
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

W1: Swift Collection Protocols

- Be at least comfortable in Swift (the stronger the better)
- Read our Linked List tutorial https://www.raywenderlich.com/144083/swift-algorithm-club-swift-linked-list-data-structure
- Understand the basics of Big-O Notation https://github.com/raywenderlich/swift-algorithm-club/blob/master/Big-O%20Notation.markdown
- Be acquainted with data structures and sorting algorithms

W1: Swift Collection Protocols Workshop

A stroll down an alleyway of the Swift standard library. In this workshop, you'll learn about the Swift collection protocols which powers much of the standard library.

You'll walk away with advanced knowledge and techniques that will augment your daily Swift development and impress your interviewers.

Swift Collection Protocols Workshop: Demo 1

By Kelvin Lau & Vincent Ngo

In this demo, you will get acquainted with the Linked List.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Checkpoint1\starter**.

1) Implement the Linked List

In the playground's **Sources** folder, create a new file named **LinkedList.swift**. Add the following inside the file:

```
public enum LinkedList<T> {
  case end
  indirect case node(value: T, next: LinkedList)
}
```

2) Adding new values

Write the following just below the two cases:

```
public mutating func push(_ value: T) {
  self = .node(value: value, next: self)
}

extension LinkedList: CustomStringConvertible {
  public var description: String {
    guard case let .node(value, next) = self else { return "end" }
    return "\(value) -> \(next.description)"
  }
```

```
}
```

3) Creating a Linked List

Navigate to the playground page named **1. LinkedList** and write the following at the top of the file:

```
example(of: "building a LL") {
  var list = LinkedList<Int>.end
  list.push(3)
  list.push(2)
  list.push(1)
  print(list)
}
```

You should see the following output in the console:

```
---Description of: building a LL---
1 -> 2 -> 3 -> end
```

4) Challenge

Given a Linked List, compute the `count` of the list. The main goal here is to get fluent with associated values. There are many ways to access the associated values of an enum case:

```
if case let .node(value, next) = list {
  // ...
}
```

```
guard case let .node(value, next) = list else { ... }
// ...
```

```
switch list {
case let .node(value, next): // ...
case .end:
}
```

```
while case let .node(value, next) = list {
  // ...
}
```

5) That's it!

Congrats, at this time you should have a good understanding of linked lists! It's

time to move onto the `Sequence` protocol.

Swift Collection Protocols Workshop: Demo 2

By Kelvin Lau & Vincent Ngo

In this demo, you will learn about Sequence and it's capabilities.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: You may continue with your current playground, or with the starter project in **Checkpoint2\starter**.

1) Implementing an Iterator

Navigate to **2. Sequence**. Inside the **Sources** folder, create a new file named **LinkedListIterator.swift**. Write the following inside the file:

```
public struct LinkedListIterator<T> {
  public var current: LinkedList<T>

  public init(current: LinkedList<T>) {
    self.current = current
  }
}

// MARK: - IteratorProtocol
extension LinkedListIterator: IteratorProtocol {

  // return the next value of the iterator
  public mutating func next() -> T? {

    // pattern match against node case, extract associated values
    guard case let .node(value, next) = current else { return nil }

    // set current to next for next iteration
    current = next
    return value
```

```
  }
}
```

2) Using an Iterator

Inside the **2. Sequence** playground page, write the following:

```
var list = LinkedList<Int>.end
list.push(3)
list.push(2)
list.push(1)

example(of: "using iterator directly") {
  var iterator = LinkedListIterator(current: list)

  while let next = iterator.next() {
    print(next)
  }
}
```

3) Adopting the Sequence protocol

In the **Sources** folder for **2. Sequence**, create a new file named **LinkedList+Sequence.swift**. Write the following inside the file:

```
extension LinkedList: Sequence {
  public func makeIterator() -> LinkedListIterator<T> {
    return LinkedListIterator(current: self)
  }
}
```

4) Looping using Sequence

Navigate back to **2. Sequence** and write the following at the bottom of the file:

```
example(of: "looping") {
  // for-in loop
  var forIn = ""
  for value in list {
    forIn += "\(value) "
  }
  print(forIn)

  // map
  let array = list.map { $0 }
  print(array)

  // filter
```

```
    let divisibleBy2 = list.filter { $0 % 2 == 0 }
    print(divisibleBy2)

    // reduce
    let sum = list.reduce(0, +)
    print(sum)
  }
```

You can check out all the other capabilities by using **CMD + SHIFT + 0** and searching for `Sequence`.

