

Swift Generics



Problem

```
1 func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

Problem

```
1 func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

```
1 var someInt = 3  
2 var anotherInt = 107  
3 swapTwoInts(&someInt, &anotherInt)  
4 print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")  
5 // Prints "someInt is now 107, and anotherInt is now 3"
```

Problem

```
1  func swapTwoStrings(_ a: inout String, _ b: inout String) {  
2      let temporaryA = a  
3      a = b  
4      b = temporaryA  
5  }  
6  
7  func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {  
8      let temporaryA = a  
9      a = b  
10     b = temporaryA  
11 }
```

Решение

Generic Functions

```
1  func swapTwoValues<T>(_ a: inout T, _ b: inout T) {  
2      let temporaryA = a  
3      a = b  
4      b = temporaryA  
5  }
```

Generic Functions

```
1 func swapTwoValues<T>(_ a: inout T, _ b: inout T) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

```
1 func swapTwoInts(_ a: inout Int, b: inout Int)  
2 func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

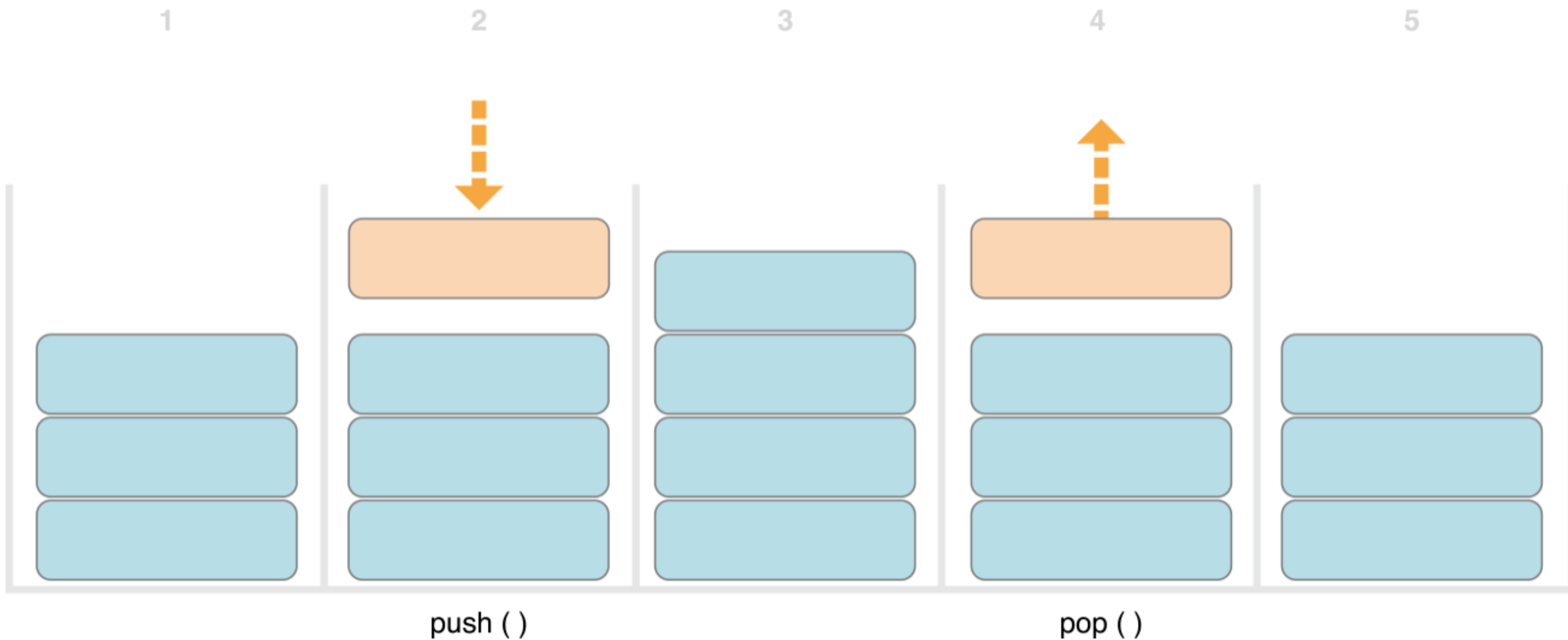

Generic Functions

```
1  var someInt = 3
2  var anotherInt = 107
3  swapTwoValues(&someInt, &anotherInt)
4  // someInt is now 107, and anotherInt is now 3
5
6  var someString = "hello"
7  var anotherString = "world"
8  swapTwoValues(&someString, &anotherString)
9  // someString is now "world", and anotherString is now "hello"
```

