Advanced Swift

Mariusz Lisiecki

Agenda

1. All about collections
2. Optionals - going deeper
3. Copy-on-write explained
4. Generics in more details
5. Advanced protocol stuff
6. Basic reflection in Swift

What's in it for me?

Increase consciousness about the langauge, to be able to answer such questions:

- How does sth work?
- Why does sth work?
- Why does sth not work?

Such knowledge is essential to write cleaner, safer and more readable code in Swift.

1. All about collections

Collections

1. Built-in collections and their capabilities
2. Creating custom collections - conforming to `Sequence` and `Collection`
3. Indices
4. Specialized Collections
5. Lazy Sequences
6. Ranges
7. Slices

Built-In Collections

- Abstract types - protocols:
  - `Sequence`
  - `Collection`
- Concrete types:
  - `Array`
  - `Dictionary`
  - `Set`

Sequence VS Collection

From Apple docs:

Sequence*: A type that provides sequential, iterated access to its elements.*

Collection*: A **sequence** whose elements can be traversed multiple times, nondestructively, and accessed by indexed subscript.*

Sequence Protocol

- `Sequence` protocol **requirements**
  - `map(_:)`, `filter(_:)`, `forEach(_:)`
  - `drop(while:)` / `dropFirst(_:)` / `dropLast(_:)`
  - `prefix(_:)` / `prefix(while:)` / `suffix(_:)`
  - `underestimatedCount`
  - `split(maxSplits:omittingEmptySubsequences:whereSeparator:)`
  - **`makeIterator()`**

Sequence Protocol

- `Sequence` protocol **default implementations**:
  - all required methods except `makeIterator`
  - `contains(_:) / contains(where:)`
  - `elementsEqual(_:) / elementsEqual(by:)`
  - `enumerated()`
  - `sorted() / sorted(by:)`
  - `reversed()`
  - `min(), min(by:), max(), max(by:)`

Collection Protocol

- `Collection` protocol provides (default implementations):
  - everything from `Sequence` (`Collection` inherits from `Sequence`)
  - `count / isEmpty`
  - `subscript(Range)`
  - `distance(from:to:)`
  - more...

Collection Protocol

- Types from Swift Standard Library that conforms to `Collection`
  - `Array`
  - `Dictionary`
  - `Set`
  - `String` (in Swift 4)
    - in Swift 3 -> `String.CharacterView` only!

Exercises

Do all exercises from Built-in Collections group.

Unstable Sequence

- Sequence that may produce different results across multiple traversals (i.e. for ... in loops)

```swift
struct ReadLineIterator: IteratorProtocol {
    func next() -> String? {
        guard let line = readLine() else { return nil }
        return line.characters.count > 0 ? line : nil
    }
}

struct ReadLineSequence: Sequence {
    func makeIterator() -> ReadLineIterator {
        return ReadLineIterator()
    }
}

let sequnce = ReadLineSequence()
var firstLoopCounter = 0
var secondLoopCounter = 0

for _ in sequnce {
    firstLoopCounter += 1
}

for _ in sequnce {
    secondLoopCounter += 1
}

print("Counters equal? \(firstLoopCounter == secondLoopCounter)")
```

Infinite Sequence

- Sequence which has iterator that never returns nil.

```swift
struct FibonacciIterator: IteratorProtocol {
    var fib1 = 1
    var fib2 = 1
    mutating func next() -> Int? {
        let current = fib1
        fib1 = fib1 + fib2
        fib2 = current
        return current
    }
}

extension FibonacciIterator: Sequence {}

let fibonacciSequence = FibonacciIterator()

for fib in fibonacciSequence {
    print("\(fib)")
} // This loop never ends (crashes not counted ;))
```

NOTE: You should not call `reversed()` on infinite sequences (otherwise crash)

Sequence VS Collection

Feature	`Sequence`	`Collection`
Multiple Traversal	Not guaranteed, sequence may be unstable (traversing same sequence multiple times may yield different results)	Guaranteed
Infinity	Sequences can be infinite	Can not be infinite - they have `count` property
Direct access to random element by its index	Not guaranteed or may take O(n) time	Guaranteed in O(1) time (provided we have proper index)

Conforming to Sequence protocol

- Has to be adopted by any type that we want to enumerate using `for ... in` loop
- Requires iterator - type implementing `IteratorProtocol`

Conforming to IteratorProtocol protocol

- Adopting requires only one function

```
mutating func next() -> Self.Element?
```

- `Element` is `associatedtype`
- NOTE: In Swift 4, `Sequence` has also such definition:

```
associatedtype Element where
Self.Element == Self.Iterator.Element
```

Conforming to Sequence protocol

- Conforming to it is easy - you only have to return some iterator:

```
func makeIterator() -> Self.Iterator
```

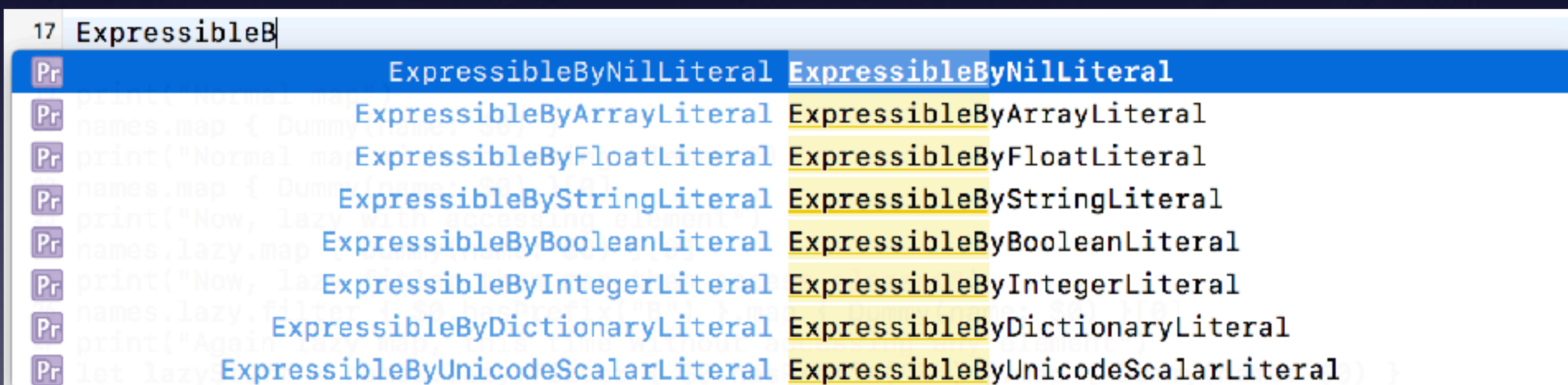
- `Iterator` is `associatedtype`
- HINT: Every `Iterator` implementation could be marked as conforming to `Sequence` "for free".

