SWIFT GRAMMAR SUMMARY

Swift
요약집

CONTENTS

PART 01

Swift 기본

Swift 요약집

Swift에 대하여

- Swift는 2014년 애플이 발표한 신규 프로그래밍 언어입니다. 이전에는 iOS 앱 개발에 Objective-C를 사용하였으며, 일부 회사나 오래된 라이브러리의 경우 Objective-C 코드가 일부 남아있는 것을 확인할 수 있습니다.

  Swift 프로그래밍 언어는 Apple에서 "Objective without C" 라고 설명했듯이, C 언어 라이브러리 및 프레임워크와 호환되지만, C언어를 기반으로 하지 않습니다. 따라서 배우고 이해하기 매우 쉽습니다.

Swift의 대표적인 특징

- **가독성 및 유지 관리**

  Objective-C 보다 Swift 언어가 선호되는 가장 중요한 이유 중에 하나입니다. 초보자도 코드를 배우고 이해하고 작성하기가 매우 간단하고 쉽습니다. 코드가 적기 때문에 Swift 언어는 간결하고 광범위하게 재사용할 수 있습니다.

- **빠른 앱 개발**

  Swift는 이름 그대로, Swift 언어는 Objective-C 또는 Python과 같은 동시대의 언어에 비해 매우 빠릅니다. Swift의 성능은 산술 계산에서 가장 빠른 C++의 성능과 가장 비슷합니다.

- **Objective-C와의 호환성**

  Swift는 C 또는 Objective-C와 관련이 없지만, Objective-C의 라이브러리 및 프레임워크와 호환됩니다. Swift를 사용하면서 Objective-C 프레임워크를 가져와 사용할 수 있습니다.

- **안전하고 오류가 적다**

  Swift는 컴파일 자체에서 오류를 발생시켜서 개발자가 즉시 수정할 수 있습니다. 개발자가 코드 작업과 동시에 버그를 작성, 컴파일 및 수정하여 더 나은 품질 관리를 구현하도록 도와줍니다. 코드 작업 자체에서 오류가 수정되고 테스트되기 때문에 앱이 훨씬 빠르고 효율적이므로 안전과 보안이 보장됩니다.

- **여러 장치 지원**

  Swift는 iOS 플랫폼 개발에만 국한되지 않습니다. 크로스 플랫폼 언어인 Swift는 Windows 및 Linux 운영 체제에서도 사용할 수 있습니다.

주석 (Comment)

- 주석은 프로그램 소스 코드에 정보를 남기는 목적으로 사용합니다. 주로 코드를 다시 봤을 때 필요한 중요 메모나 다른 개발자에게 설명하기 위한 메모등을 주석으로 남깁니다.
- Swift 에서의 한 줄 주석은 슬래시 (/) 두 개를 사용하여 나타내고, 두줄 주석은 슬래시와 별표를 사용하여 나타냅니다.

```
// 한 줄 주석입니다.

/* 여러 줄 주석입니다.
   여러 줄 주석입니다.
 */

/* 한 줄 주석도 가능합니다.*/
```

키워드

- 다음 키워드들은 Swift 에서 미리 정한 예약어이기 때문에, 변수나 타입의 이름으로 사용할 수 없습니다.

선언에 사용되는 키워드

Class	deinit	Enum	extension
Func	import	Init	internal
Let	operator	private	protocol
public	static	struct	subscript
typealias	var		

statements 에 사용되는 키워드

break	case	continue	default
do	else	fallthrough	for

if	in	return	switch
where	while		

● 표현식이나 타입에 사용되는 키워드

as	dynamicType	false	is
nil	self	Self	super
true	_COLUMN_	_FILE_	_FUNCTION_
LINE			

● 특정 문맥에 사용되는 키워드

associativity	convenience	dynamic	didSet
final	get	infix	inout
lazy	left	mutating	none
nonmutating	optional	override	postfix
precedence	prefix	Protocol	required
right	set	Type	unowned
weak	willSet		

PART 02

기본 연산자 Basic Operators

Swift에서는 산술 연산자(+, -, /, %)와 논리 연산자(&&, ||), 그리고 범위 연산자(a..<b, a..b)를 사용할 수 있습니다.

할당 연산자 (=)

- 할당 연산자는 상수나 변수의 값을 초기화 시키거나 변경합니다.

```
let b = 10
var a = 5
a = b
// a 값은 10
```

사칙 연산자 (+, -, *, /)

```
1 + 2       // 3
5 - 3       // 2
2 * 3       // 6
10.0 / 2.5  // 4.0
```

나머지 연산자 (%)

```
9 % 4       // 1
-9 % 4      // -1
```

합성 할당 연산자 (+=, -=)

- a = a + 2와 같이 할당 연산(=)과 덧셈 연산(+) 이나 뺄셈 연산(-)을 합성해 +=, -= 형태로 축약해서 사용이 가능합니다.

```
var a = 1
a + = 2
// a는 3
```

비교 연산자 (==, !=, >, <, >=, <=)

- 같다 (a == b)
- 같지 않다 (a != b)
- 크다 (a > b)
- 작다 (a < b)
- 크거나 같다 (a >= b)
- 작거나 같다 (a <= b)

```
var a = 1
a + = 2
// a는 3
```

삼항 조건 연산자

- 삼항 조건 연산자는 question ? answer1 : answer2 의 구조를 갖습니다. 그래서 question의 조건이 참인 경우, answer1이 실행되고, 거짓인 경우에는 answer2가 실행됩니다.

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight는 90 (40 + 50)
```

nil 병합 연산자

- nil 병합 연산자는 a ?? b 형태를 갖는 연산자입니다.
- 옵셔널 변수인 a를 unwrapping 하여 만약 a가 nil 인 경우에 b를 반환합니다.

```
let defaultColorName = "red"
var userDefinedColorName: String?   // 이 값은 defaults 값 nil입니다.

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorNam이 nil이므로 colorNameToUse 값은
defaultColorName인 "red"가 설정 됩니다.
```

논리 연산자 (&&, || ,!)

- && 연산자는 논리 AND 연산자라고 합니다. 두 피연산자가 모두 0이 아니면 조건이 참이 됩니다.
- || 연산자는 논리 OR 연산자라고 합니다. 두 피연산자 중 하나가 0이 아니면 조건이 참이 됩니다.
- ! 연산자는 논리 NOT 연산자라고 합니다. 피연산자의 논리 상태를 반전하는 데에 사용합니다. 조건이 참이면 논리 NOT 연산자를 거짓으로 만듭니다.

연산자의 우선 순위

- 연산자 우선 순위는 식에서 용어 그룹화를 결정합니다. 이는식이 평가되는 방식에 영향을줍니다. 특정 연산자는 다른 연산자보다 우선 순위가 높습니다. 예를 들어 곱셈 연산자는 더하기 연산자보다 우선 순위가 높습니다.

  예를 들어, x = 7 + 3 * 2; 여기서 x는 연산자 *가 +보다 우선 순위가 높기 때문에 20이 아닌 13이 할당됩니다. 따라서 먼저 3 * 2를 곱한 다음 7에 더합니다.

  여기에서 우선 순위가 가장 높은 연산자는 테이블 맨 위에 표시되고 가장 낮은 연산자는 맨 아래에 표시됩니다. 식 내에서 우선 순위가 높은 연산자가 먼저 평가됩니다.

PART 03

문자열 String

문자열은 “Hello, World!”와 같이 정렬된 문자들의 모음입니다. 문자열은 Swift에 String 타입으로 나타낼 수 있습니다.

문자열은 큰 따옴표(“)로 묶어 표현합니다.

여러 줄의 문자열은 큰 따옴표 3개(“””)로 묶어서 표현할 수 있습니다.

```
let something = "Some string literal value"

let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

빈 문자열

- 빈 문자열 리터럴을 사용하거나 아래와 같이 String 클래스의 인스턴스를 생성하여 빈 문자열을 생성 할 수 있습니다. Boolean 속성 isEmpty를 사용하여 문자열이 비어 있는지 여부를 확인할 수도 있습니다.

```
// Empty string creation using String literal
var stringA = ""

if stringA.isEmpty {
  print( "stringA is empty" )
```

```
} else {
  print( "stringA is not empty" )
}

// Empty string creation using String instance
let stringB = String()

if stringB.isEmpty {
  print( "stringB is empty" )
} else {
  print( "stringB is not empty" )
}
```

문자열 상수

- 문자열을 변수에 할당하여 수정 (또는 변경) 할 수 있는지 여부를 지정할 수 있습니다. 또는 아래와 같이 let 키워드를 사용하여 상수에 할당하여 상수가 될 수 있습니다.

```
// stringA 변수는 수정할 수 있습니다.
var stringA = "Hello, Swift!"
stringA + = "--Readers--"
print( stringA )

// stringB 변수는 수정할 수 없습니다.
let stringB = String("Hello, Swift!")
stringB + = "--Readers--"
print( stringB )
```

문자열 삽입

- 문자열 보간은 문자열 리터럴 내에 값을 포함하여 상수, 변수, 리터럴 및 표현식의 혼합에서 새 문자열 값을 구성하는 방법입니다.
  문자열 리터럴에 삽입하는 각 항목 (변수 또는 상수)은 백 슬래시 접두사가 붙은 괄호 쌍으로 둘러싸여 있습니다.

```swift
var varA = 20
let constA = 100
var varC: Float = 20.0

var stringA = "\(varA) times \(constA) is equal to \(varC * 100)"
print(stringA)
// 20 times 100 is equal to 2000.0
```

문자열 연결

- + 연산자를 사용하여 두 문자열 또는 문자열과 문자 또는 두 문자를 연결할 수 있습니다.

```swift
let a = "Hello,"
let b = "World!"

var string = a + b
print(string) // Hello,World!
```

문자열 길이

- Swift 문자열에는 길이 속성이 없지만 전역 count () 함수를 사용하여 문자열의 문자 수를 계산할 수 있습니다.

```swift
let a = "Hello, Swift!"

print("\(a), length is \((a.count))") // Hello, Swift!, length is 13
```

문자열 비교

- == 연산자를 사용하여 두 문자열 변수 또는 상수를 비교할 수 있습니다.

```swift
let a = "Hello, Swift!"
let b = "Hello, World!"

if a == b {
    print("\(a) and \(b) are equal")
} else {
    print("\(a) and \(b) are not equal")
}
// Hello, Swift! and Hello, World! are not equal
```