5) Challenge

Given a linked list, find the *middle* value of the list.

Bonus: Can you implement it in an extension of the `Sequence` protocol? Here's the interface to get you started:

```
extension Sequence {

  var middle: Element? {
    // return the middle element
  }
}
```

6) That's it!

Congrats, at this time you should have a good understanding of the `Sequence` protocol! Take a break, and then it's time to move onto learning about `Collection`.

Swift Collection Protocols Workshop: Demo 3

By Kelvin Lau & Vincent Ngo

In this demo, you will learn about `Collection` and it's capabilities.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: You may continue with your current playground, or with the starter project in **Checkpoint3\starter**.

1) Conforming to Collection

Navigate to the **Sources** folder of the playground page named **3. Collection**. Create a new file named **LinkedList+Collection.swift**. Write the following inside the file:

```
extension LinkedList: Collection {

  public var startIndex: Int {
    return 0
  }

  public var endIndex: Int {
    var current = self
    var count = 0

    // pattern match against node case and extract associated values
    while case .node(_, let next) = current {
      count += 1
      current = next
    }

    return count
  }
```

```
  public subscript(position: Int) -> T {
    var current = self
    var count = 0

    while case let .node(value, next) = current {
      if count == position {
        return value
      }

      count += 1
      current = next
    }

    fatalError("Index out of bounds")
  }

  public func index(after i: Int) -> Int {
    return i + 1
  }
}
```

2) Trying it out

Write the following inside the playground page for `Collection`:

```
var list = LinkedList<Int>.end
list.push(3)
list.push(2)
list.push(1)

example(of: "using collection") {
  print("List: \(list)")
  print("First element: \(list[list.startIndex])")
}
```

You should see the following inside the console:

```
undefined
```

3) Update example to use RefLinkedList

Find the line where you initialized the list. Update it to the following:

```
var list = LinkedList<Int>()
```

4) Challenge

Create a function that checks if a collection is in another collection. If it is, return

the **start index** that matches the collection. Here's the interface to get you started:

```
extension Collection where Element: Equatable {

  func index<Elements: Collection>(for elements: Elements)
    -> Index? where Elements.Element == Element {
    // return an index, or nil
  }
}
```

5) That's it!

Congrats, at this time you should have a good understanding of the `Collection` protocol!

Swift Collection Protocols Workshop: Demo 4

By Kelvin Lau & Vincent Ngo

In this demo, you will learn about `BidirectionalCollection` and it's capabilities.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: You may continue with your current playground, or with the starter project in **Checkpoint4\starter**.

1) Improving Linked-List

Before you conform to `index(before:)`, you need to improve your Linked-List. With the current implementation of a **singly** linked-list. You will have to traverse the whole list every time just to get the index **before**.

Instead you should use a **doubly** linked-list, where you can simply reference the **previous** node.

Improving your implementation from *O(n)* to _O(1)_.

2) Update LinkedListNode to include previous pointer.

