옵셔널 체이닝
(Optional Chaining)

과제:옵셔널을 언래핑하는 여러가지 방법

- var x : String? = "Hi"
- forced unwrapping : unsafe.

```
x!
```

- optional binding : safe.

```
if let a = x {
  print(a)
}
```

- nil coalescing operator : safe.

```
let c = x ?? ""
```

- optional chaining : safe.

```
let b = x?.count
```

```
//The number of characters in a string.
```

```
//변형하여 실습 결과 자세히 설명
//너무 중요한 실습입니다!!
var x : String? = "Hi"//Hi지우고도 실습
print(x, x!)
if let a = x {
 print(a)
}
let b = x!.count
print(type(of:b),b)

let b1 = x?.count
print(type(of:b1),b1, b1!)

let c = x ?? ""
print(c)
```

Optional Chaining이란?

- https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html
- Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil.
- If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil.
  - 옵셔널 변수(메서드, subscript)가 값이 있으면 호출이 성공하고, 아니면 nil반환
- Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.
- cell.textLabel?.text
  - 성공하면 옵셔널 값, 실패하면 nil

Optional Chaining 예

- https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html
- 옵셔널형의 프로퍼티나 메서드 호출 뒤에 "?" 사용
  - 지금까지 옵셔널 변수를 언래핑할 때는 !를 사용했다.
- (pLocation?.coordinate.latitude)!
- tabBarController?.selectedIndex = 1
- cell.textLabel?.text = items[(indexPath as NSIndexPath).row]
- rectangleAdView?.delegate = self
- audioPlayer?.volume = volumeControl.value
- audioRecorder?.recording
- audioRecorder?.record()
- locationManager?.requestWhenInUseAuthorization()

Optional Chaining 예

```
class Person {
  var name: String
  var age: Int
  init(name: String, age: Int) {
   self.name = name
   self.age = age
  }
}
let kim: Person = Person(name: "Kim", age: 20)
print(kim.age)
let han: Person? = Person(name: "Han", age: 25)
print(han.age)
//print(han!.age)
// print(han?.age) //Optional(25), 옵셔널 체이닝
// print((han?.age)!)
// if let hanAge = han?.age {
// print(hanAge)
// } else {
// print("nil")
// }
```

옵셔널 체이닝을 쓰는 이유

- 옵셔널 타입으로 정의된 값이 프로퍼티나 메서드를 가지고 있을 때, 다중 if를 쓰지 않고 간결하게 코드를 작성하기 위해
- 옵셔널 타입의 데이터는 연산이 불가능
  - 연산을 하기 위해서는 옵셔널을 해제 해야 하는데, 많은 양의 옵셔널 타입의 데이터의 경우 다시 한번 옵셔널 타입으로 변경을 하면서 해제를 시켜줌

```
if let s = p.sns {
    if let f = s.fb {
        print("\(f.account)")
    }
}
```

```
print("\(p.sns!.fb!.account)")    //crash 가능성 있음
print("\(p.sns?.fb?.account)")    //옵셔널 체인, 최종은 옵셔널 타입이 됨
```

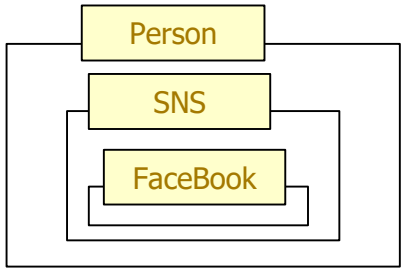
옵셔널 체이닝 예: p.sns?.fb?.account

```
class Person {
    var name : String?
    var age : Int?
    var sns : SNS? = SNS()
}
class SNS {
    var fb : FaceBook? = FaceBook()
    var tt : Twitter?
}
class FaceBook {
    var account : String = "aaa@bbb.com"
}
class Twitter {
    var account : String = ""
}
```

