APP SCHOOL : iOS

앱개발 기초

Swift 문법기초

타입

타입

Swift의 타입은 세 가지 기본 그룹으로 구분되는데, 구조체, 열거형, 클래스이다.

Swift의 구조체나 열거형은 대다수 언어보다 더 현저하게 강력하다

- 프로퍼티, 이니셜라이저, 메서드 지원뿐만 아니라 확장하거나 프로로콜을 따를 수 있다

구조체

```
struct MyStruct {
    // 프로퍼티
    // 이니셜라이저
    // 메서드
}
```

열거형

```
enum MyEnum {
    // 프로퍼티
    // 이니셜라이저
    // 메서드
}
```

클래스

```
class MyStruct: SuperClass {
    // 프로퍼티
    // 이니셜라이저
    // 메서드
}
```

각 진수에 따라 정수를 표현하는 방법

10진수 : 우리가 평소에 쓰던 숫자와 동일하게 작성

2진수 : 접두어 0b를 사용

8진수 : 접두고 0o를 사용

16진수 : 접두어 0x를 사용

나의 플레이그라운드
My Playground

```
let decimalIntger: Int = 28
let binaryInteger: Int = 0b11100        // 2진수로 10진수 28을 표현
let octalInteger: Int = 0o34            // 8진수로 10진수 28을 표현
let hexadecimalInteger: Int = 0x1c      // 16진수로 10진수 28을 표현
```

내 코드 실행

let var if for while func binaryInteger decimalIntger hexadecimalInteger octalInteger

정수와 부동소수점 변환

Integer and Floating-Point Conversion

정수와 부동 소수점 숫자 타입의 변환은 명시적으로 변환해야 한다.

이 코드에서 상수 three 는 타입 Double 의 새로운 값으로 생성하는데 사용되어 덧셈의 양쪽이 동일한 타입이다.

이 변환이 없으면 덧셈이 허용되지 않는다.

나의 플레이그라운드
My Playground

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type
 Double
```

내 코드 실행

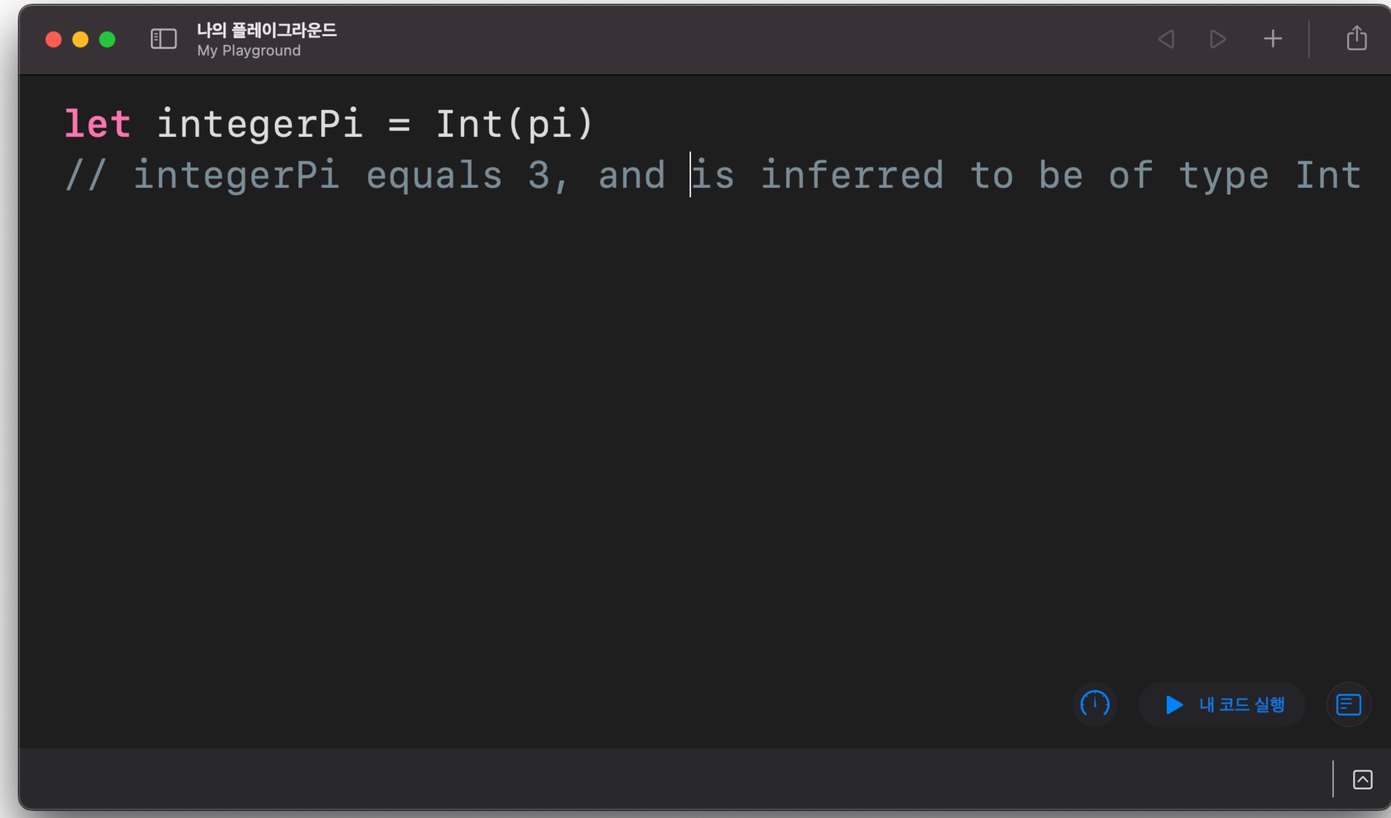
정수와 부동소수점 변환

Integer and Floating-Point Conversion

부동 소수점을 정수로 변환하는 것 또한 명시적으로 변환해야 한다.

부동 소수점 값은 새로운 정수 값으로 초기화할 때 소수점 아래를 버림한다.

이것은 4.75 는 4, 그리고 -3.9 는 -3 이 된다는 의미이다.



문자열

연산자를 통한 문자열 결합

+ 연산자를 이용해 결합

나의 플레이그라운드
My Playground

```
// 연산자를 통한 문자열 결합
let hello: String = "Hello"
let ned: String = "ned"
var greeting: String = ned + " " + ned + "!"
print(greeting) // Hello ned!

greeting = hello
greeting += " "
greeting += ned
greeting += "!"
print(greeting) // Hello ned!
```

내 코드 실행

문자열

연산자를 통한 문자열 비교

== 연산자를 이용해 비교

나의 플레이그라운드
My Playground

```
// 연산자를 통한 문자열 비교
var isSameString: Bool = false

isSameString = hello == "Hello"
print(isSameString) // true

isSameString = hello == "hello"
print(isSameString) // false

isSameString = ned == "ned"
print(isSameString) // true

isSameString = ned == hello
print(isSameString) // false
```

내 코드 실행

문자열

메서드를 통한 접두어 확인

hasPrefix 메소드를 이용해 확인

나의 플레이그라운드
My Playground

```
// 메서드를 통한 접두어 확인
var hasPrefix: Bool = false
hasPrefix = hello.hasPrefix("He")
print(hasPrefix)    // true

hasPrefix = hello.hasPrefix("HE")
print(hasPrefix)    // false

hasPrefix = greeting.hasPrefix("Hello")
print(hasPrefix)    // true

hasPrefix = ned.hasPrefix("ed")
print(hasPrefix)    // false

hasPrefix = hello.hasPrefix("Hello")
print(hasPrefix)    // true
```

내 코드 실행

문자열

메서드를 통한 접미어 확인

hasSuffix 메소드를 이용해 확인

나의 플레이그라운드
My Playground

```
// 메서드를 통한 접미어 확인
var hasSuffix: Bool = false
hasSuffix = hello.hasSuffix("He")
print(hasSuffix) // false

hasSuffix = hello.hasSuffix("llo")
print(hasSuffix) // ture

hasSuffix = greeting.hasSuffix("ned")
print(hasSuffix) // false

hasSuffix = greeting.hasSuffix("ned!")
print(hasSuffix) // true

hasSuffix = ned.hasSuffix("ed")
print(hasSuffix) // true
```

내 코드 실행

문자열

빈 문자열 확인

isEmpty 프로퍼티를 이용해 확인

나의 플레이그라운드
My Playground

```
// 프로퍼티를 통한 빈 문자열 확인
var isEmptyString: Bool = false
isEmptyString = greeting.isEmpty
print(isEmptyString) // false

greeting = "안녕"
isEmptyString = greeting.isEmpty
print(isEmptyString) // false

greeting = ""
isEmptyString = greeting.isEmpty
print(isEmptyString) // true
```

내 코드 실행

문자열

문자열 길이 확인

count 프로퍼티를 이용해 확인

나의 플레이그라운드
My Playground

```
// 프로퍼티를 이용해 문자열 길이 확인

print(greeting.count) // 0

greeting = "안녕하세요"
print(greeting.count) // 5

greeting = "안녕!"
print(greeting.count) // 3
```

내 코드 실행

집합

Sets

집합 *(set)* 은 콜렉션에 순서와 상관없이 같은 타입의 다른 값을 저장한다.

아이템의 순서가 중요하지 않거나 아이템이 반복되면 안될 때 배열 대신에 집합을 사용할 수 있다.

나의 플레이그라운드
My Playground

```
var letters = Set<Character>()
print("letters is of type Set<Character> with
 \(letters.count) items.")
// Prints "letters is of type Set<Character> with 0 items."


letters.insert("a")
// letters now contains 1 value of type Character
letters = []
// letters is now an empty set, but is still of type
 Set<Character>


var favoriteGenres: Set<String> = ["Rock", "Classical",
 "Hip hop"]
// favoriteGenres has been initialized with three initial
 items
```

내 코드 실행

집합 접근과 수정

Accessing and Modifying a Set

읽기 전용 *count* 프로퍼티로 집합의 아이템 갯수를 알 수 있습니다.

count 프로퍼티가 *0* 과 같은지를 부울 *isEmpty* 프로퍼티를 사용해 확인할 수 있다.

집합의 *insert(_:)* 메서드를 호출하여 집합에 새로운 아이템을 추가할 수 있다.

나의 플레이그라운드
My Playground

```
print("I have \(favoriteGenres.count) favorite music
 genres.")
// Prints "I have 3 favorite music genres."


if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}
// Prints "I have particular music preferences."


favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
```

내 코드 실행

집합 접근과 수정

Accessing and Modifying a Set

집합의 *remove(_:)* 메서드를 호출하여 집합의 아이템을 삭제할 수 있다.

remove(_:) 메서드는 집합에 아이템이 있을 경우 삭제하고 삭제된 값을 반환하거나 해당 아이템이 없을 경우 *nil* 을 반환한다.

또한 *removeAll()* 메서드를 사용하여 전체 아이템을 삭제할 수 있다.

contains(_:) 메서드를 사용하여 집합에 특정 아이템이 포함되어 있는지 알 수 있다.

나의 플레이그라운드
My Playground

```
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."

if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."
```

내 코드 실행

집합 반복

Iterating Over a Set

for-in 루프와 함께 집합에 값을 반복할 수 있다.

*Swift*의 *Set* 타입은 정의된 순서를 가지고 있지 않다.

특정 순서로 집합의 값을 반복하려면 집합의 요소를 < 연산자를 사용하여 정렬하여 반환하는 *sorted()* 메서드를 사용해야 한다.

나의 플레이그라운드
My Playground

```
for genre in favoriteGenres {
    print("\(genre)")
}
// Classical
// Jazz
// Hip hop


for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

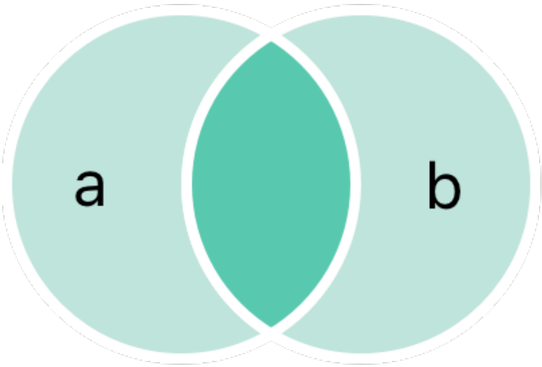
내 코드 실행

집합 연산 수행

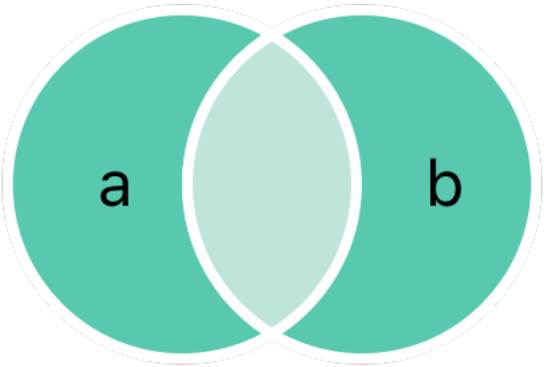
Performing Set Operations

두 집합을 합치거나 두 집합의 공통 값을 구하거나 두 집합이 모두 같은 값을 갖고 있거나 한쪽에만 존재하거나 아예 없거나와 같은 기본적인 집합 연산을 효율적으로 수행할 수 있다.

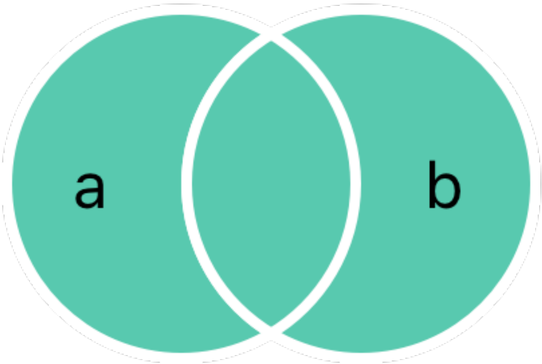
a.intersection(b)



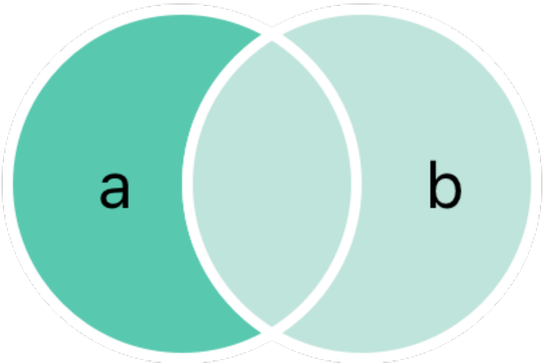
a.symmetricDifference(b)



a.union(b)



a.subtracting(b)



집합 연산 수행

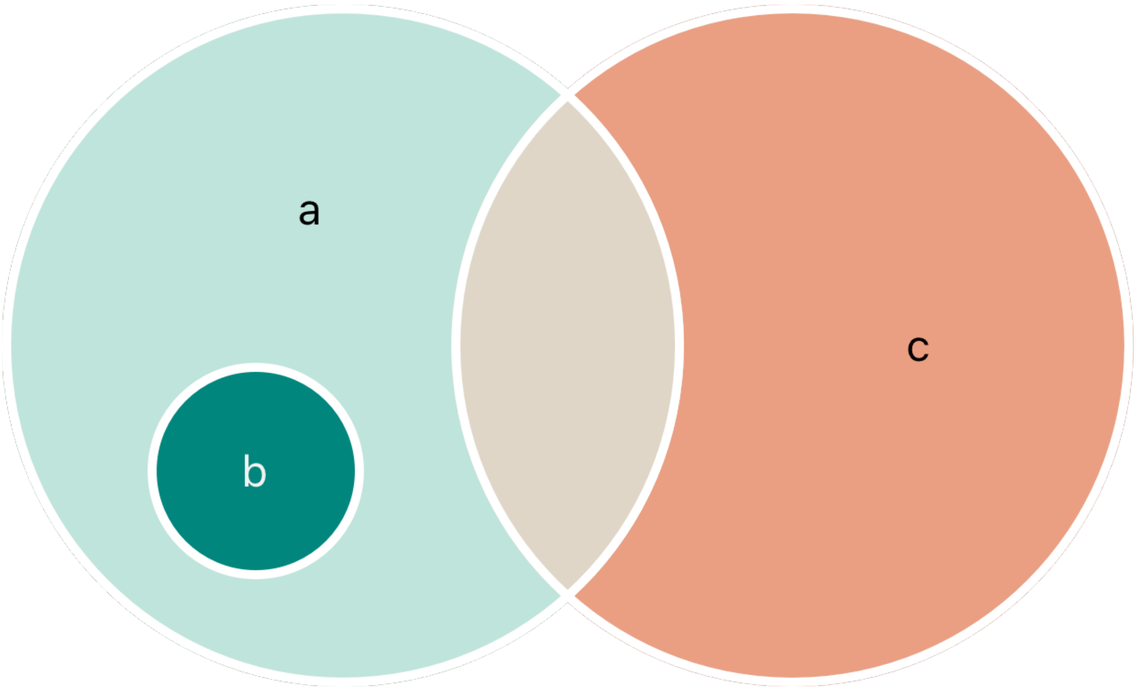
Performing Set Operations

그림은 세 집합 *a, b, c* 를 나타내며 집합의 공통 요소는 겹쳐서 표현하고 있다.

a 는 *b* 의 모든 요소를 포함하므로 집합 *a* 는 *b* 의 초집합 *(superset)* 이라 한다.