문자열 반복

- 문자열도 배열이므로, for 문을 사용하여 문자열을 반복할 수 있습니다.

```swift
for chars in "ThisString" {
  print(chars, terminator: " ")
}

// T h i s S t r i n g
```

문자열 함수 및 연산자

함수/속성 및 연산자	설명
isEmpty	문자열이 비어 있는지 여부를 결정하는 Bool 값입니다
hasPrefix(prefix: String)	**주어진 매개 변수 문자열이 문자열의 접두어로 존재하는지 여부를 확인하는 함수입니다.**
hasSuffix(suffix: String)	주어진 매개 변수 문자열이 문자열의 접미사로 존재하는지 여부를 확인하는 함수 입니다.
toInt()	**숫자 문자열 값을 정수로 변환하는 함수입니다.**
count()	**문자열의 문자 수를 계산하는 전역 함수입니다.**

utf8	문자열의 UTF-8 표현을 반환하는 속성입니다.
utf16	문자열의 UTF-16 표현을 반환하는 속성입니다.
+	두 문자열, 문자열과 문자 또는 두 문자를 연결하는 연산자입니다.
+=	기존 문자열에 문자열 또는 문자를 추가하는 연산자입니다.
==	두 문자열의 동일성을 결정하는 연산자입니다.
<, >	한 문자열이 다른 문자열보다 작거나 큰 것으로 비교되는지 확인하기 위해 사전 식 비교를 수행하는 연산자입니다.
startIndex	문자열의 시작 인덱스에서 값을 가져옵니다.
endIndex	문자열의 끝 인덱스에서 값을 가져옵니다.
insert("Value", at: position)	위치에 값을 삽입합니다.
remove(at: position) removeSubrange(range)	위치에서 값을 제거하거나 문자열에서 값 범위를 제거합니다.
reversed()	문자열의 역을 반환합니다.

PART 04

컬렉션 타입 Collection Types

Swift 요약집

배열 (Array)

- Swift에서의 배열은 동일한 값의 순서가 지정된 값의 목록을 저장하는 데에 사용할 수 있습니다. Swift는 실수로도 잘못된 타입의 값을 배열에 입력할 수 없도록 엄격하게 검사합니다.
- 생성된 배열을 변수(var)에 할당하면 항상 변경이 가능합니다. 즉, 항목을 추가하거나 제거 또는 수정하여 변경할 수 있습니다. 그러나 배열을 상수(let)에 할당하면 해당 배열은 변경할 수 없으며 크기와 내용 또한 변경할 수 없습니다.

배열 생성하기

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight는 90 (40 + 50)
```

배열 접근하기

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight는 90 (40 + 50)
```

배열 순회하기

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight는 90 (40 + 50)
```

셋 (Set)

- 셋은 동일한 타입의 고유한 값을 저장하는 데 사용되지만, 배열과 다르게 명확한 순서가 없습니다. 항목의 순서가 중요하지 않거나 중복 값이 없는지 확인하려는 경우에는 배열 대신 셋을 이용할 수 있습니다. (셋은 고유한 값만 허용합니다.)
- Set 형태로 저장되기 위해서는 반드시 타입이 hashable 이어야만 합니다. Swift에서는 String, Int, Double, Bool 같은 기본 타입은 기본적으로 hashable 입니다.

Set 생성하기

```
var someSet = Set<Character>()
```

Set 접근 및 수정하기

- count는 셋의 항목의 개수를 표현하는 데에 사용할 수 있습니다.

```
someSet.count
```

- insert 함수를 사용하여 셋에 값을 삽입할 수 있습니다.

```
someSet.insert("c")
```

- 배열과 마찬가지로, isEmpty 를 사용하여 셋이 비어있는지 확인할 수 있습니다.

```
someSet.isEmpty
```

- remove 함수를 이용하여 셋에서 값을 제거할 수 있습니다.

```
someSet.remove("c")
```

- contains 함수를 사용하여 셋에 값이 있는지 확인할 수 있습니다.

```
someSet.contains("c")
```

딕셔너리 (Dictionary)

- 딕셔너리는 정렬되지 않은 동일한 타입의 값의 목록을 저장하는 데에 사용할 수 있습니다.
- 딕셔너리는 Key 라는 고유 식별자를 사용합니다. 동일한 Key를 통해 값을 참조하고, 조회할 수 있습니다. Key는 정수나 문자열이 될 수 있으며 딕셔너리 내에서 고유한 값이어야 합니다.
- 딕셔너리를 변수(var)에 할당하면 항상 변경 가능하며, 상수(let)에 할당하면 변경할 수 없습니다.

Dictionary 생성하기

```
var namesOfIntegers = [Int: String]()
```

PART 05

제어문 Collection Types

Swift 요약집

for-In 문

- for-in 문은 배열을 순서대로 순회하기 위해 사용합니다.

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

- 범위 연산자와 함께 사용할 수 있습니다.

```
let minutes = 60
for tickMark in 0..< minutes {
    // render the tick mark each minute (60 times)
}
```

While 문

- While 문은 조건이 거짓일 때까지 구문을 반복합니다.

```
while condition {
    statements
}
```

Repeat-While 문

- repeat-while 문은 repeat 안에 있는 구문을 최소 한 번 이상 실행하고, while 조건이 거짓일 때까지 반복합니다.

```
repeat {
   statements
} while condition
```

If 문

- if 문은 어떠한 값이 특정 조건에 해당 할 때만 선택적으로 코드를 실행합니다.

```
if condition {
   statements 1
} else {
   statements 2
}
```

Switch 문

- break이 필수적이지 않지만, case 안에 특정 지점에서 멈추도록 하기 위해 break 을 사용할 수 있습니다.
- case 안에는 최소 하나의 실행 구문이 반드시 있어야 합니다.

```
switch some value to consider {
case value 1:
   respond to value 1
case value 2,
    value 3:
   respond to value 2 or 3
default:
   otherwise, do something else
}
```

Guard 문

- Guard 문은 원하는 조건이 충족되지 않으면, 불필요한 코드는 실행하지 않고, 일찍 종료할 수 있습니다.

```
guard condition else {
    statements
}
```

이용가능한 API 버전 확인

- Swift 에서는 기본으로 특정 플랫폼 (iOS, macOS, tvOS, watchOS)과 특정 버전을 확인하는 구문을 제공해줍니다. 이 구문을 활용해 특정 플랫폼과 버전을 사용하는 기기에 대한 처리를 따로 할 수 있습니다.

```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

PART 06

함수 Function

Swift 요약집

함수 정의하기

- 함수는 특정 작업을 수행하기 위해 함께 구성된 일련의 명령문입니다.
- 함수를 선언할 때는 가장 앞에 func 키워드를 붙이고 (person: String) 파라미터와 형 그리고 -> String 형태로 반환형을 정의합니다.

```
func greet(person: String) -> String {
 let greeting = "Hello, " + person + "!"
 return greeting
}
```

함수 호출하기

- 예를 들어 함수 이름이 'display'인 함수가 정수 데이터 유형을 보유하는 인수 'no1'로 초기화 된 숫자를 표시하기 위해 'display'라는 함수를 정의했다고 가정 해 보겠습니다. 그런 다음 'no1'인수가 'a'인수에 할당되며 이후 동일한 데이터 유형 정수를 가리 킵니다. 이제 인수 'a'가 함수에 반환됩니다. 여기서 display () 함수는 정수 값을 보유하고 함수가 호출 될 때마다 정수 값을 반환합니다.

```
func display(no1: Int) -> Int {
  let a = no1
  return a
}

print(display(no1: 100)) // 100
print(display(no1: 200)) // 200
```

함수 파라미터와 반환 값

- 파라미터가 없는 함수도 있습니다. 다음은 파라미터가 없는 함수의 예시입니다.

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

- 복수의 파라미터를 사용하는 함수

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

- **반환 있는 함수**

  함수는 문자열, 정수 및 부동 데이터 타입 값을 반환 유형으로 반환하는데도 사용됩니다. 주어진 배열에서 가장 큰 숫자와 가장 작은 숫자를 찾기 위해 함수 'ls'는 크고 작은 정수 데이터 유형으로 선언됩니다.

  배열은 정수 값을 보유하도록 초기화됩니다. 그런 다음 배열이 처리되고 배열의 모든 값을 읽고 이전 값과 비교합니다. 값이 이전 값보다 작을 경우 'small'인수에 저장되고, 그렇지 않으면 'arge' 인수에 저장되고 함수를 호출하여 값이 반환됩니다.

```swift
func ls(array: [Int]) -> (large: Int, small: Int) {
  var lar = array[0]
  var sma = array[0]

  for i in array[1..<array.count] {
    if i < sma {
      sma = i
    } else if i > lar {
      lar = i
    }
  }
  return (lar, sma)
}

let num = ls(array: [40,12,-5,78,98])
print("Largest number is: \(num.large) and smallest number is: \(num.small)")
// Largest number is: 98 and smallest number is: -5
```

반환 값이 없는 함수

일부 함수에는 반환 값없이 함수 내부에 선언 된 인수가있을 수 있습니다. 다음 프로그램은 sum () 함수에 대한 인수로 a 와 b 를 선언 합니다. 함수 자체 내에서 인수 a 및 b 의 값 은 함수 호출 sum ()을 호출하여 전달되고 해당 값이 인쇄되어 반환 값을 제거합니다.

```swift
func sum(a: Int, b: Int) {
  let a = a + b
  let b = a - b
  print(a, b)
}

sum(a: 20, b: 10) // 30 20
sum(a: 40, b: 10) // 50 40
sum(a: 24, b: 6) // 30 24
```

함수 인자 라벨과 파라미터 이름

- 함수 호출시 적절한 파라미터 이름을 지정해 함수 내부와 함수 호출시 사용할 수 있습니다.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // 함수 내부에서 firstParameterName와 secondParameterName의 인자를 사용합니다.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

인자 라벨 지정

- 파라미터 앞에 인자 라벨을 지정해 실제 함수 내부에서 해당 인자를 식별하기 위한 이름과 함수 호출시 사용하는 이름을 다르게 해서 사용할 수 있습니다.

```
func someFunction(argumentLabel parameterName: Int) {
    // 함수 안에서 parameterName로 argumentLabel의 인자값을 참조할 수 있습니다.
}
```

인자 생략

- 파라미터 앞에 _ 을 붙여 함수 호출시 인자 값을 생략할 수 있습니다.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // 함수 안에서 firstParameterName, secondParameterName
    // 인자로 입력받은 첫번째, 두번째 값을 참조합니다.
}
someFunction(1, secondParameterName: 2)
```

기본 파라미터 값

- 함수의 파라미터 값에 기본 값을 설정할 수 있습니다. 기본 값이 설정되어 있는 파라미터는 함수 호출시 생략할 수 있습니다. 기본 값을 사용하지 않는 파라미터를 앞에 위치 시켜야 함수를 의미 있게 사용하기 쉽습니다.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
   // 함수 호출시 두번째 인자를 생략하면 함수안에서
   // parameterWithDefault값은 12가 기본 값으로 사용됩니다.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //
parameterWithDefault는 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault는 12
```

인-아웃 파라미터 (In-Out Parameters)

- 기본적으로 함수는 매개 변수를 '상수'로 간주하는 반면 사용자는 함수에 대한 인수를 변수로 선언 할 수도 있습니다. 상수 매개 변수를 선언하기 위해 'let'키워드를 사용하고 'var'키워드로 변수 매개 변수를 정의하는 것에 대해 이미 논의했습니다.