Exercises

Do exercises 1 - 5 from Sequence & Collection group.

Expressible By Literal

There are a lot of `ExpressibleByXXXLiteral` protocols:



One can adopt them to simplify initialization of custom types.

Expressible By Literal

```swift
struct Person {
    let name: String
    let age: Int
    let isMale: Bool
}

extension Person: ExpressibleByArrayLiteral {
    typealias Element = Any
    public init(arrayLiteral elements: Element...) {
        name = elements[0] as! String
        age = elements[1] as! Int
        isMale = elements[2] as! Bool
    }
}

let p: Person = [ "Mariusz", 30, true ]
```

Converting to String

`CustomStringConvertible`

- property `description` returns textual representation suitable when converting to string

`CustomDebugStringConvertible`

- property `debugDescription` returns textual representation suitable for debugging

Converting to String

```
struct Dummy {}

extension Dummy: CustomDebugStringConvertible {
    var debugDescription: String {
        return "debugDescription"                    (7 times)
    }
}

extension Dummy: CustomStringConvertible {
    var description: String {
        return "description"                         (2 times)
    }
}

let d = Dummy()                                      debugDescription
d                                                    debugDescription
"" + "\(d)"                                          "description"
print(d)                                             "description\n"
debugPrint(d)                                        "debugDescription\n"
```

Conforming to Collection protocol

- Conforming to `Collection` may seem not so easy at first:
  - 4 associated types
  - 4 properties
  - 7 instance methods
  - 2 subscripts
- Luckily, a lot of default implementations are provided.

Conforming to Collection protocol

```swift
protocol Collection: Sequence {
    associatedtype IndexDistance = Int
    associatedtype Iterator = IndexingIterator<Self>
    associatedtype SubSequence: Sequence = Slice<Self>
        where Self.Element == Self.SubSequence.Element,
              Self.SubSequence == Self.SubSequence.SubSequence
    associatedtype Indices : Sequence = DefaultIndices<Self>
        where Self.Index == Self.Indices.Element,
              Self.Indices == Self.Indices.SubSequence,
              Self.Indices.Element == Self.Indices.Index,
              Self.Indices.Index == Self.SubSequence.Index

    var first: Self.Element? { get }
    var indices: Self.Indices { get }
    var isEmpty: Bool { get }
    var count: Self.IndexDistance { get }

    func makeIterator() -> Self.Iterator
    func prefix(through position: Self.Index) -> Self.SubSequence
    func prefix(upTo end: Self.Index) -> Self.SubSequence
    func suffix(from start: Self.Index) -> Self.SubSequence
    func distance(from start: Self.Index, to end: Self.Index) -> Self.IndexDistance
    func index(_ i: Self.Index, offsetBy n: Self.IndexDistance) -> Self.Index
    func index(_ i: Self.Index, offsetBy n: Self.IndexDistance, limitedBy limit: Self.Index) -> Self.Index?

    subscript(position: Self.Index) -> Self.Element { get }
    subscript(bounds: Range<Self.Index>) -> Self.SubSequence { get }
}
```

Conforming to Collection protocol

- What is really needed?
  - `startIndex` and `endIndex` properties,
  - `subscript` that provides at least read-only access to elements by index (in constant time)
  - `index(after:)` method for advancing an index

Conforming to Collection protocol

- What is really needed?

```
protocol Collection: Sequence {

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

    func index(after i: Self.Index) -> Self.Index

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

Conforming to Collection protocol

- In fact only `subscript` is the requirement from `Collection` protocol

- `startIndex`, `endIndex` and `index(after:)` used to be inherited from `Indexable` protocol ...

- ... but the protocol is removed in Swift 4.0
  - and now its requirements are defined nowhere...
    - ... or at least I can't find it ;)

Conforming to Collection protocol

Expected performance (from documentation):

Types that conform to Collection are expected to provide the startIndex and endIndex properties and subscript access to elements as O(1) operations.
Types that are not able to guarantee that expected performance must document the departure, because many collection operations depend on O(1) subscripting performance for their own performance guarantees.

Exercises

Do exercise 6 from Sequence & Collection group.

Things to remember

- Calling `reversed()` on infinite sequences results in runtime error
- Indices should be ascending (`startIndex <= endIndex`)
    - This requirement may be uncomfortable when working with collections that use underlying storage which stores elements in reversed order (like our `Stack`)

Collection - Associated types

Associatedtype	Type	Inherited from	Default	Comments
Element	–	Sequence	–	-
Iterator	IteratorProtocol	Sequence	IndexingIterator <Self>	No reason to change to sth other than default.
SubSequence	Sequence	Sequence (+ refined)	Slice<Self>	Should be Collection itself (not yet expressible in Swift). Performance gains if it has the same type as Self.
Index	Comparable?	?	?	Type of elements stored as indices.
IndexDistance	–	–	Int	No reason to change to sth other than default.
Indices	Sequence	–	DefaultIndices <Self>	Should be Collection itself (not yet expressible in Swift). Perfomance gains if does not have reference to Self.

Indices

Let's look a little bit deeper at indices of the collection.

- `Index` represent position in the collection (remember it has to be definite)
- `startIndex` -> first element in the collection
- `endIndex` -> index **after** last element in the collection
- Indices should be **ascending**

Indices

Indices are not always `Int`s.

`Dictionary` is a `Collection`, yet its indices are not `Int`s.

They are not dictionary's keys also, because:

- How to tell what is next index (key) after given key?
- Subscripting on index should provide direct access, without hashing or searching.

Indices

Index of dictionary is opaque -> its of type `Dictionary.Index`

It is worth noting that dictionary has two type of subscripts:

- By index -> returns non-optional key & value pair.
- By key -> returns optional value
  - NOTE: This subscript is part of `Dictionary`, not `Collection`!

Indices

`Index` of dictionary is opaque -> its of type `Dictionary.Index`

