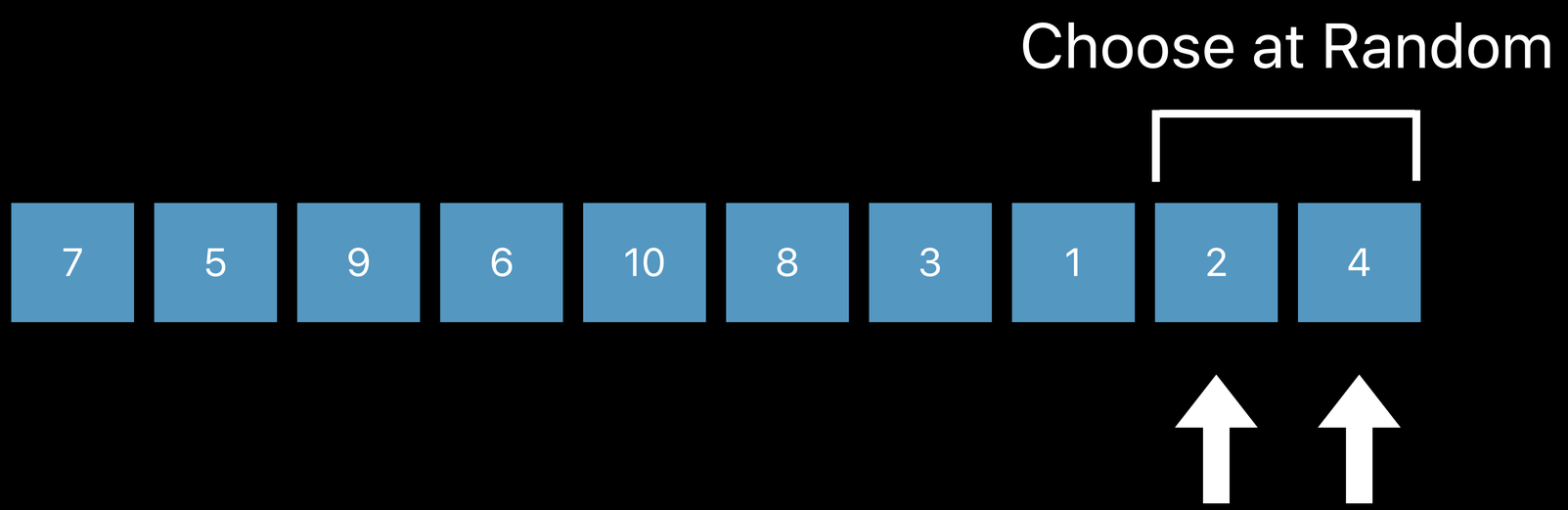
Fisher-Yates Shuffle



Fisher-Yates Shuffle



Fisher-Yates Shuffle



Fisher-Yates Shuffle


```
extension ???Collection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Fisher-Yates Shuffle


```
extension ???Collection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Fisher-Yates Shuffle


```
extension ???Collection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Fisher-Yates Shuffle


```
extension ShuffleCollection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Fisher-Yates Shuffle


```
extension ShuffleCollection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Keep Distinct Capabilities Separate

`shuffle()` is using random access and element mutation

`UnsafeBufferPointer` provides random access without element mutation

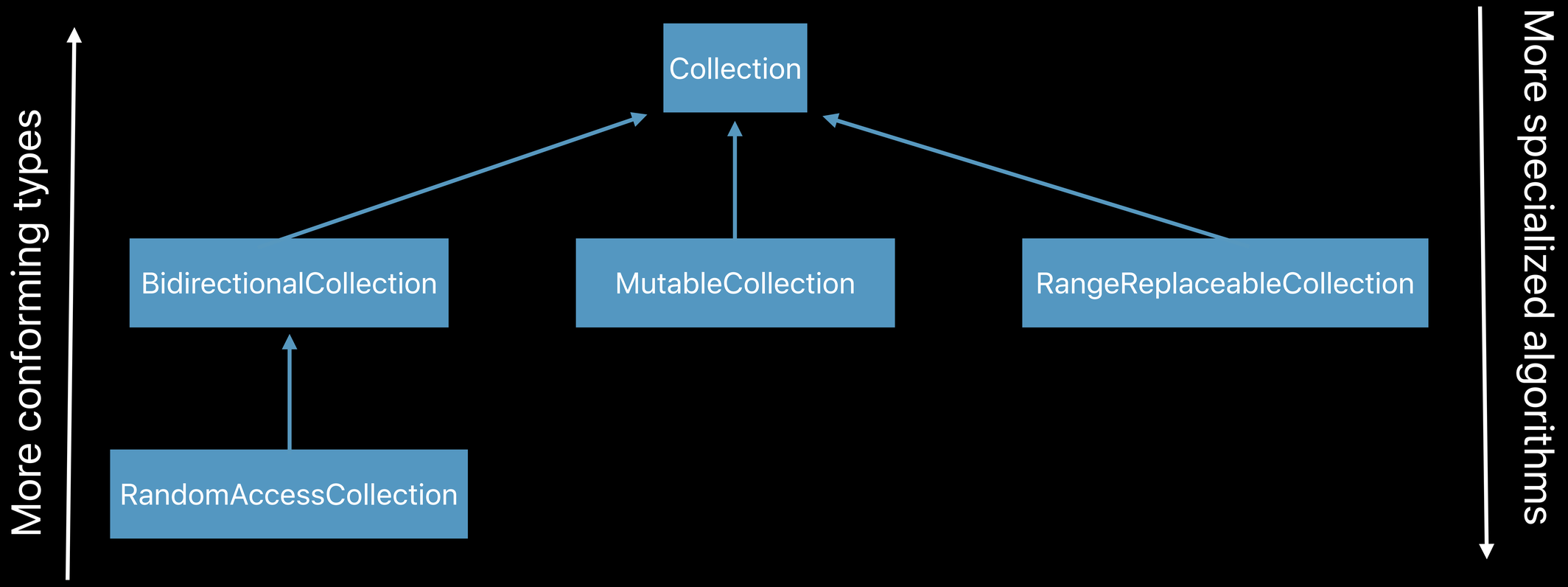
`SinglyLinkedList` provides element mutation but not random access

