Using Collections Effectively

Session 229

Michael LeHew, Foundation
Fundamentals
Indices and Slices
Lazy
Mutability and Multithreading
Foundation and Bridging
// A World Without Arrays
// A World Without Arrays

let bear1 = "Grizzly"
// A World Without Arrays

let bear1 = "Grizzly"
let bear2 = "Panda"
let bear3 = "Polar"
let bear4 = "Spectacled"
let bear1 = "Grizzly"
let bear2 = "Panda"
let bear3 = "Polar"
let bear4 = "Spectacled"

print("\(bear1) bear") // Grizzly bear
print("\(bear2) bear") // Panda bear
print("\(bear3) bear") // Polar bear
print("\(bear4) bear") // Spectacled bear
// A World Without Dictionaries

func habitat(for bear: String) -> String? {

}
func habitat(for bear: String) -> String? {
    if bear == "Polar" {
        return "Arctic"
    } else if bear == "Grizzly" {
        return "Forest"
    } else if bear == "Brown" {
        return "Forest"
    } else if /* all the other bears */
    ...
    return nil
}
// A World Without Dictionaries

func habitat(for bear: String) -> String? {
    if bear == "Polar" {
        return "Arctic"
    } else if bear == "Grizzly" {
        return "Forest"
    } else if bear == "Brown" {
        return "Forest"
    } else if /* all the other cases */
        ...
    return nil
}

let bear = "Polar"
print("\(bear) bears live in the \(habitat(for: bear)).")
let bear = ["Grizzly", "Panda", "Polar", "Spectacled"]

let habitats = ["Grizzly": "Forest", "Polar": "Arctic"]
// Our World Has Collections

let bear = ["Grizzly", "Panda", "Polar", "Spectacled"]
let habitats = ["Grizzly": "Forest", "Polar": "Arctic"]

for bear in bears {
    print("\(bear) Bear")
}
// Our World Has Collections

let bear = "Grizzly", "Panda", "Polar", "Spectacled"
let habitats = "Grizzly": "Forest", "Polar": "Arctic"

for bear in bears {
    print("\(bear) Bear")
}

let bear = bears[2]
let habitat = habitats[bear] ??"
print("\(bear) bears live in the \(habitat)")
protocol Collection
Collections Store Elements

A B C D E
Collections Store Elements

startIndex
Collections Store Elements

startIndex

endIndex
Collections Store Elements

startIndex A B C D E endIndex
Collections Store Elements

startIndex

A B C D E

endIndex
Collections Store Elements

```
subscript[index]
```

```
startIndex       endIndex
A   B   C   D   E
```

Elements are stored within a collection, where `subscript[index]` represents the access to the stored elements. The `startIndex` and `endIndex` define the range of elements within the collection.
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
// Declaration of Collection

protocol Collection : Sequence {
    associatedtype Element
    associatedtype Index : Comparable

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

    var startIndex: Index { get }
    var endIndex: Index { get }

    func index(after i: Index) -> Index
}
 Protocol Extensions

  indices()
  distance(from:, to:)
  makeIterator()     forEach
  starts(with:)
  index(of:)        last     isEmpty
  first             dropLast()
  count             dropLast(n:)
  dropFirst()       elementsEqual()
  dropFirst(n:)     index(where:)
  reversed()        reversed()
  map    filter    reduce
  split()
Protocol Extensions

- indices()
- distance(from:, to:)
- makeIterator()
- forEach
- starts(with:)
- index(of:)
- first
- last
- isEmpty
- dropFirst()
- count
- dropLast()
- dropFirst(n:)
- dropLast(n:)
- elementsEqual()
- index(where:)
- reversed()
- map
- filter
- reduce
- split()
Protocol Extensions

indices()
distance(from:, to:)
makeIterator() forEach
starts(with:)
index(of:)
first
last
isEmpty
dropFirst()
count
dropLast()
dropFirst(n:)
dropLast(n:)
elementsEqual()
index(where:)
reversed()
map
filter
reduce
split()
Protocol Extensions

indices()
distance(from:, to:)
makeIterator()    forEach
    starts(with:)
index(of:)
    first    last
    isEmpty
dropFirst()
    count
    dropLast()
    dropLast(n:)
dropFirst(n:)
    elementsEqual()
index(where:)
    reversed()
    map    filter    reduce
split()
Every Other Element

A B C D E
Every Other Element
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self[startIndex]
        let end = self[endIndex]

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex
        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) }
extension Collection {
    func everyOther(_ body: (Element) -> Void) {
        let start = self.startIndex
        let end = self.endIndex

        var iter = start
        while iter != end {
            body(self[iter])
            let next = index(after: iter)
            if next == end { break }
            iter = index(after: next)
        }
    }
}

(1...10).everyOther { print($0) } // 1, 3, 5, 7, 9
Collections Protocol Hierarchy