반대로 *b* 의 모든 요소가 *a* 에 포함되어 있으므로 집합 *b* 는 집합 *a* 의 부분집합 *(subset)* 이라 한다.

집합 *b* 와 집합 *c* 는 공통 요소가 없으므로 분리집합 *(disjoint)* 이라 한다.



집합 연산 수행

Performing Set Operations

두 집합이 같은 값을 모두 포함하고 있는지 판단하려면 "같음" 연산자 *(==)*를 사용한다.

isSubset(of:) 메서드를 사용하면 집합이 특정 집합에 모든 값이 포함되어 있는지 판단한다.

isSuperset(of:) 메서드를 사용하면 집합에 특정 집합에 모든 값을 포함 하는지 판단한다.

isStrictSubset(of:) 또는 *isStrictSuperset(of:)* 메서드를 사용하면 집합이 특정 집합과 같지 않고 부분집합인지 초집합인지 판단한다.

isDisjoint(with:) 메서드를 사용하면 두 집합이 공통 값이 없는지 판단한다.

나의 플레이그라운드
My Playground

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐷", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐟", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

내 코드 실행

열거형

열거형은 값들의 집합으로 이루어진 타입이다.

파이를 묘사하는 enum을 정의해보자.

나의 플레이그라운드
My Playground

```
// 파이를 묘사하는 열거형
enum PieType {
    case Apple
    case Cherry
    case Pecan
}

let favoritePie = PieType.Apple
```

내 코드 실행

열거형

Swift는 enum 값들을 매칭하기에 좋은 강력한 switch 문을 가지고 있다.

switch문은 모든 case를 포함해야 한다.

switch 표현식의 경우 각각의 값은 명시적이든지 default: case든지 반드시 처리되어야 한다.

Swift는 C언어와 달리 switch 문의 case들이 다른 case로 빠지지 않는다. 일치되는 case의 코드만 실행된다.

나의 플레이그라운드
My Playground

```
// 파이를 묘사하는 열거형
enum PieType {
    case Apple
    case Cherry
    case Pecan
}
let favoritePie = PieType.Apple

let name: String
switch favoritePie {
case .Apple:
    name = "Apple"
case .Cherry:
    name = "Cherry"
case .Pecan:
    name = "Pecan"
}
```

내 코드 실행

열거형

Swift의 enum은 case에 연관된 원시 값을 가질 수 있다.

타입이 명시된 enum을 가지고 rawValue를 사용하여 PieType의 인스턴스를 요청할 수 있다.

그리고 그 값을 enum 타입을 초기화할 수 있다. 이것은 enum의 실제 case에 상응하는 원시 값이 없을 수 있기 때문에 옵셔널을 반환한다.

따라서 이 경우가 옵셔널 바인딩의 좋은 예다.

나의 플레이그라운드
My Playground

```
// 파이를 묘사하는 열거형에 원시값 적용
enum PieType: Int {
    case Apple = 0
    case Cherry
    case Pecan
}

let pieRawValue = PieType.Pecan.rawValue
// pieRawValue는 2를 값으로 가진 Int 타입이다

if let pieType = PieType(rawValue: pieRawValue) {
    // 'pieType'!이 유효한 값을 가지면
}
```

내 코드 실행

열거형

enum의 원시값은 보통 Int지만 정수나 부동 소수점 수 타입뿐만 아니라 String, Character 타입도 될 수 있다.

원시 값이 정수 타입일 때 값을 명시적으로 주지 않으면 그 값은 자동으로 증가한다.

PieType에서는 Apple case에만 값을 명시했다. Cherry와 Pecan은 자동으로 각각 1과 2로 rawValue가 할당된다.

열거형의 각 case는 연관된 값을 가질 수 있다.

나의 플레이그라운드
My Playground

```
// 파이를 묘사하는 열거형에 원시값 적용
enum PieType: Int {
    case Apple = 0
    case Cherry
    case Pecan
}

let pieRawValue = PieType.Pecan.rawValue
// pieRawValue는 2를 값으로 가진 Int 타입이다

if let pieType = PieType(rawValue: pieRawValue) {
    // 'pieType'!이 유효한 값을 가지면
}
```

내 코드 실행

프로퍼티

타입

프로퍼티 관찰자 (Property observers)

프로퍼티 관찰자 (Property observers)는 프로퍼티의 값이 변경되는지 관찰하고 응답한다.

프로퍼티 관찰자는 프로퍼티의 현재 값이 새로운 값과 같더라도 프로퍼티의 값이 설정될 때 호출된다.

아래의 위치에 프로퍼티 관찰자를 추가할 수 있다.

- 상속된 프로퍼티의 경우 하위 클래스의 프로퍼티를 재정의하여 프로퍼티 관찰자를 추가한다.
- 정의한 계산된 프로퍼티의 경우 관찰자를 생성하는 대신에 프로퍼티의 setter를 이용하여 값 변경을 관찰하고 응답한다.

정의한 저장된 프로퍼티	상속한 저장된 프로퍼티	상속한 계산된 프로퍼티

타입

프로퍼티 관찰자 (Property observers)

프로퍼티에 관찰자를 정의하는 방법은 2가지 선택사항을 가지며 둘다 정의할 수도 있다.

willSet	didSet
값이 저장되기 직전에 호출	새로운 값이 저장되자마자 호출

willSet 관찰자를 구현한다면 상수 파라미터로 새로운 프로퍼티 값이 전달된다.

- willSet 구현의 일부로 이 파라미터에 특정 이름을 가질 수 있다.
- 파라미터 명과 구현 내에 소괄호를 작성하지 않으면 파라미터는 newValue 의 기본 파라미터 명으로 만들어 질 수 있다.

유사하게 didSet 관찰자를 구현한다면 예전 프로퍼티 값을 포함한 상수 파라미터가 전달된다.

- 파라미터 명을 사용하거나 oldValue 인 기본 파라미터 명을 사용할 수 있다.
- didSet 관찰자 내의 프로퍼티에 값을 할당한다면 새로운 값으로 방금 설정한 값을 대체한다.

나의 플레이그라운드
My Playground

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

내 코드 실행

제너릭

제너릭

Generic

제너릭 코드 (Generic code) 는 정의한 요구사항에 따라 모든 타입에서 동작할 수 있는 유연하고 재사용 가능한 함수와 타입을 작성할 수 있다.

중복을 피하고 명확하고 추상적인 방식으로 의도를 표현하는 코드를 작성할 수 있다.

제너릭은 Swift의 강력한 특징 중 하나이고 Swift 표준 라이브러리 대부분은 제너릭 코드로 되어 있다.

사실 모르고 있더라도 Language Guide 전체에서 제너릭을 사용한다.

- 예를 들어 Swift의 Array 와 Dictionary 타입은 둘다 제너릭 콜렉션 이다.
- Int 값을 가진 배열, 또는 String 값을 가진 배열 또는 실제로 Swift에서 생성될 수 있는 다른 모든 타입에 대한 배열을 생성할 수 있다.
- 마찬가지로 모든 지정된 타입의 값을 저장하기 위한 딕셔너리를 생성할 수 있고 해당 타입에 대한 제한은 없다.

제너릭이 해결하는 문제

The Problem That Generics Solve

다음의 두 Int 값을 바꾸는 swapTwoInts(_:_:) 라는 제너릭이 아닌 함수를 보자.

이 함수는 In-Out 파라미터 (In-Out Parameters) 에서 설명한 대로 a 와 b 의 값을 바꾸기 위해 in-out 파라미터를 사용하여 만든다.

swapTwoInts(_:_:) 함수는 b 의 값을 a 로 그리고 a 의 값을 b 로 바꾼다.

2개의 Int 변수의 값을 바꾸기 위해 이 함수를 호출할 수 있다

나의 플레이그라운드
My Playground

```
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"
```

제너릭이 해결하는 문제

The Problem That Generics Solve

swapTwoInts(_:_:) 함수는 유용하지만 Int 값만 사용이 가능하다.

2개의 String 값 또는 2개의 Double 값을 바꾸길 원하면 swapTwoStrings(_:_:) 와 swapTwoDoubles(_:_:) 함수와 같이 더 많은 함수를 작성해야 한다.

swapTwoInts(_:_:), swapTwoStrings(_:_:), 그리고 swapTwoDoubles(_:_:) 함수의 바디가 동일하다는 것을 알 수 있다.

차이점은 받아들이는 값의 타입만 (Int, String, 그리고 Double) 다르다.

모든 타입 의 2개의 값을 바꾸는 단일 함수로 작성하면 더 유용하고 더 유연합니다. 제너릭 코드는 이러한 함수를 작성할 수 있다.

나의 플레이그라운드
My Playground

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


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
}
```

내 코드 실행

제너릭 함수

Generic Functions

제너릭 함수 (Generic functions) 는 모든 타입과 함께 동작할 수 있다.

다음은 swapTwoValues(_:_:) 라는 위의 swapTwoInts(_:_:) 함수의 제너릭 버전이다.

나의 플레이그라운드
My Playground

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

내 코드 실행

제너릭 함수

Generic Functions

swapTwoValues(_:_:) 함수의 바디는 swapTwoInts(_:_:) 함수의 바디와 동일하다.

그러나 swapTwoValues(_:_:) 의 첫번째 줄은 swapTwoInts(_:_:) 와 약간 다르다.

나의 플레이그라운드
My Playground

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

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

내 코드 실행

제너릭 함수

Generic Functions

함수의 제너릭 버전은 Int, String, 또는 Double 와 같은 실제 타입 이름 대신에 이 경우 T 라는 임의의 타입 이름을 사용한다.

이 임의의 타입 이름은 T 가 무엇이어야 하는지 아무 말도 하지 않지만 T 가 무엇을 나타내든 a 와 b 는 모두 같은 타입 T 여야 한다고 말한다.

T 의 실제 타입은 swapTwoValues(_:_:) 함수가 호출될 때마다 결정된다.

제너릭 함수와 제너릭이 아닌 함수 사이의 다른 차이점은 제너릭 함수의 이름 (swapTwoValues(_:_:))에 바로 임의의 타입 이름 (T)이 꺾쇠 괄호 내 (<T>)에 위치한다는 것이다.

이 괄호는 T 는 swapTwoValues(_:_:) 함수 정의 내에서 임의의 타입 이름이라고 Swift에게 말한다.

T 는 임의의 타입이므로 Swift는 T 라는 실제 타입을 찾지 않는다.

나의 플레이그라운드
My Playground

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

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

내 코드 실행

제너릭 함수

Generic Functions

swapTwoValues(_:_:) 함수는 이제 swapTwoInts 와 동일한 방식으로 호출될 수 있지만 두 값이 서로 동일한 타입이면 모든 타입의 두 값을 전달할 수 있다는 점이 다르다.

swapTwoValues(_:_:) 가 호출될 때마다 T 로 사용한 타입은 함수에 전달된 값의 타입으로 부터 유추된다.

예제에서 T 는 각각 Int 와 String 으로 추론된다.

나의 플레이그라운드
My Playground

```
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"
```

내 코드 실행

타입 파라미터

Type Parameters

swapTwoValues(_:_:) 예제에서 임의의 타입 T 는 타입 파라미터 (type parameter) 의 예시다.

타입 파라미터는 임의의 타입을 지정하고 이름을 지정하며 꺾쇠 괄호 (예: <T>) 사이에 기록하고 함수의 이름 바로 뒤에 작성된다.

타입 파라미터를 지정하면 함수의 파라미터 (swapTwoValues(_:_:) 함수의 a 와 b 와 같이) 의 타입을 정의하기 위해 사용하거나 함수의 반환 타입이나 함수의 바디내에 타입 주석으로 사용할 수 있다.

각각의 경우 타입 파라미터는 함수가 호출될 때마다 실제 타입으로 대체된다.

- swapTwoValues(_:_:) 에서 T 는 첫번째 함수가 호출될 때 Int 로 대체되고 두번째 호출될 때 String 으로 대체된다.

콤마로 구분된 꺾쇠 괄호 안에 여러개 타입 파라미터를 작성하여 하나 이상의 타입 파라미터를 제공할 수 있다.

나의 플레이그라운드
My Playground

```
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"
```

내 코드 실행

타입 파라미터 이름

Naming Type Parameters

대부분의 경우 타입 파라미터는 타입 파라미터와 제너릭 타입 간의 관계나 함수 간의 관계를 나타내기 위해 Dictionary<Key, Value> 에서 Key 와 Value 그리고 Array<Element> 에서 Element 와 같이 설명이 포함된 이름이 있다.

그러나 의미있는 관계가 없을 때는 swapTwoValues(_:_:) 함수에서 T 와 같이 T, U, 그리고 V 와 같은 단일 문자를 사용하여 이름을 지정하는 것이 일반적이다.

나의 플레이그라운드
My Playground

```
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"
```

내 코드 실행

클로저

일단 클로저 다시보고 돌아옵시다

Swift의 클로저

- 컴퓨터 공학 용어에서의 클로저(closure)는 함수나 클로저 표현식과 같은 독립적인 코드 블록과 코드 블록 주변에 있는 하나 이상의 변수가 결합된 것을 말한다.

- 오른쪽 코드에서 functionA는 functionB라는 이름의 함수를 반환한다.
- 사실 functionB는 functionB 내부 영역 밖에 선언된 counter 변수에 의존하기 때문에 functionA는 클로저를 반환하고 있다.
- 다시 말해, functionB는 counter 변수를 “잡고 있다(captured)” 또는 “가두고 있다(closed over)”라고 말할 수 있으므로 전통적인 컴퓨터 공학 용어인 클로저(closure)로 간주된다.

- Swift에서는 클로저(closure)와 클로저 표현식(closure expression) 용어가 혼용되기 시작했지만, 어쨌든 둘 다 지원한다.

