



데이터 타입 안정

- 애플이 Swift에서 관리 경찰했던 안전성이 가장 뛰어나는 특징이다.

Swift는 type이 정해져 있고 서로 다른 데이터 타입 간에 고친은

꼭 type-casting을 거쳐야 한다.

Swift에서 값타입의 데이터고친은 명시이 말해 type-casting이 아님

새로운 인스턴스를 생성하여 학습하는 것이다.

• 데이터 타입 안정이란?

- Swift는 데이터 타입을 안정하고 사용할 수 있는 (Type-Safe) 언어이다.

타입을 안정하고 사용할 수 있다는 말은 그만큼 성수도 줄일 수 있다는 말인데,

예를 들어, Int 타입 변수에 할당하려는 값이 Character 타입이라면 컴파일 오류가 발생한다.

이런 오류는 프로그래밍 도중에 놓치기 어렵기 때문에 결과물에 영향주지 않으면

나중에 오류를 찾아내기도 쉽지 않다.

이렇게 Swift가 결과일 시 타입을 확인하는 것을 **타입 확인**이라고 한다.

타입 확인을 통해 여러 타입은 선이 사용할 때 발생할 수 있는 런타임 오류를 피할 수도 있다.

• 타입 추론

- Swift에서는 변수나 상수를 선언할 때 특정타입을 명시하지 않아도

컴파일러가 합당한 값을 기준으로 변수나 상수의 타입을 결정한다.

// 타입을 지정하지 않았으나 타입 추론을 통하여 name은 String 타입으로 선언된다.

var name = "Hoon"

// 앞서 타입 추론에 의해 name은 String 타입의 변수로 지정되었기 때문에

// 정수를 할당하려고 시도하면 오류가 발생한다.

name = 100

✖ Cannot assign value of type 'Int' to type 'String'.

• 타입 벽장

- 사용자가 임의로 뜯든 데이터타입에는 이미 존재하는 타입에는
데이터타입에 임의로 벽장을 브여볼 수 있다.

typealias MyInt = Int
typealias YourInt = Int
typealias MyDouble = Double

let age: MyInt = 100 // MyInt는 Int의 또 다른 이름이다.
var year: YourInt = 2088 // YourInt도 Int의 또 다른 이름이다.

// MyInt도, YourInt도 Int이기 때문에 같은 타입으로 취급된다.
year = age

let month: Int = 7 // 물론 기본의 Int도 사용 가능하다.
let percentage: MyDouble = 99.9 // Int 외에 다른 자료형도 모두 벽장 사용이 가능하다.

• 튜플 (Tuple)

- 튜플은 타입의 이름이 따로 지정되어 있지 않은, 프로그래밍 사용자를 만드는 타입이다.

'지정된 데이터의 모음'이라고 표현할 수 있다. (여기로 예로 들면 원시구조화 할 수 있다).

튜플은 타입 이름이 따로 없으므로 일상타입의 나열번호로 생성해줄 수 있다.

(포함된 데이터의 개수는 자유로울)

// String, Int, Double 타입을 갖는 튜플
var person: (String, Int, Double) = ("hoon", 100, 175.3)

// 인덱스를 통해서 값을 빼올 수 있다.

print("이름: \(person.0), 나이: \(person.1), 신장: \(person.2)")

이름: hoon, 나이: 100, 신

장: 175.3

person.1 = 99 // 인덱스를 통해 값을 할당할 수 있다.
person.2 = 186.3

print("이름: \(person.0), 나이: \(person.1), 신장: \(person.2)")
이름: hoon, 나이: 99, 신
장: 186.3

• 튜플 벽장 지정

- 매번 같은 모양의 튜플을 사용해야 한다면 벽장 지정이 꼬여는 것이다.

컬렉션

- Swift에는 둘째 외에도 많은 수의 컬렉션을 목록처럼 저장하고

관리할 수 있는 컬렉션 타입을 제공한다.

컬렉션 타입에는 배열(Array), 딕셔너리(Dictionary), 세트(Set) 등이 있다.