For a double linked list, a node now has two pointers that refer to it's previous and next node. Navigate to **RefLinkedList.swift** and add a property to store the `previous` node in `LinkedListNode`:

```
public class LinkedListNode<Element> {
  public var data: Element
```

```
  public var next: LinkedListNode?
  weak public var previous: LinkedListNode?

  public init(_ data: Element, next: LinkedListNode? = nil, previous:
LinkedListNode? = nil) {
    self.data = data
    self.next = next
    self.previous = previous
  }
}
```

3) Update Description

Update the description to include an arrow back to reflect that it's a doubly-linked list:

```
extension LinkedListNode: CustomStringConvertible {

  public var description: String {
    guard let next = next else {
      return "\(data)"
    }
    return "\(data) <-> " + String(describing: next) + " "
  }
}
```

4) Update Push

Every time you add a new node to the doubly-linked list you must also update the `previous` pointer.

```
public mutating func push(_ value: Element) {
  let previousHead = head // 1
  head = Node(value, next: previousHead) // 2

  if tail == nil { // 1
    tail = head
  } else {
    previousHead?.previous = head // 2
  }
}
```

1. Sticking to head first insertion, you are going to store the previous head.
2. You are going to create a new node for the given value, and set it's next pointer to the `previousHead`.
3. If the tail is nil, that means this is the first element added.
4. Update the `previous` pointer of `previousHead` node.

5) Update Index Comparable == method

Don't forget the previous pointer:

```
static public func ==(lhs: Index, rhs: Index) -> Bool {
  switch (lhs.node, rhs.node) {
  case let (left?, right?):
    return left.next === right.next && left.previous === right.previous
  case (nil, nil):
    return true
  default:
    return false
  }
}
```

6) Reversing a Collection

Navigate to the **Sources** folder and go to **RefLinkedList.swift**. Add a print statement in the `subscript` function like so:

```
public subscript(position: Index) -> Element {
  print("position: \(position)")
  return position.node!.data
}
```

7) Example of calling reversed on a collection

Calling the `reversed` function on `Collection`. Write the following inside the playground page for `BidirectionalCollection`:

```
example(of: "using BidirectionalCollection") {
  var list = RefLinkedList<Int>()
  for i in 0...9 {
    list.push(i)
  }

  print(list)

  print("Reverse the list, and get the first 4 elements:")
  for value in list.reversed().suffix(4) {
    print(value)
  }
}
```

You should see the following inside the console:

```
---Description of: using BidirectionalCollection---
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
Reverse the list, and get the first 4 elements:
position: Index(node: Optional(9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1
-> 0         ))
position: Index(node: Optional(8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
))
position: Index(node: Optional(7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
))
position: Index(node: Optional(6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0      ))
position: Index(node: Optional(5 -> 4 -> 3 -> 2 -> 1 -> 0     ))
position: Index(node: Optional(4 -> 3 -> 2 -> 1 -> 0    ))
position: Index(node: Optional(3 -> 2 -> 1 -> 0   ))
position: Index(node: Optional(2 -> 1 -> 0  ))
position: Index(node: Optional(1 -> 0 ))
position: Index(node: Optional(0))
6
7
8
9
```

Notice when calling the `reversed()` function on a collection that it goes through all the elements.

8) Conforming to BidirectionalCollection

Navigate to the **Sources** folder and go to **RefLinkedList.swift**. Change `RefLinkedList` to conform to `BidirectionalCollection` instead of `Collection` like so:

```
extension RefLinkedList: BidirectionalCollection {
  ...
}
```

9) Implementing index(before:)

Instead of looping through the singly linked-list, with a doubly-linked list, you can simply reference the `previous` node. Add the following right after `index(after:)`:

```
public func index(before i: Index) -> Index {
  if i == endIndex { // 1
    return Index(node: tail)
  } else { // 2
    return Index(node: i.node?.previous)
  }
}
```

1. If the current index is equal to the `endIndex`, return the `Index` that includes the `tail` node.

2. Return the Index of the current node's `previous` node.

10) Trying it out

Navigate back to `BidirectionalCollection` playground page and notice the console once it reruns:

```
---Description of: using BidirectionalCollection---
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
Reverse the list, and get the first 4 elements:
position: Index(node: Optional(6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0      ))
6
position: Index(node: Optional(7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
))
7
position: Index(node: Optional(8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
))
8
position: Index(node: Optional(9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1
-> 0         ))
9
```

11) Challenge

Given a word, find out if it is a palindrome.