옵셔널 체인의 특성은
옵셔널 체인으로 클래스나 구조체의 프로퍼티를 참조할 경우
결과 값이 nil 이어도 오류가 발생하지 않는다는 것과
옵셔널 체인으로 읽어낸 마지막 값이 일반 타입이라도
모두 옵셔널 타입으로 리턴 된다는 것이다.
메서드의 경우 괄호 다음에 ?함 : p.getM()?
리턴값을 옵셔널 체인으로 사용

```
let p = Person()

if let s = p.sns {
    if let f = s.fb {
        print("1: \(f.account)")
    }
}
if let account = p.sns?.fb?.account {
//옵셔널 체이닝 결과는 옵셔널값이므로 다시한번 옵셔널 바인딩
    print("2: \(account)")
}
print("3: \((p.sns?.fb?.account)!)")//옵셔널 체이닝,비추
print("4: \(p.sns!.fb!.account)")
//print("5: \(p.sns?.tt?.account)")  //nil
//print("6: \(p.sns!.tt!.account)")  //오류!!!!
```



오류 처리
Error Handling

Swift에서 오류 처리

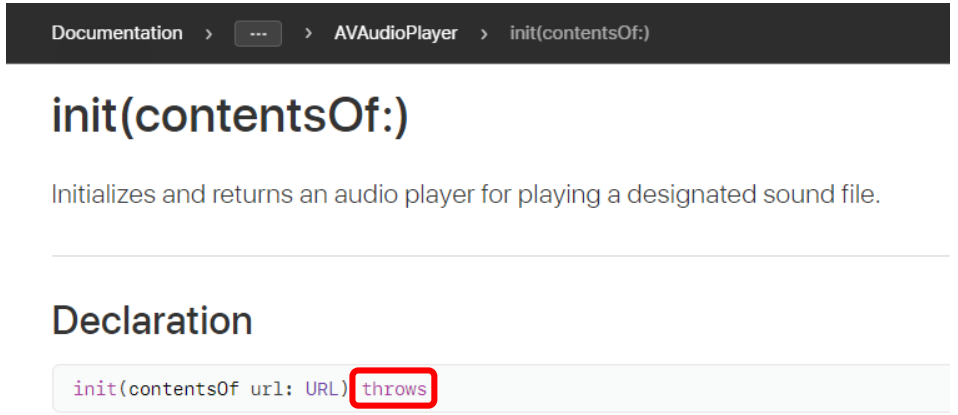
- https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html
- 예외처리(exception handling)
- 런타임 시 오류를 발견하여 응답하고 복구하는 과정
- Swift에서는 optional을 사용하여 값의 유무를 전달함으로써 작업의 성공/실패 유무를 판단할 수 있지만 작업이 실패할 때 코드가 적절히 응답할 수 있도록 함으로써 오류의 원인을 이해하는 데 도움을 줄 수 있다.
- 디스크상의 파일을 읽어서 처리하는 작업에서 발생할 수 있는 오류
  - '존재하지 않는 파일', '읽기 권한 없음', '호환되는 형식이 아님' 등 다양
  - 오류의 원인에 따라 다양한 대응이 필요한 경우, 오류의 정보를 정확히 전달함으로써 오류를 복구하는데 도움을 줄 수 있음
- Swift 2.0 이후부터는 error handling을 도입

Swift 오류 제어

- 오류가 발생하면 문제를 보정하는 방법을 시도하거나 사용자에게 알림으로써 오류를 처리
- Swift에서 오류를 처리 할 수 있는 네 가지 방법
  - Throwing Functions을 이용한 오류 전파(Propagating Errors Using Throwing Functions)
  - Do-Catch를 이용한 오류 처리(Handling Errors Using Do-Catch)
  - Error를 Optional Values로 변환(Converting Errors to Optional Values)
  - 오류 전파 비활성화(Disabling Error Propagation)

throwing function

- To indicate that a function, method, or initializer can throw an error, you write the throws keyword in the function's declaration after its parameters.
- A function marked with throws is called a throwing function.
- 매개변수 괄호 다음에 throws라는 키워드가 있는 함수는 그냥 사용할 수 없고 error handing을 해야 함
- func can() throws
  - 리턴값이 없는 throwing function
- func canThrowErrors() throws -> String
  - error handing을 해야하는 함수
- func cannotThrowErrors() -> String
  - error handing할 수 없는 함수