• 배열

- 배열 탐색 방법: [let |] 사용해 상수로 선언후에 변경할 수 없는 배열 사용)

var (변경 가능한 배열)

배열을 사용하면 Array라는 키워드와 타입 예약의 조건,
또는 대괄호로 값을 묶어 Array 타입명을 표현할 수 있다.

- 빈 배열: 빈 배열은 여러 성격의 또는 리터럴 문법을 통해 생성 가능하다

isEmpty 표기로 비어있는 배열인지를 알 수 있다.

(count 표기로 배열에 존재하는 요소의 개수를 알 수 있다.

// 대괄호를 사용하여 배열임을 표현.

var names: Array<String> = ["hoon", "yagom", "chulsoo", "hoon"]

// 위 선언과 정확히 동일한 표현법. [String]은 Array<String>의 축약 표현이다.

var names2: [String] = ["hoon", "yagom", "chulsoo", "hoon"]

var emptyArray: [Any] = [Any]() // Any 데이터를 요소로 갖는 빈 배열을 생성함.

var emptyArray2: [Any] = Array<Any>() // 위 선언과 정확히 같은 동작을 하는 코드.

// 배열의 타입을 정확히 명시해줬다면 []만으로도 빈 배열을 생성할 수 있다.

var emptyArray3: [Any] = []

print(emptyArray3.isEmpty) // true

print(names3.count) // 4

NOTE 스위프트의 Array

스위프트의 Array는 C 언어의 배열처럼 바파이며입니다. 단 C 언어처럼 한 번 선언하면 크기가 고정되며
버퍼가 아니라, 필요에 따라 자동으로 버퍼의 크기를 조절해주므로 요소의 삽입 및 삭제가 자유롭습니다. 스
위프트는 이런 리스트 타입을 Array, 즉 배열이라고 표현합니다. 기존 언어의 배열과는 조금 다른 특성도 있
지만 이 책에서도 Array를 배열이라고 표현하겠습니다.

- 또한 배열은 각 요소에 인덱스를 통해 접근 가능하다.

잘 못된 인덱스에 접근하면 **Exception Error**가 발생된다.

또, 맨 처음과 마지막 요소는 **first**와 **last** 프로퍼티를 통해 가져올 수 있다.

index(of:) 메서드를 사용하면 해당 요소의 index를 알아낼 수 있다.

만약 중복된 요소가 있다면 제일 먼저 발견된 요소의 index를 반환한다.

맨 뒤에 요소를 추가하고 싶다면 **append(_:)** 메서드를 사용한다.

중간에 요소를 추가하고 싶다면 **insert(_:at:)** 메서드를 사용한다.

var names: Array<String> = ["yagom", "chulsoo", "younghee", "yagon"]

names[0] // younghee
print(names[0]) // jenny

print(names[1]) // index 0의 범위를 벗어났기 때문에 오류가 발생함.
names[1] // elsa

names.append("elsa") // 배열에 elsa가 추가됨. (3)
names.contentsOf: ["john", "max"]) // 맨 마지막에 john max가 추가됨. (4)
names.insert("happy", at: 2) // index 2에 happy가 추가됨. (5)

// index 0에 위치해 minsoo가 삽입됨. (6)
names.insert(contentsOf: ["jinhae", "minsoo"], at: 5)

print(names[0]) // yagom
print(names.index(of: "yagom")) // 0
print(names.index(of: "christa")) // nil
print(names.first) // yagom
print(names.last) // max

let firstItem: String = names.removeFirst() // (7)
let lastItem: String = names.removeLast() // (8)
let indexZeroItem: String = names.remove(at: 0) // (9)

print(firstItem) // yagom
print(lastItem) // max
print(indexZeroItem) // chulsoo
print(names[1...3]) // ["jenny", "yagom", "jinhae"]