```
func functionA() -> () -> Int {

    var counter = 0

    func functionB() -> Int {
        return counter + 10
    }

    return functionB
}

let myClosure = functionA()
let result = functionB()
```

클로저 (Closures)

클로저 (Closures) 는 코드에서 주변에 전달과 사용할 수 있는 자체 포함된 기능 블럭입니다. Swift의 클로저는 C와 Objective-C의 블럭과 다른 프로그래밍 언어의 람다와 유사합니다.

클로저는 정의된 컨텍스트에서 모든 상수와 변수에 대한 참조를 캡처하고 저장할 수 있습니다. 이러한 상수와 변수를 *폐쇄 (closing over)* 라고 합니다. Swift는 캡처의 모든 메모리 관리를 처리합니다.

> NOTE
> 캡처에 대한 개념이 생소하더라도 걱정하지 마십시오. 아래 캡처값 (Capturing Values) 에서 자세히 다루도록 하겠습니다.

함수 (Functions) 에서 소개한 전역과 중첩 함수는 클로저의 특별한 케이스 입니다. 클로저는 3가지 형태 중 하나를 취합니다:

- 전역 함수는 이름을 가지고 어떠한 값도 캡처하지 않는 클로저입니다.
- 중첩 함수는 이름을 가지고 둘러싼 함수로 부터 값을 캡처할 수 있는 클로저입니다.
- 클로저 표현식은 주변 컨텍스트에서 값을 캡처할 수 있는 경량 구문으로 작성된 이름이 없는 클로저입니다.

Swift의 클로저 표현식은 일반 시나리오에서 간단하고 깔끔한 구문을 장려하는 최적화를 통해 깔끔하고 명확한 스타일을 가지고 있습니다. 이러한 최적화에는 다음이 포함됩니다:

- 컨텍스트에서 파라미터와 반환값 타입 유추
- 단일 표현식 클로저의 암시적 반환
- 약식 인자 이름
- 후행 클로저 구문

클로저 표현식 (Closure Expressions)

중첩 함수 (Nested Functions) 에서 소개된 중첩 함수는 더 큰 함수에 부분으로 자체 포함된 코드 블럭의 이름을 지정하고 정의하기 편리한 수단입니다. 그러나 완전한 선언과 이름없이 함수와 유사한 구조의 짧은 버전을 작성하는 것이 때때로 유용합니다. 함수를 하나 이상의 인자로 사용하는 함수 또는 메서드로 작업할 때 특히 그렇습니다.

클로저 표현식 (Closure expressions) 은 간단하고 집중적인 구문으로 인라인 클로저로 작성하는 방법입니다. 클로저 표현식은 명확성이나 의도를 잃지 않고 짧은 형태로 클로저를 작성 하기위한 몇가지 구문 최적화를 제공합니다. 아래의 클로저 표현식 예제는 여러 반복에 걸쳐 `sorted(by:)` 메서드의 단일 예제를 구체화하는 최적화를 나타냅니다. 각 예제는 동일한 기능을 보다 간결한 방식으로 표현합니다.

정렬 메서드 (The Sorted Method)

Swift의 표준 라이브러리는 사용자가 제공하는 정렬 클로저의 출력을 기반으로 알려진 타입의 값 배열을 정렬하는 `sorted(by:)` 라는 메서드를 제공합니다. 정렬 프로세스가 완료되면 `sorted(by:)` 메서드는 원본 배열과 같은 타입과 같은 크기의 올바르게 정렬된 요소의 새로운 배열로 반환합니다. 기존 배열은 `sorted(by:)` 메서드로 수정되지 않습니다.

아래 예제의 클로저 표현식은 알파벳 역순으로 `String` 값의 배열을 정렬하기 위해 `sorted(by:)` 메서드를 사용합니다. 다음은 정렬하기 위한 초기화 배열입니다:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

`sorted(by:)` 메서드는 배열 내용과 동일한 타입의 두 인자를 사용하는 클로저를 허용하고 값이 정렬된 후 첫번째 값이 두번째 값의 앞 또는 뒤에 표시되어야 하는지 여부를 나타내는 `Bool` 값을 반환합니다. 정렬 클로저는 첫번째 값이 두번째 값 앞에 나타나야 하는 경우 `true` 를 반환하고 그렇지 않으면 `false` 를 반환해야 합니다.

이 예제는 `String` 값의 배열을 정렬하고 정렬 클로저는 `(String, String) -> Bool` 타입의 함수를 필요로 합니다.

정렬 클로저를 제공하는 한가지 방법은 올바른 타입의 일반 함수를 작성하고 `sorted(by:)` 메서드에 인자로 전달하는 것입니다:

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

첫번째 문자열 (`s1`)이 두번째 문자열 (`s2`)보다 크다면 `backward(_:_:)` 함수는 `true` 를 반환하고 이것은 정렬된 배열에 `s1` 은 `s2` 전에 나타납니다. 문자열의 문자가 "더 크다"는 "알파벳 순으로 더 뒤에 나타난다"는 의미입니다. 이것은 문자 `B` 는 문자 `A` 보다 "더 크다"이고 문자열 `"Tom"` 은 문자열 `"Tim"` 보다 더 큽니다. 알파벳 역순으로 정렬될 때 `"Barry"` 는 `"Alex"` 보다 앞에 위치합니다.

그러나 이것은 본질적으로 단일 표현식 함수 (`a > b`)를 작성하는 다소 긴 방식입니다. 이 예제에서는 클로저 표현식 구문을 사용하여 정렬 클로저를 인라인으로 작성하는 것이 좋습니다.

클로저 표현구 (Closure Expression Syntax)

클로저 표현구는 아래와 같이 일반적인 형태를 가지고 있습니다:

```
{ (parameters) -> return type in
    statements
}
```

Closure Expression Syntax

클로저 표현구의 *파라미터* 는 in-out 파라미터 일 수 있지만 기본값을 가질 수 없습니다. 가변 파라미터의 이름을 지정하면 가변 파라미터를 사용할 수 있습니다. 튜플은 파라미터 타입과 반환 타입으로 사용될 수도 있습니다.

아래 예제는 위에서 `backward(_:_:)` 함수의 클로저 표현 버전입니다:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

이 인라인 클로저를 위한 파라미터와 반환 타입의 선언은 `backward(_:_:)` 함수에서 선언한 것과 동일합니다. 두 경우 모두 `(s1: String, s2: String) -> Bool` 로 작성합니다. 그러나 인라인 클로저 표현식을 위한 파라미터와 반환 타입은 중괄호 바깥이 아닌 *안에* 작성합니다.

클로저의 바디의 시작은 `in` 키워드로 시작합니다. 이 키워드는 클로저의 파라미터와 리턴 타입 정의가 끝남을 나타내며 클로저의 바디가 시작함을 나타냅니다.

클로저의 바디가 너무 짧기 때문에 한줄로 작성할 수 있습니다:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

이것은 `sorted(by:)` 메서드에 대한 전체 호출이 동일하게 유지되었음을 보여줍니다. 소괄호는 여전히 메서드의 전체 인자를 둘러싸고 있습니다. 그러나 인자는 이제 인라인 클로저입니다.

컨텍스트로 타입 유추 (Inferring Type From Context)

정렬 클로저는 메서드에 인자로 전달되기 때문에 Swift는 파라미터 타입과 반환되는 값의 타입을 유추할 수 있습니다. `sorted(by:)` 메서드는 문자열 배열에서 호출되므로 인자는 `(String, String) -> Bool` 타입의 함수이어야 합니다. 이는 `(String, String)` 과 `Bool` 타입을 클로저 표현식 정의에 일부러 작성할 필요가 없음을 의미합니다. 모든 타입은 유추할 수 있기 때문에 반환 화살표 (`->`)와 파라미터의 이름을 둘러싼 소괄호를 생략할 수 있습니다:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

함수나 메서드에 클로저를 인라인 클로저 표현식으로 전달할 때 항상 파라미터 타입과 반환 타입을 유추할 수 있습니다. 결과적으로 클로저가 함수 또는 메서드 인자로 사용될 때 완전한 형태로 인라인 클로저를 작성할 필요가 없습니다.

그럼에도 불구하고 원하는 경우 타입을 명시적으로 만들 수 있으며 코드를 읽는 자가 모호성을 피할 수 있다면 그렇게 하는 것이 좋습니다. `sorted(by:)` 메서드의 경우 정렬이 발생한다는 사실에서 클로저의 목적이 명확하며 문자열 배열의 정렬을 지원하기 때문에 코드를 읽는 사람이 클로저가 `String` 값으로 작동할 가능성이 있다고 가정하는 것이 안전합니다.

단일 표현 클로저의 암시적 반환 (Implicit Returns from Single-Expression Closures)

단일 표현 클로저 (Single-expression closures)는 이전 예제에서 `return` 키워드를 생략하여 단일 표현식으로 암시적으로 값을 반환할 수 있습니다:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

여기서 `sorted(by:)` 메서드의 인자의 함수 타입은 클로저에서 `Bool` 값이 반환되어야 하기 때문에 명확합니다. 클로저의 바디에 `Bool` 값을 반환하는 단일 표현식 (`s1 > s2`)가 포함되므로 모호하지 않고 `return` 키워드를 생략할 수 있습니다.

짧은 인자 이름 (Shorthand Argument Names)

Swift는 인라인 클로저에 `$0`, `$1`, `$2` 등 클로저의 인자값으로 참조하는데 사용할 수 있는 자동적으로 짧은 인자 이름 (shorthand argument names)을 제공합니다.

클로저 표현식에 이런 짧은 인자 이름을 사용한다면 선언에 클로저의 인자 목록을 생략할 수 있고 짧은 인자 이름의 수와 타입은 함수 타입에서 유추됩니다. 클로저 표현식이 전체 바디로 구성되기 때문에 `in` 키워드를 생략할 수도 있습니다:

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

여기서 `$0` 와 `$1` 은 클로저의 첫번째와 두번째 `String` 인자를 참조합니다. `$1` 이 짧은 인자에서 가장 높은 숫자이므로 클로저는 2개의 인자가 있다고 이해합니다. 여기서 `sorted(by:)` 함수는 인자가 모두 문자열인 클로저로 기대하므로 짧은 인자 `$0` 과 `$1` 은 모두 타입 `String` 입니다.

연산자 메서드 (Operator Methods)

실제로 위의 클로저 표현식을 *더 짧게* 작성하는 방법이 있습니다. Swift의 `String` 타입은 보다 큰 연산자 (`>`)의 문자열 별 구현을 `String` 타입의 파라미터 2개가 있는 메서드로 정의하고 `Bool` 타입의 값을 반환합니다. 이것은 `sorted(by:)` 메서드에 필요한 메서드 타입과 정확하게 일치합니다. 따라서 간단하게 보다 큰 연산자를 전달할 수 있고 Swift는 문자열 특정 구현을 사용하기 원한다고 유추합니다:

```
reversedNames = names.sorted(by: >)
```

자세한 내용은 연산자 메서드 (Operator Methods) 를 참고 바랍니다.

후행 클로저 (Trailing Closures)

함수의 마지막 인자로 함수에 클로저 표현식을 전달해야하고 클로저 표현식이 긴 경우 *후행 클로저 (trailing closure)* 로 작성하는 것이 유용할 수 있습니다. 후행 클로저는 함수의 인자이지만 함수 호출의 소괄호 다음에 작성합니다. 후행 클로저 구문을 사용할 때 함수 호출의 일부로 첫번째 클로저 인자 라벨을 작성하지 않아도 됩니다. 함수 호출은 여러개의 후행 클로저를 포함할 수 있지만 아래 몇가지 예제에서는 단일 후행 클로저를 사용합니다.

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

위의 클로저 표현구 (Closure Expression Syntax) 섹션에 문자열 정렬 클로저는 후행 클로저로 `sorted(by:)` 메서드의 소괄호 바깥에 작성될 수 있습니다:

```
reversedNames = names.sorted() { $0 > $1 }
```

후행 클로저로 표현식이 함수와 메서드의 유일한 인자일 경우 함수를 호출할 때 함수 또는 메서드 이름 뒤에 소괄호 `()` 를 작성하지 않아도 됩니다:

```
reversedNames = names.sorted { $0 > $1 }
```

후행 클로저는 클로저가 길어서 한줄로 인라인으로 작성이 불가능할 때 유용합니다. 예를 들어 Swift의 `Array` 타입은 단일 인자로 클로저 표현식을 가지는 `map(_:)` 메서드가 있습니다. 이 클로저는 배열의 각 아이템에 대해 한번 호출되고 아이템에 대해 매핑된 대체값 (다른 타입일 수 있음)이 반환됩니다. `map(_:)` 에 전달한 클로저에 작성된 코드에 따라 매핑 특성과 반환된 값의 타입을 지정합니다.

제공된 클로저에 각 배열의 요소를 적용한 후에 `map(_:)` 메서드는 기존 배열에 해당값과 같은 순서로 새로 매핑된 값의 새로운 배열을 반환합니다.

다음은 `Int` 값의 배열을 `String` 값의 배열로 변환하기 위해 후행 클로저와 `map(_:)` 메서드를 어떻게 사용하는지 나타냅니다. 배열 `[16, 58, 510]` 은 새로운 배열 `["OneSix", "FiveEight", "FiveOneZero"]` 을 생성하는데 사용됩니다:

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

위 코드는 정수와 그 정수에 맞는 영어 표기를 매핑하는 딕셔너리를 생성합니다. 문자열로 변환하기 위한 정수의 배열도 정의합니다.

`number` 배열을 사용하여 후행 클로저로 `map(_:)` 메서드로 클로저 표현식을 전달하여 `String` 값의 배열을 생성할 수 있습니다:

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_:)` 메서드는 배열에 각 아이템을 위해 클로저 표현식을 호출합니다. 매핑할 배열의 값에서 유추할 수 있으므로 클로저의 입력 파라미터 인 `number` 타입을 지정할 필요가 없습니다.

이 예에서 변수 `number` 는 클로저의 `number` 파라미터의 값으로 초기화되기 때문에 값은 클로저 바디내에서 수정될 수 있습니다 (함수와 클로저의 파라미터는 항상 상수입니다). 클로저 표현식은 출력 매핑된 출력 배열에 저장될 타입을 나타내기 위해 `String` 타입도 반환 타입으로 지정합니다.

클로저 표현식은 호출될 때마다 `output` 이라는 문자열을 만듭니다. 나머지 연산자 (`number % 10`)를 이용하여 `number` 의 마지막 숫자를 계산하고 `digitNames` 딕셔너리에 적절한 숫자 문자열을 찾습니다. 클로저는 0보다 큰 정수에 대한 문자열 표현을 생성하는데 사용할 수 있습니다.

> NOTE
> 딕셔너리 서브 스크립트는 키가 존재하지 않는경우에 값을 찾는 것을 실패하기 위해 옵셔널 값을 반환합니다. 그래서 `digitNames` 딕셔너리의 서브 스크립트를 호출할 때는 느낌표를 붙여 줍니다 (`!`). 위의 예제에서 `digitNames` 딕셔너리에 `number % 10` 은 항상 유효한 서브 스크립트 키를 보장하므로 느낌표는 서브 스크립트의 옵셔널 반환 값에 저장된 `String` 값을 강제로 언래핑 하는데 사용됩니다.

`digitNames` 딕셔너리에서 반환된 문자열이 `output` *앞에* 추가되어 숫자의 문자열 버전을 역순으로 효과적으로 빌드합니다 (표현식 `number % 10` 은 `16` 의 경우 `6`, `58` 의 경우 `8`, `510` 의 경우 `0` 을 제공합니다).

그러면 `number` 변수는 `10` 으로 나누어 집니다. 이것은 정수이기 때문에 나누면 버림이 되고 `16` 은 `1`, `58` 은 `5`, `510` 은 `51` 이 됩니다.

이 프로세스는 `number` 가 `0` 이 될때까지 반복하고 그 때 `output` 문자열은 클로저로 부터 반환되고 `map(_:)` 메서드로 부터 출력 배열에 추가됩니다.

위의 예에서 후행 클로저 구문을 사용하면 클로저가 지원하는 함수 바로 뒤에 있는 클로저의 기능을 깔끔하게 캡슐화 합니다. 전체 클로저를 `map(_:)` 메서드의 바깥 소괄호로 감쌀 필요가 없습니다.

함수가 여러개의 클로저를 가지고 있다면 첫번째 후행 클로저의 인자 라벨을 생략하고 남은 후행 클로저의 라벨은 표기합니다. 예를 들어 아래의 함수는 사진 갤러리에서 사진 하나를 불러옵니다:

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

하나의 사진을 불러오기 위해 이 함수를 호출할 때 2개의 클로저를 제공합니다. 첫번째 클로저는 사진 다운로드 완료후에 디스플레이 하는 완료 처리입니다. 두번째 클로저는 오류를 표시하는 오류 처리기 입니다.