  Swift의 인-아웃 파라미터는 함수 호출 후 값이 수정 되더라도 매개 변수 값을 유지하는 기능을 제공합니다. 함수 매개 변수 정의 시작 부분에 'inout'키워드가 선언되어 멤버 값을 유지합니다.

  값이 함수에 'in'으로 전달되고 해당 값이 함수 본문에 의해 액세스 및 수정되고 원래 인수를 수정하기 위해 함수의 'out'으로 반환되기 때문에 키워드 'inout'이 파생됩니다.

  변수는 값만 함수 내부와 외부에서 수정되기 때문에 in-out 매개 변수에 대한 인수로만 전달됩니다. 따라서 문자열과 리터럴을 in-out 매개 변수로 선언 할 필요가 없습니다. 변수 이름 앞의 '&'는 인수를 in-out 매개 변수로 전달하고 있음을 나타냅니다.

```
func temp(a1: inout Int, b1: inout Int) {
  let t = a1
  a1 = b1
  b1 = t
}
```

```
var no = 2
var co = 10
temp(a1: &no, b1: &co)
print("Swapped values are \(no), \(co)")
// Swapped values are 10, 2
```

중첩 함수

- 중첩 함수는 내부 함수를 호출하여 외부 함수를 호출하는 기능을 제공합니다.

```
func calcDecrement(forDecrement total: Int) -> () -> Int {
  var overallDecrement = 0
  func decrementer() -> Int {
    overallDecrement -= total
    return overallDecrement
  }
  return decrementer
}

let decrem = calcDecrement(forDecrement: 30)
print(decrem()) // -30
```

PART 07

클로저 Closures

클로저

- 클로저는 중괄호({})로 감싸진 실행 가능한 코드 블럭입니다. Swift의 클로저는 Objective-C의 블럭(blocks)과 다른 프로그래밍 언어의 람다(lamdas)와 비슷한 개념입니다.
  클로저는 어떠한 상수나 변수의 참조를 캡쳐(capture)해서 저장할 수 있습니다. Swift는 이 캡쳐와 관련한 모든 메모리를 알아서 처리합니다.
  파라미터를 받을 수 있고, 반환 값이 존재할 수 있다는 점에서 함수와 동일합니다.

- Swift 클로저는 최적화를 통해 깔끔하고 명확한 스타일을 가지고 있습니다. 최적화는 다음과 같습니다.
  - 문맥(context)에서 인자 타입과 반환 타입의 추론
  - 단일 표현식 클로저의 암시적 반환
  - 축약된 인자 이름
  - 후위 클로저 문법

클로저 문법

- 클로저 표현 문법은 일반적으로 아래의 형태를 가지고 있습니다.
- 인자로 넣을 parameters 가 있고, 인자 값으로 처리할 내용을 기술하는 statements , 그리고 return type 입니다.

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

PART 08

옵셔널 Optional

Swift 요약집

옵셔널 타입

- Swift에서는 Optional은 변수의 값이 nil 일 수도 있다라는 것을 표현합니다. 반대로 옵셔널이 아니라면 해당 값은 절대 nil 이 될 수 없음을 의미합니다. Swift 에서는 옵셔널은 말 그대로 선택적이며, 기본 값은 non-Optional 입니다.

Int형 Optional 변수 선언하기

```
var perhapsInt: Int?
```

String형 Optional 변수 선언하기

```
var perhapsStr: String?

// 위의 코드는 nil로 명시적으로 초기화한 것과 동일합니다.
var perhapsStr: String? = nil
```

강제 언래핑

- 변수를 optional 로 정의했다면 이 변수에서 값을 가져오려면 래핑 을 해제 해야 합니다. 이것은 변수 끝에 느낌표를 넣는 것을 의미합니다.

```
var optionalString: String? = "안녕하세요."

print(optionalString) // Optional("안녕하세요.")
print(optionalString!) // 안녕하세요.
```

자동 언래핑

- 물음표 대신 느낌표를 사용하여 선택적 변수를 선언할 수 있습니다. 이러한 선택적 변수는 자동으로 해제되며 할당된 값을 가져오기 위해 변수 끝에 추가 느낌표를 사용할 필요가 없습니다.

```
var optionalString: String? = "안녕하세요."

print(optionalString) // Optional("안녕하세요.")
print(optionalString!) // 안녕하세요.
```

선택적 바인딩

- 물음표 대신 느낌표를 사용하여 선택적 변수를 선언할 수 있습니다. 이러한 선택적 변수는 자동으로 해제되며 할당된 값을 가져오기 위해 변수 끝에 추가 느낌표를 사용할 필요가 없습니다.

```
if let constantName = someOptional {
  statements
}
```

PART 09

열거형 Enumerations

Swift 요약집

열거형 (Enumerations)

- 열거형은 관련된 값으로 이루어진 그룹을 공통의 형으로(type) 선언해 형 안전성(type-safety)을 보장하는 방법으로 코드를 다룰 수 있게 해줍니다. C나 Objective-C가 Integer값 들로 열거형을 구성한 것에 반해 Swift에서는 case값이 string, character, integer, floting 값들을 사용할 수 있습니다. 열거형은 1급 클래스 형(first-class types)이어서 계산된 프로퍼티(computed properties)를 제공하거나 초기화를 지정하거나, 초기 선언을 확장해 사용할 수 있습니다.

열거형 문법

- enum 키워드를 사용하여 열거형을 정의합니다.

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

- 다음과 같이 요일을 case로 갖는 열거형을 정의할 수 있습니다.

```
enum DaysofaWeek {
    case sunday
    case monday
    case tuesDay
    case wednesDay
    case thursDay
    case friDay
    case saturday
}
```

- 여러 case를 콤마(,)로 구분하여 한줄에 적을 수도 있습니다.

```
enum DaysofaWeek {
    case sunday, monday, tuesDay, wednesDay, thursDay, friDay, saturday
}
```

- 열거형은 case 기본 값을 할당하지 않습니다. 열거형의 이름은 다른 타입들과 마찬가지로 대문자로 시작해야 합니다. 열거형으로 정의된 변수가 해당 열거형 타입으로 한 번 정의가 되면, 다음에 값을 할당할 때 형을 생략한 점 문법(dot syntax)를 이용해 값을 할당하는 축약형 문법을 사용할 수 있습니다.

```
var weekDay = DaysofaWeek.sunday
weekDay = .saturday
```

Switch 구문에서 열거형 값 매칭하기

- 각 열거형 값을 Switch 문에서 매칭할 수 있습니다. switch 문은 반드시 열거형의 모든 case를 완전히 포함해야 합니다. 만약 모든 case 처리를 하지 않아도 된다면, switch 구문의 default case를 이용합니다.

```
switch weekDay {
case .sunday:
    print("Today is Sunday")
case .monday:
    print("Today is Monday")
case .tuesDay:
    print("Today is TuesDay")
default:
    print("What day of the week is it?")
}
```

관련 값 (Associated Values)

- 열거형의 각 case에는 custom type의 추가적인 정보를 저장할 수 있습니다.

```
enum Student {
   case Name(String)
   case Mark(Int,Int,Int)
}

var studDetails = Student.Name("Swift")
var studMarks = Student.Mark(98,97,95)

switch studMarks {
case .Name(let studName):
   print("Student name is: \(studName).")
case .Mark(let Mark1, let Mark2, let Mark3):
   print("Student Marks are: \(Mark1),\(Mark2),\(Mark3).")
}

// Student Marks are: 98,97,95.
```

Raw 값 (Raw Values)

- Raw 값은 String, Character, Integer, Float 등의 타입을 사용할 수 있습니다. 각 Raw 값은 열거형 내에서 고유한 값을 가져야 합니다.
- Raw 값은 rawValue 를 사용하여 접근할 수 있습니다.
- Raw 값을 Integer 타입으로 사용한다면, 일부 case에 값이 지정되지 않아도 자동으로 값을 증가시킵니다.

```
enum Month: Int {
   case January = 1, February, March, April, May, June, July, August,
September, October, November, December
}

let yearMonth = Month.May.rawValue
print("Value of the Month is: \(yearMonth).")
// Value of the Month is: 5.
```

- Raw 값을 String 타입으로 사용한다면, case에 값이 지정되지 않아도 case 의 텍스트가 자동으로 raw 값으로 할당될 수 있습니다.

```
enum Month: String {
    case January, February, March, April, May, June, July, August, September, October,
November, December
}

let yearMonth = Month.May.rawValue
print("Value of the Month is: \(yearMonth).")
// Value of the Month is: May.
```

PART 10

클래스와 구조체 Class and Structure

Swift 요약집

클래스와 구조체

- 구조체와 클래스는 프로그램의 코드를 추상화 하기 위해 사용됩니다. 구조체와 클래스는 프로퍼티와 메소드를 사용하여 구조화된 데이터와 기능을 가질 수 있습니다. 새로운 사용자 정의 데이터 타입을 만드는 것입니다.