Collection
Collections Protocol Hierarchy

- Collection
  - BidirectionalCollection
  - RandomAccessCollection
  - MutableCollection
  - RangeReplaceableCollection
func index(after: Self.Index) -> Self.Index
func index(after: Self.Index) -> Self.Index
Bidirectional Collections

```swift
func index(before: Self.Index) -> Self.Index
```

![Diagram showing bidirectional collection with elements A, B, C, D, E and an arrow pointing to element B]
Bidirectional Collections

```swift
func index(before: Self.Index) -> Self.Index
```

Diagram:

```
A -> B -> C -> D -> E
```
Bidirectional Collections

```swift
func index(before: Self.Index) -> Self.Index
```

![Bidirectional Collection Diagram]
Random Access Collections

// constant time
func index(_ idx: Index, offsetBy n: Int) -> Index
func distance(from start: Index, to end: Index) -> Int
Random Access Collections

// constant time
func index(_ idx: Index, offsetBy n: Int) -> Index
func distance(from start: Index, to end: Index) -> Int
Random Access Collections

// constant time
func index(_ idx: Index, offsetBy n: Int) -> Index
func distance(from start: Index, to end: Index) -> Int
Random Access Collections

// constant time
func index(_ idx: Index, offsetBy n: Int) -> Index
func distance(from start: Index, to end: Index) -> Int
Collections In Swift

Array    Set    Dictionary
# Collections In Swift

<table>
<thead>
<tr>
<th>Array</th>
<th>Set</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Range</td>
<td>String</td>
</tr>
</tbody>
</table>
Collections In Swift

Array    Set    Dictionary

IndexPath    IndexSet

Data    Range    String
Collections In Swift

NSArray
NSPointerArray
NSData
ContiguousArray
NSIndexSet

NSSet
IndexSet

NSCountedSet

NSDictionary
NSHashTable

Array
Set
Dictionary
Data
String
Range
IndexPath
Indices

Each collection defines its own index

Must be Comparable

Think of these as opaque
What is the first element of an Array?
array[0]
What is the first element of a Set?
set[0]
Cannot subscript a value of type 'Set' with an index of type 'Int'
set[set.startIndex]
set[set.startIndex]
array[array.startIndex]
set[set.startIndex]
array[array.startIndex]

Fatal error: Index out of range
array.first
set.first
What is the second element of a Collection?
Find the Second Element of a Collection

extension Collection {
    var second: Element? {
    }
}
extension Collection {
    var second: Element? {
        return self[1]
    }
}
extension Collection {
    var second: Element? {
        return self[1]
    }
}

Cannot subscript a value of type 'Collection' with an index of type 'Int'
extension Collection {
    var second: Element? {
        return self[self.startIndex + 1]
    }
}
extension Collection {
    var second: Element? {
        return self[self.startIndex + 1]
    }
}

Binary operator '+' cannot be applied to operands of type 'Index' and 'Int'
extension Collection {
    var second: Element? {
        // Is the collection empty?

        // Get the second index

        // Is that index valid?

        // Return the second element

    }
}
Find the Second Element of a Collection

extension Collection {
    var second: Element? {
        // Is the collection empty?

        // Get the second index

        // Is that index valid?

        // Return the second element

    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        // Is that index valid?
        // Return the second element
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }

        // Get the second index

        // Is that index valid?

        // Return the second element
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        // Return the second element
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        // Return the second element
        }
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        guard index != self.endIndex else { return nil }
        // Return the second element
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        guard index != self.endIndex else { return nil }
        // Return the second element
    }
}
extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard selfstartIndex != self endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        guard index != self.endIndex else { return nil }
        // Return the second element
        return self[index]
    }
}

extension Collection {
    var second: Element? {
        // Is the collection empty?
        guard self.startIndex != self.endIndex else { return nil }
        // Get the second index
        let index = self.index(after: self.startIndex)
        // Is that index valid?
        guard index != self.endIndex else { return nil }
        // Return the second element
        return self[index]
    }
}
Forming a Slice

[Diagram of a slice with labeled parts A, B, C, D, E]
Forming a Slice

A B C D E
Forming a Slice

startIndex   endIndex
Forming a Slice

[startIndex, endIndex]
Forming a Slice

startIndex | endIndex
A | B | C | D | E
Forming a Slice

subscript[index]
// Slice Share Indices with Original Collection

let array = [1, 2, 3, 4, 5]

let subarray = array.dropFirst()
let secondIndex = array.index(after: array.startIndex)

print(secondIndex == subarray.startIndex)
// Slice Share Indices with Original Collection

let array = [1, 2, 3, 4, 5]

let subarray = array.dropFirst()
let secondIndex = array.index(after: array.startIndex)

print(secondIndex == subarray.startIndex)
// Slice Share Indices with Original Collection

let array = [1, 2, 3, 4, 5]