```
protocol RandomAccessCollection: BidirectionalCollection {
  func index(_ position: Index, offsetBy n: Int) -> Index
  func distance(from start: Index, to end: Index) -> Int
}
protocol MutableCollection: Collection {
  subscript (index: Index) -> Element { get set }
  mutating func swapAt(_: Index, _: Index) { }
}
```

Clients Can Compose Multiple Protocols

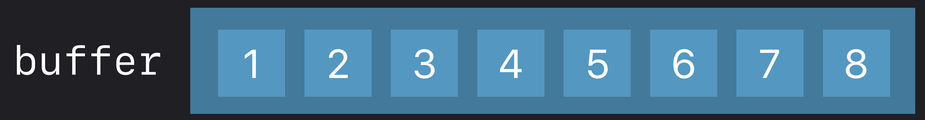
```
extension RandomAccessCollection where Self: MutableCollection {
  mutating func shuffle() {
    let n = count
    guard n > 1 else { return }
    for (i, pos) in indices.dropLast().enumerated() {
      let otherPos = index(startIndex, offsetBy: Int.random(in: i..<n))
      swapAt(pos, otherPos)
    }
  }
}
```

Collection Protocol Hierarchy



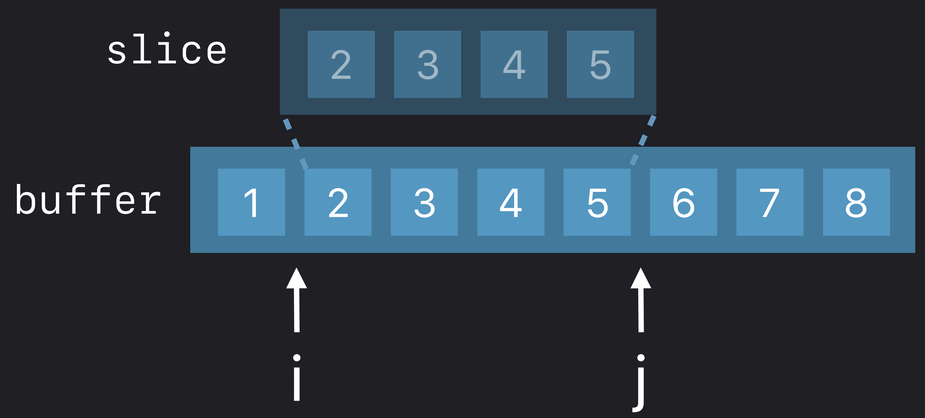
Conditional Conformance