클래스와 구조체의 공통점

- 프로퍼티와 메소드를 사용하여 구조화된 데이터와 기능을 가집니다.
- 새로운 사용자 정의 타입을 만들어 줍니다.
- 초기화(init)을 정의하여 초기 상태를 설정할 수 있습니다.
- 확장(extension)할 수 있습니다.
- 프로토콜(protocol) 사용이 가능합니다.
- 특정 값에 접근할 수 있는 서브스크립트(subscript) 문법이 사용이 가능합니다.
- 프로퍼티 값에 접근하고 싶다면, 점(dot) 문법을 사용합니다.

클래스와 구조체의 차이점

- **클래스**
  - 참조 타입 (Reference Type) 입니다. 참조 타입이란, 변수나 상수에 값을 할당하거나 함수에 인자로 전달될 때 그 값이 복사되지 않고 참조된다는 의미입니다.
  - Call by reference : 할당 또는 파라미터 전달 시에 객체를 가리키고 있는 메모리의 주소 값이 복사됩니다.
  - Heap Memory 영역에 할당됩니다. 따라서 속도가 느립니다.
  - 런타임에 직접 allocate 하며 reference counting 을 통해 deallocate가 필요합니다.
  - 상속이 가능합니다.
  - 런타임에 타입캐스팅을 통해 클래스 인스턴스에 따라 여러 동작이 가능합니다.
  - deinitializer 가 존재합니다.

- **구조체**
  - 값 타입 (Value Type) 입니다. 값 타입이란, 함수에서 상수나 변수로 전달될 때 그 값이 복사되서 전달된다는 의미입니다.
  - Call by value : 할당 또는 파라미터 전달 시 value copy 가 일어납니다.
  - 값 타입이기 때문에 멀티 스레드 환경에서 공유 변수로 인한 문제를 일으킬 확률이 적습니다.
  - Stack Memory 영역에 할당되므로 속도가 빠릅니다.
  - Scope based lifetime : 컴파일 타임에 컴파일러가 언제 메모리를 할당/해제할지 정확히 알고있습니다.
  - Data loclity : CPU 캐쉬 히트율이 높습니다.
  - 상속이 불가능합니다.
  - AnyObject로 타입 캐스팅이 불가합니다.

구조체를 사용하는 것이 유리한 경우

- 연관된 값들의 집합을 캡슐화하는 것이 목적인 경우
- 캡슐화된 값을 참조하는 것보다 복사하는 것이 합당할 때
- 다른 타입으로부터 상속받거나 자신이 상속될 필요가 없을 때
- JSON 필드와 1:! mapping 되는 간단한 모델이 필요할 때

선언하기

- 클래스는 class 키워드를, 구조체는 struct 키워드를 이름 앞에 적어서 선언합니다.

```
class SomeClass {
    // 프로퍼티와 메소드를 정의합니다.
}
struct SomeStructure {
    // 프로퍼티와 메소드를 정의합니다.
}
```

클래스와 구조체 인스턴스

- 클래스와 구조체 이름 뒤에 빈 괄호를 적으면 각각의 인스턴스를 생성할 수 있습니다.

```
let someStructure = SomeStructure() // 구조체 인스턴스 생성
let someClass = SomeClass() // 클래스 인스턴스 생성
```

프로퍼티 값 접근하기

- 점(dot) 문법을 통해 클래스나 구조체 인스턴스의 프로퍼티의 값에 접근할 수 있습니다.

```
class MarksStruct {
  var mark: Int

  init(mark: Int) {
    self.mark = mark
  }
}

class studentMarks {
  var mark = 300
}

let marks = studentMarks()
print("Mark is \(marks.mark)")
```

식별 연산자 (Identity Operators)

- 클래스는 참조 타입이기 때문에 여러 상수와 변수에서 같은 인스턴스를 참조할 수 있습니다. 상수와 변수가 같은 인스턴스를 참조하는지 비교하기 위해 식별 연산자를 사용할 수 있습니다.

===	!==
두 개의 상수 또는 변수가 동일한 인스턴스를 가리키는 경우 true를 반환합니다.	**두 개의 상수 또는 변수가 다른 인스턴스를 가리키는 경우 true를 반환합니다.**

PART 11

프로퍼티 Properties

Swift 요약집

프로퍼티

- 프로퍼티는 클래스, 열거형, 구조체에 관련한 값입니다. 프로퍼티는 저장 프로퍼티(Stored Properties)와 계산된 프로퍼티(Computed Properties)로 분류할 수 있습니다.
- 저장 프로퍼티는 값을 저장하고 있는 프로퍼티이고, 계산된 프로퍼티는 값을 저장하고 있지 않고 계산한 값을 반환해주는 프로퍼티입니다.
- 저장 프로퍼티는 클래스와 구조체에서만 사용 가능하며, 계산된 프로퍼티는 클래스, 구조체, 열거형 모두에서 사용 가능합니다.

저장 프로퍼티 (Stored Properties)

- 저장 프로퍼티는 단순히 값을 저장하고 있는 프로퍼티입니다. 상수는 'let' 키워드로 정의되고, 변수는 'var' 키워드로 정의됩니다.

```
struct Number {
  var digits: Int
  let pi = 3.1415
}

var number = Number(digits: 12345)
number.digits = 67

print("\(number.digits)") // 67
print("\(number.pi)") // 3.1415
```

지연 저장 프로퍼티 (Lazy Stored Properties)

- 지연 저장 프로퍼티는 처음 초기화 될 때, 초기 값을 계산하지 않는 프로퍼티입니다. 지연 저장 프로퍼티는 반드시 변수(var)로 선언되어야 하며, 앞에 'lazy' 키워드를 붙여야 합니다.
- 지연 저장 프로퍼티는 객체 생성을 지연시키기 위해 사용하거나 프로퍼티가 특정 요소에 의존적이어서 그 요소가 끝나기 전에 적절한 값을 알지 못하는 경우에 유용합니다.

```
class Sample {
  lazy var no = Number()
}

class Number {
  var name = "Swift"
}

var firstsample = Sample()
print(firstsample.no.name) // Swift
```

계산된 프로퍼티 (Computed Properties)

- 계산된 프로퍼티는 저장 프로퍼티와 다르게, 값을 저장하고 있는 것이 아니라 getter와 선택적인 setter를 이용하여 값을 탐색하고 간접적으로 다른 프로퍼티 값을 설정할 수 있는 방법을 제공합니다.

```
class Sample {
  var no1 = 0.0, no2 = 0.0
  var length = 300.0, breadth = 150.0

  // 계산된 프로퍼티
  var middle: (Double, Double) {
    get {
      return (length / 2, breadth / 2)
    }

    set(axis){
      no1 = axis.0 - (length / 2)
      no2 = axis.1 - (breadth / 2)
```

```
    }
  }
}

var result = Sample()
print(result.middle) // (150.0, 75.0)
result.middle = (0.0, 10.0)

print(result.no1) // -150.0
print(result.no2) // -85.0
```

읽기 전용 계산된 프로퍼티 (Read-Only Computed Properties)

- 읽기 전용 계산된 프로퍼티는 getter는 있지만 setter는 없는 프로퍼티입니다. 항상 값을 반환하는 데 사용되며, 변수는 '.'을 통해 접근할 수 있습니다.

```
class Movie {
  var title = ""
  var time = 0.0
  var info: [String: String] {
    return [
      "title": self.title,
      "time": "\(self.time)"
    ]
  }
}

var movie = Movie()
movie.title = "Swift"
movie.time = 3.09

print(movie.info["title"] ?? "")
print(movie.info["time"] ?? "")
```

프로퍼티 옵저버 (Property Observers)

- 프로퍼티에는 새로운 값이 설정될 때마다 이러한 이벤트를 감지할 수 있는 프로퍼티 옵저버를 제공합니다.
- 프로퍼티 옵저버는 새 값이 이전 값과 동일하더라도 항상 호출될 수 있습니다.
- 프로퍼티 옵저버는 지연 저장 프로퍼티에는 사용할 수 없습니다.
- 계산된 프로퍼티는 setter 에서 값의 변화를 감지 할 수 있기 때문에 옵저버를 따로 정의할 필요가 없습니다.
- 프로퍼티 옵저버는 두 가지 종류가 있습니다.
  - willSet : 값을 저장하기 전에 호출됩니다. 새로운 값의 파라미터 이름을 따로 지정하지 않으면, 기본 값으로 "newValue" 라는 이름을 사용할 수 있습니다.
  - didSet : 새로운 값이 저장되고 난 이후에 호출됩니다. 바뀌기 전 값의 파라미터 이름을 따로 지정하지 않으면, 기본 값으로 "oldValue" 라는 이름을 사용할 수 있습니다.

```
class Sample {
  var counter: Int = 0 {
    willSet(newTotal){
      print("Total Counter is: \(newTotal)")
    }

    didSet {
      if counter > oldValue {
        print("Newly Added Counter \(counter - oldValue)")
      }
    }
  }
}

let newCounter = Sample()
newCounter.counter = 100
// Total Counter is: 100
// Newly Added Counter 100

newCounter.counter = 800
// Total Counter is: 800
// Newly Added Counter 700
```

지역 변수와 전역 변수 (Local and Global Variables)

- 프로퍼티와 프로퍼티 옵저버는 전역 변수와 지역 변수 모두에서 이용 가능합니다.

지역 변수	전역 변수
함수, 메소드, 클로저 혹은 타입 컨텍스트 내부에 정의된 변수입니다.	**함수, 메소드, 클로저 혹은 타입 컨텍스트 외부에 정의된 변수입니다.**

타입 프로퍼티 (Type Properties)

- 타입 프로퍼티는 특정 인스턴스에 속한 프로퍼티를 말합니다.
- 값 유형에 대한 타입 프로퍼티를 선언하기 위해서는 'static' 키워드를 사용합니다.
- 클래스 유형에 대한 타입 프로퍼티를 선언하기 위해서는 'static'와 'class' 키워드를 사용할 수 있습니다.
- 타입 프로퍼티도 '.'을 이용해 프로퍼티의 값을 가져오고 할당할 수 있습니다.

```
struct Structname {
    static var storedTypeProperty = " "
    static var computedTypeProperty: Int {
        return 5 + 5
    }
}

enum Enumname {
    static var storedTypeProperty = " "
    static var computedTypeProperty: Int {
        return 5 + 5
    }
}

class Classname {
    class var computedTypeProperty: Int {
        return 5 + 5
    }
}
```

PART 12

상속 Inheritance

Swift 요약집

상속

- 상속은 어떠한 클래스가 가지고 있는 프로퍼티나 메소드를 다른 클래스가 이어 받는 것을 의미합니다.
- 상속을 받는 클래스를 '서브 클래스'나 '자식 클래스'라고 부르며, 주는 클래스를 '베이스 클래스', '슈퍼 클래스', '부모 클래스'라고 부릅니다.