```
loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
} onFailure: {
    print("Couldn't download the next picture.")
}
```

이 예제에서 `loadPicture(from:completion:onFailure:)` 함수는 네트워크 작업을 백그라운드로 전달하고 네트워크 작업이 완료되면 두 완료 처리기 중 하나를 호출합니다. 이러한 방식으로 함수를 작성하면 두 상황을 모두 처리하는 하나의 클로저를 사용하는 대신 성공적인 다운로드 후 사용자 인터페이스를 업데이트 하는 코드에서 네트워크 오류를 처리하는 코드를 명확하게 분리할 수 있습니다.

> NOTE 완료 핸들러 (Completion handlers) 는 특히 여러 핸들러가 중첩되어 있으면 읽기 어려울 수 있습니다. 이것을 대체하기 위해선 동시성 (Concurrency) 에서 설명되어진 것과 같이 비동기 코드를 사용하면 됩니다.

캡처값 (Capturing Values)

클로저는 정의된 둘러싸인 컨텍스트에서 상수와 변수를 *캡처 (capture)* 할 수 있습니다. 그러면 클로저는 상수와 변수를 정의한 원래 범위가 더이상 존재하지 않더라도 바디 내에서 해당 상수와 변수의 값을 참조하고 수정할 수 있습니다.

Swift에서 값을 캡처할 수 있는 가장 간단한 클로저 형태는 다른 함수의 바디 내에 작성하는 중첩 함수입니다. 중첩 함수는 바깥 함수의 어떠한 인자도 캡처할 수 있고 바깥 함수 내에 정의된 상수와 변수를 캡처할 수도 있습니다.

아래는 `incrementer` 라는 중첩 함수가 포함된 `makeIncrementer` 라는 함수의 예입니다. 중첩된 `incrementer()` 함수는 둘러싸인 컨텍스트에 `runningTotal` 과 `amount` 인 2개의 값을 캡처합니다. 이 값을 캡처한 후에 `incrementer` 는 호출될 때마다 `amount` 로 `runningTotal` 을 증가시키는 클로저로 `makeIncrementer` 에 의해 반환됩니다.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

`makeIncrementer` 의 반환 타입은 `() -> Int` 입니다. 이것은 단순한 값이 아닌 *함수* 를 반환한다는 의미입니다. 반환하는 함수에는 파라미터가 없으며 호출될 때마다 `Int` 값을 반환합니다. 함수가 다른 함수를 반환하는 방법을 알아보려면 반환 타입으로의 함수 타입 (Function Types as Return Types) 을 참고 바랍니다.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

`makeIncrementer` 의 반환 타입은 `() -> Int` 입니다. 이것은 단순한 값이 아닌 *함수* 를 반환한다는 의미입니다. 반환하는 함수에는 파라미터가 없으며 호출될 때마다 `Int` 값을 반환합니다. 함수가 다른 함수를 반환하는 방법을 알아보려면 반환 타입으로의 함수 타입 (Function Types as Return Types) 을 참고 바랍니다.

`makeIncrementer(forIncrement:)` 함수는 반환될 현재 증가분을 저장하기 위해 `runningTotal` 이라는 정수 변수를 정의합니다. 이 변수는 `0` 으로 초기화 됩니다.

`makeIncrementer(forIncrement:)` 함수는 `forIncrement` 의 인자 라벨의 하나의 `Int` 파라미터를 가지고 `amount` 라는 파라미터 이름을 가지고 있습니다. 이 파라미터에 전달된 인자값은 반환된 증가 함수가 호출 될 때마다 `runningTotal` 을 얼마나 증가시켜야 하는지 지정합니다. `makeIncrementer` 함수는 실제 증가를 수행하는 `incrementer` 라는 중첩 함수를 정의합니다. 이 함수는 간단하게 `amount` 를 `runningTotal` 에 더하고 그 결과를 반환합니다.

단독으로 고려할 때 `incrementer()` 함수는 비정상적으로 보일 수 있습니다:

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

`incrementer()` 함수는 파라미터가 없으며 함수 바디내에 `runningTotal` 과 `amount` 를 참조하고 있습니다. 둘러싸인 함수에 `runningTotal` 과 `amount` 대한 *참조 (reference)* 를 캡처하고 함수 내에서 사용합니다. 참조를 캡처하는 것은 `makeIncrementer` 호출이 종료될 때 `runningTotal` 과 `amount` 가 사라지지 않고 다음에 `incrementer` 함수가 호출될 때 `runningTotal` 을 사용할 수 있습니다.

> NOTE
> 최적화로 Swift는 값이 클로저에 의해 변경되지 않고 클로저가 생성된 후 값이 변경되지 않는 경우 값의 복사본을 캡처하고 저장할 수 있습니다.
>
> Swift는 더이상 필요하지 않을때 변수를 처리하는 것과 관련된 모든 메모리 관리도 처리합니다.

다음은 `makeIncrementer` 수행에 대한 예입니다:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

이 예제에서는 호출될 때마다 `runningTotal` 변수에 `10` 을 더하는 증가 함수를 참조하도록 `incrementByTen` 이라는 상수를 설정합니다. 함수를 여러번 호출하면 이 동작이 수행되는 동작을 보여줍니다:

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

두번째 증가를 생성하면 새로운 분리된 `runningTotal` 변수에 참조 저장됩니다:

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7
```

기존 증가 (`incrementByTen`)을 다시 호출하면 그것의 `runningTotal` 변수는 이어서 증가되고 `incrementBySeven` 으로 캡처된 변수는 영향을 주지 않습니다:

```
incrementByTen()
// returns a value of 40
```

> NOTE
> 클래스 인스턴스의 프로퍼티에 클로저를 할당하고 클로저가 인스턴스 또는 멤버를 참조하여 해당 인스턴스를 캡처하면 클로저와 인스턴스 사이에 강한 참조 사이클이 생성됩니다. Swift는 캡처 목록을 사용하여 이러한 강한 참조 사이클을 깨뜨립니다. 자세한 내용은 클로저의 강한 참조 사이클 (Strong Reference Cycles for Closures) 을 참고 바랍니다.

클로저는 참조 타입 (Closures Are Reference Types)

위 예제에서 `incrementBySeven` 과 `incrementByTen` 은 상수이지만 이러한 상수가 참조하는 클로저는 캡처한 `runningTotal` 변수를 계속 증가시킬 수 있습니다. 이는 함수와 클로저가 *참조 타입 (reference types)* 이기 때문입니다.

함수 또는 클로저를 상수 또는 변수에 할당할 때마다 실제로 해당 상수 또는 변수를 함수 또는 클로저에 대한 *참조* 로 설정합니다. 위의 예에서 `incrementByTen` 은 클로저 자체의 내용이 아니라 상수를 가리키는 클로저 선택입니다.

이것은 또한 서로 다른 2개의 상수 또는 변수에 클로저를 할당한다면 2개의 상수 또는 변수는 모두 같은 클로저를 참조한다는 의미입니다.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50

incrementByTen()
// returns a value of 60
```

위의 예제는 `alsoIncrementByTen` 호출은 `incrementByTen` 호출과 같음을 보여줍니다. 2개 모두 같은 클로저를 참조하기 때문에 둘다 증가하고 같은 러닝 합계를 반환합니다.

이스케이프 클로저 (Escaping Closures)

함수에 인자로 클로저를 전달하지만 함수가 반환된 후 호출되는 클로저를 함수를 *탈출 (escape)* 하다 라고 말합니다. 클로저를 파라미터로 갖는 함수를 선언할 때 이 클로저는 탈출을 허락한다는 의미로 파라미터의 타입 전에 `@escaping` 을 작성할 수 있습니다.

클로저가 탈출할 수 있는 한가지 방법은 함수 바깥에 정의된 변수에 저장되는 것입니다. 예를 들어 비동기적 작업을 시작하는 대부분의 함수는 완료 핸들러로 클로저를 사용합니다. 이 함수는 작업을 시작한 후에 반환되지만 작업이 완료될때까지 클로저가 호출되지 않습니다. 클로저는 나중에 호출하려면 탈출해야 합니다. 예를 들어:

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
```

`someFunctionWithEscapingClosure(_:)` 함수는 인자로 클로저를 가지고 있고 함수 바깥에 선언된 배열에 추가합니다. 함수의 파라미터에 `@escaping` 을 표시하지 않으면 컴파일 시 에러가 발생합니다.

`self` 를 참조하는 이스케이프 클로저은 `self` 가 클래스의 인스턴스를 참조하는 경우 특별한 고려가 필요합니다. 이스케이프 클로저에 `self` 캡처는 강한 참조 사이클이 생기기 쉽습니다. 참조 사이클에 대한 자세한 내용은 자동 참조 카운팅 (Automatic Reference Counting) 을 참조 바랍니다.

일반적으로 클로저는 클로저 내부에서 변수를 사용하여 암시적으로 변수를 캡처하지만 이 경우에는 명시적이어야 합니다. `self` 를 캡처하려면 사용할 때 명시적으로 `self` 를 작성하거나 클로저의 캡처 목록에 `self` 를 포함합니다. `self` 를 명시적으로 작성하는데 의도를 표현하고 참조 사이클이 없음을 확인하도록 상기시켜 줍니다. 예를 들어 아래 코드에서 `someFunctionWithEscapingClosure(_:)` 에 전달된 클로저는 명시적으로 `self` 를 참조합니다. 반대로 `someFunctionWithNonescapingClosure(_:)` 에 전달된 클로저는 비이스케이프 클로저입니다. 즉 암시적으로 `self` 를 참조할 수 있습니다.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

다음은 클로저의 캡처 목록에 포함하여 `self` 를 캡처하고 암시적으로 `self` 를 참조하는 `doSomething()` 입니다:

```
class SomeOtherClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { [self] in x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}
```

`self` 가 구조체 또는 열거형 인스턴스이면 항상 암시적으로 `self` 를 참조할 수 있습니다. 그러나 이스케이프 클로저는 구조 또는 열거 인스턴스이면 `self` 에 대한 변경 가능한 참조를 캡처할 수 없습니다. 구조와 열거는 구조체와 열거형은 값타입 (Structures and Enumerations Are Value Types) 에서 설명 했듯이 공유 변경을 허용하지 않습니다.

```
struct SomeStruct {
    var x = 10
    mutating func doSomething() {
        someFunctionWithNonescapingClosure { x = 200 }  // Ok
        someFunctionWithEscapingClosure { x = 100 }     // Error
    }
}
```

위의 예제에서 `someFunctionWithEscapingClosure` 함수 호출은 변경 가능한 메서드 내부에 있기 때문에 에러이고 `self` 는 변경 가능합니다. 이것은 이스케이프 클로저는 구조체인 `self` 를 변경가능한 참조로 캡처할 수 없다는 규칙을 위반합니다.

자동 클로저 (Autoclosures)

자동 클로저 (autoclosure) 는 함수에 인자로 전달되는 표현식을 래핑하기 위해 자동으로 생성되는 클로저입니다. 인자를 가지지 않으며 호출될 때 내부에 래핑된 표현식의 값을 반환합니다. 이러한 구문상의 편의를 통해 명시적 클로저 대신에 일반 표현식을 작성하여 함수의 파라미터 주위의 중괄호를 생략할 수 있습니다.

자동 클로저를 가지는 함수를 *호출* 하는 것은 일반적이지만 이러한 함수를 *구현* 하는 것은 일반적이지 않습니다. 예를 들어 `assert(condition:message:file:line:)` 함수는 `condition` 과 `message` 파라미터에 대한 자동 클로저를 가집니다. `condition` 파라미터는 오직 디버그 빌드인지 판단하고 `message` 파라미터는 `condition` 이 `false` 인지만 판단됩니다.

클로저가 호출될 때까지 코드 내부 실행이 되지 않기때문에 자동 클로저는 판단을 지연시킬 수 있습니다. 판단 지연은 코드 판단 시기를 제어할 수 있기 때문에 사이드 이펙트가 있거나 계산이 오래 걸리는 코드에 유용합니다. 아래 코드는 클로저가 어떻게 판단을 지연하는지 보여줍니다.

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

클로저 내부의 코드에 의해 `customersInLine` 배열의 첫번째 요소는 삭제되지만 클로저가 실제로 호출되기 전까지 삭제되지 않습니다. 클로저가 호출되지 않으면 클러저 내부의 표현식은 판단되지 않습니다. 이것은 배열의 요소가 삭제되지 않는다는 의미입니다. `customerProvider` 타입은 `String` 이 아니고 파라미터가 없고 문자열을 반환하는 `() -> String` 입니다.

함수의 인자로 클로저를 전달하면 위와 같은 지연 판단과 동일한 동작을 가질 수 있습니다.

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Prints "Now serving Alex!"
```

`serve(customer:)` 함수는 소비자의 이름을 반환하는 명시적 클로저를 가집니다. 아래 `serve(customer:)` 의 버전은 같은 동작을 수행하지만 명시적 클로저를 가지는 대신에 파라미터 타입에 `@autoclosure` 속성을 표기하여 자동 클로저를 가집니다. 이제 클로저 대신 `String` 인수를 받는 것처럼 함수를 호출할 수 있습니다. `customerProvider` 파라미터의 타입은 `@autoclosure` 속성으로 표시되므로 인자는 자동으로 클로저로 변환됩니다.

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Ewa!"
```

> NOTE
> 자동 클로저 남용은 코드 이해를 어렵게 만들 수 있습니다. 컨텍스트와 함수 이름은 판단이 연기되고 있음을 분명히 해야합니다.

자동 클로저가 이스케이프를 허용하길 원한다면 `@autoclosure` 와 `@escaping` 속성을 둘다 사용하면 됩니다. `@escaping` 속성은 위의 이스케이프 클로저 (Escaping Closures) 에서 설명되어 있습니다.

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

위의 코드에서 `customerProvider` 인자로 전달된 클로저를 호출하는 대신에 `collectCustomerProviders(_:)` 함수는 클로저를 `customerProviders` 배열에 추가합니다. 이 배열은 함수의 범위 밖에 선언됩니다. 이것은 배열에 클로저는 함수가 반환된 후에 실행될 수 있다는 의미입니다. 그 결과 `customerProvider` 인자의 값은 함수의 범위를 벗어날 수 있어야 합니다.

