4.4.1 배열

배열은 같은 타입의 데이터를 일렬로 나열한 후 순서대로 저장하는 형태의 컬렉션 타입입니다. 각기 다른 위치에 같은 값이 들어갈 수도 있음을 알아두세요.

배열 타입을 선언해줄 방법은 다양합니다. let 키워드를 사용해 상수로 선언하면 변경할 수 없는 배열이 되고, var 키워드를 사용해 변수로 선언해주면 변경 가능한 배열이 됩니다. 실제로 배열을 사용할 때는 Array라는 키워드와 타입 이름의 조합으로 사용합니다. 또, 대괄호로 값을 묶어 Array 타입임을 표현할 수도 있습니다. 빈 배열은 이니셜라이저 또는 리터럴 문법을 통해 생성해줄 수 있는데 isEmpty 프로퍼티로 비어있는 배열인지 확인해볼 수 있습니다. 그리고 배열에 몇 개의 요소가 존재하는지 알고 싶으면 count 프로퍼티를 확인하면 됩니다.

> **NOTE_ 스위프트의 Array**
>
> 스위프트의 Array는 C 언어의 배열처럼 버퍼Buffer입니다. 단, C 언어처럼 한 번 선언하면 크기가 고정되던 버퍼가 아니라, 필요에 따라 자동으로 버퍼의 크기를 조절해주므로 요소의 삽입 및 삭제가 자유롭습니다. 스위프트는 이런 리스트 타입을 Array, 즉 배열이라고 표현합니다. 기존 언어의 배열과는 조금 다른 특성도 있지만 이 책에서도 Array를 배열이라고 표현하겠습니다.

[코드 4-6]과 [코드 4-7]의 각 줄에 표시된 번호는 [그림 4-1]의 각 번호에 해당하는 코드입니다(91쪽).

코드 4-6 배열의 선언과 생성

```
// 대괄호를 사용하여 배열임을 표현합니다. ①
var names: Array<String> = ["yagom", "chulsoo", "younghee", "yagom"]

// 위 선언과 정확히 동일한 표현입니다. [String]은 Array<String>의 축약 표현입니다.
var names: [String] = ["yagom", "chulsoo", "younghee", "yagom"]

var emptyArray: [Any] = [Any]()      // Any 데이터를 요소로 갖는 빈 배열을 생성합니다.
var emptyArray: [Any] = Array<Any>() // 위 선언과 정확히 같은 동작을 하는 코드입니다.

// 배열의 타입을 정확히 명시해줬다면 []만으로도 빈 배열을 생성할 수 있습니다.
var emptyArray: [Any] = []
print(emptyArray.isEmpty)   // true
print(names.count)          // 4
```

배열은 각 요소에 인덱스를 통해 접근할 수 있습니다. 인덱스는 0부터 시작합니다. 잘못된 인덱스로 접근하려고 하면 익셉션 오류Exception Error가 발생합니다. 또, 맨 처음과 맨 마지막 요소는 first와 last 프로퍼티를 통해 가져올 수 있습니다. index(of:) 메서드를 사용하면 해당 요소의 인덱스를 알아낼 수도 있습니다. 만약 중복된 요소가 있다면 제일 먼저 발견된 요소의 인덱스를 반환*합니다. 맨 뒤에 요소를 추가하고 싶다면 append(_:) 메서드를 사용합니다.

중간에 요소를 삽입하고 싶다면 insert(_:at:) 메서드를 사용하면 됩니다. 요소를 삭제하고 싶다면 remove(_:) 메서드를 사용하게 되는데, 메서드를 사용하면 해당 요소가 삭제된 후 반환됩니다.

코드 4-7 배열의 사용

```
print(names[2])          // younghee
names[2] = "jenny"       // ②
print(names[2])          // jenny
print(names[4])          // 인덱스의 범위를 벗어났기 때문에 오류가 발생합니다.

names[4] = "elsa"        // 인덱스의 범위를 벗어났기 때문에 오류가 발생합니다.
names.append("elsa")     // 마지막에 elsa가 추가됩니다. ③
names.append(contentsOf: ["john", "max"])    // 맨 마지막에 john과 max가 추가됩니다. ④
names.insert("happy", at: 2)                 // 인덱스 2에 삽입됩니다. ⑤
// 인덱스 5의 위치에 jinhee와 minsoo가 삽입됩니다. ⑥
names.insert(contentsOf: ["jinhee", "minsoo"], at: 5)

print(names[4])                                  // yagom
print(names.index(of: "yagom"))                  // 0
print(names.index(of: "christal"))               // 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", "jinhee"]
```

* 반환에 관한 내용은 함수와 메서드(152쪽)를 참고하세요.