```
let sampleDict = [ 1 : "one ", 2 : "two", 3 : "three" ]

let subscriptByKey = sampleDict[1]
print(type(of: subscriptByKey))
// Prints "Optional<String>"

let sampleIndex = sampleDict.index(after: sampleDict.startIndex)
print(type(of: sampleIndex))
// Prints "Index"

let subscriptByIndex = sampleDict[sampleIndex]
print(type(of: subscriptByIndex))
// Prints "(key : Int, value : String)"
```

Indices

- Indices may store the reference to the base collection.
  - But better if not! E.g. indices of array does not need reference to the array itself.
- One may use indexes of some collection to access other collection.
  - However this is risky and thus discouraged.
- Indices are shared between `Collection` and its `Slice`
  - At least that is what documentation says.

Indices

Indices may become invalid when the collection is mutated.

- index may point to other element
- index may no longer be valid for the collection

Specialized Collections

Please note that pure `Collection` protocol does not have methods / properties / subscripts to do following stuff:

- advancing index backwards
  - Thus: reverse traversal
- being able to calculate any index in constant time
  - Thus: allowing access to any element in constant time (unless you somehow already have correct index)
- mutability
  - Including mutability of ranges of elements of the collection

Specialized Collections

There are 4 base types of specialized `Collection`:

- `BidirectionalCollection` (backward traversal)
- `RandomAccessCollection` (calculating random index in constant time)
- `MutableCollection` (mutability of single random element while preserving length and indices)
- `RangeReplacableCollection` (mutability of ranges, possibility to change collection's length, while preserving internal location of the elements)

Specialized Collections

`Collection`
VS `BidirectionalCollection`
VS `RandomAccessCollection`

	Accessing next element	Accessing prev element	Accessing middle element
LinkedList	V	X	X
DoublyLinkedList `String`	V	V	X
`Array`	V	V	V

Specialized Collections

`Collection`
VS `BidirectionalCollection`
VS `RandomAccessCollection`

	`index(after:)`	`index(before:)`	`index(_:offsetBy:)` `distance(to:from:)`
`Collection`	O(1)	Not required by protocol	O(n)
`Bidirectional-Collection`	O(1)	O(1)	O(n)
`RandomAccess-Collection`	O(1)	O(1)	O(1)

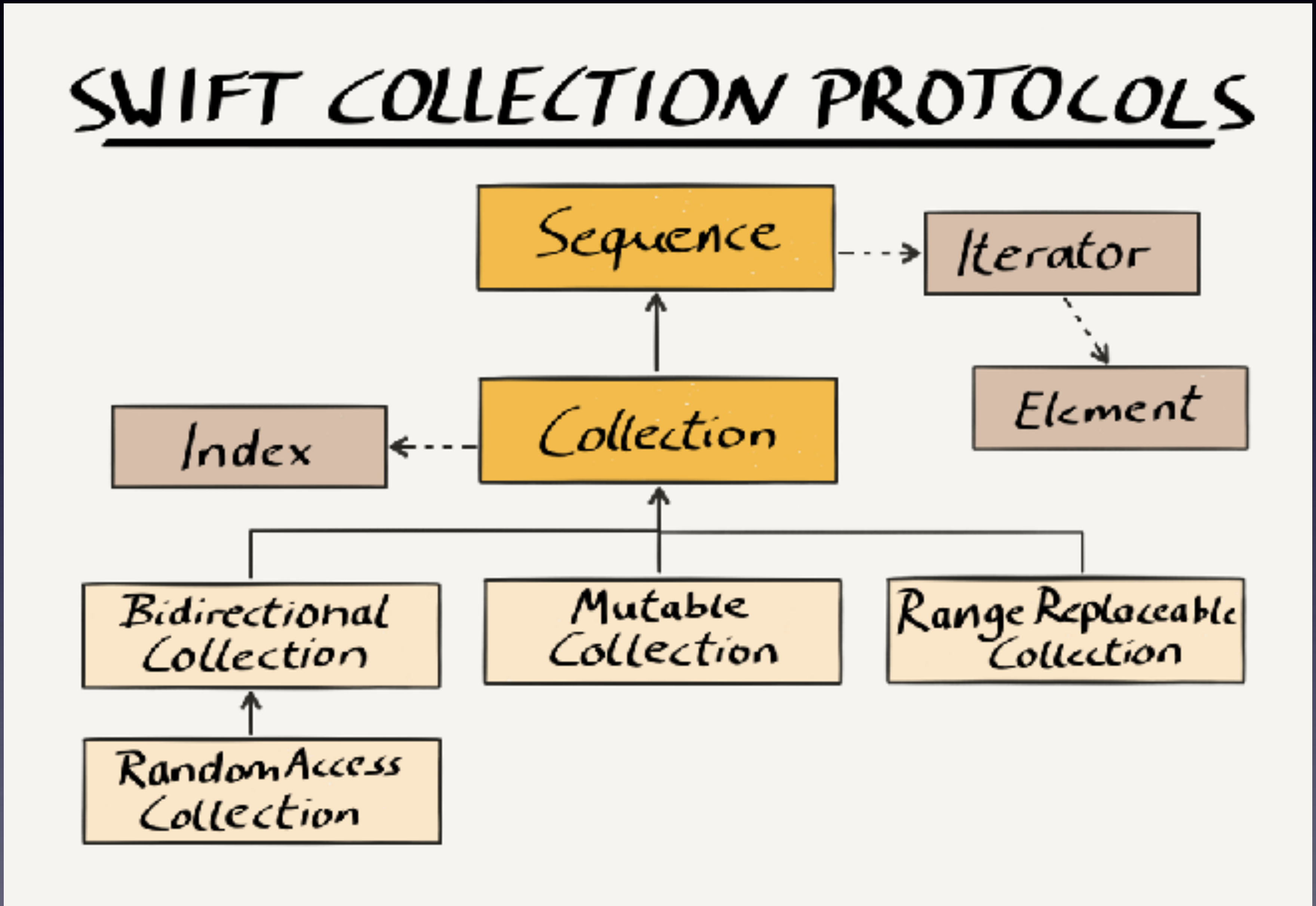
Specialized Collections

`Collection`
VS `MutableCollection`
VS `RangeReplaceableCollection`

	Accessing element	Mutating single element while preserving order and length and indices	Mutating arbitrary ranges of elements while preserving internal order - "location" (may change length)
`Set, Dictionary`	V	X (mutating may change length or internal order)	X (mutating may change internal order)
`String.CharacterView / String` in Swift 4	V	X (replacing "a" with "❤️" may change index of the element)	V
`UnsafeMutableBuffer-Pointer`	V	V	X (memory can be modified by the type, but size of the memory cannot)
`Array`	V	V	V

Specialized Collections

Summary:



Source (and suggested further reading):
https://oleb.net/blog/2017/02/why-is-dictionary-not-a-mutablecollection/
https://oleb.net/blog/2017/02/why-is-string-characterview-not-a-mutablecollection/

Exercises

Do exercise 1 from Specialized Collections group.

Specialized Collections

So we have:

- 3 possibilities of traversal (forward-only, backward, random access)

- 4 possibilities of mutability (not mutable, mutable, range replaceable, mutable-and-range replaceable)

- For every combination, there may be dedicated `Slice` type:

