SWIFT
Generics

Bill Kim(김정훈) | ibillkim@gmail.com

목차

Generics

Generic Functions

Type Parameters

Generic Types

Type Constraints

Associated Types

Generic Where Clauses

References

Generics

Generic을 사용하면 재사용 가능하고 유연한 코드를 작성할 수 있습니다.

Swift에서 제공하는 C++에서의 템플릿과 유사한 개념이라고 생각할 수 있습니다.

Generic Functions

Generic을 함수에서 사용하면 아래와 같이 사용할 수 있습니다.

```
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
}

// Generics 함수를 사용하여 다음과 같이 다양한 타입의 파라미터를 받을 수 있습니다.
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

swapTwoValues(&someInt, &anotherInt)
swapTwoValues(&someString, &anotherString)
```

Type Parameters

앞에서 사용한 플레이스 홀더 T는 타입 파라미터의 예입니다.

타입 파라미터는 플레이스 홀더 타입의 이름을 명시하고 함수명 바로 뒤에 적어줍니다. 그리고 와 같이 꺽쇄로 묶어 줍니다.

타입 파라미터를 한번 선언하면 이 것을 함수의 타입으로 사용할 수 있습니다.

복수의 타입 파라미터를 사용할때는 콤마로 구분해 줍니다.

Dictionary의 Key, Value와 같이 엘리먼트 간의 서로 상관관계가 있는 경우 의미가 있는 이름을 파라미터 이름으로 붙이고 그렇지 않은 경우는 T, U, V와 같은 단일 문자로 파라미터 이름을 짓습니다.

Generic Types

제네릭 함수에 추가로 Swift에서는 제네릭 타입을 정의할 수 있습니다.

제네릭 타입이란 “타입을 파라미터화해서 컴파일시 구체적인 타입이 결정되도록 하는 것“이란 뜻으로 Swift에서도 제네릭 타입을 지원합니다.

```swift
// EEItem: 임의로 설정한 이름으로 어떠한 타입도 받을 수 있음
func makeArray<EEItem>(repeating item: EEItem, numberOfTimes: Int) ->
[EEItem] {
    var result = [EEItem]() // 배열의 초기화
    for _ in 0 ..< numberOfTimes {
        result.append(item)
    }
    return result
}

let res1 = makeArray(repeating: "knock", numberOfTimes: 5)
print(res1)
```

Type Constraints

Swift의 Dictionary 타입은 key값을 사용합니다.

이때 key는 유일한 값이어야 하기 때문에 hashable이라는 프로토콜을 반드지 따라야 합니다.

이와 같이 Generics은 특정 타입이 반드시 어떤 프로토콜을 따라야 하도록 타입에 대한 제한을 설정할 수 있습니다.

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

        // == 등호 메소드를 사용하기 위해서는 두 값 혹은 객체가 반드시
        // Equatable 프로토콜을 따라야한다.
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Associated Types

연관타입(Associated Types)은 프로토콜의 일부분으로 타입에 플레이스홀더 이름을 부여합니다.

다시 말해 특정 타입을 동적으로 지정해 사용할 수 있습니다.

associatedtype를 사용하여 지정하면 해당 Item은 어떤 타입도 될 수 있습니다.

```
protocol Container {
    associatedtype Item
    var count: Int { get }
}

struct IntStack: Container {
    // Item을 Int형으로 선언해 사용합니다.
    typealias Item = Int
    var items = [Int]()

    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    var count: Int {
        return items.count
    }
}
```

Generic Where Clauses

제네릭에서도 where절을 사용할 수 있습니다.

```
// Container C1, C2를 비교하며 모든 값이 같을 때 true를 반환
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: 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're equivalent.
        for i in 0..< someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // All items match, so return true.
        return true
}
```

Generic Where Clauses

제네릭의 익스텐션을 선언할때 where절을 포함시킬 수 있습니다.

```
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        // Element 을 Equatable 프로토콜을 따르도록 하였기에 컴파일 에러 미발생
        return topItem == item
    }
}
```

Generic Where Clauses

연관 타입(Associated Types)에도 where절을 적용해 제한을 둘 수 있습니다.

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

    associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
    func makeIterator() -> Iterator
}

// 다른 프로토콜을 상속하는 프로토콜에도 where절로 조건을 부여할 수 있습니다.
protocol ComparableContainer: Container where Item: Comparable { }
```

Generic Where Clauses

제네릭의 서브스크립트에도 조건(where)을 걸 수 있습니다.

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

```
extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
```

References

[1] [Swift]Generics 정리 : http://minsone.github.io/mac/ios/swift-generics-summary

[2] Swift) Generic : https://zeddios.tistory.com/226

[3] Generic Parameters and Arguments : https://docs.swift.org/swift-book/ReferenceManual/GenericParametersAndArguments.html

[4] 제네릭(Generics) : https://kka7.tistory.com/128

[5] Swift Generics Tutorial: https://www.raywenderlich.com/3535703-swift-generics-tutorial-getting-started

References

[6] [Swift] 프로토콜과 제네릭 그리고 열거형 : https://baked-corn.tistory.com/133

[7] Generics in Swift 4 : https://medium.com/developermind/generics-in-swift-4-4f802cd6f53c

[8] Swift) 제네릭(Generics) : https://atelier-chez-moi.tistory.com/80

[9] Generics : https://www.swiftbysundell.com/basics/generics/

[10] Swift - Generics : https://www.tutorialspoint.com/swift/swift_generics.htm

Thank you!