옵셔널 체이닝

일단 옵셔널 다시보고 돌아옵시다

옵셔널 타입

옵셔널 타입

Optional type

- Swift 옵셔널 데이터 타입은 대부분의 다른 프로그래밍 언어에 없는 새로운 개념이다.
- 옵셔널 타입의 목적은 변수 또는 상수에 값이 할당되지 않은 상황을 처리하기 위해 안전하고 일관된 접근 방식을 제공하는 것이다.
- 변수를 선언할 때, 데이터 타입 선언 다음에 '?' 문자를 두어 옵셔널이 되게 한다.

```
var index: Int?
```

- 이제 index 변수는 정숫값이 할당되거나 아무런 값도 할당되지 않을 수 있다. 내부적으로 컴파일러와 런타임의 관점에서 볼 때 어떤 값도 할당되지 않은 옵셔널은 실제로 **nil**의 값을 갖는다.

옵셔널 타입

Optional type

- 옵셔널은 할당된 값이 있는지를 식별하기 위한 테스트를 다음과 같이 쉽게 할 수 있다.

```
var index: Int?

if index != nil {
  // index 변수는 값이 할당되어 있다.
} else {
  // index 변수는 값이 할당되어 있지 않다.
}
```

강제 언래핑

Forced unwrapping

- 만약 옵셔널에 값이 할당되었다면 해당 값이 옵셔널 내에서 '**래핑되었다(wrapped)**'고 말한다.
- 옵셔널 안에 래핑된 값을 사용할 때는 **강제 언래핑(forced unwrapping)**이라는 개념을 이용하게 된다.
  - 래핑된 값은 옵셔널 데이터 타입에서 옵셔널 이름 뒤에 느낌표(!)를 두어 추출되게 한다.

```swift
var index: Int?

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
  print(treeArray[index!])
} else {
  print("index does not contain a value")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Birch
```

강제 언래핑

Forced unwrapping

- 옵셔널로 할당된 값을 강제언래핑하지 않고 직접 접근한다면 에러가 생긴다

```swift
var index: Int?

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
  print(treeArray[index])
} else {
  print("index does not contain a value")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
/main.swift:8:19: error: value of optional type 'Int?' must be unwr
e of type 'Int'
  print(treeArray[index])
                  ^
/main.swift:8:19: note: coalesce using '??' to provide a default wh
 value contains 'nil'
  print(treeArray[index])
                  ^
                        ?? <#default value#>
/main.swift:8:19: note: force-unwrap using '!' to abort execution i
value contains 'nil'
  print(treeArray[index])
                  ^
                        !
*** No output. ***
```

강제 언래핑

Forced unwrapping

- 강제언래핑을 시도할 때 부주의하면 에러가 생길 수 있다
  - 가능하면 사용하지 않는 것이 좋다

```swift
var index: Int?

index = 5

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
  print(treeArray[index!])
} else {
  print("index does not contain a value")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Birch
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Swift/ContiguousArrayBuffer.swift:600: Fatal error: Index out of r
Current stack trace:
0    libswiftCore.so                    0x00007fb7d65a58c0 _swift_
alErrorInFile + 112
1    libswiftCore.so                    0x00007fb7d626f161 <unavai
2    libswiftCore.so                    0x00007fb7d626ee86 <unavai
3    libswiftCore.so                    0x00007fb7d626ec7a <unavai
4    libswiftCore.so                    0x00007fb7d626e6a0 _assert
ile:line:flags:) + 315
```

옵셔널 바인딩

Optional binding

- 강제 언래핑 대신, 옵셔널로 할당된 값은 옵셔널 바인딩(optional binding)을 이용하여 임시 변수나 상수에 할당할 수 있다.
- 다음의 코드는 두 가지 작업을 수행한다
  - 지정된 옵셔널이 값을 가지고 있는지를 확인한다
  - 옵셔널 변수가 값을 가지고 있는 경우에 선언된 상수 또는 변수에 그 값을 할당하고 코드가 실행된다

```
if let constantName = optionalName {

}

if var variableName = optionalName {

}
```

옵셔널 바인딩

Optional binding

```swift
var index: Int?

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if let myValue = index {
  print(treeArray[myValue])
} else {
  print("index does not contain a value")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Birch
```

- 위 코드는 index 변수에 할당된 값이 언래핑되어 myValue라는 **임시 상수**에 할당되어 배열에 대한 인덱스로 사용된다.
- 임시 상수는 if 구문 안에서만 유효하다. (if 구문 실행이 끝나면 이 상수는 더 이상 존재하지 않게된다)
- 위의 이유로 옵셔널로 할당된 동일한 이름을 사용해도 충돌이 발생하지 않는다

```swift
if let index = index {
  print(treeArray[index])
} else {
  print("index does not contain a value")
```

옵셔널 바인딩

Optional binding

- 옵셔널 바인딩은 여러 개의 옵셔널을 언래핑하고 조건문을 포함하는 데 사용될 수도 있다.

if let 상수명1 = 옵셔널이름1, let 상수명2 = 옵셔널이름2, let 상수명3 = 옵셔널이름3, … <조건식> {

}

```
var pet1: String?
var pet2: String?

pet1 = "cat"
pet2 = "dog"

if let firstPet = pet1, let secondPet = pet2 {
  print(firstPet)
  print(secondPet)
} else {
  print("insufficient pets")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Birch
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
cat
dog
```

옵셔널 바인딩

Optional binding

- 다음의 코드는 조건문을 사용하는 예제다.
  - petCount가 1보다 크지 않다면 옵셔널 바인딩이 수행되지 않을 것이다.

```swift
var pet1: String?
var pet2: String?

pet1 = "cat"
pet2 = "dog"

let petCount = 15

if let firstPet = pet1, let secondPet = pet2, petCount > 1 {
    print(firstPet)
    print(secondPet)
} else {
    print("insufficient pets")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
cat
dog
```

강제적으로 언래핑

- 강제적으로 언래핑되도록 옵셔널을 선언할 수도 있다.
- 이 방식으로 옵셔널을 선언하면 강제 언래핑이나 옵셔널 바인딩을 하지 않아도 값에 접근할 수 있다.
- 옵셔널을 선언할 때 물음표(?) 대신에 느낌표(!)를 사용하여 강제적으로 언래핑되도록 하는 것이다.

```swift
var index: Int!   // 이제 옵셔널은 강제적으로 언래핑된다

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
    print(treeArray[index]) // 이제 index를 언래핑 없이 바로 쓸 수 있다
} else {
    print("index does not contain a value")
}
```

```
Swift version 5.7 (swift-5.7-RELEASE)
Target: x86_64-unknown-linux-gnu
Birch
```

옵셔널 주의점

- 할당된 값이 없거나 nil을 할당할 수 있는 것은 옵셔널 타입뿐이다.
- 옵셔널이 아닌 변수 또는 상수에는 nil을 할당할 수 없다.
- 다음의 코드는 모두 유효하지 않다. (옵셔널로 선언된 변수가 아니다)

~~var myInt = nil~~ ✗

~~var myString: String = nil~~ ✗

~~let myConstant = nil~~ ✗

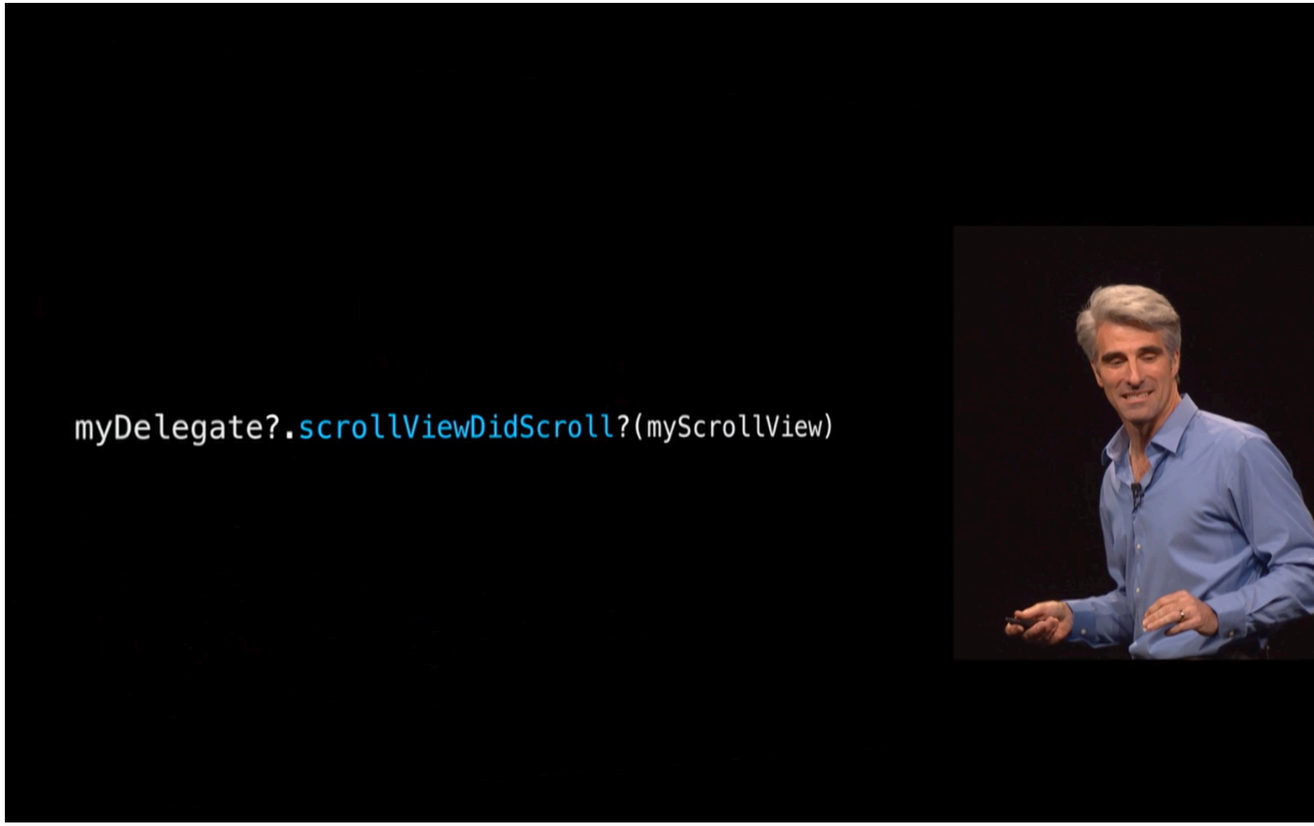
옵셔널 체이닝

Optional Chaining

옵셔널 체이닝 (Optional chaining) 은 현재 nil 일 수 있는 옵셔널 인 프로퍼티, 메서드, 그리고 서브 스크립트를 조회하고 호출하기 위한 프로세스 입니다.

옵셔널에 값이 포함되어 있으면 프로퍼티, 메서드, 또는 서브 스크립트는 호출에 성공합니다. 옵셔널이 nil 이면 프로퍼티, 메서드, 또는 서브 스크립트 호출은 nil 을 반환합니다.

여러 조회는 함께 연결될 수 있고 체인에 어느 부분이라도 nil 이면 전체 체인은 실패합니다.



강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

옵셔널이 nil 이 아닌 경우 프로퍼티, 메서드 또는 서브 스크립트를 호출하려는 옵셔널 값 뒤에 물음표 (?)를 배치하여 옵셔널 체이닝을 지정합니다.

이것은 값에 강제 언래핑을 하기 위해 옵셔널 값 뒤에 느낌표 (!)를 배치하는 것과 유사합니다.

이것의 주요 차이점은 옵셔널이 nil 일 때 옵셔널 체이닝은 실패하는 반면에 강제 언래핑은 런타임 에러를 트리거 합니다.

옵셔널 체이닝은 nil 값에 대해 호출될 수 있다는 사실을 반영하기 위해 조회하는 프로퍼티, 메서드, 또는 서브 스크립트가 옵셔널 값이 아닌 값을 반환하더라도 항상 옵셔널 값을 반환합니다.

옵셔널 반환 값으로 옵셔널 체이닝 호출이 성공 (반환된 옵셔널 체이닝에 값이 포함됨)했는지 실패 (반환된 옵셔널 값은 nil)했는지 확인할 수 있습니다.

특히 옵셔널 체이닝 호출의 결과는 예상되는 반환값과 동일한 타입이지만 옵셔널로 래핑됩니다.

일반적으로 Int 로 반환하는 프로퍼티는 옵셔널 체이닝을 통해 접근하면 Int? 를 반환합니다.

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

다음의 코드들은 옵셔널 체이닝이 강제 언래핑과 어떻게 다른지 보여주고 성공여부를 확인할 수 있도록 합니다.

첫번째, Person 과 Residence 라는 2개의 클래스는 정의 됩니다.

Residence 인스턴스는 기본값이 1 인 numberOfRooms 라는 Int 프로퍼티를 가지고 있습니다.

Person 인스턴스는 Residence? 타입의 옵셔널 residence 프로퍼티를 가지고 있습니다.

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

내 코드 실행

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

새로운 Person 인스턴스를 생성하면 residence 프로퍼티는 옵셔널 규칙에 따라 nil 로 초기화 됩니다.

이 코드에서 john 은 nil 의 residence 프로퍼티 값을 가지고 있습니다.

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()
```

내 코드 실행

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

값에 강제 언래핑을 하기 위해 residence 뒤에 느낌표를 배치하여 사람의 residence 에 numberOfRooms 프로퍼티를 접근하면 residence 값이 없기 때문에 런타임 에러를 트리거 합니다.

John.residence 가 nil 값이 아니고 roomCount 에 방의 적절한 숫자를 포함한 Int 로 설정하면 위의 코드는 정상동작 합니다. 그러나 residence 가 nil 이면 이 코드는 항상 런타임 에러를 트리거 합니다.

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

내 코드 실행

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

옵셔널 체이닝은 numberOfRooms 의 값에 접근하기 위한 대안으로 제공합니다. 옵셔널 체이닝을 사용하기 위해 느낌표 위치에 물음표를 사용합니다.

이것은 Swift가 옵셔널 residence 프로퍼티를 "체인"하고 residence 가 존재하면 numberOfRooms 값을 조회하도록 합니다.

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

내 코드 실행

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

numberOfRooms 접근하기 위한 시도는 실패할 수 있으므로 옵셔널 체이닝은 Int? 타입이나 "옵셔널 Int"의 값을 반환합니다.

예제에서 처럼 residence 가 nil 일 경우 numberOfRooms 접근이 불가능 한 사실을 반영하기 위해 옵셔널 Int 는 nil 입니다.

옵셔널 Int는 언래핑 한 정수에 옵셔널 바인딩으로 접근되고 roomCount 상수에 옵셔널이 아닌 값을 할당합니다.

numberOfRooms 가 옵셔널 Int 가 아니더라도 마찬가지입니다. 옵셔널 체인으로 조회 된다는 것은 numberOfRooms 호출은 항상 Int 대신 Int? 를 반환한다는 의미입니다.

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

내 코드 실행

강제 언래핑의 대한 대안으로 옵셔널 체이닝

Optional Chaining as an Alternative to Forced Unwrapping

더이상 nil 값을 갖지 않기 위해 john.residence 에 Residence 인스턴스를 할당할 수 있습니다.

john.residence 는 nil 이 아닌 실제 Residence 인스턴스를 포함합니다.

이전 처럼 같은 옵셔널 체이닝으로 numberOfRooms 에 접근하면 기본 numberOfRooms 값인 1 의 Int? 를 반환할 것입니다.

나의 플레이그라운드
My Playground

```
john.residence = Residence()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

하나 이상의 레벨 깊이 인 프로퍼티, 메서드, 그리고 서브 스크립트를 호출하기 위해 옵셔널 체이닝을 사용할 수 있습니다.