let subarray = array.dropFirst()
let secondIndex = array.index(after: array.startIndex)

print(secondIndex == subarray.startIndex)
// Slice Share Indices with Original Collection

let array = [1, 2, 3, 4, 5]

let subarray = array.dropFirst()

let secondIndex = array.index(after: array.startIndex)

print(secondIndex == subarray.startIndex)
// Slice Share Indices with Original Collection

let array = [1, 2, 3, 4, 5]

let subarray = array.dropFirst()
let secondIndex = array.index(after: array.startIndex)

print(secondIndex == subarray.startIndex) // true
Find the Second Element of a Collection

```swift
var second: Element? {
    // Is the collection empty?
    guard self.startIndex != self.endIndex else { return nil }
    // Get the second index
    let index = self.index(after: self.startIndex)
    // Is the second index valid?
    guard index != self.endIndex else { return nil }
    // Return the second element
    return self[index]
}
```
Find the Second Element of a Collection

```swift
var second: Element? {
    return self.dropFirst().first
}
```
Find the Second Element of a Collection

```swift
var second: Element? {
    return self.dropFirst().first
}
```
Find the Second Element of a Collection

```swift
var second: Element? {
    return self.dropFirst().first
}
```
Find the Second Element of a Collection

```swift
var second: Element? {
    return self.dropFirst().first
}
```
Produce Collection-like peers of original collection
Slices

Produce Collection-like peers of original collection

Array \rightarrow \text{ArraySlice}
Slices

Produce Collection-like peers of original collection

<table>
<thead>
<tr>
<th>Type</th>
<th>Mapping</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>\rightarrow</td>
<td>ArraySlice</td>
</tr>
<tr>
<td>String</td>
<td>\rightarrow</td>
<td>Substring</td>
</tr>
</tbody>
</table>
Slices

Produce Collection-like peers of original collection

<table>
<thead>
<tr>
<th>Original Type</th>
<th>Slices Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>ArraySlice</td>
</tr>
<tr>
<td>String</td>
<td>Substring</td>
</tr>
<tr>
<td>Set</td>
<td>Slice&lt;Set&gt;</td>
</tr>
</tbody>
</table>
**Slices**

Produce Collection-like peers of original collection

<table>
<thead>
<tr>
<th>Type</th>
<th>Produces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>ArraySlice</td>
</tr>
<tr>
<td>String</td>
<td>Substring</td>
</tr>
<tr>
<td>Set</td>
<td>Slice&lt;Set&gt;</td>
</tr>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Range</td>
<td>Range</td>
</tr>
</tbody>
</table>
Slices Keep Underlying Storage Alive
Slices Keep Underlying Storage Alive
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}
extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf
array = []
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf // [1, 2, 3, 4]
array = []
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf  // [1, 2, 3, 4]
array = []
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf // [1, 2, 3, 4]
array = []

print(firstHalf.first!) // 1
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf // [1, 2, 3, 4]
array = []

print(firstHalf.first!) // 1

let copy = Array(firstHalf) // [1, 2, 3, 4]
firstHalf = []
print(copy.first!)
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf // [1, 2, 3, 4]
array = []

print(firstHalf.first!) // 1

let copy = Array(firstHalf) // [1, 2, 3, 4]
firstHalf = []
print(copy.first!)
// Slicing Keeps Underlying Storage

extension Array {
    var firstHalf: ArraySlice<Element> {
        return self.dropLast(self.count / 2)
    }
}

var array = [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf // [1, 2, 3, 4]
array = []

print(firstHalf.first!) // 1

let copy = Array(firstHalf) // [1, 2, 3, 4]
firstHalf = []
print(copy.first!) // 1
Copying a Slice

var array = [...]

[ 1 2 3 4 5 6 7 8 ]
var array = [...]

var firstHalf = array.firstHalf
Copying a Slice

```swift
var array = [...] // [1, 2, 3, 4, 5, 6, 7, 8]
var firstHalf = array.firstHalf
let copy = Array(firstHalf) // [1, 2, 3, 4]
```
var array = []

var firstHalf = array.firstHalf

let copy = Array(firstHalf)
Copying a Slice

```javascript
var array = []

var firstHalf = []

let copy = Array(firstHalf)
```
Copying a Slice

var array = []

var firstHalf = []

let copy = Array(firstHalf)

[ 1 2 3 4 ]
Eager Functions

```swift
let items = (1...4000).map { $0 * 2 }.filter { $0 < 10 }
```
Eager Functions

```swift
let items = (1...4000).map { $0 * 2 }.filter { $0 < 10 }
```

1...4000
Eager Functions

```
let items = (1...4000).map { $0 * 2 }.filter { $0 < 10 }
```

```
1...4000

\[ \downarrow \]

\[ [2, 4, 6, 8, \ldots, 7998, 8000] \]
```
Eager Functions

```swift
let items = (1...4000).map { $0 * 2 }.filter { $0 < 10 }
```