그림 4-1 [코드 4-6]과 [코드 4-7]의 names Array 모식도

[코드 4-7]의 맨 아래 줄의 `names[1 ... 3]` 표현은 범위 연산자(109쪽)를 사용하여 `names` 배열의 일부만 가져온 것입니다. 코드처럼 읽기만 가능한 것이 아니라 `names[1 ... 3] = ["A", "B", "C"]`와 같이 범위에 맞게 요소를 바꾸는 것도 가능합니다.

스위프트의 배열을 비롯한 컬렉션 타입을 활용할 때 서브스크립트Subscript 기능을 많이 사용합니다. 서브스크립트 문법은 나중에 조금 더 자세히 다루겠습니다(301쪽). 조금 더 재미있고 다양한 배열의 활용 방법은 반복문을 배울 때 다루겠습니다(136쪽).

4.4.2 딕셔너리

딕셔너리는 요소들이 순서 없이 키와 값의 쌍으로 구성되는 컬렉션 타입입니다. 딕셔너리에 저장되는 값은 항상 키와 쌍을 이루게 되는데, 딕셔너리 안에는 키가 하나이거나 여러 개일 수 있습니다. 단, 하나의 딕셔너리 안의 키는 같은 이름을 중복해서 사용할 수 없습니다. 쉽게 말해서 [코드 4-8]에서 `"yagom"`이라는 키가 두 번 쓰일 수 없다는 뜻입니다. 즉, 딕셔너리에서 키는 값을 대변하는 유일한 식별자가 되는 것입니다.

딕셔너리는 `Dictionary`라는 키워드와 키의 타입과 값의 타입 이름의 조합으로 써줍니다. 대괄호로 키와 값의 타입 이름의 쌍을 묶어 딕셔너리 타입임을 표현합니다. `let` 키워드를 사용하여 상수로 선언하면 변경 불가능한 딕셔너리가 되고, `var` 키워드를 사용하여 변수로 선언해주면 변경 가능한 딕셔너리가 됩니다. 빈 딕셔너리는 이니셜라이저 또는 리터럴 문법을 통해 생성할 수 있습니다. `isEmpty` 프로퍼티를 통해 비어있는 딕셔너리인지 확인할 수 있습니다. 그리고 `count` 프로퍼티로 딕셔너리의 요소 개수를 확인할 수 있습니다.

[코드 4-8]과 [코드 4-9]의 각 줄에 표시된 번호는 [그림 4-2]의 각 번호에 해당하는 코드입니다(94쪽). [코드 4-8]에 사용된 `typealias` 키워드는 타입 별칭 키워드입니다. 타입 별칭에 대한 자세한 설명은 타입 별칭(86쪽)에 있습니다.