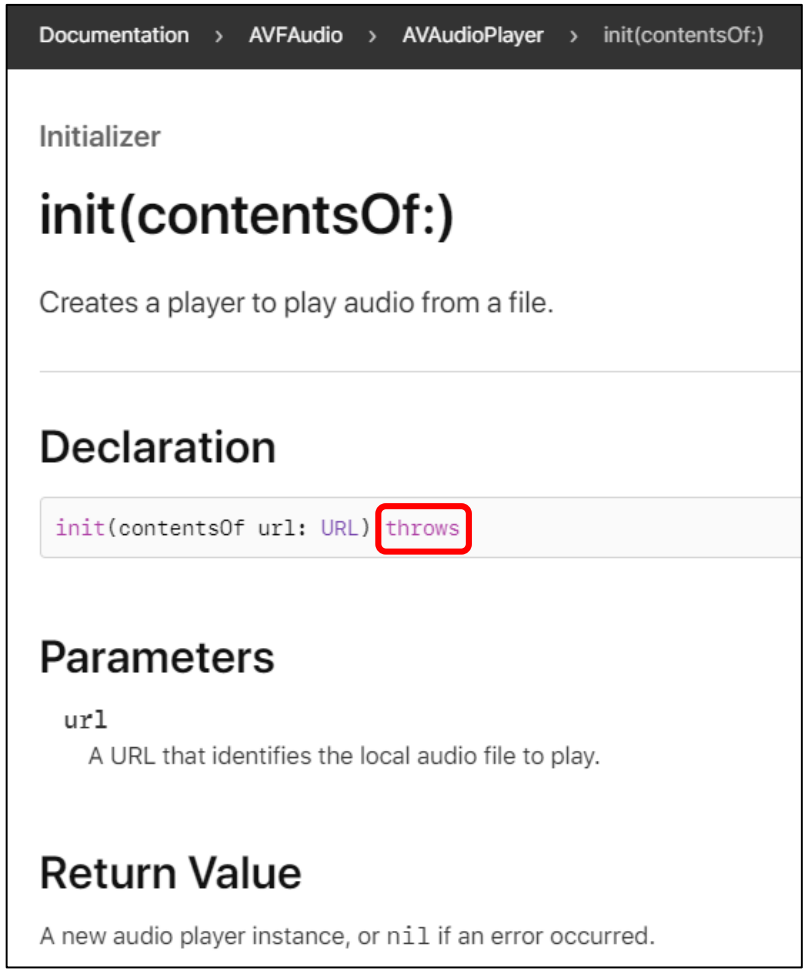
https://developer.apple.com/documentation/avfoundation/avaudioplayer

오류 발생 가능 함수의 호출 방식(do~try~catch)

```
do {
    audioPlayer = try AVAudioPlayer(contentsOf: audioFile)
} catch let error as NSError {
    print("Error-initPlay : \(error)")
}
```

- 이렇게 그냥 호출할 수는 없음
  - AVAudioPlayer(contentsOf: audioFile)
- do~try~catch로 error handing해야 함
  - 하지 않으면 Call can throw, but it is not marked with 'try' and the error is not handled" 오류가 발생

https://developer.apple.com/documentation/avfoundation/avaudioplayer



do~catch을 이용한 error handling

```
do {
    try 오류 발생 코드
    오류가 발생하지 않으면 실행할 코드
} catch 오류패턴1 {
    처리 코드
} catch 오류패턴2 where 조건 {
    처리 코드
} catch {
    처리 코드
}
```

throwing function을 error handling하지 않으면 오류

- do~try~catch로 error handing해야 함
  - 하지 않으면 Call can throw, but it is not marked with 'try' and the error is not handled" 오류가 발생

Generic

<>

Swift Generic

- https://en.wikipedia.org/wiki/Generic_programming
- https://docs.swift.org/swift-book/LanguageGuide/Generics.html
  - Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code.
- `var a : [Int] = [1,2,3,4]`
- `var b : Array<Int>  = [1,2,3,4]`
  - https://developer.apple.com/documentation/swift/array
  - Array
  - Generic Structure
  - @frozen struct Array<Element>
- https://developer.apple.com/documentation/uikit/uiresponder/1621142-touchesbegan
  - func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)