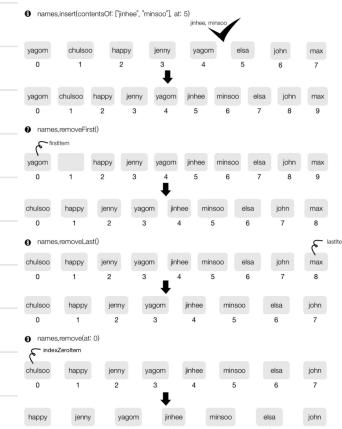
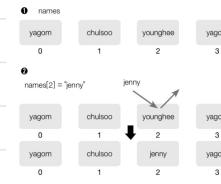
방법의 일관성을 4장에서 **name 배열의 일부**로 복습

개념:

또는 방식에 맞게 모든 내용 수 있음

ex) names[1...3] = ["A", "B", "C"]

• names Array 모식도



• Dictionary

- 특징

- Dictionary는 요소들이 순서없이 키와 값의 쌍으로 구성되는 컬렉션 타입.
- 같은 Dictionary 안에서 key의 이름은 중복될 수 없음.
즉, key는 값을 대변하는 유일한 식별자임.

- 사용

- Dictionary라는 키워드와 키의 타입과 값의 타입 이름의 조합으로 써줌.
대괄호로 키와 값의 이름의 쌍을 묶어주면 가능.
- let : 변경 불가능한 Dictionary
var : 변경 가능
- **비교**: 배열과 문자열을 통해 비교
`isEmpty()` 표시되는 통해 빈여있는 Dictionary인지 알수 있음,
`Count` : 표시 개수

• Dictionary 선언과 사용

```
// typealias를 통해 조금 더 단순하게 표현할 수 있다.
typealias StringIntDictionary = [String: Int] ↪ 타입 별칭 키워드

// 키는 String, 값은 Int 타입인 빈 딕셔너리를 생성.
var numberForName1: Dictionary<String, Int> = Dictionary<String, Int>()

// 위 선언과 같은 표현, [String: Int]는 Dictionary<String, Int>의 측약 표현이다.
var numberForName2: [String: Int] = [String: Int]()

// 위 코드와 같은 동작.
var numberForName3: StringIntDictionary = StringIntDictionary()

// dictionary의 키와 값 타입을 정확히 명시해준다면 [:]으로 빈 딕셔너리를 생성할 수 있다.
var numberForName4: [String: Int] = [:] ↪ 리터럴

// 초기값을 주어 생성 가능. ①
var numberForName: [String: Int] = ["yagom": 100, "chulsoo": 200, "jenny": 300]

print(numberForName.isEmpty) // false
print(numberForName.count) // 3
```

. Dictionary의 API

```
print(numberForName["chulsoo"]) // 200
print(numberForName["minji"]) // nil

numberForName["chulsoo"] = 150 // ②
print(numberForName["chulsoo"]) // 150

numberForName["max"] = 999 // max라는 키로 999라는 값을 추가해줌 ③
print(numberForName["max"]) // 999

print(numberForName.removeValue(forKey: "yagom")) // 100 ④

// 위에서 yagom 키에 해당하는 값이 이미 삭제되었으므로 nil로 반환됨.
// 키에 해당하는 값이 없으면 기본값을 돌려주도록 할 수도 있음.
print(numberForName.removeValue(forKey: "yagom", default: 0)) // 0
```

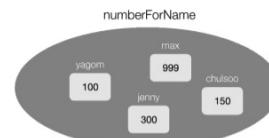
• Dictionary는 Array와는 다르게 Dictionary 내부에 값은 키로 접근해도 오류가 발생하지 않는다. 키를 빼는 걸 뺄때다.

• NumberForName Dictionary 모식도

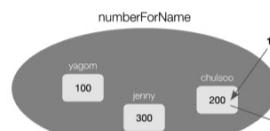
① var numberForName: [String: Int] =
["yagom": 100, "chulsoo": 200, "jenny": 300]
numberForName



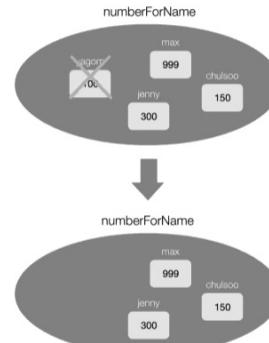
① numberForName["max"] = 999



② numberForName["chulsoo"] = 150



① numberForName.removeValue(forKey: "yagom")



• 111트

- 특징

- 같은 type의 Data를 순서없이 하나의 모음으로 저장하는 형태

컬렉션 type.