```
public struct MutableRangeReplaceableBidirectionalSlice<Base> :
    BidirectionalCollection, MutableCollection, RangeReplaceableCollection
    where Base : _BidirectionalIndexable, Base : _MutableIndexable,
          Base : _RangeReplaceableIndexable
```

Specialized Collections

Ok, but why the hell we need to know this?

- If you create your custom `Collection`, you should know which specializations can you adopt.
  - Taking into account performance of operations!
- A lot of `Collection` methods are available only for specific specializations.
  - Example `sort()` vs `sorted()`

Specialized Collections

`sorted()` VS `sort()`

- `sorted()` is defined in extension of `Sequence` where `Iterator.Element: Comparable`
- `sort()` is defined in extension of `MutableCollection` where Self is also `RandomAccessCollection` (otherwise sort may be slow) and where `Iterator.Element: Comparable`

Lazy Collections

Every `Sequence` has `.lazy` property

- This property returns a sequence that contains same elements as the target sequence
- However, some operations (like `map` or `filter`) on the returned sequence may be implemented lazily.
- Same thing for `Collection` -> `LazyCollection`

Lazy Collections

What does it mean to be implemented lazily?

```
struct Dummy {
    let name: String
    init(name: String) {
        print("Initializing self with name: \(name)")
        self.name = name
    }
}

let names = [ "Ann", "Beth", "Celine" ]
let result1 = names.map { Dummy(name: $0) }[0]      // Initializes 3 Dummies
let result2 = names.map { Dummy(name: $0) }         // Initializes 3 Dummies
let result3 = names.lazy.map { Dummy(name: $0) }[0] // Initializes 1 Dummy
let result4 = names.lazy.map { Dummy(name: $0) }    // Initializes 0 Dummies
```

Further reading:
https://news.realm.io/news/slug-brandon-kase-grokking-lazy-sequences-collections/

Exercises

Do all exercises from Lazy Collections group.

Things to remember

- If operations like `map` OR `filter` are chained on lazy collection, they wrap themselves like matryoshka ;)
- `LazySequence` / `LazyCollection` are not quite print-friendly
  - `Array` has initializers that allow to workaround that ;)
- Some specialized lazy sequences / collections sometimes are **not** Int-indexable.

Quiz

We all know following code:

```
for i in 0..<10 {
    // Do sth with i
}
```

But do you know what is the type of this?

```
let whatIsThis = 0..<10
```

Or this: `let whatIsThis2 = "a"..."z"`

Ranges

- They are ranges
- There used to be 4 types of ranges:

Elements are	Half-open range	Closed range
`Comparable`	`Range`	`ClosedRange`
`Strideable` (with integer steps)	`CountableRange`	`CountableClosed-Range`

Ranges

Swift 4 adds 4 more Ranges + Protocol:

- `PartialRangeFrom` (`X...` operator)
- `PartialRangeThrough` (`...X` operator)
- `PartialRangeUpTo` (`..<X` operator)
- `CountablePartialRangeFrom` (same as `PartialRangeFrom`, but only for types that are `Strideable` with `SignedInteger` as `Stride`)
- protocol `RangeExpression` (all ranges conform to it)

Ranges

Swift 4 adds 4 more Ranges + Protocol:

```
var partialRangeUpTo: PartialRangeUpTo<Float> = ..<5.0
var partialRangeThrough: PartialRangeThrough<Float> = ...5.0
var partialRangeFrom: PartialRangeFrom<Float> = 5.0...

// Does not compile, Float does not conform to SignedInteger
// var countablePartialRangeFrom: CountablePartialRangeFrom<Float> = 5.0...

// This compiles
var countablePartialRangeFrom: CountablePartialRangeFrom<Int> = 5...
```

Do all exercises from Ranges group.

Ranges

There are some quirks about ranges:

- Ranges that does not have `Countable` word in its name are not even `Sequence`s
- `CountablePartialRangeFrom` is infinite sequence
  - One should NOT call methods like `map`, `filter`, `suffix`, `reversed`, etc.
- `CountableRange` and `CountableClosedRange` are `RandomAccessCollection`s

Ranges

There are some quirks about ranges (part 2):

- According to docs, both `Range` and `ClosedRange` have `count` property...
  - ... and if fact, they do have, but it is defined in extension with generic constraints
- Elements of `CountableRange` are indices to the collection itself
  - Thus, index 0 may NOT be the index to the first element!

Ranges

There are some quirks about Ranges (part 3):

- `CountableRange<Int>` cannot be subscripted (compile time error) - eventhough it is `Collection`
  - However, it could be workarounded (see documentation)
- `CountableClosedRange` can be subscripted
  - But not with integers, but with instances of `ClosedRangeIndex`

Future

Actually, stuff like `Countable(Closed)Range` may be declared that way:

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

- However, this is still not yet possible in Swift

Quiz

Is the following code safe?

```
let someCollection = [ 1, 2, 3 ].dropFirst()

if someCollection.count > 0 {
    print("\(someCollection[0])")
}
```

Slices

- Objective-C:

`[NSArray subarrayWithRange:]` returns `NSArray`

- Swift