코드 4-8 딕셔너리의 선언과 생성

```
// typealias를 통해 조금 더 단순하게 표현해볼 수도 있습니다.
typealias StringIntDictionary = [String: Int]

// 키는 String, 값은 Int 타입인 빈 딕셔너리를 생성합니다.
var numberForName: Dictionary<String, Int> = Dictionary<String, Int>()
```

```
// 위 선언과 같은 표현입니다. [String: Int]는 Dictionary<String, Int>의 축약 표현입니다.
var numberForName: [String: Int] = [String: Int]()

// 위 코드와 같은 동작을 합니다.
var numberForName: StringIntDictionary = StringIntDictionary()

// 딕셔너리의 키와 값 타입을 정확히 명시해줬다면 [:]만으로도 빈 딕셔너리를 생성할 수 있습니다.
var numberForName: [String: Int] = [:]

// 초깃값을 주어 생성해줄 수도 있습니다. ①
var numberForName: [String: Int] = ["yagom": 100, "chulsoo": 200, "jenny": 300]

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

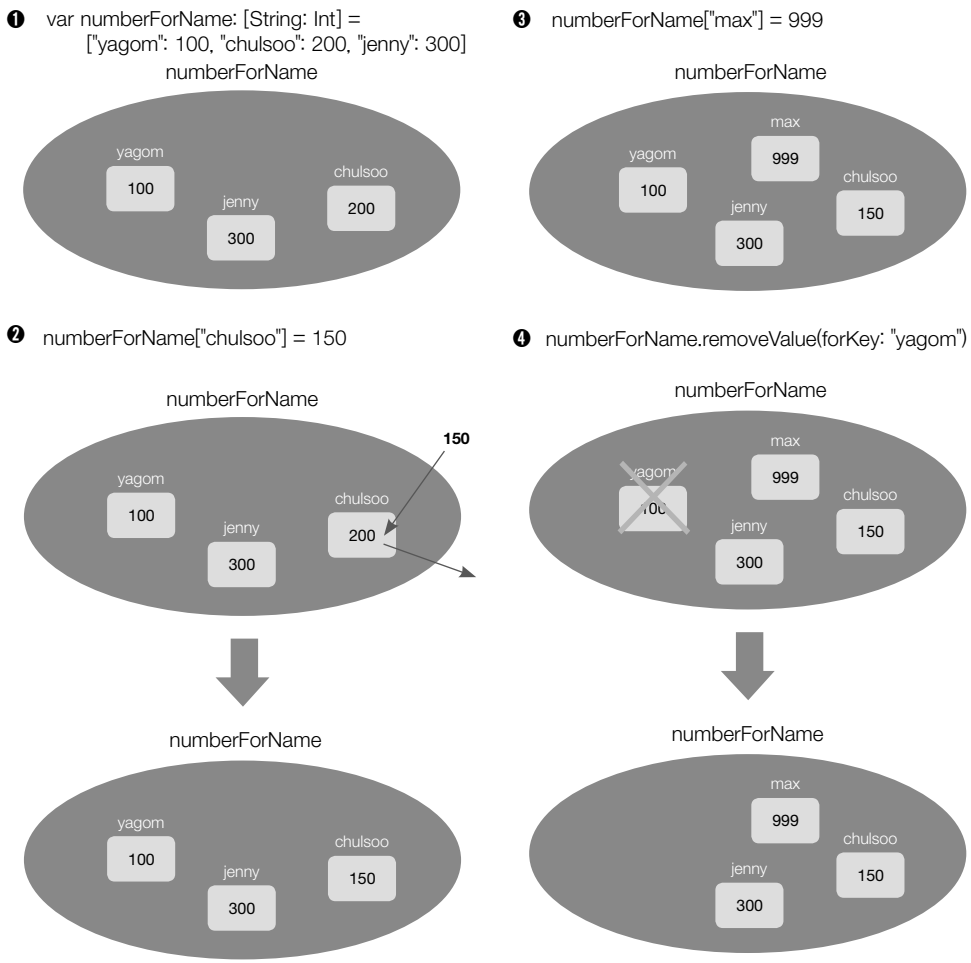
딕셔너리는 각 값에 키로 접근할 수 있습니다. 딕셔너리 내부에서 키는 유일해야 하며, 값은 유일하지 않습니다. 딕셔너리는 배열과 다르게 딕셔너리 내부에 없는 키로 접근해도 오류가 발생하지 않습니다. 다만 `nil`을 반환할 뿐이죠. 특정 키에 해당하는 값을 제거하려면 `removeValue(forKey:)` 메서드를 사용합니다. 키에 해당하는 값이 제거된 후 반환됩니다.

코드 4-9 딕셔너리의 사용

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

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

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

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

// 위에서 yagom 키에 해당하는 값이 이미 삭제되었으므로 nil이 반환됩니다.
// 키에 해당하는 값이 없으면 기본값을 돌려주도록 할 수도 있습니다.
print(numberForName.removeValue(forKey: "yagom"))

// yagom 키에 해당하는 값이 없으면 기본으로 0이 반환됩니다.
print(numberForName["yagom", default: 0])  // 0
```

조금 더 재미있고 다양한 딕셔너리의 활용 방법은 반복문에서 다룹니다(138쪽).

그림 4-2 [코드 4-8]과 [코드 4-9]의 numberForName Dictionary 모식도



4.4.3 세트

세트는 같은 타입의 데이터를 순서 없이 하나의 묶음으로 저장하는 형태의 컬렉션 타입입니다. 세트 내의 값은 모두 유일한 값, 즉 중복된 값이 존재하지 않습니다. 그래서 세트는 보통 **순서가 중요하지 않거나 각 요소가 유일한 값이어야 하는 경우**에 사용합니다. 또, 세트의 요소로는 **해시 가능한 값***이 들어와야 합니다.

* 스위프트 표준 라이브러리의 Hashable 프로토콜을 따른다는 것을 의미합니다. 스위프트의 기본 데이터 타입은 모두 해시 가능한 값입니다.