1...4000

↓

[2, 4, 6, 8, ..., 7998, 8000]

↓

[2, 4, 6, 8]
Eager Functions

```swift
let items = (1...4000).map { $0 * 2 }.filter { $0 < 10 }
```

[2, 4, 6, 8]
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```

1...4000
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```

1...4000

↓

LazyCollection<Range<Int>>
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```

1...4000
\[
\downarrow
\]
LazyCollection<Range<Int>>
\[
\downarrow
\]
LazyMapCollection<Range<Int>>
Let's break down the code and visualize the structure of the lazy collection:

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```

1. **Lazy Collection**: This is the result of applying the `lazy` modifier to the range `1...4000`. It delays the computation of the elements until they are needed.
2. **Lazy Map Collection**: Each element of the range is mapped to its double using the `map` function. This is another lazy collection because the `lazy` modifier is applied.
3. **Lazy Filter Collection**: The mapped values are then filtered to keep only those less than 10. This is the final lazy collection.

This visualization helps understand how lazy collections work in a functional programming context, allowing efficient computation for large datasets.
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
```

LazyFilterCollection  LazyMapCollection  1...4000
Lazy Functions

```
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
```

LazyFilterCollection  LazyMapCollection  1...4000
```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
```

Lazy Functions

LazyFilterCollection

LazyMapCollection

1...4000
Lazy Functions

let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
```

LazyFilterCollection LazyMapCollection 1...4000
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first

LazyFilterCollection
LazyMapCollection
1...4000
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first

LazyFilterCollection
LazyMapCollection
1...4000
```
Lazy Functions

```swift
let items = (1...4000).lazy.map { $0 * 2 }.filter { $0 < 10 }
items.first // 2
```
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)\")
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)"")
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)\'\")
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)\")
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'")
    return $0.contains("Bear")
}

print(redundantBears.first!)
let bears = "Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"

let redundantBears = bears.lazy.filter {
    print("Checking 
          \($0)\)
          // Checking 'Grizzly'
    return $0.contains("Bear")
}

print(redundantBears.first!)
let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)\") // Checking 'Grizzly'
    return $0.contains("Bear") // false
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = \["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"\]

let redundantBears = bears.lazy.filter {
    print("Checking \'$0\'") // Checking 'Panda'
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter { print("Checking '($0)'") // Checking 'Panda'
                                          return $0.contains("Bear") } // false

print(redundantBears.first!)
// Lazy Defers Computation

let bears = [
    "Grizzly", 
    "Panda", 
    "Spectacled", 
    "Gummy Bears", 
    "Chicago"
]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'")
    return $0.contains("Bear")
}

print(redundantBears.first!)
let bears = \["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \'\($0)\'\") // Checking 'Spectacled'
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'") // Checking 'Spectacled'
    return $0.contains("Bear") // false
}

print(redundantBears.first!)
let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)\'")
    return $0.contains("Bear")
}

print(redundantBears.first!)
let bears = 
["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'")  // Checking 'Gummy Bears'
    return $0.contains("Bear")
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'") // Checking 'Gummy Bears'
    return $0.contains("Bear") // true
}

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)\'")
    return $0.contains("Bear")
}

print(redundantBears.first!) // Gummy Bears
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)\'")
    return $0.contains("Bear")
}

print(redundantBears.first!) // Gummy Bears
// Lazy Defers Computation

let bears = "Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"

let redundantBears = bears.lazy.filter {
    print("Checking \("$0"\)"")
    return $0.contains("Bear")
}

print(redundantBears.first!) // Gummy Bears
print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \"\$(0)\"\")
    return $0.contains("Bear")
}

print(redundantBears.first!) // Gummy Bears

print(redundantBears.first!)
// Lazy Defers Computation

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'")
    return $0.contains("Bear")
}

print(redundantBears.first!) // Gummy Bears
let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)\")
    return $0.contains("Bear")
}

let filteredBears = Array(redundantBears)
print(filteredBears.first!)
// Be Lazy, Exactly Once

let bears = [
    "Grizzly",
    "Panda",
    "Spectacled",
    "Gummy Bears",
    "Chicago"
]

let redundantBears = bears.lazy.filter {
    print("Checking \($0)\")
    return $0.contains("Bear")
}

let filteredBears = Array(redundantBears)

print(filteredBears.first!)
// Be Lazy, Exactly Once

let bears = "Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"

let redundantBears = bears.lazy.filter {
    print("Checking \($0)"")
    return $0.contains("Bear")
}

let filteredBears = Array(redundantBears) // ["Gummy Bears"]
print(filteredBears.first!)
// Be Lazy, Exactly Once

let bears = ["Grizzly", "Panda", "Spectacled", "Gummy Bears", "Chicago"]

let redundantBears = bears.lazy.filter {
    print("Checking '\($0)'")
    return $0.contains("Bear")
}

let filteredBears = Array(redundantBears)  // ["Gummy Bears"]

print(filteredBears.first!)  // Gummy Bears
Advice: When to Be Lazy?

Chained computation

Only need part of a result

No side effects

Avoid API boundaries
Collections Protocol Hierarchy

- Collection
  - BidirectionalCollection
  - MutableCollection
  - RangeReplaceableCollection
  - RandomAccessCollection
// constant time
subscript(_: Self.Index) -> Element { get set }
Mutable Collection