- IDE 내의 값은 모두 원본 값 - 중복 X

그러나 세트는 보통 순서에 양의 양과 유일한 값이어야 하는 경우에 사용함.

세트의 요소는 해시 가능한 값이 들어와야 함.

TIP

<https://applecider2020.tistory.com/14>

- 사용

- Set 키워드와 type 이름의 조합으로 작성.
- All와 마찬가지로 대괄호로 값을 묶어 사용.

But, All와 달리 중에서 표현할 수 있는
즉각형 (ex) `Allarray<Int> => [Int]` 이
없다.

- 빈 세트 : 이니셜라이저 or 리터럴

`isEmpty, count` 사용 가능

• 세트의 선과 사용

```
var names1: Set<String> = Set<String>()           // 빈 세트 생성 (이니셜라이저)
var names2: Set<String> = []                         // 빈 세트 생성 (리터럴)

// Array와 마찬가지로 대괄호 사용 ①
var names: Set<String> = ["yagom", "chulsoo", "younghee", "yagom"]

// 그렇기 때문에 타입 추론을 사용하게 되면 컴파일러는 Set가 아닌 Array로 타입을 지정한다.
var numbers = [100, 200, 300]
print(type(of: numbers)) // Array<Int>

print(names.isEmpty)        // false
print(names.count)          // 3 - 중복된 값은 허용되지 않아 yagom은 1개만 남는다.
```

• 세트의 활용 - 집합연산

```
let englishClassStudents: Set<String> = ["john", "chulsoo", "yagom"]           // ④
let koreanClassStudents: Set<String> = ["jenny", "yagom", "chulsoo",
                                         "hana", "minsoo"] // ④

// ④ 교집합 {"yagom", "chulsoo"}
let intersectSet: Set<String> =
englishClassStudents.intersection(koreanClassStudents)

// ④ 차집합 {"john", "jenny", "hana", "minsoo"} - {"yagom", "chulsoo", "hana", "minsoo"} // ④
let symmetricDiffSet: Set<String> =
englishClassStudents.symmetricDifference(koreanClassStudents)

// ④ 합집합 {"minsoo", "jenny", "john", "yagom", "chulsoo", "hana"} - {"yagom", "chulsoo", "hana", "minsoo"} // ④
let unionSet: Set<String> = englishClassStudents.union(koreanClassStudents)

// ④ 차집합 {"john"} - {"yagom", "chulsoo", "hana", "minsoo"} // ④
let subtractSet: Set<String> =
englishClassStudents.subtracting(koreanClassStudents)

print(unionSet.sorted()) // ["chulsoo", "hana", "jenny", "john", "minsoo", "yagom"]
```

• 세트는 자신의 내부의 값들에 모두 원본함을 보장하고,
집합연산은 표현하고자 할때 유용하게 쓰일 수 있으며
두 세트의 교집합, 합집합 등을 인식하기에 매우 유용하다