```
// Slicing Collections
```



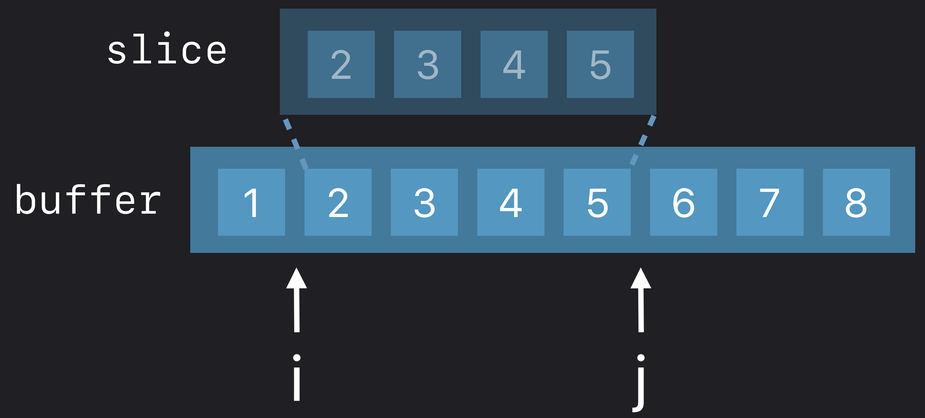

```
// Forming a slice
let slice = buffer[i..<j]
```

```
// Slicing Collections
```




```
// Forming a slice
let slice = buffer[i..<j]
```

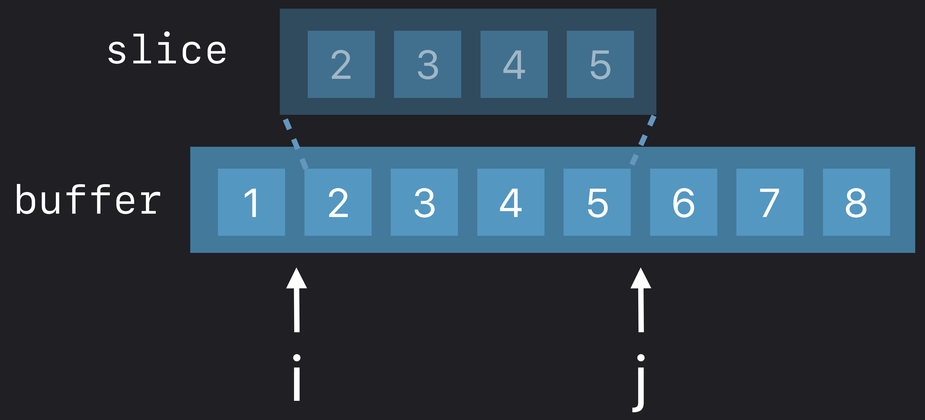
```
// Slicing Collections
```




```
// Forming a slice
let slice = buffer[i..<j]
```

```
struct Slice<Base: Collection>: Collection { … }
```

```
// Slicing Collections
```




```
// Forming a slice
let slice = buffer[i..<j]
```

```
struct Slice<Base: Collection>: Collection { … }
```

```
// Forward search
buffer.index(where: { $0.isEven })
slice.index(where: { $0.isEven })

// Backward search
buffer.lastIndex(where: { $0.isEven })
slice.lastIndex(where: { $0.isEven })
```

No member named lastIndex(where:) on type ‘Slice<Buffer<Element>>’

Conformance to BidirectionalCollection

```
extension Slice: BidirectionalCollection {
  func index(before idx: Index) -> Index { return base.index(before: idx) }
}
```

No member named index(before:) on type ‘Base’

Conditional Conformance to BidirectionalCollection

Conformance depends on additional requirements

```
extension Slice: BidirectionalCollection where Base: BidirectionalCollection {
  func index(before idx: Index) -> Index { return base.index(before: idx) }
}

extension Slice: RandomAccessCollection where Base: RandomAccessCollection {
  func index(_ idx: Index, offsetBy n: Int) -> Index { return base.index(idx, offsetBy: n) }
  func distance(from s: Index, to e: Index) -> Int { return base.distance(from: s, to: e) }
}
```

Ranges

```
let doubleRange = 2.71828 ..< 3.14159
doubleRange.contains(3.0)

let intRange = 17 ..< 42
intRange.contains(25)

for i in intRange { … }
```

Ranges

```swift
let doubleRange = 2.71828 ..< 3.14159
doubleRange.contains(3.0)

let intRange = 17 ..< 42
intRange.contains(25)

for i in intRange { … }
```

```swift
struct Range<Bound: Comparable> {
  let lowerBound: Bound
  let upperBound: Bound
  func contains(_ value: Bound) -> Bool { … }
}
```

```swift
struct CountableRange<Bound: Strideable>
  where Bound.Stride: SignedInteger {
  let lowerBound: Bound
  let upperBound: Bound
  func contains(_ value: Bound) -> Bool { … }
}

extension CountableRange: RandomAccessCollection { … }
```

Range as a Collection

Range conditionally conforms to RandomAccessCollection

```
extension Range: RandomAccessCollection
      where Bound: Strideable, Bound.Stride: SignedInteger {
  // …
}
```

Conditional conformance of type 'Range<Bound>' to protocol 'RandomAccessCollection' does not imply conformance to inherited protocol 'BidirectionalCollection'

Range as a Collection

Range conditionally conforms to RandomAccessCollection

```
extension Range: Collection, BidirectionalCollection, RandomAccessCollection
     where Bound: Strideable, Bound.Stride: SignedInteger {
  // …
}
```

Range as a Collection

Range conditionally conforms to RandomAccessCollection

```
extension Range: Collection, BidirectionalCollection, RandomAccessCollection
     where Bound: Strideable, Bound.Stride: SignedInteger {
  // …
}
```

CountableRange is a convenient alias for Ranges that are Collections