```swift
// constant time
subscript(_: Self.Index) -> Element { get set }
```
Range Replaceable Collections

replaceSubrange(_:, with:)

F  B  C  D  E
Range Replaceable Collections

replaceSubrange(_: with:)

| F | E |
Range Replaceable Collections

replaceSubrange(_:, with:)

| F | E |
Range Replaceable Collections

replaceSubrange(\_, \_, with:\_)

| F | E | D | B |
Why did this collection code crash?
Follow-Up Questions

Are you mutating your collection?

Are your collections accessed from multiple threads?
Crashing Collection Code

```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```
Crashing Collection Code

```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```
Crashing Collection Code

```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```

Crashing Collection Code
Crashing Collection Code

```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])
```
```swift
var array = ["A", "B", "C", "D", "E"]
let index = array.firstIndex(of: "E")!
array.remove(at: array.startIndex)
print(array[index])

Fatal Error: Index out of range.
```

![Array with one element remaining](image)
```swift
var array = ["A", "B", "C", "D", "E"]
array.remove(at: array.startIndex)
if let idx = array.firstIndex(of: "E") {
    print(array[idx])
}
```
Avoid Index Invalidation

```swift
var array = ["A", "B", "C", "D", "E"]
array.remove(at: array.startIndex)
if let idx = array.firstIndex(of: "E") {
    print(array[idx])
}
```
Avoid Index Invalidation

```swift
var array = ["A", "B", "C", "D", "E"]
array.remove(at: array.startIndex)
if let idx = array.firstIndex(of: "E") {
    print(array[idx])
}
```
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex])

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex])
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex])

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex])
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert": "honey ice cream",
    "sleep": "hibernation",
    "food": "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex])  // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex])
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex]) // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex])
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex]) // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex]) // (key: "sleep", value: "hibernation")
// Reusing Invalid Dictionary Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!

print(favorites[foodIndex]) // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

print(favorites[foodIndex]) // (key: "sleep", value: "hibernation")

Fatal error: Attempting to access Dictionary elements using an invalid Index
// Always Work With Up-To-Date Indices

var favorites: [String : String] = [
    "dessert" : "honey ice cream",
    "sleep"   : "hibernation",
    "food"    : "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex]) // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

if let foodIndex = favorites.index(forKey: "food") {
    print(favorites[foodIndex])
}
// Always Work With Up-To-Date Indices

```swift
var favorites: [String: String] = [
    "dessert": "honey ice cream",
    "sleep": "hibernation",
    "food": "salmon"
]

let foodIndex = favorites.index(forKey: "food")!
print(favorites[foodIndex]) // (key: "food", value: "salmon")

favorites["accessory"] = "tie"
favorites["hobby"] = "stealing picnic supplies"

if let foodIndex = favorites.index(forKey: "food") {
    print(favorites[foodIndex]) // (key: "food", value: "salmon")
}
```
Advice: Indices and Slices

Use caution when keeping indices/slices

Mutation invalidates

Calculate only as needed
Are your collections reachable from multiple threads?
Multithreaded Mutable Collections

Our collections optimized for single-threaded access

This is a Good Thing™

Undefined behavior without mutual exclusion
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()

let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }  queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }  queue.async { sleepingBears.append("Cub") }

sleepingBears
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }

sleepingBears

["Grandpa", "Cub"]
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }

sleepingBears

["Grandpa", "Cub"]
["Cub", "Grandpa"]
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") } queue.async { sleepingBears.append("Cub") }

sleepingBears

["Grandpa", "Cub"]
["Cub", "Grandpa"]
["Grandpa"]
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") } queue.async { sleepingBears.append("Cub") }
// Example of Thread-Unsafe Practices

var sleepingBears = [String]()
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }  // sleepingBears now contains "Grandpa"
queue.async { sleepingBears.append("Cub") }  // sleepingBears now contains "Grandpa" and "Cub"

sleepingBears

["Grandpa", "Cub"]
["Cub", "Grandpa"]
["Grandpa"]
["Cub"]

malloc: *** error for object 0x100586238: pointer being freed was not allocated
// Example of Thread-Unsafe Practices

```swift
var sleepingBears = [String]() // Thread 1
let queue = DispatchQueue.global()

queue.async { sleepingBears.append("Grandpa") }  // Thread 2
queue.async { sleepingBears.append("Cub") }     // Thread 3

sleepingBears
```

```
["Grandpa", "Cub"]
["Cub", "Grandpa"]
["Grandpa"]
["Cub"]
```

```
malloc: *** error for object 0x100586238: pointer being freed was not allocated
```
WARNING: ThreadSanitizer: Swift access race

Modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 3:

...