• Tip

설명에서 일의 요소 추출과 위치

```
스위프트 4.2 버전에서 집합연산에서 일의 요소를 추출하는 randomElement() 메서드와 협력선의 요소를  
임의로 바꾸는 shuffle() 메서드가 추가되었습니다. 또 자신의 요소는 그대로 듣고 새로운 집합연산에 임의  
의 요소로 바꿔서 전달하는 shuffled() 메서드도 추가되었습니다.

var array: [Int] = [0, 1, 2, 3, 4]
var set: Set<Int> = [0, 1, 2, 3, 4]
var dictionary: [String: Int] = ["a": 1, "b": 2, "c": 3]
var string: String = "string"

print(array.randomElement()) // 일의 요소
print(array.shuffle()) // 뒤섞여진 배열 [4, 2, 3, 1, 0] - array 내부의 요소는 그  
대로입니다.

array.shuffle() // [0, 1, 2, 3, 4]
array.shuffled() // array 자체를 뒤섞여버리게 되기
print(array) // 뒤섞여진 배열 [0, 4, 3, 2, 1]

print(set.randomElement()) // 세트를 뒤섞여버리게 해주세요 전환해줍니다.
// set.shuffle() // 오류 발생. 세트는 순서가 있기 때문에 스스로 뒤섞을 수 없습니다.
print(dictionary.shuffle()) // 딕셔너리를 뒤섞으면 키, 값이 영수 이론 품질의 배  
열로 전환해줍니다.
print(string.shuffle()) // String도 뒤섞여버립니다.
```

• 세트의 사용

```
print(names.count) // 3
names.insert("jenny") // ②
print(names.count) // 4

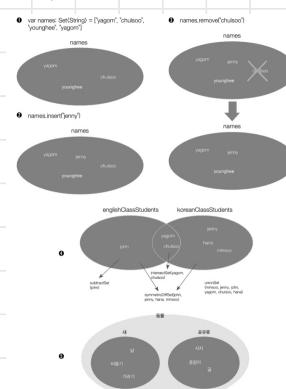
print(names.remove("chulsoo")) // chulsoo ③
print(names.remove("john")) // nil
```

• 세트의 활용 - 포함관계 연산

```
let set: Set<String> = ["박기", "김", "기리기"]
let 포함: Set<String> = ["사자", "호랑이", "곰"]
let 동물: Set<String> = 새.union(포유류) // 새와 포유류의 합집합 // ⑤

print(새.isDisjoint(with: 포함)) // 서로 베타적이지 - true
print(새.isSubset(of: 동물)) // 새가 동물의 부분집합인가요? - true
print(동물.isSuperSet(of: 포함)) // 동물은 포유류의 전체집합인가요? - true
print(동물.isSuperSet(of: 새)) // 동물은 새의 전체집합인가요? - true
```

• Set 모색



TIP **컬렉션에서 임의의 요소 추출과 뒤섞기**

스위프트 4.2 버전에서 컬렉션에서 임의의 요소를 추출하는 `randomElement()` 메서드와 컬렉션의 요소를 임의로 뒤섞는 `shuffle()` 메서드가 추가되었습니다. 또, 자신의 요소는 그대로 두 채 새로운 컬렉션에 임의의 순서로 쉬어서 반환하는 `shuffled()` 메서드도 추가되었습니다.

```
var array: [Int] = [0, 1, 2, 3, 4]
var set: Set<Int> = [0, 1, 2, 3, 4]
var dictionary: [String: Int] = ["a": 1, "b": 2, "c": 3]
var string: String = "string"

print(array.randomElement()) // 임의의 요소
print(array.shuffled()) // 뒤섞박죽된 배열 [4, 2, 3, 1, 0] - array 내부의 요소는 그 대로 있습니다.
print(array) // [0, 1, 2, 3, 4]
array.shuffle() // array 자체를 뒤섞박죽으로 뒤섞기
print(array) // 뒤섞박죽된 배열 [0, 4, 3, 2, 1]

print(set.shuffled()) // 세트를 뒤섞으면 배열로 반환해줍니다.
//set.shuffle() // 오류 발생! 세트는 순서가 없기 때문에 스스로 뒤섞을 수 없습니다.
print(dictionary.shuffled()) // 딕셔너리를 뒤섞으면 키, 값이 다른 키값의 배열로 반환해줍니다.
print(string.shuffled()) // String도 컬렉션입니다
```

4.5 - 열거형

- 연관된 항목들을 묶어서 표현하는 `Type`.
- `Array`나 `Dictionary`와는 다르게 프로그래밍에 정해진 항목 값 외에는 추가 / 수정이 불가능.