Generic Functions

1. `func` funcWithMultiplePlaceholders<T, U, V>(_ a: T, _ b: U, _ c: V)
2. `public struct` Array<Element>

Generic Types



Generic Types

```
1  struct IntStack {  
2      var items = [Int]()  
3      mutating func push(_ item: Int) {  
4          items.append(item)  
5      }  
6      mutating func pop() -> Int {  
7          return items.removeLast()  
8      }  
9  }
```

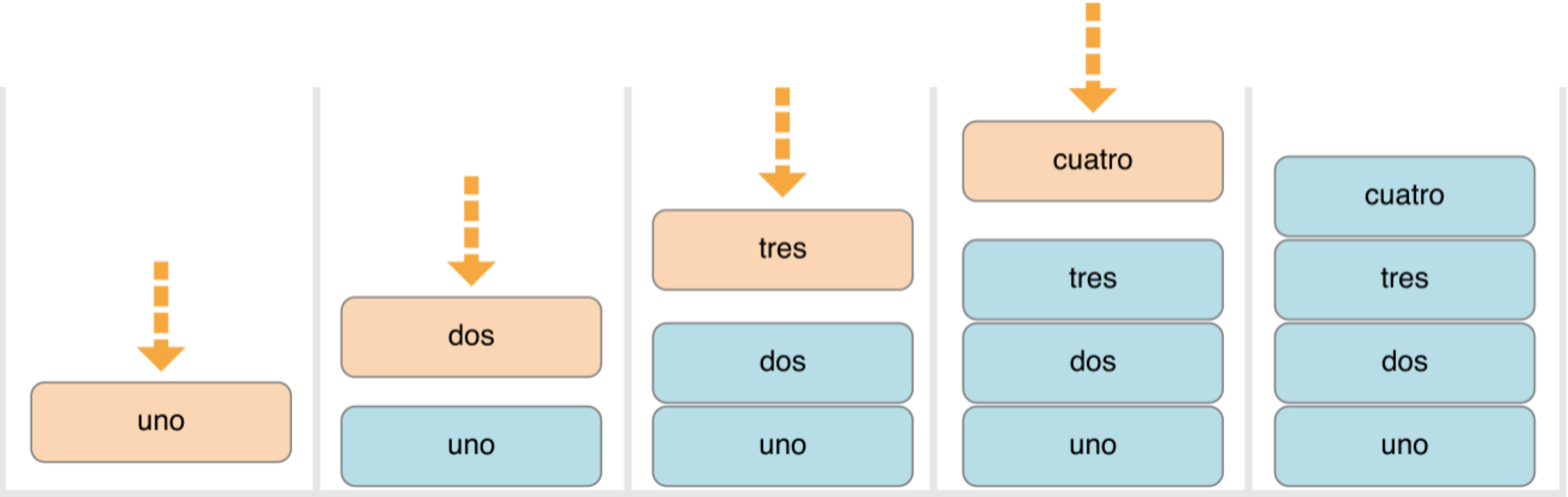

Generic Types

```
1  struct Stack<Element> {  
2      var items = [Element]()  
3      mutating func push(_ item: Element) {  
4          items.append(item)  
5      }  
6      mutating func pop() -> Element {  
7          return items.removeLast()  
8      }  
9  }
```