오류가 발생하지 않도록 함수를 수정하시오.

```
func myPrint(a: Int, b: Int) {
    print(b,a)
}
myPrint(a:1,b:2)
myPrint(a:2.5,b:3.5)
//error: cannot convert value of type 'Double' to expected argument type 'Int'
```

기능은 같고 매개변수형만 다른 함수는 제네릭 함수로 구현

```
func myPrint<T>(a: T, b: T) {
    print(b,a)
}
myPrint(a:1,b:2)
myPrint(a:2.5,b:3.5)
//myPrint(a:"Hi",b:"Hello")  //가능?
```

기능은 같고 매개변수형만 다른 함수

```
func swapInt(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
var x = 10
var y = 20
swapInt(&x, &y)
print(x,y)
func swapDouble(_ a: inout Double, _ b: inout Double) {
    let temp = a
    a = b
    b = temp
}
var xd = 10.3
var yd = 20.7
swapDouble(&xd, &yd)
print(xd,yd)
```

```
func swapString(_ a: inout String, _ b: inout String)
{
    let temp = a
    a = b
    b = temp
}
var xs = "Hi"
var ys = "Hello"
swapString(&xs, &ys)
print(xs,ys)
```

기능은 같고 매개변수형만 다른 함수

```
func swapInt(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
func swapDouble(_ a: inout Double, _ b: inout Double) {
    let temp = a
    a = b
    b = temp
}
func swapString(_ a: inout String, _ b: inout String) {
    let temp = a
    a = b
    b = temp
}
```

```
func swapAny<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
} //T는 type이름
```

주의 : swift에는 이미 swap함수가 있음으로 다른 이름 사용해야 함

```
public func swap<T>(_ a : inout, _ b : inout T)
```

기능은 같고 매개변수형만 다른 함수 : generic 함수

```
func swapAny<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}
var x = 10
var y = 20
swapAny(&x, &y)
print(x,y)
var xd = 10.3
var yd = 20.7
swapAny(&xd, &yd)
print(xd,yd)
var xs = "Hi"
var ys = "Hello"
swapAny(&xs, &ys)
print(xs,ys)
```

Int형 스택 구조체

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}

//구조체는 value타입이라 메서드 안에서
//프로퍼티 값 변경불가
//mutating 키워드를 쓰면 프로퍼티 값 변경 가능
```

```
var stackOfInt = IntStack()
print(stackOfInt.items) //[]
stackOfInt.push(1)
print(stackOfInt.items) //[1]
stackOfInt.push(2)
print(stackOfInt.items) //[1,2]
stackOfInt.push(3)
print(stackOfInt.items) //[1,2,3]
print(stackOfInt.pop()) //3
print(stackOfInt.items) //[1,2]
print(stackOfInt.pop()) //2
print(stackOfInt.items) //[1]
print(stackOfInt.pop()) //1
print(stackOfInt.items) //[]
```

일반 구조체 vs. generic 구조체

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}

//구조체는 value타입이라 메서드 안에서
//프로퍼티 값 변경불가
//mutating 키워드를 쓰면 가능
```

```
struct Stack <T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

Generic 스택 구조체에서 Int형 사용

```
struct Stack <T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

```
var stackOfInt = Stack<Int>()
//var stackOfInt = IntStack()
print(stackOfInt.items)  //[]
stackOfInt.push(1)
print(stackOfInt.items) //[1]
stackOfInt.push(2)
print(stackOfInt.items)
stackOfInt.push(3)
print(stackOfInt.items)
print(stackOfInt.pop())  //3
print(stackOfInt.items)
print(stackOfInt.pop())
print(stackOfInt.items)
print(stackOfInt.pop())
print(stackOfInt.items) //[]
```