```
typealias CountableRange<Bound: Strideable> = Range<Bound>
   where Bound.Stride: SignedInteger
```

Range as a Collection

Range conditionally conforms to RandomAccessCollection

```
extension CountableRange: Collection, BidirectionalCollection, RandomAccessCollection {
  // …
}
```

CountableRange is a convenient alias for Ranges that are Collections

```
typealias CountableRange<Bound: Strideable> = Range<Bound>
   where Bound.Stride: SignedInteger
```

Recursive Constraints

Recursive Constraints

```
protocol Collection {
  // …
  associatedtype SubSequence: Collection
}
```

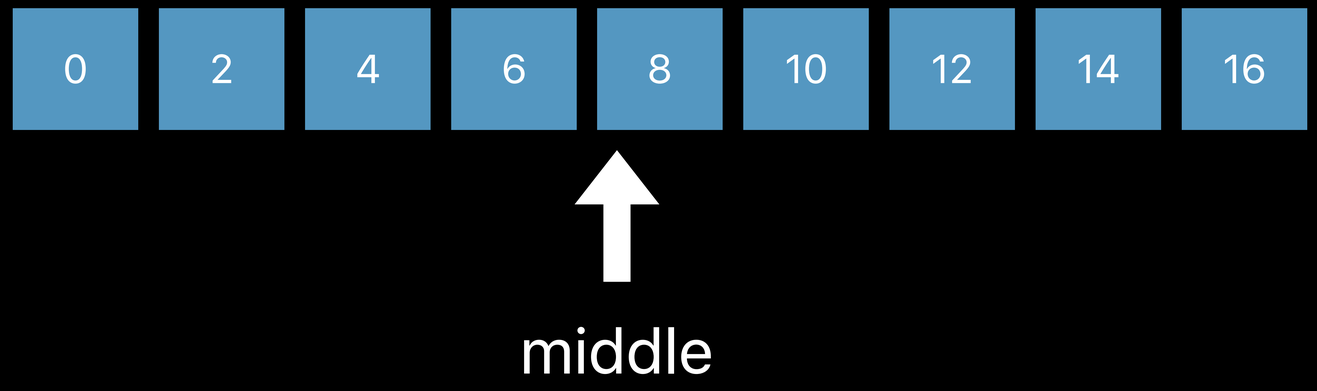
Insertion into a Sorted Collection

Where should we insert a new value in an already-sorted array?

```
var array = [0, 2, 4, 6, 8, 10, 12, 14, 16]
array.insert(11, at: array.sortedInsertionPoint(of: 11))
print(array) // [0, 2, 4, 6, 8, 10, 11, 12, 14, 16]
```

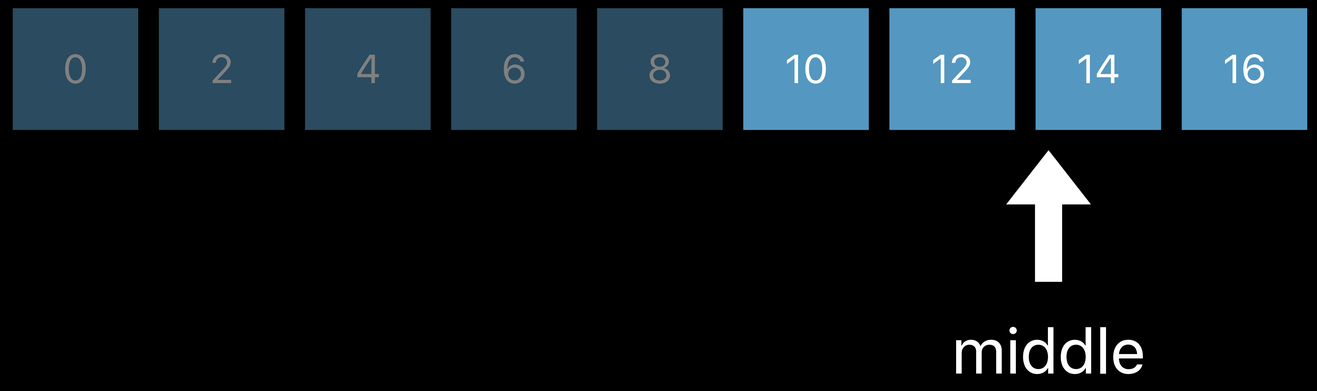
Divide-and-Conquer Binary Search