Bonus: Can you implement it in an extension of the `BidirectionalCollection` protocol? Here's the interface to get you started:

```
extension BidirectionalCollection where Element: Equatable {

  var isPalindrome: Bool {
    // return true or false
  }
}
```

12) That's it!

Congrats, at this time you should have a good understanding of the `BidirectionalCollection` protocol! Let's now move on to `MutableCollection`.

Swift Collection Protocols Workshop: Demo 5

By Kelvin Lau & Vincent Ngo

In this demo, you will learn about `MutableCollection` and it's capabilities.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: You may continue with your current playground, or with the starter project in **Checkpoint5\starter**.

1) Conforming to MutableCollection

```
extension RefLinkedList: BidirectionalCollection, MutableCollection {
  ...
}
```

2) Read and Write Subscript

```
public subscript(position: Index) -> Element {
   get {
     guard let node = position.node else {
       fatalError("Node does not exist")
     }
     return node.data
   }

   set {
     guard let node = position.node else {
       fatalError("Node does not exist")
     }
     node.data = newValue
   }
```

```
  }
```

3) Trying it Out

Write the following inside the playground page for `MutableCollection`:

```
example(of: "using MutableCollection") {
  var list = RefLinkedList<Int>()

  for i in 0...9 {
    list.push(i)
  }

  print(list)

  let index3 = list.index(of: 3)!
  let index4 = list.index(of: 9)!

  list[index3] = 100
  print(list)

  list.swapAt(index3, index4)
  print(list)

  print(list.sorted(by: <))
}
```

4) Challenge

Generalize Selection Sort as an extension of Mutable Collection

```
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    // Add Solution here.
  }
}
```

5) That's it!

Congrats, at this time you should have a good understanding of the `MutableCollection` protocol! Let's now move on to `RangeReplaceableCollection`.

RangeReplaceableCollection Protocol: Demo 6

By Kelvin Lau & Vincent Ngo

In this demo, you will learn about `RangeReplaceableCollection` and it's capabilities.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: You may continue with your current playground, or with the starter project in **Checkpoint6\starter**.

1) Conforming to RangeReplaceableCollection

Navigate to the **Sources** folder. In **RefLinkedList.swift**, add **RangeReplceableCollection** as another conformance:

```
extension RefLinkedList: RangeReplaceableCollection,
BidirectionalCollection, MutableCollection {
...
}
```

2) Implementing replaceSubrange

Add the following within **RefLinkedList**:

```
public mutating func replaceSubrange<C>(_ subrange: Range<Index>, with
newElements: C) where C : Collection, Element == C.Element {

}
```

3) Handling removal cases

Add the following within `replaceSubrange(:with)`:

```
// removeFirst(), removeSubrange(range), removeFirst(n:Int),
removeAll(capacity)
 if newElements.isEmpty {
   if subrange.lowerBound.node === head { // 1
     head = subrange.upperBound.node
     if subrange.upperBound.node === nil {
       tail = nil
     }
   } else if subrange.upperBound.node === nil { // 2
     tail = subrange.lowerBound.node?.previous
     tail?.next = nil
   } else { // 3
     subrange.lowerBound.node?.previous?.next = subrange.upperBound.node
     subrange.upperBound.node?.previous =
subrange.lowerBound.node?.previous
   }

   return
 }
```

The `newElements` will return a `Collection` of type `EmptyCollection`. You want to remove a bunch of nodes for a given `range`. A range has a `lowerBound`, and a `upperBound`.

1. Removing a subrange of nodes from the head.
2. Removing a subrange of nodes from the tail.
3. Removing a subrange of nodes between the head and tail of a linkedList.

4) Trying out removal.