Previous modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 2:

...

Location is heap block of size 24 at 0x7b0800023ce0 allocated by main thread:

...

SUMMARY: ThreadSanitizer: Swift access race main.swift:515 in closure #1 in gotoSleep()
WARNING: ThreadSanitizer: Swift access race

Modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 3:

... 

Previous modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 2:

... 

Location is heap block of size 24 at 0x7b0800023ce0 allocated by main thread:

... 

SUMMARY: ThreadSanitizer: Swift access race main.swift:515 in closure #1 in gotoSleep()
WARNING: ThreadSanitizer: Swift access race

Modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 3:
...

Previous modifying access of Swift variable at 0x7b0800023cf0 by thread Thread 2:
...

Location is heap block of size 24 at 0x7b0800023ce0 allocated by main thread:
...

SUMMARY: ThreadSanitizer: Swift access race main.swift:515 in closure #1 in gotoSleep()
// Avoid concurrent mutation

var sleepingBears = [String]()

let queue = DispatchQueue(label: "Bear-Cave")

queue.async { sleepingBears.append("Grandpa") } queue.async { sleepingBears.append("Cub") }
// Avoid concurrent mutation

var sleepingBears = [String]()

let queue = DispatchQueue(label: "Bear-Cave")

queue.async { sleepingBears.append("Grandpa") }  
queue.async { sleepingBears.append("Cub") }
// Avoid concurrent mutation

var sleepingBears = [String]()

let queue = DispatchQueue(label: "Bear-Cave")

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
// Avoid concurrent mutation

var sleepingBears = [String]()
let queue = DispatchQueue(label: "Bear-Cave")

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
queue.async { print(sleepingBears) }
// Avoid concurrent mutation

var sleepingBears = [String]()
let queue = DispatchQueue(label: "Bear-Cave")

queue.async { sleepingBears.append("Grandpa") }
queue.async { sleepingBears.append("Cub") }
queue.async { print(sleepingBears) }

"Grandpa", "Cub"
Advice: Multithreading

Prefer state accessible from a single thread

When this is not possible:

• Ensure mutual exclusion
• Use TSAN
Advice: Prefer Immutable Collections

Easier to reason about data that can't change

Less surface area for bugs

Emulate mutation with slices and lazy

The compiler will help you
Advice: Forming New collections

Use capacity hints if possible

- Array.reserveCapacity(_ :)
- Set(minimumCapacity:)
- Dictionary(minimumCapacity:)

Table: Array Capacity Distribution
Advice: Forming New collections

Use capacity hints if possible

- Array.reserveCapacity(_:)
- Set(minimumCapacity:)
- Dictionary(minimumCapacity:)

[Graph showing allocation of memory]
Advice: Forming New collections

Use capacity hints if possible

- `Array.reserveCapacity(_:)
- `Set(minimumCapacity:)
- `Dictionary(minimumCapacity:)`
Advice: Forming New collections

Use capacity hints if possible

Array.reserveCapacity(_:)
Set(minimumCapacity:)
Dictionary(minimumCapacity:)

Unused Storage

Unused Storage
Advice: Forming New collections

Use capacity hints if possible

Array.reserveCapacity(_:)
Set(minimumCapacity:)
Dictionary(minimumCapacity:)

Unused Storage
Foundation Collections
Reference Type Collections

NSArray
NSMutableArray
NSPointerArray
NSData
NSSet
NSMutableSet
NSCountedSet
NSMutableOrderedSet
NSHashTable
NSIndexSet
NSCharacterSet
NSDictionary
NSMutableDictionary
NSMapTable
Value and Reference Collections
Value and Reference Collections

// Value

// Reference
<table>
<thead>
<tr>
<th>// Value</th>
<th>// Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var x: [String] = []</code></td>
<td><code>let x = NSNSMutableArray()</code></td>
</tr>
</tbody>
</table>
### Value and Reference Collections

<table>
<thead>
<tr>
<th>Value</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>var <code>x</code>: <code>[String] = []</code></td>
<td>let <code>x = NSMutableArray()</code></td>
</tr>
</tbody>
</table>

![Diagram](https://via.placeholder.com/150)
### Value and Reference Collections

<table>
<thead>
<tr>
<th>// Value</th>
<th>// Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var x: [String] = []</code></td>
<td><code>let x = NSMutableArray()</code></td>
</tr>
</tbody>
</table>

![Diagram showing the difference between value and reference collections](image-url)
// Value
var x: [String] = []
x.append("🐻")

// Reference
let x = NSMutableArray()
x.add("🐻")
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")