```
let array = ["a", "b", "c", "d", "e"]
let subarray = array[1...3]
```

- Type of `subarray` is `ArraySlice`

Slices

- `ArraySlice` is not a new `Array`, instead it is "a view onto the storage of a larger array" (Apple doc)
- Has the same interface as `Array`
- However, these two types are not equal (and does not have any common ancestor)
  - But you can easily construct new `Array` from its `ArraySlice`
    - That's not true for `Dictionary`!

Exercises

Do exercises 1 - 7 from Slices & SubSequences group.

Slices

- `ArraySlice` is the type of slices for `Array`
- Other types (like `Dictionary`) simply have `Slice`
- Both `ArraySlice` and `Slice` conform to `Collection`

Slices

- Slices share the indexes with the base collection
- Accessing an index that is not contained within slice (eventhough it may be correct for the base collection) -> CRASH
  - That means `[0]` index is not always safe if the slice is not empty
  - Use `startIndex`, `endIndex`, `index(after:)` instead

Slices

Slices inherit semantics

- If you get a slice from a collection that has value semantics, the slice has also value semantics.
  - Thus, if you mutate original collection that has value semantics, the slice remain unchanged.

Slices

- A slice may hold a reference to the entire storage of a larger collection.

- Long-term storage of a slice may therefore prolong the lifetime of elements that are no longer otherwise accessible, which can erroneously appear to be memory leakage.

SubSequence

- In fact, `(Array)` `Slice` is just `SubSequence`
- `SubSequence` is `associatedType` on `Sequence` (just like `Iterator` is)
- It is possible however to define sequences where `SubSequence == Self`
  - This is the case of `String`

Exercises

Do exercises 8 - 9 from Slices & SubSequences group.

Hashable requirement

If some type has to be used as:

- key of `Dictionary`
- element of `Set`

it has to conform to `Hashable` protocol

- NOTE: This protocol inherits from `Equatable` protocol

Hashable requirement

Beware!

- If your reference type allows mutability, you should ensure instances of it does NOT get mutated while being stored in set OR dictionary
  - In particular if `hashValue` depends on the property/ies being mutated.

Exercises

Do exercise 1 from Hashable requirement group.

2. Optionals - going deeper

Optionals

1. Nested optionals

2. Mapping optionals

3. Comparing optionals

Exercises

Do exercises 1-3 from Optionals group.

Definition

Optionals are nothing more but generic enums.

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

Nested Optionals

Thus, nothing stops you from creating such values:

```
let doubleNestedOptionalInt: Int?? = .some(.some(1))

var tripleNestedOptionalInt: Int???
tripleNestedOptionalInt = .some(.some(.some(20)))
tripleNestedOptionalInt = 20    // Same as above
tripleNestedOptionalInt = .some(.some(.none))
tripleNestedOptionalInt = .some(.some(nil)) // Same as above
tripleNestedOptionalInt = .some(.none)  // Sth different

tripleNestedOptionalInt == nil // false
// tripleNestedOptionalInt == .some(.none) // Does not compile
```

Nested Optionals

You can create also collections with `nil` values:
(but please note another level of optionality)

```
let arrayWithNils: [Int?] = [ 0, nil, nil ]
print(arrayWithNils.first)
// prints "Optional(Optional(0))"
print(arrayWithNils.count)
// prints "3"

let dictWithNils: [String: Int?] = [ "one" : 1, "two" : 2, "fee" : nil ]
print(dictWithNils["one"])
// prints "Optional(Optional(1))"
print(dictWithNils["fee"])
// prints "Optional(nil)"
```

Mapping optionals

One can `map` optionals:

```
let possibleNumber: Int? = Int("42")
let possibleSquare = possibleNumber.map { $0 * $0 }
print(possibleSquare)
// Prints "Optional(1746)"

let noNumber: Int? = nil
let noSquare = noNumber.map { $0 * $0 }
print(noSquare)
// Prints "nil"
```

Mapping optionals

One can also `flatMap` optionals:

```
let possibleNumber: Int? = Int("42")
let nonOverflowingSquareTransform = { (x: Int) -> Int? in
    let (result, overflowed) = Int.multiplyWithOverflow(x, x)
    return overflowed ? nil : result
}
let mapTransformResult = possibleNumber.map(nonOverflowingSquareTransform)
let flatMapTransformResult = possibleNumber.flatMap(nonOverflowingSquareTransform)

print("\(mapTransformResult)")
// Prints "Optional(Optional(1746))"

print("\(flatMapTransformResult)")
// Prints "Optional(1746)"
```

Optionals and collections

Because optional is in fact enum, in contrary to Objective-C we can store `nil`s in collections:

```
var arrayOfOptionals = [ 1, 2, nil ]

var dictWithOptionals = [ 0 : nil, 1 : "one"]
```

Optionals and collections

But be careful with dictionaries

```
var dwo = [ 0 : nil, 1 : "one" ]

dwo[1] = nil  // Removes value for key 1

dwo[1]? = nil  // Updates value for key 1
```

OR

```
dwo[1] = Optional(nil)  // Updates value for
key 1
```

Comparing Optionals

- Optionals could be checked for equality if they wrap same type that is `Equatable`.

- One can also compare optional value and non-optional value
  - This is because compiler implicitly converts non-optional to optional.

```swift
let nonOpt = 1
let opt: Int? = 1

nonOpt == opt    // true
// What compiler does under the hood
Optional(nonOpt) == opt // Same as above
```

Comparing Optionals

- However, `Optional` does not implement `Equatable` protocol
  - Because it can be checked for equality **only** when the `Wrapped` type is `Equatable` - which is not always true because `Optional` may store any type.
- That's why array of optionals cannot be compared.
  - Function that check equality of two arrays require `Element` of the `Array` to conform to `Equatable`

Comparing Optionals

```
let arr1 = [ 1, 2, 3 ]
let arr2 = [ 1, 2, 3 ]
let arr3 = [ 1, 2, nil ]
let arr4 = [ 1, 2, nil ]

arr1 == arr2
arr3 == arr4    // Does not compile
arr1 == arr3    // Does not compile as well
```

One can workaround this by creating dedicated function

OR
wait for future versions of Swift, where the problem may be solved ;)

Comparing Optionals

One final note:

- Although titles of recent slides were "Comparing Optionals", optionals are not `Comparable` ;>
- nor they provide `<`, `>`, etc. functions

Exercises

Do exercises 4-5 from Optionals group.

3. Copy-on-write explained

Quiz

How the code below will behave?

Will it not compile | crash | execute 1 or 3 times?