Generic 스택 구조체에서 Int형, String형 사용

```
struct Stack <T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}

//다양한 자료형에 대해 같은 알고리즘 적용
```

```
var stackOfInt = Stack<Int>()
stackOfInt.push(1)
print(stackOfInt.items)
stackOfInt.push(2)
print(stackOfInt.items)
print(stackOfInt.pop())
print(stackOfInt.items)
print(stackOfInt.pop())

var stackOfString = Stack<String>()
stackOfString.push("일")
print(stackOfString.items)
stackOfString.push("이")
print(stackOfString.items)
print(stackOfString.pop())
print(stackOfString.items)
print(stackOfString.pop())
```

swift의 Array도 generic 구조체

- `var x : [Int] = []  //빈 배열`
- `var y = [Int]()`
- `var z : Array<Int> = []`

- `var a : [Int] = [1,2,3,4]`
- `var b : Array<Int>  = [1,2,3,4]`
- `var c : Array<Double>  = [1.2,2.3,3.5,4.1]`

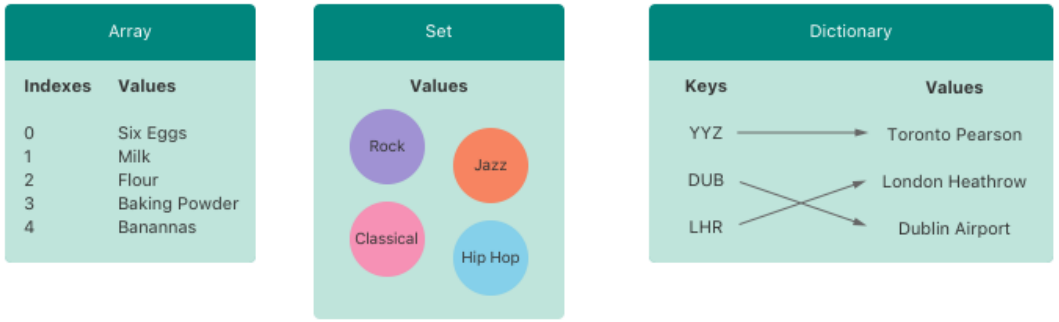
- `@frozen struct Array<Element>`
  - `@frozen`은 저장프로퍼티 추가, 삭제 불가

Collection Type

Collection Type

- https://en.wikipedia.org/wiki/Collection_(abstract_data_type)
- https://docs.swift.org/swift-book/documentation/the-swift-programming-language/collectiontypes/

Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.



Arrays, sets, and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you can't insert a value of the wrong type into a collection by mistake. It also means you can be confident about the type of values you will retrieve from a collection.

> NOTE
>
> Swift's array, set, and dictionary types are implemented as *generic collections*. For more about generic types and collections, see Generics.

Array

Array	
Indexes	**Values**
0	Six Eggs
1	Milk
2	Flour
3	Baking Powder
4	Banannas

Array

- 연결리스트(linked list)
- https://developer.apple.com/documentation/swift/array

Generic Structure

Array

An ordered, random-access collection.

Declaration

```
@frozen struct Array<Element>
```

Overview

Arrays are one of the most commonly used data types in an app. You use arrays to organize your app's data. Specifically, you use the `Array` type to hold elements of a single type, the array's `Element` type. An array can store any kind of elements—from integers to strings to classes.

swift의 배열은 generic 구조체

- `var x : [Int] = []  //빈 배열`
- `var y = [Int]()`
- `var z : Array<Int> = []`

- `var a : [Int] = [1,2,3,4]`
- `var b : Array<Int>  = [1,2,3,4]`
- `var c : Array<Double>  = [1.2,2.3,3.5,4.1]`

- `@frozen struct Array<Element>`
  - @frozen은 저장프로퍼티 추가, 삭제 불가

Array의 자료형

```
let number = [1, 2, 3, 4]   //타입 추론
let odd : [Int] = [1, 3, 5]
let even : Array<Int> = [2, 4, 6]

print(type(of:number)) //Array<Int>
print(number)//[1, 2, 3, 4]

print(type(of:odd))
print(odd)

print(type(of:even))
print(even)

let animal = ["dog", "cat","cow"]
print(type(of:animal))//Array<String>
print(animal)
```

빈 배열(empty array)

```
var number : [Int] = []
var odd = [Int]()
var even : Array<Int> = Array()
print(number) //[]
```