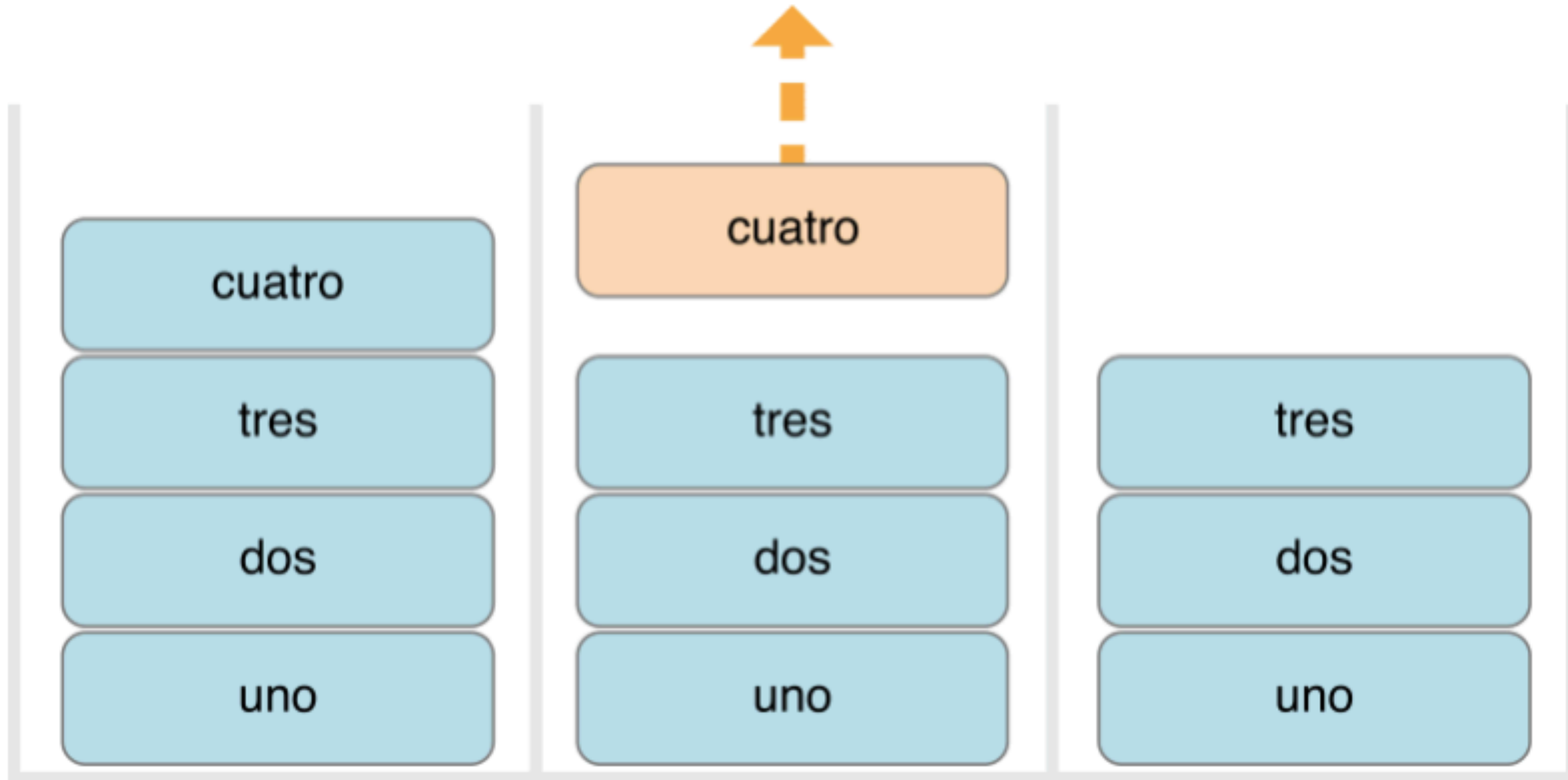
Generic Types

```
1  var stackOfStrings = Stack<String>()  
2  stackOfStrings.push("uno")  
3  stackOfStrings.push("dos")  
4  stackOfStrings.push("tres")  
5  stackOfStrings.push("cuatro")  
6  // the stack now contains 4 strings
```


Generic Types



Generic Types



```
1  let fromTheTop = stackOfStrings.pop()  
2  // fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```


Extending a Generic Type

```
1  extension Stack {  
2      var topItem: Element? {  
3          return items.isEmpty ? nil : items[items.count - 1]  
4      }  
5  }
```