```
var mutableArray = [ 1, 2, 3 ]
for _ in mutableArray {
    mutableArray.removeAll()
    print("For loop executed")
}
```

* What will happen if we change the type of array to `NSMutableArray`?

Exercises

Do exercise 1 from Copy-on-write group.

Structs - reminder

- Structs are value types, not managed by ARC.
  - Classes are reference types, managed by ARC.
- Structs are immutable, modifying struct declared as var is simply assigning totally new struct to the variable.
- If we pass value to or from function, we in fact pass a copy.
  - However, compiler can optimize out redundant copies.

Structs - reminder

- Optimizing out redundant copies (e.g. passing constant struct by reference instead of passing by value) is done **automatically by compiler**.
- This is **not** the same as copy-on-write behaviour of a type that has value semantics.
    - It has to be implemented by developer by detecting shared references.
- Array has copy-on-write technique implemented.

Copy-on-write

IF:

- You have var array of something (could be structs).
- There are **no other** references to the array (uniqueness).
- You modify the array / one of the "something" that is being stored.
- THEN no copy is made -> memory is modified in place.

Copy-on-write

IF:

- You have var array of something (could be structs).
- You create another variable array and assign the first array to it.
- THEN no copy is made -> variables share the memory they point to.
- UNLESS one of the vars is modified, then real deep copy gets made.

Copy-on-write

`isKnownUniquelyReferenced()` function may be useful when implementing the technique in custom type.

- However, the function works only with pure Swift classes.
  - Does not compile for structs.
  - Returns false for classes inheriting from NSObject.
  - Requires the instance to be declared as variable (`var`).

Exercises

Do exercise 2 from Copy-on-write group.

Further reading

More advanced example of implementing Copy-on-write for custom type:

https://www.cocoawithlove.com/blog/2016/09/22/deque.html

4. Generics in more details

Overloading

Overloading a function / method could be done:

- by parameter type

```
extension Int {
    mutating func multiply(by multiplier: Double) {
        let doubleSelf = Double(self)
        let doubleResult = doubleSelf * multiplier
        self = Int(doubleResult)
    }

    mutating func multiply(by multiplier: Int) {
        self *= multiplier
    }
}

var a = 3; var b = 3; var c = 3
a.multiply(by: 2.5)       // 7
b.multiply(by: Int(2.5))  // 6
c.multiply(by: 2)         // 6
```

Overloading

Overloading a function / method could be done:

- by return type

```swift
extension Double {
    func half() -> Int {
        return Int(self) / 2
    }

    func half() -> Double {
        return Double(self) / 2.0
    }
}

let toBeHalved = 1.0
let half: Double = toBeHalved.half()    // 0.5
let halfAsInt: Int = toBeHalved.half()  // 0
```

Overloading

Overloading a function / method could be done:

- by generic constraints

Let's find out by doing exercises 1-2 from Generics group.

Overloading

Overloading a function / method could be done:

- by generic constraints

```swift
extension Sequence where Iterator.Element: Equatable {
    func isSubset<S>(of other: S) -> Bool
    where S: Sequence, S.Iterator.Element == Self.Iterator.Element
    {
        for element in self {
            guard other.contains(element) else { return false }
        }
        return true
    }
}

extension Sequence where Iterator.Element: Hashable {
    func isSubset<S>(of other: S) -> Bool
    where S: Sequence, S.Iterator.Element == Self.Iterator.Element
    {
        let otherSet = Set(other)
        for element in self {
            guard otherSet.contains(element) else { return false }
        }
        return true
    }
}
```

Overloading

Generic overloading - resolution

- pick the most specific one
- done at compile time

```swift
func describe<T>(_ value: T) {
    print("It is a value \(value)")
}

func describe(_ int: Int) {
    print("It is integer \(int)")
}

describe(1)     // Non-generic version
describe(1.0)   // Generic version

let intsArray: [Int] = [ 1, 2, 3 ]
let miscArray: [Any] = [ 1, 1.0, "1" ]
intsArray.forEach(describe)     // Non-generic version
miscArray.forEach(describe)     // Generic version
```

Overloading

Generic overloading - resolution

- For operators, compiler always favor non-generic one.
- Compiles ⬇️ | Does not compile ⬇️

```swift
func square(_ v: Float) -> Float {
    return v * v
}

func square(_ v: Double) -> Double {
    return v * v
}

func square<I: Numeric>(_ v: I) -> I {
    return v * v
}

square(2.0)
square(2)
```

```swift
postfix operator ^^

postfix func ^^(_ v: Float) -> Float {
    return v * v
}

postfix func ^^(_ v: Double) -> Double {
    return v * v
}

postfix func ^^<I: Numeric>(_ v: I) -> I {
    return v * v
}

2.0^^
2^^      ❗ Ambiguous use of operator '^^'
```

Overloading

Generic overloading - nice & deep article on the topic, that also explains why following does not compile in Swift:

```swift
//: Playground - noun: a place where people can play

import Foundation

let a: Double = -(1 + 2) + -(3 + 4) + -(5 + 6) + 7
```

Expression was too complex to be solved in reasonable time; consider breaking up the expression into distinct sub-expressions

Expression was too complex to be solved in reasonable time; consider breaking up the expression into distinct sub-expressions

https://www.cocoawithlove.com/blog/2016/07/12/type-checker-issues.html

Generic Constraints

One can create two types of generic constraints:

- `==` (e.g. `IndexDistance == Int`, `Self.Iterator.Element == Other.Iterator.Element`, `T == UIView`, etc.)

- `:` (e.g. `Self: BidirectionalCollection`, `T: Comparable`, `T: UIView` etc.)

Generic Constraints

== VS :

```swift
extension Comparable where Self == Int {
    func someFunc() {

    }
}

extension Comparable where Self: SignedInteger {
    func otherFunc() {

    }
}

let int: Int = 1
let int8: Int8 = 1
int.someFunc()
int.otherFunc()
//int8.someFunc()    // Does not compile, Int8 is NOT Int
int8.otherFunc()     // Compiles, Int8 conforms to SignedInteger
```

Generic Constraints

== VS :

```swift
class Holder<T> {
    let value: T
    init(value: T) {
        self.value = value
    }
}

extension Holder where T: UIView {
    func someFunc() {}
}

extension Holder where T == UIView {
    func otherFunc() {}
}

let viewHolder = Holder(value: UIView())
let labelHolder = Holder(value: UILabel())
viewHolder.someFunc()
viewHolder.otherFunc()
labelHolder.someFunc()
// labelHolder.otherFunc() // Does not compile
```

Generic Constraints

Summary:

- == for concrete types (both value types and class types) and associated types
- **:** for protocols and class types

Exercises

Do exercise 3 from Generics group.

Do exercises 4-5 from Generics group.

Generics Limitations

Does not compile (being implemented):

```
extension Array: Equatable where
Element: Equatable
```

(conditional protocol conformance)

https://github.com/apple/swift-evolution/blob/master/proposals/0143-conditional-conformances.md

Generics Limitations

Does not compile (cannot assign generic function to non-specified variable):