```
sortedInsertionPoint(of: 11)
```



Divide-and-Conquer Binary Search

```
sortedInsertionPoint(of: 11)
```

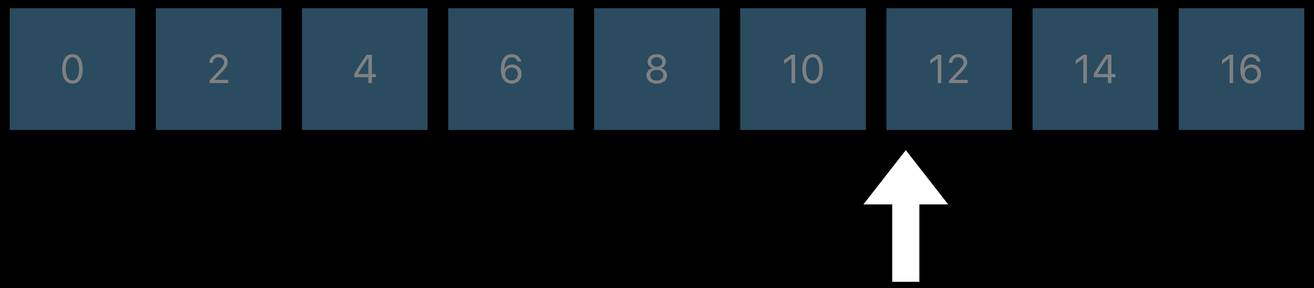


Divide-and-Conquer Binary Search

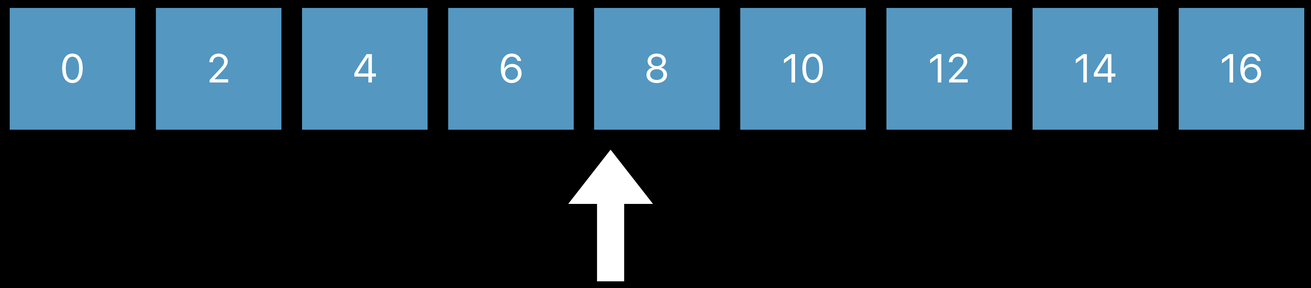
```
sortedInsertionPoint(of: 11)
```

Divide-and-Conquer Binary Search

```
sortedInsertionPoint(of: 11)
```

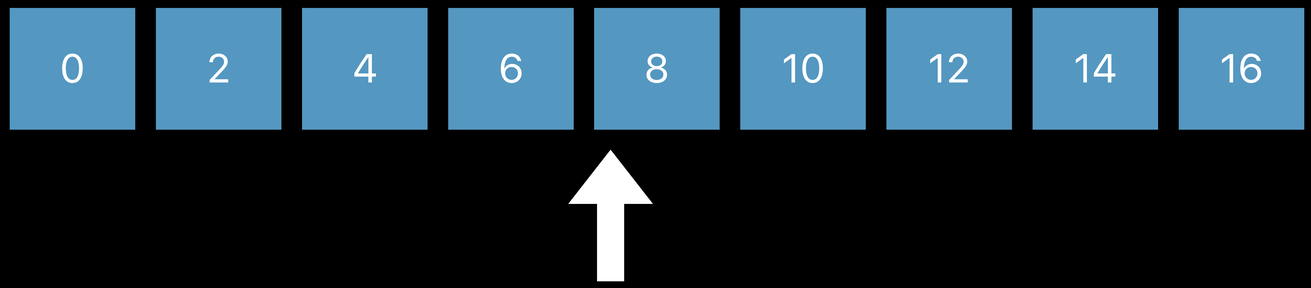


Divide-and-Conquer Binary Search




```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
    } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
    }
  }
}
```

Divide-and-Conquer Binary Search




```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
    } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
    }
  }
}
```

Divide-and-Conquer Binary Search


```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
    } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
    }
  }
}
```

Adding Slicing to Collection

```
extension Collection {
  subscript (bounds: Range<Index>) -> Slice<Self> {
    return Slice(base: self, bounds: bounds)
  }
}
```

Custom Slice Types

```
extension Collection {
  subscript (bounds: Range<Index>) -> Slice<Self> {
    return Slice(base: self, bounds: bounds)
  }
}
```

```
extension String: Collection {
  subscript (bounds: Range<Index>) -> Substring { … }
}

extension Range: Collection where Bound: Strideable, Bound.Stride: SignedInteger {
  subscript (bounds: Range<Bound>) -> Range<Bound> { return bounds }
}
```

Slicing Requirements

```
protocol Collection {
  // …
  associatedtype SubSequence
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Slicing Requirements

```
protocol Collection {
  // …
  associatedtype SubSequence
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
extension String: Collection {
  typealias SubSequence = Substring
  subscript (bounds: Range<Index>) -> SubSequence { … }
}
```