```
1  if let topItem = stackOfStrings.topItem {  
2      print("The top item on the stack is \$(topItem).")  
3  }  
4  // Prints "The top item on the stack is tres."
```

Type Constraints

Generic Structure

Dictionary

A collection whose elements are key-value pairs.

Declaration

```
struct Dictionary<Key, Value> where Key : Hashable
```

Type Constraints

```
1 func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {  
2     // function body goes here  
3 }
```

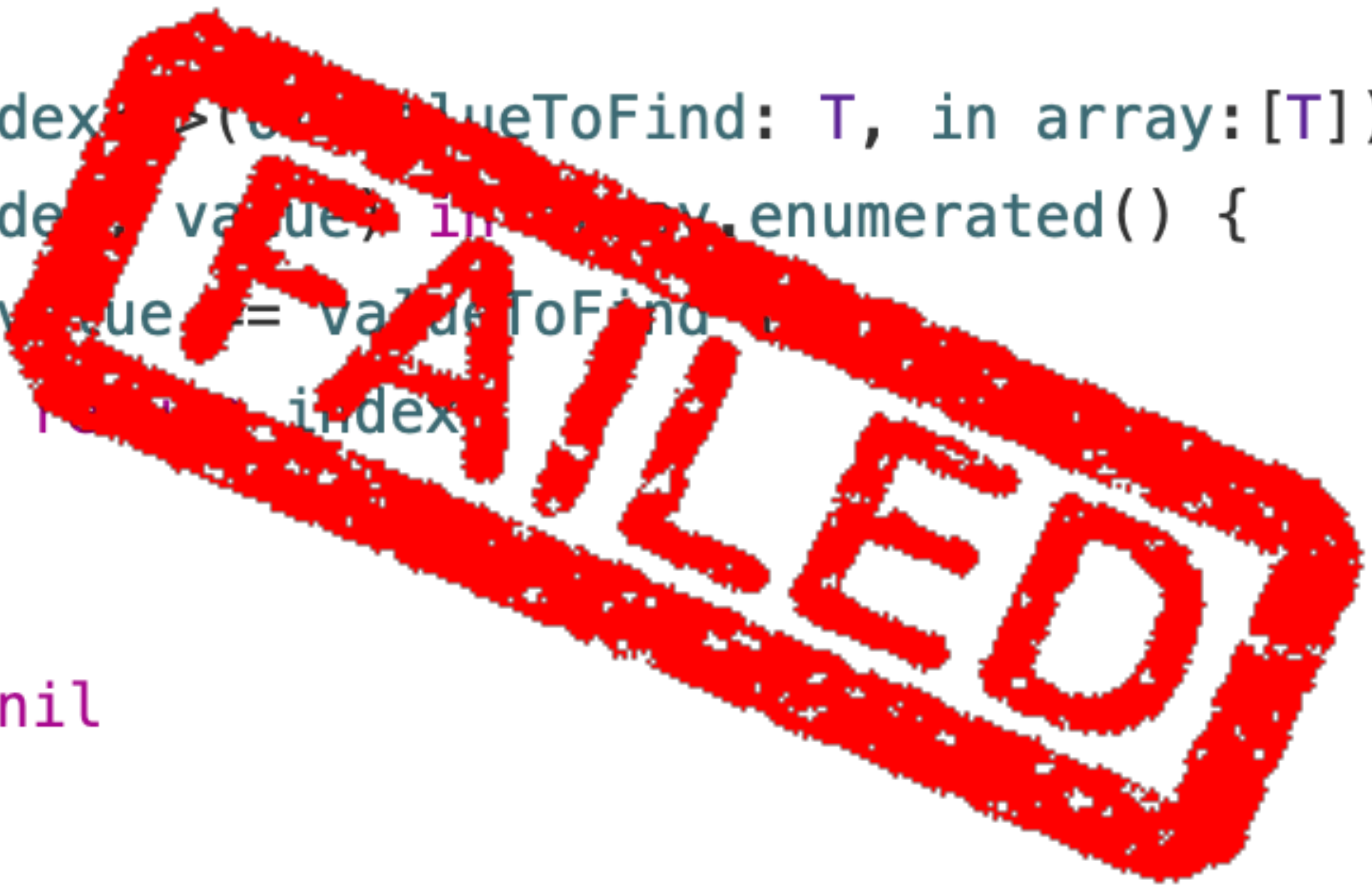

Type Constraints