서브클래싱 (Subclassing)

- 기존 클래스를 상속받아 새로운 클래스(자식 클래스)를 만드는 행위를 '서브클래싱' 이라고 합니다.
- 자식 클래스는 부모 클래스의 프로퍼티, 메소드를 상속 받을 수 있으며, 자기 자신 고유의 특성도 추가할 수 있습니다.
- 자식 클래스를 정의하려면 ":" 뒤에 부모 클래스 이름을 작성하면 됩니다.

```
class Vehicle {
    let speed: Int

    init(speed: Int) {
        self.speed = speed
    }

    func display() {
        print("speed is \(speed)")
    }
}

class Car: Vehicle {
    init() {
        super.init(speed: 120)
    }
```

```
}

let car = Car()
car.display() // speed is 120
```

오버라이딩 (Overriding)

- 자식 클래스에서는 부모 클래스에서 상속받은 것을 재정의할 수 있는데, 이를 '오버라이딩'이라고 부릅니다.
- 오버라이딩은 인스턴스 메소드, 타입 메소드, 인스턴스 프로퍼티, 타입 프로퍼티, 서브스크립트 모두에 대해 가능합니다.
- 오버라이드를 위해서는 다른 선언 앞에 'override' 키워드를 붙여줍니다.

```
class Vehicle {
    let speed: Int

    init(speed: Int) {
        self.speed = speed
    }

    func display() {
        print("speed is \(speed)")
    }
}

class Car: Vehicle {
    init() {
        super.init(speed: 120)
    }

    override func display() {
        print("car's speed : \(speed)")
    }
}

let car = Car()
car.display() // car's speed : 120
```

오버라이딩 방지

- 자식 클래스에서 특정 메소드, 프로퍼티 등이 재정의되는 것을 방지하기 위해 'final' 키워드를 사용합니다.

  재정의가 필요 없는 클래스를 정의할 때에는 final 키워드를 적용하는 것이 성능에 이점이 있습니다. final로 선언된 요소들은 직접 호출하는 반면에, 그렇지 않은 요소들은 vtable을 통해 간접 호출되어 직접 호출되는 경우보다 느리게 작동한다고 합니다.

  vtable이란 가상 메소드 테이블이며, 메소드 오버라이딩에 따라 실행 시점에 어떤 메소드를 실행할지 결정하는 동적 디스패치를 지원하기 위해 프로그래밍 언어에서 사용하는 메커니즘입니다.

```
final class Vehicle {
    let speed: Int

    init(speed: Int) {
        self.speed = speed
    }

    func display() {
        print("speed is \(speed)")
    }
}
```

PART 13

초기화 Initialization

Swift 요약집

초기화

- 초기화는 클래스, 구조체, 열거형 인스턴스를 사용하기 위해 준비 작업을 하는 단계입니다.
- 초기화 단계에서는 저장 프로퍼티의 초기 값을 설정합니다.
- 초기화 함수는 'init()' 형태로 사용합니다.
- 초기화와 반대로, 인스턴스가 할당 해제되면 메모리 관리 작업을 수행하기 위해 'deinit()' 함수도 제공합니다.

```
struct Rectangle {
    var length: Double
    var breadth: Double

    init() {
        length = 6
        breadth = 12
    }
}

var area = Rectangle()
print("area of rectangle is \(area.length * area.breadth)")
// area of rectangle is 72.0
```

초기화 파라미터 (Initialization Parameters)

- 초기화 정의에 파라미터를 정의해 사용할 수 있습니다.

```
struct Rectangle {
    var length: Double
    var breadth: Double
    var area: Double
```

```
    init(fromLength length: Double, fromBreadth breadth: Double) {
        self.length = length
        self.breadth = breadth
        area = length * breadth
    }
    init(fromLeng leng: Double, fromBread bread: Double) {
        self.length = leng
        self.breadth = bread
        area = leng * bread
    }
}


let ar = Rectangle(fromLength: 6, fromBreadth: 12)
print("area is: \(ar.area)") // area is: 72.0

let are = Rectangle(fromLeng: 36, fromBread: 12)
print("area is: \(are.area)") // area is: 432.0
```

클래스 상속 및 초기화 (Class Inheritance and Initialization)

- 모든 클래스의 저장 프로퍼티와 슈퍼 클래스로부터 상속받은 모든 프로퍼티는 초기화 단계에서 반드시 초기 값이 할당되어야 합니다. 클래스의 초기화 단계는 지정 초기자, 편리한 초기자 두 가지로 분류할 수 있습니다.

지정 초기화	편리한 초기화
클래스에 대한 기본 초기화 단계입니다.	클래스에 대한 초기화 지원 단계입니다.
모든 클래스에 대해 한 개 이상의 지정 초기자가 있어야 합니다.	클래스에 편리한 초기자를 필수로 정의할 필요는 없습니다.

init(parameters) {
statements
}
	convenience init(parameters) {
	statements
	}

```swift
class MainClass {
    var num1: Int

    init(num1: Int) {
        self.num1 = num1
    }
}

class SubClass: MainClass {
    var num2: Int

    init(num1: Int, num2: Int) {
        self.num2 = num2
        super.init(num1: num1)
    }
}

let main = MainClass(num1: 10)
let sub = SubClass(num1: 10, num2: 20)

print("\(main.num1)") // 10
print("\(sub.num1)") // 10
print("\(sub.num2)") // 20
```

PART 14

익스텐션 Extensions

Swift 요약집

익스텐션

- 익스텐션을 이용해 클래스, 구조체, 열거형 혹은 프로토콜 타입에 기능을 추가할 수 있습니다.
- 익스텐션은 타입에 새로운 기능을 추가할 수 있지만, 오버라이드는 할 수 없습니다.

```
extension SomeType {
  // 새로운 기능을 추가합니다.
}

extension SomeType: SomeProtocol, AnotherProtocol {
  // 프로토콜 요구사항을 정의합니다.
}
```

계산된 프로퍼티

- 익스텐션을 이용하여 존재하는 타입에 계산된 인스턴스 프로퍼티와 타입 프로퍼티를 추가할 수 있습니다.

```
extension Int {
   var add: Int { return self + 100 }
   var sub: Int { return self - 10 }
   var mul: Int { return self * 10 }
   var div: Int { return self / 5 }
}

let addition = 3.add
print("Addition is \(addition)") // Addition is 103

let subtraction = 120.sub
print("Subtraction is \(subtraction)") // Subtraction is 110
```

```
let multiplication = 39.mul
print("Multiplication is \(multiplication)") // Multiplication is 390

let division = 55.div
print("Division is \(division)") // Division is 11

let mix = 30.add + 34.sub
print("Mixed Type is \(mix)") // Mixed Type is 154
```

이니셜라이저

- 익스텐션을 이용해 존재하는 타입에 새로운 이니셜라이저를 추가할 수 있습니다.

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}

extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

변경 가능한 인스턴스 메소드

- 익스텐션에 추가된 인스턴스 메소드는 인스턴스 자신(self)를 변경할 수 있습니다.
- 구조체와 열거형 메소드 중 자기 자신을 변경하는 인스턴스 메소드는 반드시 mutating 으로 선언되어야 합니다.

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
```

PART 15

프로토콜 Protocols

Swift 요약집

프로토콜(protocol)은 특정 작업이나 기능에 적합한 메소드, 프로퍼티 및 기타 요구사항의 '약속'을 정의합니다. 클래스(class), 구조체(struct), 열거형(enum)에서는 어떠한 프로토콜을 채택하여 해당 프로토콜의 요구사항을 실제로 구현할 수 있습니다. 프로토콜의 요구사항을 충족시키는 모든 타입은 해당 프로토콜을 준수한다 혹은 따른다고 합니다.

프로토콜 정의하기

- protocol 키워드를 앞에 붙이고, 프로토콜의 이름을 작성합니다.
- 아래는 SomeProtocol을 정의한 코드입니다.

```
protocol SomeProtocol {
    // 프로토콜 내용
}
```

프로토콜 채택하기

- 클래스, 구조체, 열거형에서 정의한 프로토콜을 채택하기 위해서는 콜론(:)을 사용합니다.
- 하나의 타입에서 다수의 프로토콜을 채택하기 위해서는 콤마(,)를 사용합니다.
- 클래스에서 상속과 프로토콜 채택을 동시에 하려면 상속받으려는 클래스를 먼저 명시하고, 그 뒤에 채택할 프로토콜 목록을 작성합니다.

```
protocol SomeProtocol {
}

// 프로토콜 채택
struct SomeStructure: SomeProtocol {
}
```

```
// 다수의 프로토콜 채택
struct SomeStructure: SomeProtocol, AnotherProtocol {
}

// SomeSuperClass를 상속 받고 있고, SomeProtocol을 채택한 클래스 정의
class SomeClass: SomeSuperClass, SomeProtocol {
}
```

프로토콜의 프로퍼티, 메소드 요구사항

프로퍼티

- 프로퍼티는 타입과 이름만 작성합니다.
- gettable/settable 여부를 작성합니다.
- 프로퍼티는 var 로 선언해야 합니다.

메소드

- 메소드는 함수명과 반환 값을 지정할 수 있고, 메소드 내용을 작성하지 않아도 됩니다.

```
protocol SomeProtocol {
    var a: Int { get }
    var b: Int { get set }

    func setNumber(a: Int, b: Int)
    func add() -> Int
}

class SomeClass: SomeProtocol {
    var a: Int = 0
    var b: Int = 0

    func setNumber(a: Int, b: Int) {
        self.a = a
        self.b = b
    }
```

```
    func add() -> Int {
        return a + b
    }
}
```

변경 가능한 메소드 요구사항 (mutating 키워드)

- mutating 키워드를 사용하면, 값 타입 형에서 채택한 프로콜에서는 변경가능합니다.

```
protocol SomeProtocol {
    mutating func setNumber(a: Int)
}

struct SomeStructure: SomeProtocol {
    var a: Int

    mutating func setNumber(a: Int) {
        self.a = a
    }
}
```

프로토콜 이니셜라이저

- 프로토콜에서는 필수로 구현해야하는 이니셜라이저를 지정할 수 있습니다.
- 이니셜라이저가 있는 프로토콜을 채택한 타입에서는 해당 이니셜라이저에 required 키워드를 작성해야 합니다.

