

# The Swift Programming Language

Generics

osxdev (hothead)

성기평 ([sungkipyung@gmail.com](mailto:sungkipyung@gmail.com))

**gen·er·ic**

US.UK [dʒəˈnerɪk]   | UK  

저네릭 이라네요

시작 전 공지

[https://github.com/sungkiyoung/Swift\\_Generic\\_osxdev](https://github.com/sungkiyoung/Swift_Generic_osxdev)

# The Problem That Generics Solve

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

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

# The Problem That Generics Solve

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

# Generic Functions

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
var someInt = 3  
var anotherInt = 107  
swapTwoValues(&someInt, &anotherInt)  
// someInt is now 107, and anotherInt is now 3
```

```
var someString = "hello"  
var anotherString = "world"  
swapTwoValues(&someString, &anotherString)  
// someString is now "world", and anotherString is now "hello"
```

# Type Parameters syntax

// 이름을 지을 수 있어요, 그런데 좀 헷갈리죠?

```
func read<Input>() -> Input? {  
    var obj:Input? = nil  
    // generate object  
    return obj  
}
```

// 여러개를 쓸 때는 comma ", "로 분리 해주면 됩니다.

```
func process<Input, Ouput>(_input: Input) -> Ouput? {  
    // processing  
    return nil  
}
```

```
func write<Output>(output: Output) {  
    // write output  
}
```

# Generic Types

```
struct IntStack {  
    var items = [Int]()  
    mutating func push(_ item: Int) {  
        items.append(item)  
    }  
    mutating func pop() -> Int {  
        return items.removeLast()  
    }  
}  
  
struct Stack<Element> {  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
}  
  
var stackOfStrings = Stack<String>()  
stackOfStrings.push("uno")  
stackOfStrings.push("dos")  
stackOfStrings.push("tres")  
stackOfStrings.push("cuatro")  
// the stack now contains 4 strings  
  
let fromTheTop = stackOfStrings.pop()  
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

# Extending a Generic Type

```
extension Stack {  
    var topItem: Element? {  
        return items.isEmpty ? nil : items[items.count - 1]  
    }  
}  
  
if let topItem = stackOfStrings.topItem {  
    print("The top item on the stack is \(topItem).")  
}  
// Prints "The top item on the stack is tres."
```



# Type Constraints in Action

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

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

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

```
/**
 Playground execution failed: error: Type Constraints in Action .xcplaygroundpage:16:18: error:
 binary operator '==' cannot be applied to two 'T' operands
   if value == valueToFind {
   ~~~~~ ^ ~~~~~
 */
```

# Type Constraints in Action

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

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

# Associated Types

```
protocol Container {  
    associatedtype ItemType  
    mutating func append(_ item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}
```

```
struct IntStack: Container {  
    // original IntStack implementation  
    var items = [Int]()  
    mutating func push(_ item: Int) {  
        items.append(item)  
    }  
    mutating func pop() -> Int {  
        return items.removeLast()  
    }  
    // conformance to the Container protocol  
    typealias ItemType = Int  
    mutating func append(_ item: Int) {  
        self.push(item)  
    }  
    var count: Int {  
        return items.count  
    }  
    subscript(i: Int) -> Int {  
        return items[i]  
    }  
}
```

# Associated Types

```
struct Stack<Element>: Container {  
    // original Stack<Element> implementation  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
    // conformance to the Container protocol  
    mutating func append(_ item: Element) {  
        self.push(item)  
    }  
    var count: Int {  
        return items.count  
    }  
    subscript(i: Int) -> Element {  
        return items[i]  
    }  
}
```

# Extending an Existing Type to Specify an Associated Type

```
protocol Container {  
    associatedtype ItemType  
    mutating func append(_ item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}  
  
// Array는 이미 append, count, subscript 조건을 만족하기 때문에  
// 그대로 두어도 동작하게 된다.  
extension Array: Container {  
  
}
```

# Generic Where Clauses

```
func allItemsMatch<C1: Container, C2: Container>
  (_ someContainer: C1, _ anotherContainer: C2) -> Bool
  where C1.ItemType == C2.ItemType, C1.ItemType: Equatable {

    // Check that both containers contain the same number of items.
    if someContainer.count != anotherContainer.count {
      return false
    }

    // Check each pair of items to see if they are equivalent.
    for i in 0..
```

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
  print("All items match.")
} else {
  print("Not all items match.")
}
```

# Generic Where Clauses

```
class Person {  
    var name: String  
    init(name: String) {  
        self.name = name  
    }  
}  
  
//extension Person: Equatable {  
//    public static func ==(lhs: Person, rhs: Person) -> Bool {  
//        return lhs.name == rhs.name  
//    }  
//}  
  
var stackOfPerson = Stack<Person>()  
  
stackOfPerson.push(Person(name: "tom"))  
stackOfPerson.push(Person(name: "bob"))  
  
var arrayOfPersons = [Person(name: "tom"), Person(name: "bob")]  
  
if allItemsMatch(stackOfPerson, arrayOfPersons) {  
    print ("all persons match.")  
} else {  
    print ("not all persons match.")  
}
```

# Reference

- Generics (apple document)
- [https://github.com/sungkiyoung/Swift\\_Generic\\_osxdev](https://github.com/sungkiyoung/Swift_Generic_osxdev)