Slicing Requirements

```
protocol Collection {
  // …
  associatedtype SubSequence
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
extension Range: Collection where Bound: Strideable, Bound.Stride: SignedInteger {
  typealias SubSequence = Range<Bound>
  subscript (bounds: Range<Bound>) -> SubSequence { return bounds }
}
```

Associated Type Defaults

```
protocol Collection {
  // …
  associatedtype SubSequence = Slice<Self>
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Default Implementation

```
protocol Collection {
  // …
  associatedtype SubSequence = Slice<Self>
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
extension Collection {
  subscript (bounds: Range<Index>) -> Slice<Self> {
    return Slice(base: self, bounds: bounds)
  }
}
```

Default Implementation

```
protocol Collection {
  // …
  associatedtype SubSequence = Slice<Self>
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
extension Collection where Self.SubSequence == Slice<Self> {
  subscript (bounds: Range<Index>) -> Slice<Self> {
    return Slice(base: self, bounds: bounds)
  }
}
```

What Does a SubSequence Do?

```
protocol Collection {
  // …
  associatedtype SubSequence = Slice<Self>
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

What Capabilities Do Algorithms Depend On?

```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
   } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
   }
  }
}
```

What Capabilities Do Algorithms Depend On?

```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
   } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
   }
  }
}
```

Self.SubSequence

What Capabilities Do Algorithms Depend On?

```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
   } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
   }
  }
}
```

What Capabilities Do Algorithms Depend On?

```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
   } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
   }
  }
}
```

Self.SubSequence

Self.SubSequence.Element

What Capabilities Do Algorithms Depend On?

```
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    if isEmpty { return startIndex }
    let middle = index(startIndex, offsetBy: count / 2)
    if value < self[middle] {
      return self[..<middle].sortedInsertionPoint(of: value)
   } else {
      return self[index(after: middle)...].sortedInsertionPoint(of: value)
   }
  }
}
```

Self.SubSequence.Index

Constraining SubSequence

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence



  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Recursive Constraints

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection



  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Associated Type Where Clauses

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element


  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Associated Type Where Clauses

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element,
          SubSequence.Index == Index

  subscript (range: Range<Index>) -> SubSequence { get }
}
```

Can You Slice a SubSequence?

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element,
          SubSequence.Index == Index

  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
self[bounds]
```

Self.SubSequence

Can You Slice a SubSequence?

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element,
          SubSequence.Index == Index

  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
self[bounds][bounds]
```

```
Self.SubSequence.SubSequence
```

Can You Slice a SubSequence?

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element,
          SubSequence.Index == Index

  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
self[bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds]
```

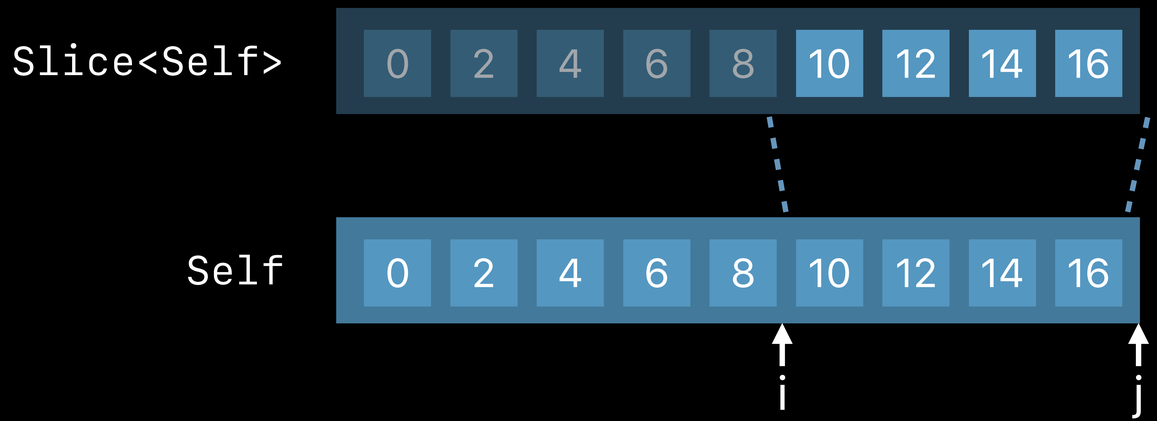
```
Self.SubSequence.SubSequence.SubSequence.SubSequence.SubSequence.SubSequence.SubSequence.SubSeque
```

```
// Non-Recursive Divide-and-Conquer
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    var slice: SubSequence = self[...]

    while !slice.isEmpty {
      let middle = slice.index(slice.startIndex, offsetBy: slice.count / 2)
      if value < slice[middle] {
        slice = slice[..<middle]
      } else {
        slice = slice[index(after: middle)...]
      }
    }
  Self.SubSequence   Self.SubSequence.SubSequence

    return slice.startIndex
  }
}
```