```
protocol SomeProtocol {
    init(someParameter: Int)
}

class SomeClass: SomeProtocol {
    required init(someParameter: Int) {

    }
}
```

프로토콜 익스텐션

- 이미 존재하는 타입에 새 프로토콜을 따르게 하기 위해서 확장(extension)을 사용할 수 있습니다.
- 기본 메소드 구현을 위해 프로토콜 익스텐션을 사용할 수 있습니다.

```
protocol SomeProtocol {
    func method()
}

extension SomeProtocol {
    func method() {
        // 기본 구현
    }
}

class SomeClass: SomeProtocol {
    // method 함수를 정의하지 않아도 에러가 발생하지 않습니다.
}
```

Delegate 패턴

- 델리게이트 패턴은 디자인 패턴 중 하나로, 프로그램 안에서 어떤 객체를 대신하여 행동한다던가, 다른 객체와 협동하여 일할 수 있게끔 권한을 위임하는 패턴입니다.
- iOS에서는 UITableViewDelegate나 UICollectionViewDelegate를 예시로 들 수 있습니다.
- Delegate 패턴을 사용하는 이유
    - 델리게이트 패턴은 한 클래스와 다른 클래스의 상호작용을 간단히 할 수 있도록 돕습니다.
    - 델리게이트 패턴은 1:1 관계에서 매우 유용합니다.
    - 델리게이트는 프로토콜을 준수하는 것만으로 구현이 가능하므로 매우 유연합니다.

프로토콜 상속

- 프로토콜은 하나 이상의 프로토콜을 상속받아 기존 프로토콜의 요구사항보다 더 많은 요구사항을 추가할 수 있습니다.
- 프로토콜 상속 문법은 클래스의 상속 문법과 유사하지만, 프로토콜은 클래스와 다르게 다중상속이 가능합니다.

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

클래스 전용 프로토콜

- 구조체, 열거형에서 사용하지 않고 클래스 타입에만 사용가능한 프로토콜을 선언하기 위해서는 프로토콜에 AnyObject 키워드를 추가합니다.

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

> 클래스 전용 프로토콜을 정의할 때, AnyObject 키워드와 class 키워드를 함께 사용할 수 있었으나 Swift 4부터는 AnyObject 키워드 사용을 권장했습니다. Xcode 12.5에서는 AnyObject 대신 class 키워드를 사용하면 warning이 발생합니다.

- 클래스 전용 프로토콜이 아닌 델리게이트 프로토콜을 클래스 안에서 Retain Cycle을 피하기 위해 weak으로 선언하면 *'weak' must not be applied to non-class-bound 'Delegate'; consider adding a protocol conformance that has a class bound* 와 같은 에러를 볼 수 있습니다. 해당 에러를 해결하기 위해서는 클래스 전용 프로토콜을 사용해야 합니다.

```
protocol MyDelegate: AnyObject {
    func method()
}

class SomeClass: MyDelegate {
    weak var delegate: MyDelegate?

    func method() {}
}
```

PART 16

에러 처리 Error Handling

Swift 요약집

프로그램 실행시 에러가 발생하면 그 상황에 대해 대응하고, 이를 복구하는 과정이 필요합니다. 이 과정을 '에러 처리' 라고 합니다.

Swift 에서는 런타임에 에러가 발생한 경우 그것의 처리를 위해 에러의 발생(throwing), 감지(catching), 전파(propagating), 조작(manipulating)을 지원하는 일급 클래스를 제공합니다.

Swift 에서 에러를 처리하는 네 가지 방법은 다음과 같습니다.

- 에러가 발생한 함수에서 반환 값으로 에러를 반환하여, 해당 함수를 호출한 코드에서 에러를 처리하도록 합니다.
- `do-catch` 구문을 사용합니다.
- 옵셔널 값을 반환합니다.
- `assert` 를 이용하여 강제로 크래쉬를 발생합니다.

`Error` 프로토콜과 열거형을 통한 에러를 표현하기

- Swift 에서 에러는 Error 프로토콜을 따르는 타입의 값으로 표현됩니다. Error 프로토콜은 비어있으며 에러 처리에 타입을 사용할 수 있음을 나타냅니다.

```
enum ServerError: Error {
    case forbidden
    case notFound
    case internalError
    case unauthorized
    // ...
}
```

throw 키워드 함수를 사용하여 에러 전파하기

- 어떤 함수, 메소드 혹은 Initializer가 에러를 발생 시킬 수 있다는 것을 알리기 위해서 throw 키워드를 함수 선언부의 파라미터 뒤에 붙일 수 있습니다. throw 키워드로 표시된 함수를 'throwing function'이라고 부르며, 만약 함수가 리턴 값을 명시했다면 throw 키워드는 리턴 값 표시 기호인 -> 전에 적습니다.
- throwing function 은 그 함수를 호출한 곳으로 에러를 전파합니다.

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

do-catch 구문 사용한 에러 처리

- 코드 블록에서 에러를 처리하기 위해 do-catch 구문을 사용할 수 있습니다. 만약 do 구문에서 에러가 발생한다면 에러의 종류를 catch 구문으로 구분해 처리할 수 있습니다.

```
do {
    try 표현식
    결과
} catch 패턴 1 {
    처리 결과
} catch 패턴 2 where 조건 {
    처리 결과
} catch {
    처리 결과
}
```

에러를 옵셔널 값으로 변환하기

- try? 키워드를 사용하면 에러를 옵셔널 값으로 변환하여 처리할 수 있습니다. 만약 try? 뒤에 있는 코드에서 에러가 발생한다면, 반환 값은 nil 이 됩니다.

```swift
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction() // 에러가 발생하면 x는 nil이 됩니다.

// x와 y는 동일한 값을 가질 수 있으며, 동일한 동작을 합니다.
let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

defer 구문

- defer 구문은 현재 코드 블록을 나가기 전에 꼭 실행되어야 하는 코드를 작성하여, 코드가 블록을 빠져나기기 전에 꼭 마무리해야 되는 작업을 할 수 있게 도와줍니다.
- defer 구문은 에러 처리 이외의 경우에도 사용할 수 있습니다.

```swift
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file) // block이 끝나기 직전에 실행, 주로 자원 해제나 정지에 사용
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

PART 17

제네릭 Generics

Swift 요약집

제네릭

- 제네릭 타입이란 타입을 파라미터화하여 컴파일 타임에 구체적인 타입이 결정되는 것을 의미합니다.
- 제네릭을 이용하면 타입에 유연하게 대처하는 것이 가능하며, 제네릭으로 구현한 기능과 타입은 재사용에 용이하며, 코드의 중복을 줄일 수 있어서 깔끔한 표현이 가능합니다.
- Swift 에서의 대표적인 제네릭 타입으로는 Array 와 Dictionary 타입이 있습니다. Array 와 Dictionary 는 Int 값을 저장할 수도 있고, String 값을 저장할 수도 있으므로, 타입에 제한이 없습니다.

```
// String 타입을 갖는 두개의 값을 바꾸는 함수
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

// Double 타입을 갖는 두개의 값을 바꾸는 함수
func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}

// 제네릭 타입을 활용하여 다양한 타입의 값을 바꾸는 함수
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

타입 파라미터 (Type Parameters)

- 위 `swapTwoValues` 함수의 예제에서 `T`는 타입 파라미터의 예입니다. 타입 파라미터는 꺽쇠 괄호 (<>) 사이에, 견본 (placeholder) 타입을 지정한 다음에 이름을 지정하고, 함수의 이름 바로 뒤에 작성합니다. 타입 파라미터를 한번 선언하면 이 것을 함수의 타입으로 사용할 수 있습니다. 복수의 타입 파라미터를 사용할때는 `<T, U>`와 같이 콤마로 구분해 줍니다.

타입 파라미터 이름짓기

- `Dictionary<Key, Value>`와 같이 엘리먼트 간의 서로 상관관계가 있는 경우 의미가 있는 이름을 파라미터 이름으로 붙이고, 그렇지 않은 경우는 `T`, `U`, `V`와 같은 단일 문자로 파라미터 이름을 사용합니다.
- 값이 아니라 타입에 대한 placeholder 임을 가리키기 위해, 항상 타입 매개변수의 이름에는 카멜 표기법을 사용합니다.

타입 제약 사항

- 타입 제약사항은 타입 매개변수가 특정 클래스에서 상속되어야 하거나, 특정 프로토콜이나 프로토콜 합성을 준수하는 것을 지정합니다.
- 예를들어, Swift의 Dictionary 타입은 딕셔너리의 키로 사용될 수있는 타입을 Hashable 로 제한합니다.
- 단일 클래스나 타입 매개변수 이름 뒤에 타입 매개변수 목록으로 콜론(:)으로 구분된, 프로토콜 제약사항을 위치시켜서 타입 제약사항을 작성합니다.

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U)
{
    // ...
}
```

연관 타입 (Associated Types)

- 연관 타입(associated type)은 프로토콜에서 사용되는 타입에 placeholder 이름을 주는 것입니다. 연관 타입에 사용하는 실제 타입은 프로토콜이 채택될때까지 지정되지 않습니다. 연관 타입은 `associatedtype` 키워드로 지정됩니다.

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

제네릭 Where절

- 제네릭 where 절은 특정 프로토콜을 반드시 준수하는 연관된 타입을 가져오거나 특정 타입 매개변수와 연관된 타입이 동일하게 하는것이 가능합니다. 제네릭 where절은 `where` 키워드로 시작합니다. 연관된 타입 또는 타입과 연관된 타입간의 동등한 관계에 대한 제약사항이 뒤따릅니다. 제네릭 where절을 타입이나 함수의 본문의 열린 중괄호({}) 바로 앞에 작성합니다.