- 이렇게 사용

- 제한된 선택지를 주고 싶을 때
- 정해진 값 외에는 입력 받고 싶지 않을 때
- 예상된 입력 값이 한정되어 있을 때

- 다른 언어와는 다르게 Swift의 열거형은
- 항목별로 값을 가질 수도, 가지치않을 수도 있음.

또한 기존의 C언어 등에서의 열거형은 주로 정수 값 `Type`의 보조 형태로 사용되며
여러 열거형을 사용할 때 프로그래머의 실수로 인한 버그가 생길 수도 있었지만,
Swift의 열거형은 각 열거형이 고유의 `Type`으로 인식되어 버그를 놓쳐날 수 있다.

ex)

- 무선통신 방식 : WIFI, 블루투스, LTE, 3G, 기타
- 학생 : 초등학생, 중학생, 고등학생, 대학생, 대학원생, 기타
- 지역 : 강원도, 경기도, 경상도, 전라도, 제주도, 충청도

- **Raw Value** : 원시값 (정수, 실수, 문자열 등)을 가질 수 있음

- **Associated Value** : 연관 값을 사용하여 다른 언어에서 공용체라고 불리는 값의 묶음을 구현할 수 있다.

- 열거형은 `switch` 와의 궁합이 좋다.

TIP 열거형과 음서널

스위프트의 주요 기능 중 하나인 음서널은 enum(열거형)으로 구현되어 있습니다. 이에 관한 내용은 음서널(8장)에서 더 자세히 알아보겠습니다.

4.5.1 - 기본 열거형

- enum으로 선언 가능

· School 열거형의 선언

```
enum School {
    case primary
    case elementary
    case middle
    case high
    case college
    case university
    case graduate
}
```

각각의 고유의 값

· 측정하여 선언

```
enum School {
    case primary, elementary, middle, high, college, university, graduate
}
```

· School 열거형 변수의 생성 및 값 변경

```
enum School {
    case primary, elementary, middle, high, college, university, graduate
}

var highestEducationLevel: School = School.university

// 위 코드와 정확히 같은 표현
var highestEducationLevel2: School = .university

// 같은 타입인 School 내부의 항목으로만 highestEducationLevel의 값을 변경해줄 수 있다.
highestEducationLevel = .graduate
```

· 열거형의 원시값 지정과 사용

```
enum School: String {
    case primary = "유치원"
    case elementary = "초등학교"
    case middle = "중학교"
    case high = "고등학교"
    case college = "대학"
    case university = "대학원"
    case graduate = "대학원"
}

let highestEducationLevel: School = School.university
print("저의 최종학력은 \(highestEducationLevel.rawValue)입니다.")
// 저의 최종학력은 대학교 졸업입니다.

enum WeekDays: Character {
    case mon = "월", tue = "화", thu = "목", fri = "금", sat = "토", sun = "일"
}

let today: WeekDays = WeekDays.fri
print("오늘은 \(today.rawValue)입니다.") // 오늘은 금요일입니다.
```

· 열거형의 원시값 일부 지정 및 작동처리

```
enum School: String {
    case primary = "유치원"
    case elementary = "초등학교"
    case middle = "중학교"
    case high = "고등학교"
    case college
    case university
    case graduate
}

let highestEducationLevel: School = School.university
print("저의 최종학력은 \(highestEducationLevel.rawValue)입니다.")
// 저의 최종학력은 university 졸업입니다.

print(School.elementary.rawValue) // 초등학교

enum Numbers: Int {
    case zero
    case one
    case two
    case ten = 10
}

print("\(Numbers.zero.rawValue), \(Numbers.one.rawValue), \(Numbers.two.rawValue), * + "
    "\ \(Numbers.ten.rawValue)") // 0, 1, 2, 10
```

· 원시값을 통한 열거형 초기화

```
let primary = School(rawValue: "유치원") // primary
let graduate = School(rawValue: "석박사") // nil

let one = Numbers(rawValue: 1) // 1
let three = Numbers(rawValue: 3) // nil
```

- 몇바지 않은 원시값을 통해 생성하려면 nil을 return 함.