타입 호환되는 복잡한 모델 내에서 하위 프로퍼티로 내려갈 수 있으며 해당 하위 프로퍼티에 프로퍼티, 메서드, 그리고 서브 스크립트에 접근 가능한지 확인할 수 있습니다.

이 코드는 여러 레벨 옵셔널 체이닝의 예를 포함하여 몇몇의 후속 예제에서 사용할 4개의 모델 클래스를 정의합니다.

이 클래스는 관련 프로퍼티, 메서드, 그리고 서브 스크립트를 가지는 Room 과 Address 클래스를 추가하여 위의 Person 과 Residence 모델을 확장합니다.

Person 클래스는 이전과 같게 정의됩니다:

나의 플레이그라운드
My Playground

```
// 옵셔널 체이닝 클래스 예시

class Person {
    var residence: Residence?
}

class Residence {
    // Residence 클래스를 정의하는 코드
}
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

Residence 클래스는 이전보다 더 복잡합니다.

이제 Residence 클래스는 [Room] 타입의 빈 배열을 가지고 초기화 되는 rooms 라는 변수 프로퍼티를 정의합니다:

나의 플레이그라운드
My Playground

```
class Residence {
    var rooms: [Room] = []
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}

class Room {
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

이 Residence 버전은 Room 인스턴스의 배열을 저장하기 때문에 numberOfRooms 프로퍼티는 저장된 프로퍼티가 아닌 계산된 프로퍼티로 구현됩니다.

계산된 numberOfRooms 프로퍼티는 rooms 배열에서 count 프로퍼티의 값을 반환합니다.

rooms 배열에 접근하는 짧은 구문을 위해 Residence 에 버전은 rooms 배열에 요청된 인덱스로 방에 접근을 제공하는 읽기-쓰기 서브 스크립트를 제공합니다.

이 Residence 버전은 주택에 있는 방의 갯수를 출력하는 printNumberOfRooms 라는 메서드도 제공합니다.

마지막으로 Residence 는 Address? 타입을 가지는 address 라는 옵셔널 프로퍼티를 정의합니다. 이 프로퍼티에 대한 Address 클래스 타입은 다음에 정의되어 있습니다.

나의 플레이그라운드
My Playground

```
class Residence {
    var rooms: [Room] = []
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}

class Room {
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

rooms 배열에 사용된 Room 클래스는 name 이라는 프로퍼티와 적절한 방 이름을 프로퍼티에 설정하는 초기화 구문을 가진 클래스 입니다.

나의 플레이그라운드
My Playground

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}

class Address {
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

이 모델의 마지막 클래스는 Address 라 합니다.

이 클래스는 String? 타입의 3개의 옵셔널 프로퍼티를 가집니다.

buildingName 과 buildingNumber 인 첫번째, 두번째 프로퍼티는 주소의 부분으로 특정 건물을 식별하기 위한 대안책입니다.

세번째 프로퍼티 인 street 는 주소에 대한 길의 이름으로 사용됩니다.

나의 플레이그라운드
My Playground

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street
         {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}
```

내 코드 실행

옵셔널 체이닝에 대한 모델 클래스 정의

Defining Model Classes for Optional Chaining

또한 Address 클래스는 String? 의 반환 타입을 가지는 buildingIdentifier() 라는 메서드도 제공합니다.

이 메서드는 주소의 프로퍼티를 확인하고 값이 있으면 buildingName 을 반환하거나 둘다 값이 있으면 street 과 연결된 buildingNumber 를 반환하고 값이 없으면 nil 을 반환합니다.

나의 플레이그라운드
My Playground

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street
         {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}
```

내 코드 실행

옵셔널 체이닝을 통해 프로퍼티 접근

Accessing Properties Through Optional Chaining

앞에서 설명 했듯이 옵셔널 값의 프로퍼티에 접근하고 프로퍼티 접근이 성공하면 검사하기 위해 옵셔널 체이닝을 사용할 수 있습니다.

새로운 Person 인스턴스를 생성하기 위해 위에 정의된 클래스를 사용하고 이전처럼 numberOfRooms 프로퍼티에 접근합니다.

John.residence 는 nil 이기 때문에 옵셔널 체이닝은 이전과 같이 실패를 호출합니다.

나의 플레이그라운드
My Playground

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

내 코드 실행

옵셔널 체이닝을 통해 프로퍼티 접근

Accessing Properties Through Optional Chaining

이 예제에서 john.residence 가 nil 이기 때문에 john.residence 에 address 프로퍼티에 설정하기 위한 시도는 실패 할 것입니다.

이 할당은 = 연산자의 우항의 코드는 평가되지 않으므로 옵셔널 체이닝의 일부입니다.

나의 플레이그라운드
My Playground

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."


let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

내 코드 실행

옵셔널 체이닝을 통해 프로퍼티 접근

Accessing Properties Through Optional Chaining

이전 예제에서 상수에 접근하는 것은 어떠한 영향도 없기 때문에 someAddress 가 평가되지 않는다는 것을 쉽게 파악할 수 없습니다.

이 코드는 같은 할당을 수행하지만 주소를 생성하기 위해 함수를 사용합니다.

이 함수는 값을 반환하기 전에 "Function was called"를 출력하여 = 연산자 우항이 평가되었음을 알 수 있습니다.

아무것도 출력되지 않기 때문에 createAddress() 함수가 호출되지 않음을 알 수 있습니다.

나의 플레이그라운드
My Playground

```
func createAddress() -> Address {
    print("Function was called.")

    let someAddress = Address()
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}

john.residence?.address = createAddress()
```

내 코드 실행

옵셔널 체이닝을 통한 함수 호출

Calling Methods Through Optional Chaining

이 메서드는 반환 타입이 지정되지 않았습니다.

그러나 반환 타입이 없는 함수와 메서드는 반환값 없는 함수(Functions Without Return Values)에서 설명했듯이 Void 의 암시적 반환 타입을 가지고 있습니다.

이것은 () 의 값 또는 빈 튜플을 반환한다는 의미입니다.

나의 플레이그라운드
My Playground

```
class Residence {
    var rooms: [Room] = []
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

내 코드 실행

옵셔널 체이닝을 통한 함수 호출

Calling Methods Through Optional Chaining

옵셔널 체이닝을 사용하여 옵셔널 값에 대해 메서드를 호출하면 반환값은 옵셔널 체이닝을 통해 호출하면 옵셔널 타입이기 때문에 메서드의 반환 타입은 Void 가 아닌 Void? 입니다.

메서드가 반환값을 정의하지 않았어도 printNumberOfRooms() 메서드 호출이 가능한지 if 구문을 사용하여 확인할 수 있습니다.

printNumberOfRooms 호출의 반환값을 nil 과 비교하여 메서드 호출이 성공했는지 확인합니다:

나의 플레이그라운드
My Playground

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

내 코드 실행

옵셔널 체이닝을 통한 함수 호출

Calling Methods Through Optional Chaining

옵셔널 체이닝을 통해 프로퍼티를 설정하려는 경우에도 마찬가지입니다.

옵셔널 체이닝을 통해 프로퍼티 접근 (Accessing Properties Through Optional Chaining) 에서 위의 예제는 residence 프로퍼티가 nil 이지만 John.residence 에 대해 address 값을 설정 하려고 합니다.

옵셔널 체이닝을 통해 프로퍼티를 설정하려는 모든 시도는 nil 과 비교하여 프로퍼티에 값이 성공적으로 설정되었는지 확인할 수 있는 Void? 타입의 값을 반환합니다:

나의 플레이그라운드
My Playground

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."


if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

내 코드 실행

옵셔널 체이닝을 통한 서브 스크립트 접근

Accessing Subscripts Through Optional Chaining

옵셔널 값의 서브 스크립트에서 값을 조회하고 설정하고 해당 서브 스크립트 호출이 성공했는지 확인하기 위해 옵셔널 체이닝을 사용할 수 있습니다.

옵셔널 체이닝을 통해 옵셔널 값의 서브 스크립트에 접근할 때 물음표는 서브 스크립트의 대괄호 전에 위치합니다. 옵셔널 체이닝 물음표는 항상 옵셔널 표현구 부분의 바로 다음에 위치합니다.

이 예제는 Residence 클래스에 정의된 서브 스크립트를 사용하여 john.residence 프로퍼티에 rooms 배열의 첫번째 방 이름을 조회합니다.

현재 john.residence 는 nil 이므로 서브 스크립트 호출은 실패합니다.

나의 플레이그라운드
My Playground

```
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

내 코드 실행

옵셔널 체이닝을 통한 서브 스크립트 접근

Accessing Subscripts Through Optional Chaining

서브 스크립트 호출에 옵셔널 체이닝 물음표는 john.residence 가 옵셔널 체이닝이 시도되는 옵셔널 값이기 때문에 john.residence 다음과 대괄호 전에 위치합니다.

유사하게 옵셔널 체이닝으로 서브 스크립트를 통해 새로운 값을 설정할 수 있습니다.

residence가 현재 nil 이므로 서브 스크립트 설정은 실패합니다.

나의 플레이그라운드
My Playground

```
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."


john.residence?[0] = Room(name: "Bathroom")
```

내 코드 실행

옵셔널 체이닝을 통한 서브 스크립트 접근

Accessing Subscripts Through Optional Chaining

rooms 배열에 하나 이상의 Room 인스턴스 가지고 john.residence 에 실제 Residence 인스턴스를 생성하고 할당하면 옵셔널 체이닝을 통해 rooms 배열에 항목을 접근하기 위해 Residence 서브 스크립트를 사용할 수 있습니다.

나의 플레이그라운드
My Playground

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."
```

내 코드 실행

옵셔널 타입에 서브 스크립트 접근

Accessing Subscripts of Optional Type

서브 스크립트가 Swift의 Dictionary 타입의 키 서브 스크립트와 같이 옵셔널 타입의 값을 반환하는 경우 옵셔널 반환값을 연결하기 위해 서브 스크립트의 닫는 대괄호 뒤에 물음표를 추가합니다.

나의 플레이그라운드
My Playground

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now
 [80, 94, 81]
```

내 코드 실행

옵셔널 타입에 서브 스크립트 접근

Accessing Subscripts of Optional Type

이 예제는 String 키를 Int 값 배열에 매핑하는 2개의 키-값 쌍을 포함하는 testScores 라는 딕셔너리를 정의합니다.

이 예제는 "Dave" 배열에 첫번째 항목에 91 을 설정하고 "Bev" 배열에 첫번째 항목에 1 을 더하고 "Brian" 키에 대한 배열에 첫번째 항목에 값을 설정하기 위해 옵셔널 체이닝을 사용합니다.

testScores 딕셔너리는 "Dave" 와 "Bev" 에 대한 키를 가지고 있으므로 첫번째 두번째 호출은 성공합니다.

testScores 딕셔너리는 "Brian" 에 대한 키를 가지고 있지 않으므로 세번째 호출은 실패합니다.

나의 플레이그라운드
My Playground

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now
 [80, 94, 81]
```

내 코드 실행

여러 수준의 체인연결

Linking Multiple Levels of Chaining

여러 수준의 옵셔널 체이닝을 연결하여 모델 내에서 프로퍼티, 메서드, 그리고 서브 스크립로 깊게 접근할 수 있습니다.

그러나 여러 수준의 옵셔널 체이닝은 반환된 값에 더 많은 수준의 옵션성을 추가하지 않습니다.

다른 말로 표현하면:

- 조회하려는 타입이 옵셔널이 아니면 옵셔널 체이닝 때문에 옵셔널이 됩니다.
- 조회하려는 타입이 이미 옵셔널이면 체이닝 때문에 더 많은 옵셔널이 되지 않습니다.

따라서:

- 옵셔널 체이닝으로 Int 값을 조회하려고 하면 사용된 체이닝의 수준과 상관없이 항상 Int? 가 반환됩니다.
- 유사하게 옵셔널 체이닝으로 Int? 값을 조회하려고 하면 사용된 체이닝의 수준과 상관없이 항상 Int? 가 반환됩니다.

여러 수준의 체인연결

Linking Multiple Levels of Chaining

이 예제는 john 의 residence 프로퍼티에 address 프로퍼티에 street 프로퍼티를 접근합니다.

여기에서는 residence 와 address 프로퍼티를 통해 연결하기 위해 2단계 수준의 옵셔널 체이이 사용되며 둘 다 옵셔널 타입입니다.

나의 플레이그라운드
My Playground

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

내 코드 실행

여러 수준의 체인연결

Linking Multiple Levels of Chaining

john.residence 의 값은 현재 유효한 Residence 인스턴스를 가지고 있습니다. 그러나 John.residence.address 의 값은 현재 nil 입니다.

이것 때문에 john.residence?.address?.street 을 호출하면 실패합니다.

여기서는 street 프로퍼티의 값을 조회하려고 합니다.

이 프로퍼티의 타입은 String? 입니다.

따라서 john.residence?.address?.street 의 반환값은 2단계 옵셔널 체이닝이 프로퍼티의 옵셔널 타입에 적용되었지만 String? 입니다.

나의 플레이그라운드
My Playground

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

내 코드 실행

여러 수준의 체인연결

Linking Multiple Levels of Chaining

john.residence.address 에 대한 값으로 Address 인스턴스를 설정하고 주소의 street 프로퍼티에 대해 값을 설정하면 여러 수준의 옵셔널 체이닝을 통해 street 프로퍼티의 값에 접근할 수 있습니다.

이 예제에서 john.residence 의 값은 현재 유효한 Residence 인스턴스 이므로 john.residence 에 address 프로퍼티에 값을 설정하는 것이 가능합니다.

나의 플레이그라운드
My Playground

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence?.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "John's street name is Laurel Street."
```

내 코드 실행

옵셔널 반환값으로 메서드 체이닝

Chaining on Methods with Optional Return Values

이전 예제는 어떻게 옵셔널 체이닝을 통해 옵셔널 타입의 프로퍼티의 값을 조회해야 하는지 보여줍니다.

옵셔널 체이닝은 또한 옵셔널 타입의 값을 반환하는 메서드를 호출하고 필요하다면 메서드의 반환값을 연결하기 위해 사용할 수 있습니다.

이 예제는 옵셔널 체이닝을 통해 Address 클래스의 buildingIdentifier() 메서드를 호출합니다.

이 메서드는 String? 타입의 값을 반환합니다.

위에서 설명 했듯이 옵셔널 체이닝 이후 이 메서드 호출의 반환 타입은 String? 입니다.

나의 플레이그라운드
My Playground

```
if let buildingIdentifier =
 john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

내 코드 실행

옵셔널 반환값으로 메서드 체이닝

Chaining on Methods with Optional Return Values

메서드의 반환값에서 더 옵셔널 체이닝을 수행하려면 메서드의 소괄호 다음에 옵셔널 체이닝 물음표를 위치시키면 됩니다.

이 예제에서 연결하려는 옵셔널 값이 buildingIdentifier() 메서드 자체가 아닌 buildingIdentifier() 메서드의 반환값이므로 소괄호 뒤에 옵셔널 체이닝 물음표를 위치시킵니다.