```
extension Container where T: hashable    {
    ...
}
```

PART 18

ARC Automatic Reference Counting

Swift 요약집

Swift에서는 메모리 사용을 관리하기 위해 'ARC(Automatic Reference Counting)' 라는 메모리 관리 전략을 사용하여, 대부분의 경우에 개발자는 메모리 관리에 신경을 쓸 필요가 없습니다.

메모리 관리는 메모리 영역 중, 힙 영역과 관련이 있습니다. 힙 영역은 참조(Reference) 형 타입인 클래스, 클로저 등이 보관됩니다. 값(Value) 타입인 구조체, 열거형 등은 메모리 관리 대상이 아닙니다.

힙 영역의 참조형 자료들이 프로그램 상에서 얼마나 참조되는지 숫자를 세어서, 메모리가 자동으로 할당 및 제거하도록 관리하는 것이 ARC 입니다.

ARC의 동작

- ARC는 클래스 인스턴스를 생성하였을 때, 메모리를 할당합니다. 그리고 클래스 인스턴스가 더이상 필요하지 않을 때, ARC는 해당 메모리를 해제합니다. 이 때, ARC는 잘못된 메모리 접근을 막기 위해 얼마나 많은 프로퍼티들이 각각의 클래스 인스턴스를 참조하고 있는지 숫자를 셉니다. Swift에서는 각 객체마다 참조 횟수를 가지고 있고, 이 개수를 통해 메모리가 해제되어야 하는지를 결정합니다. 그래서 참조 횟수가 0이 되는 순간까지 ARC는 객체를 메모리에서 해제하지 않습니다.

강한 참조 순환 (Strong Reference Cycle)

- 강한 참조 순환은 두 개의 참조형 타입의 데이터가 서로 강한 인스턴스로 참조하고 있어서, 참조 개수가 0개가 되지 않는 상황을 의미합니다.
- 이들 인스턴스는 자신을 선언한 변수가 nil 이 되어도, 서로에 대한 강한 참조 때문에 메모리를 빠져나가지 못하게 되고, 이로 인해 메모리 누수(Memory Cycle)가 일어나게 됩니다.
- 강한 참조 순환 문제를 해결하기 위해서는 두가지 방법이 있습니다. 하나는 weak 참조, 다른 하나는 unowned 참조를 사용하는 것입니다. weak 참조, unowned 참조 모두 ARC에서 참조 횟수를

증가시키지 않고 인스턴스를 참조합니다. 그래서 강한 참조 순환 문제를 해결할 수 있습니다. 약한 참조는 참조하고 있는 인스턴스가 먼저 메모리에서 해제될때 사용하고 미소유 참조는 반대로 참조하고 있는 인스턴스가 같은 시점 혹은 더 뒤에 해제될때 사용합니다.

```
class Person {
    let name: String
    var apartment: Apartment?

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(name) is being deinitialized")
    }
}

class Apartment {
    let unit: String
    var tenant: Person?

    init(unit: String) {
        self.unit = unit
    }

    deinit {
        print("Apartment \(unit) is being deinitialized")
    }
}
```

약한 참조 (Weak References)

- 약한 참조는 강한 참조와 다르게 참조 횟수를 증가 시키지 않습니다.
- 변수의 선언 앞에 weak 키워드를 작성하여 사용하며, nil 이 할당될 수 있어야 하기 때문에 항상 옵셔널 타입입니다.
- ARC에서 약한 참조에 nil을 할당하면 프로퍼티 옵저버는 실행되지 않습니다.

```swift
class Person {
    let name: String
    var apartment: Apartment?

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(name) is being deinitialized")

    }
}

class Apartment {
    let unit: String
    weak var tenant: Person?

    init(unit: String) {
        self.unit = unit
    }

    deinit {
        print("Apartment \(unit) is being deinitialized")
    }
}
```

미소유 참조 (Unowned References)

- 미소유 참조는 해당 인스턴스에 대하여 Reference Counting을 하지 말라고 선언하는 것입니다.
- 변수의 선언 앞에 unowned 키워드를 작성하여 사용하며, 옵셔널 타입 일 수도 있고, 아닐 수도 있지만 nil 일 경우에 런타임 에러가 발생할 확률이 큽니다.
- 미소유 참조는 참조하는 동안 해당 인스턴스가 메모리에서 해제되지 않으리라는 확신이 있는 경우에만 사용하는 것이 좋습니다.

```swift
class Customer {
    let name: String
    var card: CreditCard?

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(name) is being deinitialized")

    }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer

    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }

    deinit {
        print("Card #\(number) is being deinitialized")
    }
}
```

클로저에서의 강한 참조 순환

- 클래스와 마찬가지로, 참조 타입인 클로저에서도 강한 참조 순환이 발생할 수 있습니다.
- 클래스 내부에서 self 를 캡처(capture)해서 사용할 수 있기 때문에, self 를 약한 참조 혹은 미소유 참조로 지정하면 강한 참조 순환 문제를 해결할 수 있습니다.

> 클로저는 자신이 정의되는 시점에 주변 환경으로부터 여러 상수나 변수에 대한 참조를 잡아내어 저장합니다. 이를 'capture' 라고 부릅니다.

```
lazy var someClosure: () -> String = { [weak self] in
    // ..
}
```

PART 19

접근 제어 Access Control

Swift 요약집

접근 제어

- 접근 제어는 특정 코드의 접근을 다른 소스 파일이나 모듈에서 제한하는 것입니다.

모듈과 소스파일

- Swift의 접근 제어는 모듈과 소스파일에 기반을 두고 있습니다.
- 모듈은 코드를 배포하는 단일 단위로, 프레임워크나 앱이 이 단위로 배포되고 다른 모듈에서 import 키워드를 사용해 포함될 수 있습니다.
- 소스파일은 모듈 안에 있는 소스파일을 의미합니다.

접근 레벨

- Swift에서는 5개의 접근 레벨을 제공합니다.

Open	**선언한 모듈이 아닌 다른 모듈에서 사용 가능합니다.**
Public	**선언한 모듈이 아닌 다른 모듈에서 접근이 가능하지만, 다른 모듈에서 오버라이드와 서브클래싱이 불가능합니다.**
Internal	**기본 접근 레벨입니다.**
File-Private	**특정 엔티티가 선언한 파일 안에서만 사용 가능합니다.**
Private	**특정 엔티티가 선언된 괄호({ }) 안에서만 사용 가능합니다.**

```
public class SomePublicClass { }
internal class SomeInternalClass { }
private class SomePrivateClass { }

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() { }
```

PART 20

RxSwift 기본

Swift 요약집

RxSwift

- Rx는 Reactive Extensions (= Reactive X)의 약자로, 반응형 프로그래밍 패턴에 따라 코드를 작성하는 데 도움을 줍니다.
- RxSwift는 Swift 에서 Rx를 구현한 라이브러리이며, RxCocoa는 UIKit과 Cocoa 프레임워크 기반 개발을 지원하는 라이브러리입니다.
- Rx는 변화할 수 있는 상태에 쉽게 대처할 수 있고, 이벤트들의 순서의 구성과, 코드 분리, 재사용성 등을 향상시킬 수 있습니다.
- Rx의 3요소에는 Observable, Operator, Scheduler가 있습니다.

Observables

- Observable 클래스는 여러 이벤트를 생성할 수 있는 대상을 의미합니다.
  - next : 최신/다음 데이터를 전달하는 이벤트입니다. 구성 요소를 계속해서 방출할 수 있습니다.
  - completed : 성공적으로 일련의 이벤트들을 종료시키는 이벤트. 즉, Observable이 성공적으로 자신의 생명주기를 완료했으며, 추가적으로 이벤트를 생성하지 않습니다.
  - error : Observable이 에러를 발생했으며, 추가적으로 이벤트를 생성하지 않고, 완전 종료하는 이벤트입니다.

```
_ = API.download(file: "https://fastcampus.co.kr...")
 .subscribe(onNext: { data in
   // Append data to temporary file
 },
 onError: { error in
   // Display error to user
 },
 onCompleted: {
   // Use downloaded file
 })
```

Operator

- ObservableType과 Observable 클래스에는 복잡한 논리를 구현하기 위해 많은 메소드가 포함되어 있습니다. 이러한 메소드들을 Operator 라고 합니다.
- Operator는 비동기 입력을 받아 출력만 생성하기 때문에 Operator 들끼리 쉽게 혼합해서 사용이 가능합니다.
- RxOperator들은 Observable에 의해 들어온 값들을 처리하고 최종 값이 나올 때 방출합니다.

Observable을 생성하는 Operator

just

특정 Observable를 하나만 간단하고 쉽게 생성할 수 있습니다.

```
Observable.just("RxSwift")
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// RxSwift
```

from

just와 동일하지만, just는 한 번에 모든 결과를 방출하는 방면에 from은 결과를 하나씩 방출합니다.

```
Observable.from(["Rx", "Swift"])
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// Rx
// Swift
```

of

from과 동일하지만, from은 배열만 받을 수 있으며 of는 배열 뿐만 아니라 각각의 값을 받을 수 있다.

```
Observable.of("Rx", "Swift")
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// Rx
// Swift
```

방출하는 결과를 변환하는 Operator

map	**map은 방출한 결과를 원하는 으로 변형해서 방출시킬 수 있습니다.**
	`Observable.just("RxSwift")`
	`  .map { $0.count }`
	`  .subscribe(onNext: { data in`
	`  print(data)`
	`  }).disposed(by: disposeBag)`
	`// 7`

flatMap	**방출된 이벤트로부터 새로운 Observable로 변형하여 방출합니다.**
	`let sequenceInt = Observable.of(1, 2)`
	`let sequenceString = Observable.of("A", "B")`
	`sequenceInt`
	`  .flatMap { (num: Int) -> Observable<String> in`
	`  print("Int Item : \(num)")`
	`  return sequenceString`
	`  }.subscribe(onNext: { str in`
	`  print("String Item: \(str)")`
	`  }).disposed(by: disposeBag)`
	`// Int Item : 1`
	`// String Item: A`
	`// Int Item : 2`
	`// String Item: B`
	`// String Item: A`
	`// String Item: B`

flatMapLatest	**기본적으로 flatMap과 동일하게 작동하지만, 가장 최신의 값만을 다룰 수 있습니다.**

방출하는 필터링하는 Operator

filter

방출된 이벤트 중 조건에 맞는 이벤트만을 방출시킵니다.

```
Observable.from([1, 2, 3, 4, 5])
    .filter { $0 % 2 == 0 }
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// 2
// 4
```

take

처음으로 들어오는 n개의 이벤트만 방출합니다.

```
Observable.of(1, 2, 3, 4, 5, 6)
    .take(3)
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// 1
// 2
// 3
```

skip

처음으로 방출되는 n개의 이벤트는 방출하지 않고, 이후부터 방출합니다.