// Reference
let x = NSMutableArray()
x.add("🐻")
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")

// Reference
let x = NSMutableArray()
x.add("🐻")
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")
var y = x

// Reference
let x = NSMutableArray()
x.add("🐻")
let y = x
// Value
var x: [String] = []
x.append("🐻")
var y = x

// Reference
let x = NSMutableArray()
x.add("🐻")
let y = x
// Value
var x: [String] = []
x.append("🐻")
var y = x

// Reference
let x = NSMutableArray()
x.add("🐻")
let y = x
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")
var y = x
y.append("🐼")

// Reference
let x = NSMutuableArray()
x.add("🐻")
let y = x
y.add("🐼")

Value and Reference Collections
// Value
var x: [String] = []
x.append("🐻")
var y = x
y.append("🐼")

// Reference
let x = NSMutableArray()
x.add("🐻")
let y = x
y.add("🐼")
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")
var y = x
y.append("🐼")

// Reference
let x = NSMutableArray() 
x.add("🐻")
let y = x
y.add("🐼")
Value and Reference Collections

// Value
var x: [String] = []
x.append("🐻")
var y = x
y.append("🐼")

// Reference
let x = NSMutableArray()
x.add("🐻")
let y = x
y.add("🐼")
Objective-C APIs in Swift

// Objective-C
@interface NSView
@property NSArray<NSView *> *subviews;
@end
Objective-C APIs in Swift

// Objective-C
@interface NSView
@property NSArray<NSView *> *subviews;
@end

// Swift
class NSView {
    var subviews: [NSView]
}

// Swift
class NSView {
    var subviews: [NSView]
}
Bridging

Converts between runtime types

Bidirectional

Bridging of collections

• Is necessary

• Can be cheap, but is never free
How Bridging Works

Objective-C

A | B | C | D | E
How Bridging Works

Objective-C

---

Swift
How Bridging Works

Objective-C

Swift
Two Kinds of Bridging

Eager when element types are bridged

Otherwise lazy

• Bridged on first use
Bridging Examples
<table>
<thead>
<tr>
<th>Bridging Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NSArray&lt;NSData *&gt; *</code></td>
</tr>
<tr>
<td>Bridging Examples</td>
</tr>
<tr>
<td>-----------------------------------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
## Bridging Examples

<table>
<thead>
<tr>
<th>Type Description</th>
<th>Eager/Lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSArray&lt;NSData *&gt; *</td>
<td>[Data] Eager</td>
</tr>
<tr>
<td>NSArray&lt;NSView *&gt; *</td>
<td>[NSView] Lazy</td>
</tr>
<tr>
<td>NSDictionary&lt;NSString *, id&gt; *</td>
<td>[String : Any] Eager</td>
</tr>
</tbody>
</table>
Identifying Bridging Problems

Measure your performance with Instruments

Especially inside loops at language boundaries

Look for hotspots like:

- `_unconditionallyBridgeFromObjectiveC`
- `bridgeEverything`
let story = NSString(string: ""
    Once upon time there lived a family of Brown Bears. They had long brown hair.
    ...
    They were happy with their new hair cuts. The end.
    """
)
// A Story About Bridging

let story = NSString(string: ""
    Once upon time there lived a family of Brown Bears. They had long brown hair.
...
    They were happy with their new hair cuts. The end.
"")

let text = NSMutableAttributedString(string: story)

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.

...They were happy with their new hair cuts. The end."

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.

...  

They were happy with their new hair cuts. The end.

```swift
let text = NSMutableAttributedString(string: story)

let range = text.string.range(of: "Brown")!

let nsrange = NSRange(range, in: text.string)  // NSRange

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
```
// A Story About Bridging

let story = NSString(string: "Once upon time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.
"")

let text = NSMutableAttributedString(string: story)

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.

... They were happy with their new hair cuts. The end.

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon a time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.
"""

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

NSColor

text.add Attribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.

let text = NSAttributedString(string: story)
// A Story About Bridging

let story = NSString(string: ""

    Once upon time there lived a family of Brown Bears. They had long brown hair.
    ...
    They were happy with their new hair cuts. The end.
"")

let text = NSMutableAttributedString(string: story)

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

let nsrange = NSRange(range, in: text.string)

text.addAttributes([.foregroundColor: NSColor.brown], range: nsrange)
When Bridging Happens
When Bridging Happens

class NSMmutableAttributedString : ...
{
    var string: String
}

text.string
When Bridging Happens

class NSMutableAttributedString : ... {
    var string: String
}

@interface NSMutableAttributedString : ...
@property NSString *string;
@end
When Bridging Happens

```
class NSMutableAttributedString : ... {
    var string: String
}

@interface NSMutableAttributedString : ...
@property NSString *string;
@end
```

"Once upon time ... The end."