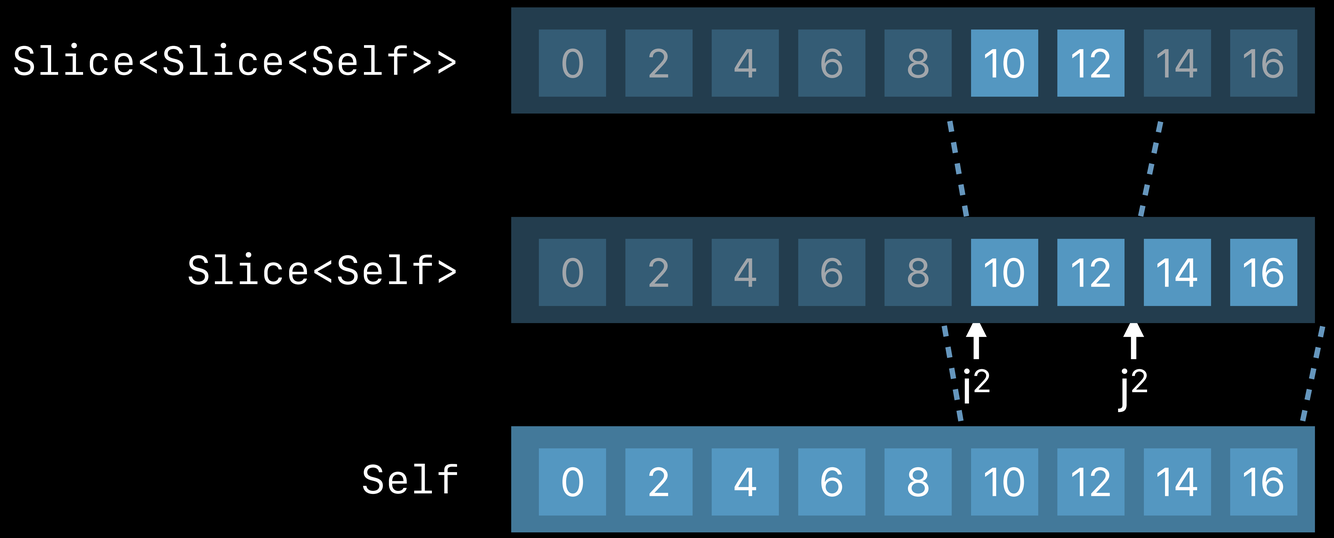
cannot assign value of type `Self.SubSequence.SubSequence` to type `Self.SubSequence`

Slicing a Slice

Slicing a Slice



Slicing a Slice



Slicing a Slice

“Tying Off” Recursion

```
protocol Collection {
  associatedtype Element
  associatedtype Index
  associatedtype SubSequence: Collection
    where SubSequence.Element == Element,
          SubSequence.Index == Index,
          SubSequence.SubSequence == SubSequence
  subscript (range: Range<Index>) -> SubSequence { get }
}
```

```
self[bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds][bounds]
```

Self.SubSequence

```swift
// Non-Recursive Divide-and-Conquer
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    var slice: SubSequence = self[...]

    while !slice.isEmpty {
      let middle = slice.index(slice.startIndex, offsetBy: slice.count / 2)
      if value < slice[middle] {
        slice = slice[..<middle]
      } else {
        slice = slice[index(after: middle)...]
      }
    }
  Self.SubSequence    Self.SubSequence
    return slice.startIndex
  }
}
```

```swift
// Non-Recursive Divide-and-Conquer
extension RandomAccessCollection where Element: Comparable {
  func sortedInsertionPoint(of value: Element) -> Index {
    var slice: SubSequence = self[...]

    while !slice.isEmpty {
      let middle = slice.index(slice.startIndex, offsetBy: slice.count / 2)
      if value < slice[middle] {
        slice = slice[..<middle]
      } else {
        slice = slice[index(after: middle)...]
      }
    }

    return slice.startIndex
  }
}
```

Tracking Protocol Inheritance

Protocols can strengthen requirements with `where` clauses

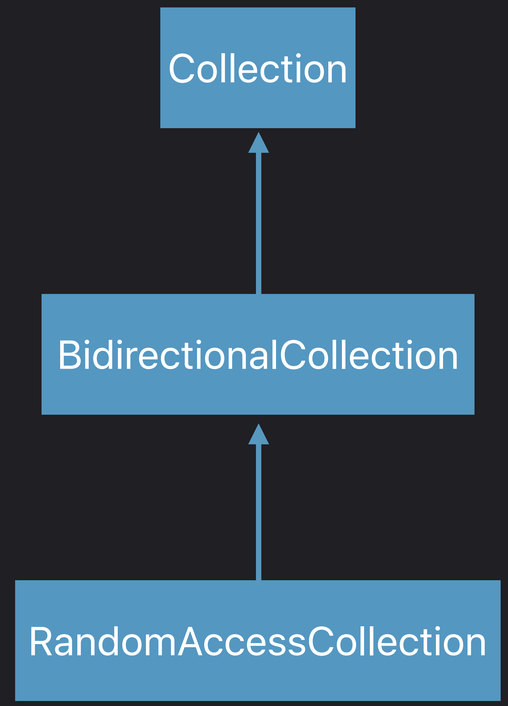
```
protocol BidirectionalCollection: Collection
    where SubSequence: BidirectionalCollection {
  // …
}
```