```
func genericOne<T>(param: T) -> Bool {
    // do sth with T
}


let functionVar = genericOne  // Does not compile


let specifiedFunc: (Int) -> Bool = genericOne // Compiles
```

How Generics work

What does compiler do for such function?

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

It lacks two pieces of information:

1. Size of params and return type (both of type `T`)
2. Address of specific < operator (function) to call

How Generics work

1. Size of params and return type (both of type `T`)

- Compiler **boxes** each generic type value in a **container**
- Container has fixed size to store the value
- If the value is too big, container stores reference

How Generics work

1. Size of params and return type (both of type `T`)

- Compiler also maintains **value witness table** for each generic type `T`

- The value witness table contains pointers to basic functions like allocation, copying, destruction etc.

- Contains also size and alignment of the type.

How Generics work

2. Address of specific < operator (function) to call

- For each protocol (in this case we have one - Comparable), compiler maintains **protocol witness table**.
- For each method or property declared by the protocol, the table contains pointer to concrete implementation for the type.

How Generics work

Value witness table and protocol witness table(s):

- Are used to dynamically dispatch function calls to proper implementations at runtime.
- We could not make any non-basic operation on type `T` without protocol witness table.
- Thus, generics are so closely related to protocols.

How Generics work

How the function may be implemented under the hood?

```
func min<T: Comparable>(_ x: TBox, _ y: TBox,
vwt: ValueWitnessTable,
cPWT: ComparableProtocolWitnessTable)
 -> TBox
{
let xCopy = vwt.copy(x)
let yCopy = vwt.copy(y)
let result = cPWT.lessThan(yCopy, xCopy) ? y : x
vwt.release(xCopy)
vwt.release(yCopy)
return result
}
```

How Generics work

How the function call may look under the hood?

```
let a = 10; let b = 20;
let m = min(TBox(a), TBox(b),
Int.valueWitnessTable,
Int.comparableProtocolWitnessTable)
```

Note: Both examples are in pseudocode, not real code.

How Generics work

Dynamic dispatch causes an overhead

- Compiler may try to alleviate this by creating specialized clones of the function for some concrete types, e.g. `Int` or `String`
  - so if we call `min` function for `Int` or `String` parameters, no dynamic dispatch will occur
  - compiler uses some heuristics to determine types for which it should specialize the function
- We can provide hint to compiler via `@_specialize`

How Generics work

Dynamic dispatch causes an overhead

- If the generic function is not visible outside the module, compiler may even not generate generic function version at all!

Whole module optimization

- Generic specialization only works if compiler can see all of particular generic function calls inside the module.
- However, .swift files are compiled individually...
- ... but this can be alleviated by merging whole swift code into one file and then compiling ...
- ... and this is Whole Module Optimization.

Whole module optimization

- WMO also enables other optimizations, e.g.:
- If there is internal class with no subclasses...
- ... it can be marked as `final` ...
- ... causing dynamic dispatch for it be replaced with static dispatch.

5. Advanced Protocols Stuff

Protocols

Static dispatch VS dynamic dispatch

```
protocol X {
    func someFunc()  // Dynamic dispatch
    func otherFunc()  // Dynamic dispatch
}

extension X {
    func otherFunc() { ... } // Dynamic dispatch
    func additionalFunc() { ... }  // Static disp.
}
```

Exercises

Do exercise 1 from Protocols group.

Protocols

Static dispatch VS dynamic dispatch

```swift
protocol X {
    func someFunc()
    func otherFunc()
}

extension X {
    func otherFunc() {
        print("Default impl")
    }
    func additionalFunc() {
        print("Default impl")
    }
}
```

```swift
struct Impl: X {
    func someFunc() {
        print("Specialized impl")
    }
    func otherFunc() {
        print("Specialized impl")
    }
    func additionalFunc() {
        print("Specialized impl")
    }
}
```

```swift
let impl = Impl()
impl.someFunc()  // Specialized
impl.otherFunc() // Specialized
impl.additionalFunc()  // Specialized

let prot: X = Impl()
prot.someFunc() // Specialized
prot.otherFunc() // Specialized
prot.additionalFunc() // Default
```

Protocols

How can we call default method, not specialized?

What is this?

```
let protocolTypeLet: Protocol =
Impl()
```

- Just like with generics compiler boxes the instance (`Impl()`).
- This box is called **existential container**.

Existential container

Existential container contains:

- buffer for the stored value (24 bytes)
  - or pointer to the value stored on heap
- metadata (8 bytes)
  - information about the type
- number of witness tables (8 bytes each)

Existential container

Class existential container (for class only protocols):

- buffer = pointer (8 bytes)
- metadata not needed (stored in the class)
- number of witness tables (8 bytes each)

Witness table

- allows dynamic dispatch
- for each method in protocol, stores pointer to concrete implementation for the particular type
- there is each witness table for each protocol (remember you can combine protocols like this: `Protocol1 & Protocol2`)

Witness table VS Subclasses

DEMO

of the real problem we faced
in our project

Witness table VS Subclasses

- Protocol witness table (PWT) is created only for the type that states it conforms to the protocol
- Thus, subclasses of the class that stated protocol conformance does NOT get their own PWTs.
- This may lead to unexpected behaviour in some scenarios.

Witness table VS Subclasses

- Protocol `P` declares function `X()`
- Protocol `P` extension provides:
  - default implementation of `X()`
  - default implementation of `Y()`, that calls `X()` inside
- `BaseClass` conforms to `P`, but uses default implementation of `X()` from protocol extension
  - In its witness table, pointer for `X()` function points to default implementation from extension

Witness table VS Subclasses

- `SubClass` of `BaseClass` refines function `X()`
  - However, it does not have its own PWT.
- When function `Y()` is called on `SubClass` instance, it needs to use PWT.
- But `SubClass` does not have PWT, so PWT from `BaseClass` is used instead...
  - ... and that means calling default implementation of `X()`.

Witness table VS Subclasses

```swift
protocol P {
    func x()
}

extension P {
    func x() { print("Default") }
    func y() { print("y() calls x()"); x(); }
}

class BaseClass: P {}

class SubClass: BaseClass {
    func x() { print("Refined") }
}

let s = SubClass()
s.y()   // "Default" is printed on the console
```

Witness table VS Subclasses

Possible solutions to the problem:

- `SubClass: BaseClass, P` - does not compile :/
- Implement function `X()` in `BaseClass`
  - That way class virtual table (used for polymorphism) will solve the problem.
- Refactor your protocol ...
- ... or get rid of inheritance :)

Protocols & Performance

- adds a level of indirection, which can cause performance overhead
- creating a function that has parameters or return type of protocol type may require additional boxing of value(s)
  - especially for collections of protocol value like `[Any]`
    - `Any` is typealias for empty protocol.

Protocols limitations

Not all protocols can be used as:

- variable types
- parameter types
- return types

```
func someFuncOnEquatable(x: Equatable) {
    // Will not compile :/
}

func someFuncOnIterator(s: IteratorProtocol) {
    // Will not compile :/
}
```

Protocol 'Equatable' can only be used as a generic constraint because it has Self o...

Protocol 'IteratorProtocol' can only be used as a generic constraint because it has...

Protocols limitations

These "limited" protocols are:

- Those that has `Self` in their declaration:

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

- Those that have associated types:

```
public protocol IteratorProtocol {
    associatedtype Element
    // ...
}
```

Protocols limitations

Why are they actually limited?