```
1 func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {  
2     for (index, value) in array.enumerated() {  
3         if value == valueToFind {  
4             return index  
5         }  
6     }  
7     return nil  
8 }
```

```
1 let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]  
2 if let foundIndex = findIndex(ofString: "llama", in: strings) {  
3     print("The index of llama is \(foundIndex)")  
4 }  
5 // Prints "The index of llama is 2"
```

Type Constraints

```
1 func findIndex(>(valueToFind: T, in array:[T]) -> Int? {  
2     for (index, value) in array.enumerated() {  
3         if value == valueToFind {  
4             return index  
5         }  
6     }  
7     return nil  
8 }
```



Type Constraints

```
1 func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {  
2     for (index, value) in array.enumerated() {  
3         if value == valueToFind {  
4             return index  
5         }  
6     }  
7     return nil  
8 }
```


Type Constraints

```
1 func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {  
2     for (index, value) in array.enumerated() {  
3         if value == valueToFind {  
4             return index  
5         }  
6     }  
7     return nil  
8 }
```

```
1 let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])  
2 // doubleIndex is an optional Int with no value, because 9.3 isn't in the  
   array  
3 let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm",  
   "Andrea"])  
4 // stringIndex is an optional Int containing a value of 2
```

Задача

Associated Types

```
1  protocol Container {  
2      associatedtype Item  
3      mutating func append(_ item: Item)  
4      var count: Int { get }  
5      subscript(i: Int) -> Item { get }  
6  }
```


Associated Types

```
1  struct IntStack: Container {
2      // original IntStack implementation
3      var items = [Int]()
4      mutating func push(_ item: Int) {
5          items.append(item)
6      }
7      mutating func pop() -> Int {
8          return items.removeLast()
9      }
10     // conformance to the Container protocol
11     typealias Item = Int
12     mutating func append(_ item: Int) {
13         self.push(item)
14     }
15     var count: Int {
16         return items.count
17     }
18     subscript(i: Int) -> Int {
19         return items[i]
20     }
21 }
```

Associated Types

```
1  struct Stack<Element>: Container {
2      // original Stack<Element> implementation
3      var items = [Element]()
4      mutating func push(_ item: Element) {
5          items.append(item)
6      }
7      mutating func pop() -> Element {
8          return items.removeLast()
9      }
10     // conformance to the Container protocol
11     mutating func append(_ item: Element) {
12         self.push(item)
13     }
14     var count: Int {
15         return items.count
16     }
17     subscript(i: Int) -> Element {
18         return items[i]
19     }
20 }
```

Associated Types

Generic Structure

Array

An ordered, random-access collection.

Declaration

```
struct Array<Element>
```

```
extension Array: Container {}
```


Associated Types

```
1  protocol Container {  
2      associatedtype Item: Equatable  
3      mutating func append(_ item: Item)  
4      var count: Int { get }  
5      subscript(i: Int) -> Item { get }  
6  }
```

Associated Types

```
1 protocol SuffixableContainer: Container {  
2     associatedtype Suffix: SuffixableContainer where Suffix.Item == Item  
3     func suffix(_ size: Int) -> Suffix  
4 }
```

Associated Types

```
1  extension Stack: SuffixableContainer {
2      func suffix(_ size: Int) -> Stack {
3          var result = Stack()
4          for index in (count-size)..<count {
5              result.append(self[index])
6          }
7          return result
8      }
9      // Inferred that Suffix is Stack.
10 }
11 var stackOfInts = Stack<Int>()
12 stackOfInts.append(10)
13 stackOfInts.append(20)
14 stackOfInts.append(30)
15 let suffix = stackOfInts.suffix(2)
16 // suffix contains 20 and 30
```

Associated Types