나의 플레이그라운드
My Playground

```
if let buildingIdentifier =
 john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."


if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The")
     {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier does not begin with
         \"The\".")
    }
}
// Prints "John's building identifier begins with "The"."
```

내 코드 실행

동시성

동시성

Concurrency

Swift 는 구조화된 방식으로 비동기 (asynchronous) 와 병렬 (parallel) 코드 작성을 지원합니다. 비동기 코드 (Asynchronous code) 는 일시적으로 중단되었다가 다시 실행할 수 있지만 한번에 프로그램의 한 부분만 실행됩니다.

프로그램에서 코드를 일시 중단하고 다시 실행하면 UI 업데이트와 같은 짧은 작업을 계속 진행하면서 네트워크를 통해 데이터를 가져오거나 파일을 분석하는 것과 같은 긴 실행 작업을 계속할 수 있습니다.

병렬 코드 (Parallel code) 는 동시에 코드의 여러부분이 실행됨을 의미합니다—예를 들어 4코어 프로세서의 컴퓨터는 각 코어가 하나의 작업을 수행하므로 코드의 4부분을 동시에 실행할 수 있습니다.

병렬과 비동기 코드를 사용하는 프로그램은 한 번에 여러 작업을 수행합니다; 외부 시스템을 기다리는 작업을 일시 중단하고 이 코드를 메모리 안전한 방식으로 더 쉽게 작성할 수 있도록 합니다.

동시성

Concurrency

병렬 또는 비동기 코드의 추가 스케줄링 유연성에는 복잡성이 증가하는 비용도 수반됩니다.

Swift 를 사용하면 일부 컴파일 시간 검사를 가능하게 하는 방식으로 표현할 수 있습니다—예를 들어 실행자를 사용하여 변경가능한 상태에 안전하게 접근할 수 있습니다.

그러나 느리거나 버그가 있는 코드에 동시성을 추가한다고 해서 코드가 빠르거나 올바르게 동작한다는 보장은 없습니다.

사실 동시성을 추가하면 코드를 디버그하기 더 어렵게 만들 수도 있습니다.

그러나 동시성이 필요한 코드에서 동시성에 대한 Swift 의 언어 수준 지원을 사용하면 Swift 가 컴파일 시간에 문제를 찾는데 도움이 될 수 있습니다.

동시성

Concurrency

이전에 동시성 코드를 작성한 적이 있다면 쓰레드 동작에 익숙할 것입니다.

Swift 에서 동시성 모델은 쓰레드의 최상단에 구축되지만 직접적으로 상호작용하지 않습니다.

Swift 에서 비동기 함수는 실행중인 쓰레드를 포기할 수 있습니다.

그러면 첫번째 함수가 차단되는 동안 해당 쓰레드에서 다른 비동기 함수가 실행될 수 있습니다.

동시성

Concurrency

Swift 의 언어 지원을 사용하지 않고 동시성 코드를 작성할 수 있지만 해당 코드는 읽기가 더 어려운 경우가 많습니다.

예를 들어 다음 코드는 사진 이름 목록을 다운로드하고 해당 목록의 첫번째 사진을 다운로드하고 해당 사진을 사용자에게 보여줍니다.

이 간단한 경우에도 완료 핸들러가 연속해서 작성되어야 하므로 결국 중첩 클로저를 작성하게 됩니다.

이 스타일에서는 더 많이 중첩된 복잡한 코드는 빠르게 다루기 어려울 수 있습니다.

나의 플레이그라운드
My Playground

```
// 동시성 예시 코드
listPhotos(inGallery: "Summer Vacation") { photoNames in
    let sortedNames = photoNames.sorted()
    let name = sortedNames[0]
    downloadPhoto(named: name) { photo in
        show(photo)
    }
}
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

비동기 함수 (asynchronous function) 또는 비동기 메서드 (asynchronous method) 는 실행 도중에 일시적으로 중단될 수 있는 특수한 함수 또는 메서드 입니다.

이것은 완료될 때까지 실행되거나 오류가 발생하거나 반환되지 않는 일반적인 동기 함수 또는 메서드 (synchronous functions and methods) 와 대조됩니다.

비동기 함수 또는 메서드는 이 세가지 중 하나를 수행하지만 무언가를 기다리고 있을 때 중간에 일시 중지될 수도 있습니다.

비동기 함수 또는 메서드의 바디 내에서 실행을 일시 중지할 수 있는 부분을 표시합니다.

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

함수 또는 메서드가 비동기 임을 나타내려면 던지는 함수 (throwing function) 를 나타내기 위해 throws 사용하는 것과 유사하게 파라미터 뒤의 선언에 async 키워드를 작성합니다.

함수 또는 메서드가 값을 반환한다면 반환 화살표 (->) 전에 async 를 작성합니다.

예를 들어 갤러리에 사진의 이름을 가져오는 방법은 다음과 같습니다.

비동기와 던지기 둘 다인 함수 또는 메서드는 throws 전에 async 를 작성합니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

비동기 메서드를 호출할 때 해당 메서드가 반환될 때까지 실행이 일시 중단됩니다.

중단될 가능성이 있는 지점을 표시하기 위해 호출 앞에 await 를 작성합니다.

이것은 에러가 있는 경우 프로그램의 흐름을 변경 가능함을 나타내기 위해 던지는 함수를 호출할 때 try 를 작성하는 것과 같습니다.

비동기 메서드 내에서 실행 흐름은 다른 비동기 메서드를 호출할 때만 일시 중단됩니다.

(중단은 암시적이거나 선점적이지 않습니다)

이것은 가능한 모든 중단 지점이 await 로 표시된다는 의미입니다.

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

예를 들어 이 코드는 갤러리에 모든 사진의 이름을 가져온 다음에 첫번째 사진을 보여줍니다.

listPhotos(inGallery:) 와 downloadPhoto(named:) 함수 모두 네트워크 요청을 필요로 하기 때문에 완료하는데 비교적 오랜시간이 걸릴 수 있습니다.

반환 화살표 전에 async 를 작성하여 둘 다 비동기로 만들면 이 코드는 그림이 준비될 때까지 기다리는 동안 앱의 나머지 코드가 계속 실행될 수 있습니다.

위 예제의 동시성을 이해하기 위한 실행 순서는 다음과 같습니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}

let photoNames = await listPhotos(inGallery: "Summer Vacation")
let sortedNames = photoNames.sorted()
let name = sortedNames[0]
let photo = await downloadPhoto(named: name)
show(photo)
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

1. 코드는 첫번째 줄에서 실행을 시작하고 첫번째 await 까지 실행됩니다. listPhotos(inGallery:) 함수를 호출하고 해당 함수가 반환될 때까지 실행을 일시 중단합니다.

2. 이 코드의 실행이 일시 중단되는 동안 같은 프로그램의 다른 동시 코드가 실행됩니다. 예를 들어 오랜시간 실행되는 백그라운드 작업이 새 사진을 목록을 업데이트 할 수 있습니다. 이 코드는 await 로 표시된 다음 중단 지점 또는 완료될 때까지 실행됩니다.

3. listPhotos(inGallery:) 가 반환된 후에 이 코드는 해당 지점에서 시작하여 계속 실행됩니다. 반환된 값을 photoNames 에 할당합니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}

let photoNames = await listPhotos(inGallery: "Summer Vacation")
let sortedNames = photoNames.sorted()
let name = sortedNames[0]
let photo = await downloadPhoto(named: name)
show(photo)
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

4 sortedNames 와 name 을 정의하는 라인은 일반적인 동기 코드 입니다. 이 라인은 await 로 표시되지 않았으므로 가능한 중단 지점이 없습니다.

5 다음 await 는 downloadPhoto(named:) 함수에 대한 호출을 표시합니다. 이 코드는 해당 함수가 반환될 때까지 실행을 다시 일시 중단하여 다른 동시 코드에 실행할 기회를 제공합니다.

6 downloadPhoto(named:) 가 반환된 후에 반환값은 photo 에 할당된 다음에 show(_:) 를 호출할 때 인수로 전달됩니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}

let photoNames = await listPhotos(inGallery: "Summer Vacation")
let sortedNames = photoNames.sorted()
let name = sortedNames[0]
let photo = await downloadPhoto(named: name)
show(photo)
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

await 로 표시된 코드의 중단이 가능한 지점은 비동기 함수 또는 메서드가 반환되기를 기다리는 동안 현재 코드 부분이 실행을 일시적으로 중단할 수 있음을 나타냅니다.

Swift 가 현재 쓰레드에서 코드의 실행을 일시 중단하고 대신 해당 쓰레드에서 다른 코드를 실행하기 때문에 이것을 쓰레드 양보 (yielding the thread) 라고도 부릅니다.

await 가 있는 코드는 실행을 일시 중단할 수 있어야 하므로 프로그램의 특정 위치에서만 비동기 함수 또는 메서드를 호출할 수 있습니다:

- 비동기 함수, 메서드 또는 프로퍼티의 바디에 있는 코드
- @main 으로 표시된 구조체, 클래스, 또는 열거형의 정적 (static) main() 메서드에 있는 코드
- 아래의 구조화되지 않은 동시성 (Unstructured Concurrency) 에 보이는 것처럼 구조화되지 않은 하위 작업의 코드

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

중단이 가능한 지점 사이의 코드는 다른 비동기 코드의 중단 가능성없이 순차적으로 실행됩니다.

예를 들어, 다음의 코드는 한 갤러리의 사진을 다른 곳으로 이동시킵니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}

let photoNames = await listPhotos(inGallery: "Summer Vacation")
let sortedNames = photoNames.sorted()
let name = sortedNames[0]
let photo = await downloadPhoto(named: name)
show(photo)

let firstPhoto = await listPhotos(inGallery: "Summer Vacation")[0]
add(firstPhoto toGallery: "Road Trip")
// At this point, firstPhoto is temporarily in both galleries.
remove(firstPhoto fromGallery: "Summer Vacation")
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

add(_:toGallery:) 와 remove(_:fromGallery:) 사이에 실행되는 다른 코드는 없습니다.

그 시간동안 첫번째 사진은 양쪽 갤러리에 모두 나타나며 앱의 불변성 중 하나를 임시적으로 위반합니다.

이 코드에 await 가 확실히 추가되지 말아야 한다는 것을 나타내기 위해 동기 함수로 코드를 리팩토링 할 수 있습니다.

이 예제에서 move(_:from:to:) 함수는 동기 함수기 때문에 중단 가능한 지점을 포함히지 않는다는 것을 보장할 수 있습니다.

이 함수에 중단 가능한 지점을 도입하기위해 비동기 코드를 추가하면 버그가 아닌 컴파일 에러가 발생합니다.

나의 플레이그라운드
My Playground

```
func move(_ photoName: String, from source: String, to destination:
 String) {
    add(photoName, to: destination)
    remove(photoName, from: source)
}
// ...
let firstPhoto = await listPhotos(inGallery: "Summer Vacation")[0]
move(firstPhoto, from: "Summer Vacation", to: "Road Trip")
```

내 코드 실행

비동기 함수 정의와 호출

Defining and Calling Asynchronous Functions

Task.sleep(until:tolerance:clock:) 메서드는 동시성 작동 방식을 배우기 위해 간단한 코드를 작성할 때 유용합니다.

이 메서드는 아무런 동작도 하지 않지만 반환되기 전에 주어진 나노 단위의 초만큼 기다립니다.

다음은 네트워크 작업 대기를 시뮬레이션하기 위해 sleep(until:tolerance:clock:) 을 사용하는 listPhoto(inGallery:) 함수의 버전입니다.

나의 플레이그라운드
My Playground

```
func listPhotos(inGallery name: String) async throws -> [String] {
    try await Task.sleep(until: .now + .seconds(2), clock:
     .continuous)
    return ["IMG001", "IMG99", "IMG0404"]
}
```

내 코드 실행

let var if for while func firstPhoto name photo photoNames sortedNames move(photoName: String, from: String, to:

비동기 시퀀스

Asynchronous Sequences

이전 섹션에서 listPhotos(inGallery:) 함수는 비동기적으로 배열의 모든 요소가 준비된 후에 전체 배열을 한번에 반환합니다.

또 다른 접근 방식은 비동기 시퀀스 (asynchronous sequence) 를 사용하여 한번에 콜렉션의 한 요소를 기다리는 것입니다.

비동기 시퀀스에 대한 조회 동작은 다음과 같습니다.

나의 플레이그라운드
My Playground

```
import Foundation

let handle = FileHandle.standardInput
for try await line in handle.bytes.lines {
    print(line)
}
```

내 코드 실행

let var if for while func handle

비동기 시퀀스

Asynchronous Sequences

일반적인 for-in 루프 대신에 위의 예제는 for 다음에 await 를 작성합니다.

비동기 함수 또는 메서드 호출할 때와 마찬가지로 await 작성은 가능한 중단 지점을 나타냅니다.

for-await-in 루프는 다음 요소를 사용할 수 있을 때까지 기다리고 각 반복이 시작될 때 잠재적으로 실행을 일시 중단합니다.

Sequence 프로토콜에 준수성을 추가하여 for-in 루프에서 자체 타입을 사용할 수 있는 것과 같은 방식으로 AsyncSequence 프로토콜에 준수성을 추가하여 for-await-in 루프에서 자체 타입을 사용할 수 있습니다.

나의 플레이그라운드
My Playground

```
import Foundation

let handle = FileHandle.standardInput
for try await line in handle.bytes.lines {
    print(line)
}
```

내 코드 실행

let var if for while func handle

비동기 함수 병렬로 호출

Calling Asynchronous Functions in Parallel

await 를 사용하여 비동기 함수를 호출하면 한번에 코드의 한 부분만 실행됩니다.

비동기 코드가 실행되는 동안 호출자는 코드의 다음 라인을 실행하기 위해 이동하기 전에 해당 코드가 완료될 때까지 기다립니다.

예를 들어 갤러리에서 처음 세 장의 사진을 가져오려면 다음과 같이 downloadPhoto(named:) 함수에 대한 세 번의 호출을 기다릴 수 있습니다.

나의 플레이그라운드
My Playground

```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

내 코드 실행

let var if for while func firstPhoto photos secondPhoto thirdPhoto

비동기 함수 병렬로 호출

Calling Asynchronous Functions in Parallel

이 방식에는 중요한 단점이 있습니다.

다운로드가 비동기이고 진행되는 동안 다른 작업을 수행할 수 있지만 downloadPhoto(named:) 에 대한 호출은 한 번에 하나만 실행됩니다.

각 사진은 다음 사진이 다운로드를 시작하기 전에 완료됩니다.

그러나 이런 작업을 기다릴 필요가 없습니다

각 사진은 개별적으로 또는 동시에 다운로드 할 수 있습니다.