```
// Int implements Equatable
let x: Equatable = 42

// String implements Equatable
let s: Equatable = "Answer to all the things"

// What should compiler do here ?
s == x
```

```
let iterator: IteratorProtocol = SomeIterator()

// What is the type of next?
let next = iterator.next()
```

Protocols limitations

How to deal with it?

```swift
struct ConstantIterator: IteratorProtocol {
    func next() -> Int? {
        return 1
    }
}
```

```swift
struct FibonacciIterator: IteratorProtocol {
    var state = (1, 1)
    mutating func next() -> Int? {
        let current = state.0
        state.0 = state.1
        state.1 = state.1 + current
        return current
    }
}
```

```swift
// We cannot give other type than [Any] ... at least not yet.
let intIteratorsArray = [ ConstantIterator(), FibonacciIterator() ] as [Any]

// Thus we cannot make sth like this
for iter in intIteratorsArray {
    print("\(iter.next()!)")
}
```

Protocols limitations

How to deal with it?

```swift
class AnyIterator<A>: IteratorProtocol {
    var nextImpl: () -> A?
    init<I: IteratorProtocol>(_ iterator: I) where I.Element == A {
        var iteratorCopy = iterator
        self.nextImpl = { iteratorCopy.next() }
    }
    func next() -> A? {
        return nextImpl()
    }
}
```

```swift
let anyIntIterators = [ AnyIterator<Int>(ConstantIterator()),
                        AnyIterator<Int>(FibonacciIterator()) ]

for iter in anyIntIterators {
    print("\(iter.next()!)")  // Prints two 1's
}
```

Protocols limitations

Such constructions are called "type erasers".

- They require storing closures wrapping each method OR reference to underlying protocol.

- There are few examples in standard library: `AnyIterator`, `AnySequence`, `AnyCollection`, etc.
  - In fact, Standard Library erases the type in some more sophisticated way, but we won't go into details.

Numeric protocols - Swift 4

```
                +-------------+   +-------------+
        +------>+   Numeric   |   | Comparable  |
        |       |   (+,-,*)   |   | (==,<,>,...)|
        |       +-------------++  +---+---------+
        |                     ^       ^
+-------+------------+        |       |
|    SignedNumeric   |      +-+-------+-----------+
|     (unary -)      |      |    BinaryInteger    |
+------+-------------+      |(words,%,bitwise,...)|
       ^                    ++---+-----+----------+
       |         +-----------^   ^     ^---------------+
       |         |               |                     |
+------+---------++    +---------+---------------+  +--+----------------+
|  SignedInteger  |    |  FixedWidthInteger      |  |  UnsignedInteger  |
|                 |    |(endianness,overflow,...)|  |                   |
+---------------+-+    +-+--------------------+--+  +-+-----------------+
                ^        ^                    ^       ^
                |        |                    |       |
                |        |                    |       |
               ++--------+-+                +-+-------+-+
               |Int family |-+              |UInt family|-+
               +-----------+ |              +-----------+ |
                 +-----------+                +-----------+
```

https://github.com/apple/swift-evolution/blob/master/proposals/0104-improved-integers.md

6. Basic reflection in Swift

Objective-C's reflection

In Swift we can use Objective-C's reflection in limited way:

- Only on classes that inherit from `NSObject`
- Using read-only functions like: `class_copyPropertyList`, `property_getName`, `method_getImplementation`, etc.
- Swizzling is possible, dynamic adding of Swift function to class - also, but it is not quite convenient.

Objective-C's reflection

When we want to use read-only reflection on pure Swift classes or structs

OR

we do not want to use unsafe and complicated Objective-C runtime API.

we can use **Mirrors.**

Mirror

Mirrors allows some kind of reflection into arbitrary type.

Mostly, we can inspect "children" of a type. Children are either:

- properties of struct / class
- elements of a collection (Array, Set, Dictionary)

Enums do not have children in mirrors ;)

Mirror.Child

- Child is a tuple `(label: String?, value: Any)`
- `label` is the name of the property (nil for collections)
- `value` is the value of the property OR element of the collection.

We can create a mirror for the value of a child and continue inspection recursively.

Mirror

There are 2 useful things in `Mirror` struct

- `children` property which returns collection (AnyCollection) of children mirrored type (may be empty).
- `descendant(...)` method which could be used to seek through descendants of the mirrored type.

descendant()

- Method accepts variable argument list (should be at least 1)
- Each argument is used to seek through descendant tree.
- The arguments could be either `Int` or `String`
  - `Int`: nth child is selected
  - `String`: child for which `label` is equal to provided value

descendant()

`descendant(1, "two", 3)` means

- take second (indices start from 0) child
- in the child's children, take the child that has label `"two"` (grandchild)
- in the grandchild's children, take fourth child (greatgrandchild)

Of course, it may return `nil` if some child cannot be matched during the search.

descendant()

```swift
let rect = CGRect(x: 10.0, y: 20.0, width: 30.0, height: 40.0)
let mirror = Mirror(reflecting: rect)

// NOTE: String(describing: ) is used to silence warnings
print(String(describing: mirror.descendant(0, 1)))              // Prints Optional(20.0)
print(String(describing: mirror.descendant("size", "width")))   // Prints Optional(30.0)
print(String(describing: mirror.descendant("origin", 0)))       // Prints Optional(10.0)
print(String(describing: mirror.descendant(1, 1, 1)))           // Prints nil
```

Mirror

More reflection is expected to come with Swift 4 (Stage 2):

https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20160725/025676.html

Further reading:

https://appventure.me/2015/10/24/swift-reflection-api-what-you-can-do/

https://makeitnew.io/reflection-in-swift-68a06ba0cf0e

Thanks!

Contact me at:

mariusz.m.lisiecki@gmail.com