NSString
When Bridging Happens

Once upon time ... The end.
When Bridging Happens

@interface NSMutableAttributedString : ...
@property NSString *string;
@end

class NSMutableAttributedString : ...
{
    var string: String
}

@interface NSString : ...
- (NSRange)rangeOfString:(NSString *)string;
@end

struct String {
    func range(of: StringProtocol) -> Range
}
When Bridging Happens

@interface NSMutableAttributedString : ...
@property NSString *string;
@end

class NSMutableAttributedString : ... {
    var string: String
}

@interface NSString : ...
- (NSRange)rangeOfString:(NSString *)string;
@end

struct String {
    func range(of: StringProtocol) -> Range
}
When Bridging Happens

@interface NSMutableAttributedString : ...
@property NSString *string;
@end

@implementation NSMutableAttributedString
var string: String
@end

@interface NSString : ...
- (NSRange)rangeOfString:(NSString *)string;
@end

struct String {
    var range: Range
}

@interface NSString : ...
@end

struct String {
    func range(of: StringProtocol) -> Range
}

// A Story About Bridging

let story = NSString(string: ""
    Once upon time there lived a family of Brown Bears. They had long brown hair.
    ...
    They were happy with their new hair cuts. The end.
""
)

let text = NSMutableAttributedString(string: story)

let range = text.string.range(of: "Brown")!
let nsrange = NSRange(range, in: text.string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.

... They were happy with their new hair cuts. The end.

let story = NSString(string: """)

let text = NSMutableAttributedString(string: story)
let string = text.string
let range = string.range(of: "Brown")!
let nsrange = NSRange(range, in: string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
// A Story About Bridging

let story = NSString(string: ""
   Once upon time there lived a family of Brown Bears. They had long brown hair.
   ...
   They were happy with their new hair cuts. The end.
"
)

let text = NSMutableAttributedString(string: story)
let string = text.string
let range = string.range(of: "Brown")!
let nsrange = NSRange(range, in: string)

let range = string.range(of: "Brown")!
let nsrange = NSRange(range, in: string)
text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
// A Story About Bridging

let story = NSString(string: "Once upon time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.
"")

let text = NSMutableAttributedString(string: story)
let string = text.string
let range = string.range(of: "Brown")!
let nsrange = NSRange(range, in: string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
// A Story About Bridging

let story = NSString(string: """
Once upon time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.
"""")

let text = NSMutableAttributedString(string: story)
let string = text.string
let range = string.range(of: "Brown")!
let nsrange = NSRange(range, in: string)

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Once upon time there lived a family of Brown Bears. They had long brown hair.

...  

They were happy with their new hair cuts. The end.

"""
Once upon time there lived a family of Brown Bears. They had long brown hair.

... They were happy with their new hair cuts. The end.

let text = NSMutabileAttributedString(string: story)
let string = text.string as NSString  // NSString

let nsrange = string.range(of: "Brown")  // NSRange

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
let story = NSString(string: "Once upon time there lived a family of Brown Bears. They had long brown hair."

...  

They were happy with their new hair cuts. The end."

"

let text = NSMutableAttributedString(string: story)  
let string = text.string as NSString  // NSString  
let NSRange  // NSRange

let NSRange = string.range(of: "Brown")  // NSRange

text.addAttribute(.foregroundColor, value: NSColor.brown, range: NSRange)  // 25 ms
// A Story About Bridging

let story = NSString(string: "Once upon time there lived a family of Brown Bears. They had long brown hair.
...
They were happy with their new hair cuts. The end.
"")

let text = NSMutableAttributedString(string: story)
let string = text.string as NSString // NSString

let NSRange = string.range(of: "Brown") // NSRange

let nsrange = string.range(of: "Brown") // NSRange

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange) // NSRange

let story = NSString(string: "Once upon time there lived a family of Brown Bears. They had long brown hair. ...
They were happy with their new hair cuts. The end. ")

let text = NSMutableAttributedString(string: story)
let string = text.string as NSString

let nsrange = string.range(of: "Brown")

let nsrange = string.range(of: "Brown") // NSRange

let NSRange

10 ms
3 ms
25 ms
1 ms

text.addAttribute(.foregroundColor, value: NSColor.brown, range: nsrange)
Advice: When to Use Foundation Collections

You need reference semantics

You are working with known proxies
  • NSFetchedResultsController
  • Core Data Managed Objects

You've measured and identified bridging costs
Now It’s Your Turn

Explore your existing collections
Measure your code
Audit your mutable state
Gain mastery in Playgrounds
More Information

https://developer.apple.com/wwdc18/229

<table>
<thead>
<tr>
<th>Lab Type</th>
<th>Location</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cocoa Lab</td>
<td>Technology Lab 7</td>
<td>Friday 11:00AM</td>
</tr>
<tr>
<td>Swift Open Hours</td>
<td>Technology Lab 10</td>
<td>Friday 3:00PM</td>
</tr>
</tbody>
</table>