빈 배열(empty array) 주의 사항

```
let number : [Int] = []
//빈 배열을 let으로 만들 수는 있지만 초기값에서 변경 불가이니 배열의 의미 없음
var odd = [Int]()
var even : Array<Int> = Array()
print(number)
number.append(100)  //let으로 선언한 불변형 배열이라 추가 불가능
//error: cannot use mutating member on immutable value: 'number' is a 'let' constant
odd.append(1)
even.append(2)
```

- 가변형(mutable)
  - var animal = ["dog", "cat","cow"]

- 불변형 (immutable)
  - 초기화 후 변경 불가
  - let animal1 = ["dog", "cat","cow"]

빈 배열(empty array) 주의 사항

```
var number : [Int] = []

//number[0]=1 //crash, 방을 만든 후 사용하라!
number.append(1)
print(number)
number[0]=10
print(number)
```

Array(repeating:count:)

- 특정값(repeating)으로 원하는 개수(count)만큼 초기화
- https://developer.apple.com/documentation/swift/array

```
var x = [0,0,0,0,0]
print(x)
var x1 = Array(repeating: 0, count: 5)
print(x1)
var x2 = [Int](repeating: 1, count: 3)
print(x2)
var x3 = [String](repeating: "A", count: 4)
print(x3)
```

for~in으로 배열 항목 접근

```
let colors = ["red", "green", "blue"]
print(colors)

for color in colors {
   print(color)
}
```

항목이 몇 개인지(count), 비어있는지(isEmpty) 알아내기

```swift
let num = [1, 2, 3, 4]
var x = [Int]()
print(num.isEmpty) //배열이 비어있나? false
print(x.isEmpty)

if num.isEmpty {
  print("비어 있습니다")
}
else {
  print(num.count) //배열 항목의 개수
}
```

```
var isEmpty: Bool
    A Boolean value indicating whether the collection is empty.

var count: Int
    The number of elements in the array.
```

Dictionary, String 등에도 있는 property

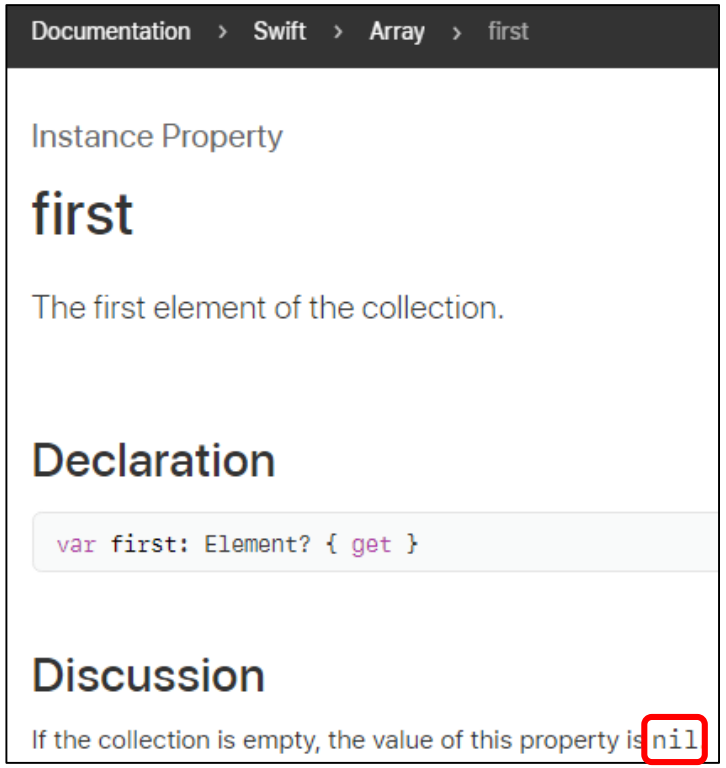
first와 last 프로퍼티

```swift
let num = [1, 2, 3, 4]
let num1 = [Int]()

print(num.first, num.last)//Optional(1) Optional(4)
print(num1.first, num1.last)//nil nil
if let f = num.first, let l = num.last {
  print(f,l) //1 4
}
```

```
var first: Element?
    The first element of the collection.

var last: Element?
    The last element of the collection.
```