나의 플레이그라운드
My Playground

```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

내 코드 실행

let var if for while func firstPhoto photos secondPhoto thirdPhoto

비동기 함수 병렬로 호출

Calling Asynchronous Functions in Parallel

비동기 함수를 호출하고 주변의 코드와 병렬로 실행하려면 상수를 정의할 때 let 앞에 async 를 작성하고 상수를 사용할 때마다 await 를 작성합니다.

나의 플레이그라운드
My Playground

```
• async let firstPhoto = downloadPhoto(named: photoNames[0])
• async let secondPhoto = downloadPhoto(named: photoNames[1])
• async let thirdPhoto = downloadPhoto(named: photoNames[2])

• let photos = await [firstPhoto, secondPhoto, thirdPhoto]
• show(photos)
```

내 코드 실행

비동기 함수 병렬로 호출

Calling Asynchronous Functions in Parallel

이 예제에서 downloadPhoto(named:) 을 호출하는 세가지는 모두 이전 호출이 완료되길 기다리지 않고 시작됩니다.

사용할 수 있는 시스템 자원이 충분하다면 동시에 실행할 수 있습니다.

코드가 함수의 결과를 기다리기 위해 일시 중단되지 않기 때문에 이러한 함수 호출 중 어느 것도 await 로 표시하지 않습니다.

대신 photos 가 정의된 라인까지 실행이 계속됩니다.

이 시점에서 프로그램은 이러한 비동기 호출의 결과를 필요로 하므로 세 장의 사진이 모두 다운로드 될 때까지 실행을 일시 중단하기 위해 await 를 작성합니다.

나의 플레이그라운드
My Playground

```
• async let firstPhoto = downloadPhoto(named: photoNames[0])
• async let secondPhoto = downloadPhoto(named: photoNames[1])
• async let thirdPhoto = downloadPhoto(named: photoNames[2])

• let photos = await [firstPhoto, secondPhoto, thirdPhoto]
• show(photos)
```

내 코드 실행

비동기 함수 병렬로 호출

Calling Asynchronous Functions in Parallel

다음은 이 두 접근 방식의 차이점에 대해 생각할 수 있는 방법입니다.

- 다음 줄의 코드가 해당 함수의 결과에 따라 달라지면 await 를 사용하여 비동기 함수를 호출합니다. 이것은 순차적으로 실행되는 작업을 생성합니다.
- 나중에 코드에서 결과가 필요하지 않을 때 async-let 을 사용하여 비동기 함수를 호출합니다. 이렇게 하면 병렬로 수행할 수 있는 작업이 생성됩니다.
- await 와 async-let 은 모두 일시 중단되는 동안 다른 코드를 실행할 수 있도록 합니다.
- 두 경우 모두 비동기 함수가 반환될 때까지 필요한 경우 실행이 일시 중단됨을 나타내기 위해 가능한 일시 중단 지점을 await 로 표시합니다.

동일한 코드에서 이 두 가지 접근 방식을 혼합할 수도 있습니다.

작업과 작업 그룹

Tasks and Task Groups

작업 (task) 은 프로그램의 일부로 비동기적으로 실행할 수 있는 작업 단위입니다.

모든 비동기 코드는 어떠한 작업의 일부로 실행됩니다.

이전 섹션에서 설명한 async-let 구문은 하위 작업을 생성합니다.

작업 그룹 (task group) 을 생성하고 해당 그룹에 하위 작업을 추가할 수도 있습니다.

그러면 우선순위와 취소를 더 잘 제어할 수 있으며 동적으로 작업의 수를 생성할 수 있습니다.

작업과 작업 그룹

Tasks and Task Groups

작업은 계층 구조로 정렬됩니다.

작업 그룹의 각 작업에는 동일한 상위 작업이 있으며 각 작업에는 하위 작업이 있을 수도 있습니다.

작업과 작업 그룹 간의 명시적 관계 때문에 이 접근 방식을 구조적 동시성 (structured concurrency) 이라고 합니다.

정확성에 대한 일부 책임을 지고 있지만 작업 간의 명시적 부모 (parent)-자식 (child) 관계를 통해 Swift 는 취소 전파 (propagating cancellation) 와 같은 일부 동작을 처리할 수 있고 Swift 는 컴파일 시간에 일부 오류를 감지할 수 있습니다.

작업 그룹에 대한 자세한 내용은 TaskGroup 을 참고 바랍니다.

나의 플레이그라운드
My Playground

```
• await withTaskGroup(of: Data.self) { taskGroup in
      let photoNames = await listPhotos(inGallery: "Summer Vacation")
      for name in photoNames {
          taskGroup.addTask { await downloadPhoto(named: name) }
      }
  }
```

내 코드 실행

.self

구조화되지 않은 동시성

Unstructured Concurrency

이전 섹션에서 설명한 동시성에 대한 구조화된 접근 방식 외에도 Swift 는 구조화되지 않은 동시성 (unstructured concurrency) 을 지원합니다.

작업 그룹의 일부인 작업과 달리 구조화되지 않은 작업 (unstructured task) 에는 상위 작업이 없습니다.

프로그램이 필요로 하는 방식으로 구조화되지 않은 작업을 관리할 수 있는 완전한 유연성이 있지만 정확성에 대한 완전한 책임도 있습니다.

현재 행위자 (actor) 에서 실행되는 구조화되지 않은 작업을 생성하려면 Task.init(priority:operation:) 초기화 구문을 호출해야 합니다.

더 구체적으로 분리된 작업으로 알려진 현재 행위자의 일부가 아닌 구조화되지 않은 작업을 생성하려면 Task.detached(priority:operation:) 클래스 메서드를 호출합니다.

이 모든 동작은 서로 상호작용 할 수 있는 작업 (task)을 반환합니다 (예를 들어 결과를 기다리거나 취소하는 경우가 해당됩니다.)

구조화되지 않은 동시성

Unstructured Concurrency

분리된 작업 (detached tasks) 관리에 대한 자세한 내용은 Task 를 참고 바랍니다.

나의 플레이그라운드
My Playground

```
let newPhoto = // ... some photo data ...
let handle = Task {
    return await add(newPhoto, toGalleryNamed: "Spring Adventures")
}
let result = await handle.value
```

내 코드 실행

작업 취소

Task Cancellation

Swift 동시성은 협동 취소 모델 (cooperative cancellation model) 을 사용합니다. 각 작업은 실행의 적절한 시점에서 취소되었는지 확인하고 적절한 방법으로 취소에 응답합니다. 수행 중인 작업에 따라 일반적으로 다음 중 하나를 의미합니다:

- CancellationError 와 같은 에러 발생
- nil 또는 빈 콜렉션 반환
- 부분적으로 완료된 작업 반환

취소를 확인하려면 작업이 취소된 경우 CancellationError 를 발생시키는 Task.checkCancellation() 을 호출하거나 Task.isCancelled 의 값을 확인하고 자체 코드에서 취소를 처리합니다.

예를 들어 갤러리에서 사진을 다운로드 하는 작업은 일부 다운로드를 삭제하고 네트워크 연결을 닫아야 할 수 있습니다.

취소를 수동으로 전파하려면 Task.cancel() 을 호출해야 합니다.

행위자

Actors

프로그램을 동시성 조각으로 분리하기위해 작업 (task) 을 사용할 수 있습니다.

작업은 서로 분리되어 있어 같은 시간 안전하게 실행될 수 있지만 작업 간에 일부 정보를 공유해야 할 수도 있습니다.

행위자 (Actors) 는 동시성 코드간에 정보를 안전하게 공유할 수 있게 해줍니다.

행위자

Actors

클래스와 마찬가지로 행위자 (actors) 는 참조 타입이므로 클래스는 참조 타입 (Classes Are Reference Types) 에서 값 타입과 참조 타입의 비교는 클래스 뿐만 아니라 행위자에도 적용됩니다.

클래스와 다르게 행위자는 한 번에 하나의 작업만 변경 가능한 상태에 접근할 수 있도록 허용하므로 여러 작업의 코드가 행위자의 동일한 인스턴스와 상호작용 하는 것은 안전합니다.

행위자

Actors

예를 들어 다음은 온도를 기록하는 행위자 입니다:

actor 키워드를 사용하여 행위자를 도입하고 중괄호로 정의합니다.

TemperatureLogger 행위자는 행위자 외부의 다른 코드가 접근할 수 있는 프로퍼티가 있으며 행위자 내부의 코드만 최대값을 업데이트 할 수 있게 max 프로퍼티를 제한합니다.

나의 플레이그라운드
My Playground

```
// 행위자 예제 코드
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}
```

내 코드 실행

행위자

Actors

구조체와 클래스와 같은 초기화 구문으로 행위자의 인스턴스를 생성합니다.

행위자의 프로퍼티 또는 메서드에 접근할 때 일시 중단 지점을 나타내기 위해 await 를 사용합니다.

예를 들어 이 예제에서 logger.max 에 접근하는 것은 일시 중단 지점으로 가능합니다.

행위자는 한 번에 하나의 작업만 변경 가능한 상태에 접근할 수 있도록 허용하므로 다른 작업의 코드가 이미 로거와 상호 작용하고 있는 경우 이 코드는 프로퍼티 접근을 기다리는 동안 일시 중단됩니다.

나의 플레이그라운드
My Playground

```
// 행위자 예제 코드
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}

let logger = TemperatureLogger(label: "Outdoors", measurement: 25)
print(await logger.max)
// Prints "25"
```

내 코드 실행

행위자

Actors

대조적으로 행위자의 부분인 코드는 행위자의 프로퍼티에 접근할 때 await 를 작성하지 않습니다.

예를 들어 새로운 온도로 TemperatureLogger 를 업데이트하는 메서드 입니다:

update(with:) 메서드는 행위자에서 이미 실행 중이므로 max 와 같은 프로퍼티에 대한 접근을 await 로 표시하지 않습니다.

이 메서드는 행위자가 변경 가능한 상태와 상호 작용하기 위해 한 번에 하나의 작업만 허용하는 이유 중 하나를 보여줍니다:

행위자의 상태에 대한 일부 업데이트는 일시적으로 불변성을 깨뜨립니다.

나의 플레이그라운드
My Playground

```
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}

extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}
```

내 코드 실행

행위자

Actors

TemperatureLogger 행위자는 온도 목록과 최대 온도를 추적하고 새로운 측정값을 기록할 때 최대 온도를 업데이트 합니다.

업데이트 도중에 새로운 측정값을 추가한 후 max 를 업데이트 하기 전에 온도 로거는 일시적으로 일치하지 않는 상태가 됩니다.

여러 작업이 동일한 인스턴스에 상호 작용하는 것을 방지하면 다음 이벤트 시퀀스와 같은 문제를 방지할 수 있습니다:

나의 플레이그라운드
My Playground

```
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}

extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}
```

내 코드 실행

행위자

Actors

1. 코드는 update(with:) 메서드를 호출합니다. 먼저 measurements 배열을 업데이트 합니다.

2. 코드에서 max 를 업데이트 하기 전에 다른 코드에서 최대값과 온도 배열을 읽습니다.

3. 코드는 max 를 변경하여 업데이트를 완료합니다.

나의 플레이그라운드
My Playground

```
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}

extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}
```

내 코드 실행

행위자

Actors

이러한 경우 다른 곳에서 실행 중인 코드는 데이터가 일시적으로 유효하지 않은 동안 update(with:) 호출 중간에 행위자에 대한 접근이 인터리브 (interleaved) 되어 잘못된 정보를 읽습니다.

Swift 행위자는 한 번에 해당 상태에 대해 하나의 작업만 허용하고 해당 코드는 await 가 일시 중단 지점으로 표시되는 위치에서만 중단될 수 있기 때문에 Swift 행위자를 사용하여 이 문제를 방지할 수 있습니다.

update(with:) 는 일시 중단 지점을 포함하지 않으므로 다른 코드는 업데이트 중간에 데이터에 접근할 수 없습니다.

나의 플레이그라운드
My Playground

```
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}

extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}
```

내 코드 실행

행위자

Actors

클래스의 인스턴스와 같이 행위자의 외부에서 프로퍼티에 접근하려고 하면 컴파일 때 에러가 발생합니다.

예를 들어, 이 코드는 await 작성 없이 logger.max 에 접근하는 것은 행위자의 프로퍼티가 해당 행위자의 분리된 로컬 상태의 부분이기 때문에 실패합니다.

Swift 는 행위자 내부의 코드만 행위자의 로컬 상태에 접근할 수 있도록 보장합니다. 이 보장을 행위자 분리 (actor isolation) 이라고 합니다.

나의 플레이그라운드
My Playground

```
extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}

print(logger.max)  // Error
```

내 코드 실행

전송 가능 타입

Sendable Types

작업 (Tasks) 와 행위자 (actors) 는 프로그램을 동시에 안전하게 실행할 수 있는 조각으로 나눌 수 있습니다.

작업 또는 행위자의 인스턴스 내에서 변수와 프로퍼티와 같은 변경 가능한 상태를 포함하는 프로그램의 일부분을 동시성 도메인 (concurrency domain) 이라고 부릅니다.

어떤 데이터는 데이터가 변경 가능한 상태를 포함하지만 동시 접근에 대해 보호되지 않으므로 동시성 도메인 간에 공유될 수 없습니다.

전송 가능 타입

Sendable Types

한 동시성 도메인에서 다른 동시성 도메인으로 공유될 수 있는 타입을 전송 가능 타입 (sendable type) 이라고 합니다.

예를 들어, 행위자 메서드로 호출될 때 인자로 전달되거나 작업의 결과로 반환될 수 있습니다.

이 챕터의 앞부분에 있는 예제들은 동시성 도메인 간에 전달되는 데이터는 항상 안전한 간단한 값 타입을 사용하기 때문에 전송 가능성에 대해 논의하지 않았습니다.

반대로 일부 타입은 동시성 도메인 간에 전달하기 위해 안전하지 않습니다.

예를 들어, 변경 가능한 프로퍼티를 포함하고 해당 프로퍼티에 순차적으로 접근하지 않는 클래스는 서로 다른 작업 클래스의 인스턴스에 전달될 때 예상할 수 없고 잘못된 결과를 생성할 수 있습니다.

전송 가능 타입

Sendable Types

Sendable 프로토콜을 선언하여 전송 가능한 타입으로 표시합니다. 이 프로토콜은 어떠한 코드 요구사항을 가지지 않지만 Swift 가 적용하는 의미론적 요구사항이 있습니다. 일반적으로 타입을 전송 가능한 것으로 나타내기 위한 세가지 방법이 있습니다:

- 타입은 값 타입이고 변경 가능한 상태는 다른 전송 가능한 데이터로 구성됩니다-예를 들어, 전송 가능한 저장된 프로퍼티가 있는 구조체 또는 전송 가능한 연관된 값이 있는 열거형이 있습니다.
- 타입은 변경 가능한 상태가 없으며 변경 불가능한 상태는 다른 전송 가능한 데이터로 구성됩니다-예를 들어, 읽기전용 프로퍼티만 있는 구조체 또는 클래스가 있습니다.
- 타입은 @MainActor 로 표시된 클래스나 특정 쓰레드나 큐에서 프로퍼티에 순차적으로 접근하는 클래스와 같이 변경 가능한 상태의 안정성을 보장하는 코드를 가지고 있습니다.

의미론적 요구사항의 자세한 리스트는 전송 가능 (Sendable) 프로토콜을 참고 바랍니다.

전송 가능 타입

Sendable Types

전송 가능한 프로퍼티만 가지는 구조체와 전송 가능한 연관된 값만 가지는 열거형과 같이 어떠한 타입은 항상 전송 가능합니다.

예를 들어, TemperatureReading 은 전송 가능한 프로퍼티만 가지는 구조체이며 public 또는 @usableFromInline 으로 표시되지 않은 구조체이므로 암시적으로 전송 가능합니다.

나의 플레이그라운드
My Playground

```
struct TemperatureReading: Sendable {
    var measurement: Int
}

extension TemperatureLogger {
    func addReading(from reading: TemperatureReading) {
        measurements.append(reading.measurement)
    }
}

let logger = TemperatureLogger(label: "Tea kettle", measurement: 85)
let reading = TemperatureReading(measurement: 45)
await logger.addReading(from: reading)
```

내 코드 실행

전송 가능 타입

Sendable Types

다음은 Sendable 프로토콜을 준수가 암시되는 구조체입니다:

나의 플레이그라운드
My Playground

```
struct TemperatureReading {
    var measurement: Int
}
```

내 코드 실행

그박의 내용들

공식 문서에서 확인합시다

중첩된 타입

불투명한 타입

자동 참조 카운팅

일급함수

mutating

감사합니다