```
1  extension IntStack: SuffixableContainer {  
2      func suffix(_ size: Int) -> Stack<Int> {  
3          var result = Stack<Int>()  
4          for index in (count-size)..<count {  
5              result.append(self[index])  
6          }  
7          return result  
8      }  
9      // Inferred that Suffix is Stack<Int>.  
10 }
```


Generic Where Clauses

```
1  func allItemsMatch<C1: Container, C2: Container>
2      (_ someContainer: C1, _ anotherContainer: C2) -> Bool
3      where C1.Item == C2.Item, C1.Item: Equatable {
4
5          // Check that both containers contain the same number of items.
6          if someContainer.count != anotherContainer.count {
7              return false
8          }
9
10         // Check each pair of items to see if they're equivalent.
11         for i in 0..
```

Generic Where Clauses

```
1  var stackOfStrings = Stack<String>()
2  stackOfStrings.push("uno")
3  stackOfStrings.push("dos")
4  stackOfStrings.push("tres")
5
6  var arrayOfStrings = ["uno", "dos", "tres"]
7
8  if allItemsMatch(stackOfStrings, arrayOfStrings) {
9      print("All items match.")
10 } else {
11     print("Not all items match.")
12 }
13 // Prints "All items match."
```

Extensions with a Generic Where Clause

```
1  extension Stack where Element: Equatable {  
2      func isTop(_ item: Element) -> Bool {  
3          guard let topItem = items.last else {  
4              return false  
5          }  
6          return topItem == item  
7      }  
8  }
```

Extensions with a Generic Where Clause

```
1  if stackOfStrings.isTop("tres") {  
2      print("Top element is tres.")  
3  } else {  
4      print("Top element is something else.")  
5  }  
6  // Prints "Top element is tres."
```


Extensions with a Generic Where Clause

```
1  extension Container where Item: Equatable {  
2      func startsWith(_ item: Item) -> Bool {  
3          return count >= 1 && self[0] == item  
4      }  
5  }
```

```
1  if [9, 9, 9].startsWith(42) {  
2      print("Starts with 42.")  
3  } else {  
4      print("Starts with something else.")  
5  }  
6  // Prints "Starts with something else."
```

Extensions with a Generic Where Clause

```
1  extension Container where Item == Double {  
2      func average() -> Double {  
3          var sum = 0.0  
4          for index in 0..  
5              sum += self[index]  
6          }  
7          return sum / Double(count)  
8      }  
9  }  
10 print([1260.0, 1200.0, 98.6, 37.0].average())  
11 // Prints "648.9"
```

Associated Types with a Generic Where Clause

```
1  protocol Container {  
2      associatedtype Item  
3      mutating func append(_ item: Item)  
4      var count: Int { get }  
5      subscript(i: Int) -> Item { get }  
6  
7      associatedtype Iterator: IteratorProtocol where Iterator.Element ==  
8      Item  
9      func makeIterator() -> Iterator  
}
```

Associated Types with a Generic Where Clause

```
protocol ComparableContainer: Container where Item: Comparable { }
```


Read

- <https://docs.swift.org/swift-book/LanguageGuide/Generics.html>
- <https://www.appcoda.com/swift-generics/>
- <https://developer.apple.com/videos/play/wwdc2018/406/>

Homework

1. В Playground **SwiftGenerics_01** раскомментировать указанный код и сделать так, чтобы он работал.

```
15
16 //let c = "ABC"
17 //let d = "DEF"
18 //
19 //let resultString = sumTwoValues(c, d)
20 //print(resultString)
21
```

2. В Playground **SwiftGenerics_02** создать систему generic-типов. Типы могут быть свои либо можно использовать предложенный ниже вариант

“Реализовать базовый протокол для контейнеров. Контейнеры должны отвечать, сколько они содержат элементов, добавлять новые элементы и возвращать элемент по индексу. На основе базового протокола реализовать универсальный связанный список и универсальную очередь (FIFO) в виде структуры или класса. ”

3. * В Playground **SwiftGenerics_03** реализовать универсальный связанный список на основе enum.

Домашнее задание принимается в срок до 19-го марта включительно в виде проекта на github или архива в облаке