Navigate to **7. RangeReplaceableCollection** and add the following:

```
example(of: "using RangeReplaceableCollection to remove elements") {
  var list = RefLinkedList<Int>()

  for i in 0...9 {
    list.push(i)
  }

  print("removing first")
  print(list)
  list.removeFirst()

  print("removing subrange")
  print(list)
  let index7 = list.index(of: 7)!
```

```
    let index4 = list.index(of: 4)!
    list.removeSubrange(index7...index4)
    print(list)

    print("removing first n")
    print(list)
    list.removeFirst(3)
    print(list)

    print("remove all")
    list.removeAll()
    print(list)
  }
```

5) Handling insertion/append cases

Next add the following just below:

```
  var lowerBound = subrange.lowerBound
  let upperBound = subrange.upperBound

  for element in newElements {
      // 1
      if lowerBound.node == nil && upperBound.node == nil { // append, or
  insert at the endIndex

      // 2
      } else { // Insert subRange

      }
    }
```

Loop through every element to be added, considering two cases:

1. If the lowerBound and upperBound is nil, that means you append or insert at the `endIndex`.
2. You insert a collection or a single node between the head and tail.

6) Appending/Inserting at the end

Add the following within the first condition statement:

```
  let newNode = LinkedListNode(element) // 1
  tail?.next = newNode // 2
  newNode.previous = tail
  tail = newNode // 3

  if head == nil {
    head = tail
  }
```

For every element:

1. Create a new node object.
2. Add the new node to the end of the tail. Make sure you properly hook up the previous and next pointers.
3. Update the `newNode` to be the tail.
4. If the `head` is `nil`, this means the linkedList didn't have elements before, so update the head to equal the tail.

7) Inserting subrange between head and tail.

Add the following code in the `else` clause:

```
let newNode = LinkedListNode(element) // 1

// 2
if lowerBound.node === head && upperBound.node === head { // Update Head
if inserting from head.
  head = newNode
}

// 3
// Connect new node with PREVIOUS node next/previous pointers.
lowerBound.node?.previous?.next = newNode
newNode.previous = lowerBound.node?.previous

// 4
// Connect new node with NEXT node's next/previous pointer
newNode.next = subrange.upperBound.node // connect new node to next node.
upperBound.node?.previous = newNode

// 5
lowerBound.node = newNode.next
upperBound.node = newNode.next // update new lower bounds if more than
one element added.
```

The code above handles inserting nodes given a subrange.

1. Create a new node for the element.
2. Check to see if the `lowerBound` and `upperBound` is the head. This means that you are inserting in front of the head. So set the `newNode` as the head.
3. Update the previous and next pointers between the `newNode` and the `lowerBound` node.
4. Connect the `newNode` next and previous pointers with the `upperBound` node.
5. If there is more than one element in the collection that needs to be added, go ahead and update the `lowerBound` node to be the `newNode`.

8) Trying out inserting subrange

Navigate to **7. RangeReplaceableCollection** and add the following:

```
example(of: "using RangeReplaceableCollection to insert/append elements")
{
  var list = RefLinkedList<Int>()

  print("appending elements")
  list.append(3)
  print(list)

  list.append(5)
  print(list)

  print("inserting a new element at a specific index")
  list.insert(8, at: list.startIndex)
  print(list)

  let index5 = list.index(of: 5)!
  list.insert(21, at: index5)
  print(list)

  print("inserting a collection at a specific index")
  var list2 = RefLinkedList<Int>()
  list2.push(9)
  list2.push(8)
  list2.push(7)
  list2.push(6)
  print("Insert \(list2) into \(list)")
  list.insert(contentsOf: list2, at: index5)
  print(list)

  print("appending a collection to the end of the list")
  list.append(contentsOf: list2)
  print(list)

}
```

Challenge 1

Create a function that remove occurrences of a given subsequence for a given collection. The collection must conform to RangeReplaceableCollection and BidirectionalCollection.

```
func removeOccurrences<T: RangeReplaceableCollection &
BidirectionalCollection>(source: inout T, matching subsequence:
T.SubSequence) where T.Element: Equatable {
  // solution here
}
```

6) That's it!

Congrats, at this time you should have a good understanding of the `RangeReplaceableCollection` protocol! Let's now move on to `RandomAccessCollection`.