Recursive requirements can “stack”

```
protocol RandomAccessCollection: BidirectionalCollection
    where SubSequence: RandomAccessCollection {
  // …
}
```

Recursive Constraints and Conditional Conformance

```
protocol Collection {
  // …
  associatedtype SubSequence: Collection = Slice<Self>
}
protocol BidirectionalCollection: Collection
    where SubSequence: BidirectionalCollection {
  // …
}
protocol RandomAccessCollection: BidirectionalCollection
    where SubSequence: RandomAccessCollection {
  // …
}
```



Generics and Classes

Class Inheritance

```
class Vehicle { … }
class Taxi: Vehicle { … }
class PoliceCar: Vehicle { … }

extension Vehicle {
  func drive() { … }
}

taxi.drive()
```

Liskov Substitution Principle

If S is a subtype of T, any instance of type T can be replaced by an instance of S

```
class Vehicle { … }
class Taxi: Vehicle { … }
class PoliceCar: Vehicle { … }

extension Vehicle {
  func drive() { … }
}

taxi.drive()
```

https://en.wikipedia.org/wiki/Barbara_Liskov

Protocol Conformances and Classes

```
protocol Drivable {
  func drive()
}

extension Vehicle: Drivable { }
```

Protocol Conformances and Classes

Protocol conformances are inherited by subclasses

A single conformance must work for *all* subclasses

```
extension Vehicle: Drivable { }

extension Drivable {
  func sundayDrive() {
    if Date().isSunday { drive() }
  }
}

PoliceCar().sundayDrive()
```

```swift
// Initializer Requirements

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

extension Decodable {
  static func decode(from decoder: Decoder) throws -> Self {
    return try self.init(from: decoder)
  }
}
```

```swift
// Initializer Requirements

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

extension Decodable {
  static func decode(from decoder: Decoder) throws -> Self {
    return try self.init(from: decoder)
  }
}

class Vehicle: Decodable {
  init(from decoder: Decoder) throws { … }
}

Taxi.decode(from: decoder)                    // produces a Taxi
```

Which Initializer Gets Called?

```
class Vehicle: Decodable {
  init(from decoder: Decoder) throws { … }
}

class Taxi: Vehicle {
  var hourlyRate: Double
}

Taxi.decode(from: decoder)
```

Which Initializer Gets Called?

```
class Vehicle: Decodable {
  init(from decoder: Decoder) throws { … }
}

class Taxi: Vehicle {
  var hourlyRate: Double
}

Taxi.decode(from: decoder)
```

Which Initializer Gets Called?

```
class Vehicle: Decodable {
  init(from decoder: Decoder) throws { … }
}

class 0.01 Vehicle {
  var hourlyRate: Double
}
    60.00    3,300.00

Taxi.decode(from: decoder)
```

Which Initializer Gets Called?

```
class Vehicle: Decodable {
  init(from decoder: Decoder) throws { … }
}

class Taxi: Vehicle {
  var hourlyRate: Double
}


Taxi.decode(from: decoder)
```

Initializer requirement 'init(from:)' can only be satisfied by a 'required' initializer in the definition of non-final class 'Vehicle'

Which Initializer Gets Called?

Required initializers must be implemented by all subclasses

```
class Vehicle: Decodable {
  required init(from decoder: Decoder) throws { … }
}

class Taxi: Vehicle {
  var hourlyRate: Double
}

Taxi.decode(from: decoder)
```

Which Initializer Gets Called?

Required initializers must be implemented by all subclasses

```
class Vehicle: Decodable {
  required init(from decoder: Decoder) throws { … }
}

class Taxi: Vehicle {
  var hourlyRate: Double
  required init(from decoder: Decoder) throws { … }
}
Taxi.decode(from: decoder)
```

Final Classes Have No Subclasses

`final` classes are exempt from these rules

Use `final` when your class is not customizable *through inheritance*

```
final class EndOfTheLine: Decodable {
  init(from decoder: Decoder) { … }                    // 'required' is not required
}
```

Summary

Swift’s generics provide code reuse while maintaining static type information

Let the push-pull between generic algorithms and conforming types guide design

- Protocol inheritance captures specialized capabilities of some conforming types
- Conditional conformance provides composition for those capabilities

Apply the Liskov Substitution Principle when working with classes

More Information

https://developer.apple.com/wwdc18/406

Everyday Algorithms	Hall 3	Thursday 2:00PM
Using Collections Effectively	Hall 2	Friday 9:00AM


WWDC18