첨자(subscript)로 항목 접근

```
var num = [1, 2, 3, 4]
print(num[0], num[3])
print(num.first!)
for i in 0...num.count-1{
 print(num[i])
}
for i in 0...num.endIndex-1{
 print(num[i])
}
print(num[1...2])
num[0...2] = [10,20,30]
print(num)
```

```
var startIndex: Int
    The position of the first element in a nonempty array.

var endIndex: Int
    The array's "past the end" position—that is, the position one greater than the last valid
    subscript argument.
```

Array: 추가/제거

```
//let x = [1, 2, 3, 4] //불변 배열은 초깃값에서 변경 불가
//x.append(5)
var num = [1,2,3]
print(num)
num.append(4)
print(num)
num.append(contentsOf: [6, 7, 8])
print(num) //[1, 2, 3, 4, 6, 7, 8]
num.insert(5, at:4)
print(num) //[1, 2, 3, 4, 5, 6, 7, 8]
num.remove(at:3)
print(num) //[1, 2, 3, 5, 6, 7, 8]
num.removeLast()
print(num) //[1, 2, 3, 5, 6, 7]
print(num.firstIndex(of:2)) //Optional(1), 2가 처음으로 나오는 첨자
if let i = num.firstIndex(of:2) { //2가 처음으로 저장된 방의 값을 20으로 바꿈
num[i] = 20 //num[1] = 20
}
print(num) //[1, 20, 3, 5, 6, 7]
num=num+num
print(num) //[1, 20, 3, 5, 6, 7, 1, 20, 3, 5, 6, 7]
num+=[8,9]
print(num) //[1, 20, 3, 5, 6, 7, 1, 20, 3, 5, 6, 7, 8, 9]
num.removeAll()
print(num) //[]
```

```
func firstIndex(of: Element) -> Int?
```

Returns the first index where the specified value appears in the collection.

Array는 구조체이므로 값 타입

```
var num = [1,2,3]
var x = num
num[0]=100
print(num)
print(x)
```

Array 요소의 최댓값 최솟값 :max(), min()

```
var num = [1,2,3,10,20]
print(num)
print(num.min())
print(num.max())
print(num.min()!)
print(num.max()!)
```

```
func min() -> Element?
```

Returns the minimum element in the sequence.

Array 요소의 정렬

```
var num = [1,5,3,2,4]
num.sort() //오름차순 정렬하여 원본 변경
print(num) //[1, 2, 3, 4, 5]
num[0...4] = [2,3,4,5,1]
num.sort(by:>) //내림차순 정렬하여 원본 변경
print(num) //[5, 4, 3, 2, 1]
num[0...4] = [2,3,4,5,1]
num.reverse() //반대로 정렬하여 원본 변경
print(num) //[1, 5, 4, 3, 2]
print(num.sorted()) //오름차순 정렬 결과를 리턴하고, 원본은 그대로, var x = num.sorted()
//[1, 2, 3, 4, 5]
print(num) //[1, 5, 4, 3, 2]
print(num.sorted(by:>)) //내림차순 정렬 결과를 리턴하고, 원본은 그대로
//[5, 4, 3, 2, 1]
print(num)//[1, 5, 4, 3, 2]
```

Array를 String으로 변환

- 고차 함수는 뒤에서 설명

```
var x = ["a","1","c","AB","한"]
var x1 = x.joined()
print(x1) //a1cAB한
print(type(of:x1)) //String
print(x.joined(separator:"-"))
var x2 = x.reduce("",{$0+$1})
print(x2)
print(type(of:x2)) //String
```

```
func joined() -> FlattenSequence<Array<Element>>
```
Returns the elements of this sequence of sequences, concatenated.

```
func joined<Separator>(separator: Separator) -> Joined
Sequence<Array<Element>>
```
Returns the concatenated elements of this sequence of sequences, inserting the given separator between each element.

```
func joined(separator: String) -> String
```
Returns a new string by concatenating the elements of the sequence, adding the given separator between each element.

String을 Array로 변환

```
let str = "Hi, Swift!"
let arr = Array(str)
print(arr) //["H", "i", ",", " ", "S", "w", "i", "f", "t", "!"]
let arr1 = str.map {String($0)}
print(arr1)
print(type(of:arr1)) //Array<String>
```