```
Observable.of(1, 2, 3, 4, 5, 6)
    .take(3)
    .subscribe(onNext: { data in
        print(data)
    }).disposed(by: disposeBag)
// 1
// 2
// 3
```

distinctUntil Changed	연속으로 동일한 값이 방출되지 않도록 합니다.
	`Observable.of(1, 1, 1, 2, 3, 4)`
	`  .distinctUntilChanged()`
	`  .subscribe(onNext: { data in`
	`  print(data)`
	`  }).disposed(by: disposeBag)`
	`// 1`
	`// 2`
	`// 3`
	`// 4`

throttle	지정한 시간동안 처음 들어오는 이벤트를 방출합니다. latest라는 변수의 값에 따라 이벤트를 다르게 방출합니다. false인 경우에는 처음에 들어온 이벤트만 방출하며, true인 경우에는 처음과 마지막 이벤트를 방출합니다.
debounce	지정한 시간동안 들어온 이벤트 중, 가장 최신의 이벤트를 방출합니다.

Observable을 합성하는 Operator

merge	여러 개의 Observable을 합치지만, 하나의 이벤트를 받을 수 있습니다. 공통의 로직을 실행해야 할 경우 사용하면 유용합니다.
	`let observable1 = Observable.of(1, 2, 3)`
	`let observable2 = Observable.of(4, 5, 6)`
	
	`Observable.merge(observable1, observable2)`
	`  .subscribe(onNext: {`
	`  print($0)`
	`  })`
	`  .disposed(by: disposeBag)`
	`// 1`
	`// 4`
	`// 2`
	`// 5`
	`// 3`
	`// 6`

combine Latest	**두 개 이상의 Observable 중, 가장 최근의 이벤트가 방출될 때 이벤트를 전부 방출합니다. 둘 중, 어느 하나라도 방출하지 않은 Observable이 있다면 방출하지 않습니다.**
	`let observable1 = Observable.of(1, 2, 3)`
	`let observable2 = Observable.of(4, 5, 6)`
	
	`Observable.merge(observable1, observable2)`
	`  .subscribe(onNext: {`
	`  print($0)`
	`  })`
	`  .disposed(by: disposeBag)`
	`// (1, 4)`
	`// (2, 4)`
	`// (2, 5)`
	`// (3, 5)`
	`// (3, 6)`

zip	**두 개 이상의 Observable 중, 이벤트를 방출하면 순서가 맞은 쌍끼리 하나의 이벤트를 방출합니다.**
	`let observable1 = Observable.of(1, 2, 3)`
	`let observable2 = Observable.of(4, 5, 6)`
	
	`Observable.merge(observable1, observable2)`
	`  .subscribe(onNext: {`
	`  print($0)`
	`  })`
	`  .disposed(by: disposeBag)`
	`// (1, 4)`
	`// (2, 5)`
	`// (3, 6)`

Schedulers

- Schedulers는 Rx에서 dispatch queue와 동일하며, 훨씬 강력하고 사용하기 쉽습니다.
- RxSwift에는 여러가지의 스케줄러가 이미 정의되어 있으며, 개발자가 새롭게 스케줄러를 정의해서 사용하는 일은 거의 없습니다.

스케줄러의 종류

MainScheduler	**ainThread에서 실행되어야 할 작업을 처리합니다.**
CurrentThreadScheduler	**현재 있는 Thread에서 작업을 처리합니다.**
SerialDispatchQueue Scheduler	**특정한 dispatch_queue_t 에서 실행되어야 할 작업을 처리합니다.**
ConcurrentDispatchQueue Scheduler	**특정한 dispatch_queue_t 에서 실행되어야 할 작업을 처리합니다. 보통 백그라운드 작업을 처리할 때 사용합니다.**
OperationQueueScheduler	**NSOperationQueue에서 실행되어야 할 작업을 처리합니다.**

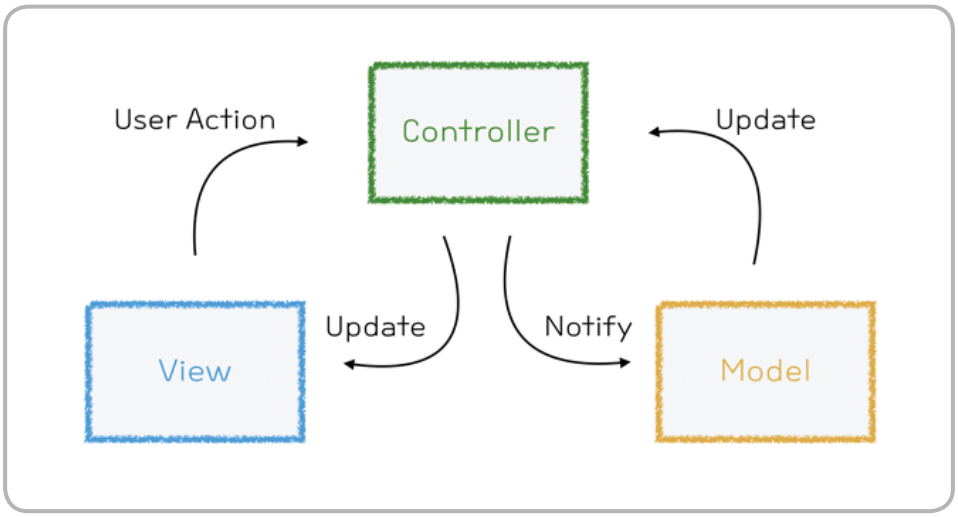
PART 21

MVC vs MVVM

Swift 요약집

MVC

- MVC란 Model View Controller의 약자로 애플리케이션을 세가지의 역할로 구분한 개발 방법론입니다. 사용자가 Controller를 조작하면 Controller는 Model을 통해서 데이터를 가져오고 그 정보를 바탕으로 시각적인 표현을 담당하는 View를 제어해서 사용자에게 전달하게 됩니다.



- Model : 어플리케이션에서 사용되는 데이터와 그 데이터를 처리하는 부분입니다. 예를 들어, 도서관이라는 어플리케이션에서의 책 모델은 책 이름, 저자, 고유 번호, 출판사 이름, 출판 일 등과 같은 정보가 모델에 저장되게 됩니다.

- View : 사용자가 Controller를 통해 전달한 요청을 모델을 통해 전달받고 사용자에게 보여주는 UI 부분입니다. View는 모델의 데이터에 어떻게 접근할 지에 대한 것만 알고 있고, 이 데이터 자체가 무엇을 의미하는지 알지 못합니다.

- Controller는 View와 Model의 사이에서, 사용자로부터 어떠한 이벤트를 받고 처리하는 부분입니다. View의 입력을 Model에 반영하고, Model의 변화를 View에 갱신하는 역할을 합니다.

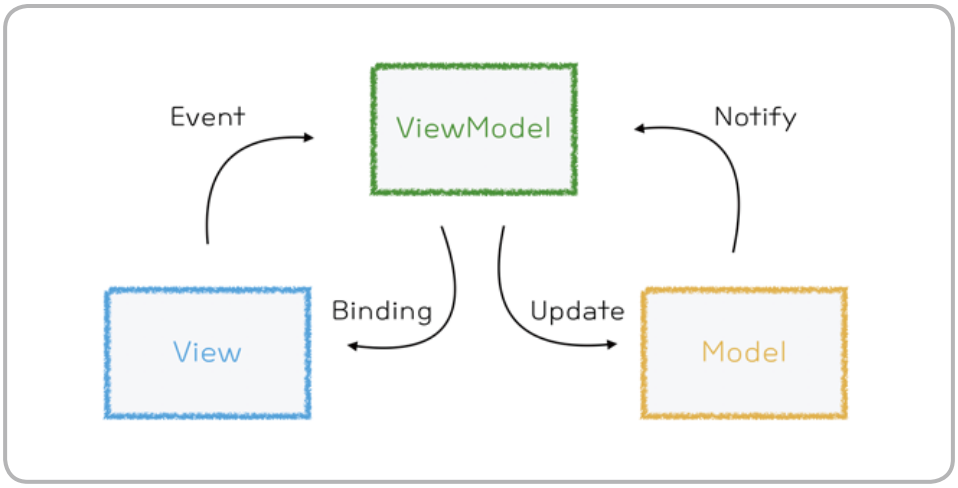
- 이렇게 mvc패턴을 사용하면 서로 분리되어 각자의 역할에 집중할 수 있게끔하여 개발을 하고 그렇게 애플리케이션을 만든다면, 유지보수성, 확장성 그리고 유연성이 증가하고, 중복 코딩이라는

문제점이 사라지게 됩니다.

- MVC 패턴의 특징
  - MVC 패턴은 가장 단순하고 보편적으로 많이 사용되는 패턴입니다.
  - View와 Model 사이의 의존성이 높기 때문에 패턴이 모호해질 수 있고, 어플리케이션의 규모가 커질수록 유지보수가 어려울 수 있습니다.
  - 기본 기능 설계를 위해 클래스들이 많이 필요하기 때문에 복잡해질 수 있습니다.

MVVM

- MVVM패턴이란, 아키텍쳐에서 Model, View, ViewModel 을 합친 용어입니다. 쉽게 말해, MVC 패턴에서 Controller를 제외하고, ViewModel을 추가했다고 볼 수 있습니다. 여기서는 ViewController가 View의 역할을 하고, ViewModel의 중간 역할을 합니다.



- Model : 어플리케이션에서 사용되는 데이터와 그 데이터를 처리하는 부분입니다.
- View : 사용자에서 보여지는 UI 부분입니다.
- View Model : View를 표현하기 위해 만든 View를 위한 Model입니다. View와 ViewModel 사이에는 Binding이라는 개념이 있는데, 연결 고리라는 의미입니다. 주로 바인딩을 할 때, Rx를 이용합니다.

- **MVVM 패턴의 특징**
  - View와 Model 사이의 의존성을 없애여, 각각의 부분이 독립적이기 때문에 테스트하기에 좋습니다.
  - ViewModel의 설계가 쉽지 않고 데이터 바인딩 기법을 통해 View를 바꿀 때 많은 코드의 작성을 필요로 하므로 간단한 View나 로직을 만들 때에도 하는 일에 비해 많은 코드를 작성해야 합니다.

SWIFT GRAMMAR SUMMARY

Swift
문법 요약집