# Swift 문법

## 1. 변수 상수

var + 변수명
let + 상수명

## 2. 연산자

### 범위 연산자

닫힌 범위 연산자(Closed range operator)
1...5 (1, 2, 3, 4, 5)

반 닫힌 범위 연산자(Half-closed range operator)
1..<5 (1, 2, 3, 4)

1>..5 (X)
범위 연산자의 왼쪽에는 작은 숫자를 오른쪽에는 큰 숫자를 배치

### 대입 연산자(assignment operator)

a = 1
a++ (X)
a += 1 (O)

# 3. 흐름 제어 구문

반복문

for ~ in 구문

```
for <루프 상수> in <순회 대상> {
   <실행할 구문>
}
```

while 구문 / repeat ~ while 구문

```
while <조건식> {
   <실행할 구문>
}

repeat {
   <실행할 구문>
}
while <조건식>
```

조건문

if 구문

```
if <조건식> {
   <실행할 구문>
} else if <조건식> {
   <실행할 구문>
} else {
   <실행할 구문>
}
```

guard 구문 (※ 코드를 중첩해서 사용하지 않아도 된다는 장점 (코드의 깊이가 깊어지지 않음))

```
guard <조건식 또는 표현식> else {
   <조건식 또는 표현식의 결과가 false일 때 실행될 코드>
}
```

#available 구문

```
if #available(<플레폼이름 버전>, <...>, <*> {
   <해당 버전에서 사용할 수 있는 API 구문>
} else {
   <API를 사용할 수 없는 환경에 대한 처리>
}
if #available <iOS 10.0, OSX 10.10, watchOS 1, *) { }
```

switch 구문

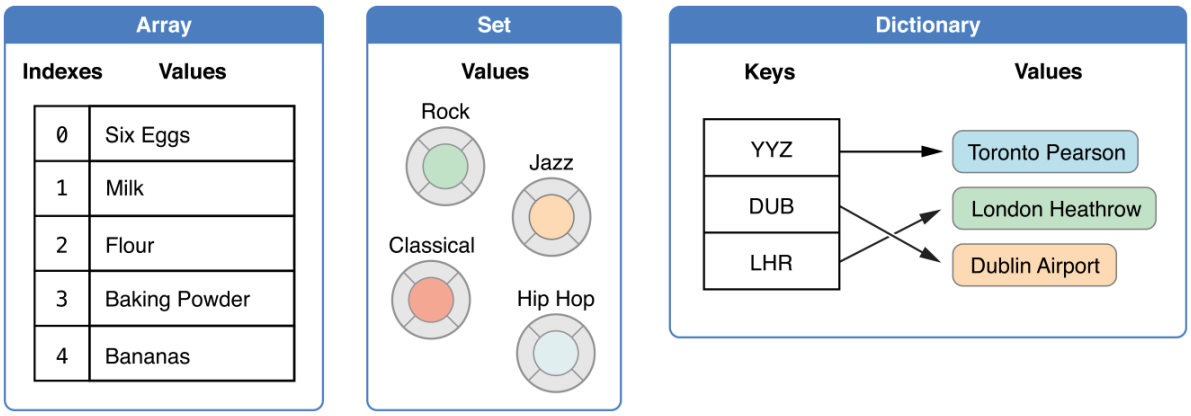
```
switch <비교 대상> {
   case <비교 패턴1>:
      <비교 패턴1이 일치했을 때 실행할 구문>
   case <비교 패턴2>, <비교 패턴3>: //fallthrough로 표현 가능
      <비교 패턴2 또는 3이 일치했을 때 실행할 구문>
   default:
      <어느 비교 패턴과도 일치하지 않았을 때 실행할 구문>
}
```

구문 레이블과 break, continue

```
<레이블 이름> : for <루프 상수> in <순회 대상> {
   <실행할 구문>
}

break < 레이블 이름>
continue <레이블 이름>
```

# 4. 집단 자료형(Collective Types)

| | |
|---|---|
| 배열(Array) | 일련번호로 구분되는 순서에 따라 데이터가 정렬된 목록 형태 |
| 집합(Set) | 중복되지 않은 유일 데이터들이 순서 없이 모인 집합 형태 |
| 튜플(Tuple) | 종류에 상관없이 데이터들을 모은 자료형. 값을 변경할 수 없음 |
| 딕션너리(Dictionary) | 키-값으로 연관된 데이터들이 순서 없이 모인 자료형 |

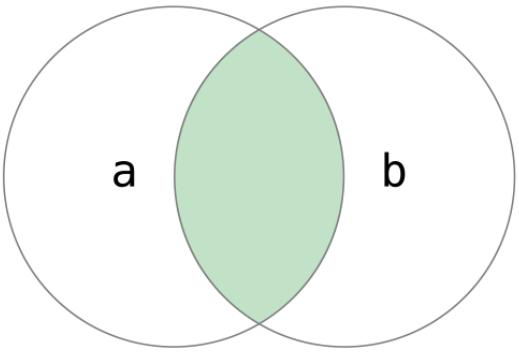



### 1) 배열(Array)

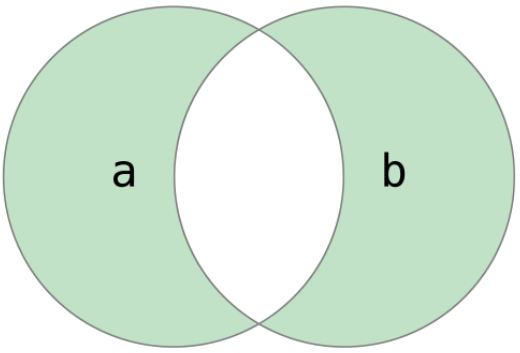
```
정적 선언
let countries = ["korea", "japan", "china"]  as Array<String>
let countries = ["korea", "japan", "china"]  as [String]
let countries: Array<String> = ["korea", "japan", "china"]
let countries: [String] = ["korea", "japan", "china"]
let countries = ["korea", "japan", "china"]

동적 선언
var countries: Array<String> = Array<String>()
var countries: [String] = [String]()
var countries: [String] = Array()
var countries = [String]()
var countries: [String] = []
```

2) 집합(Set)

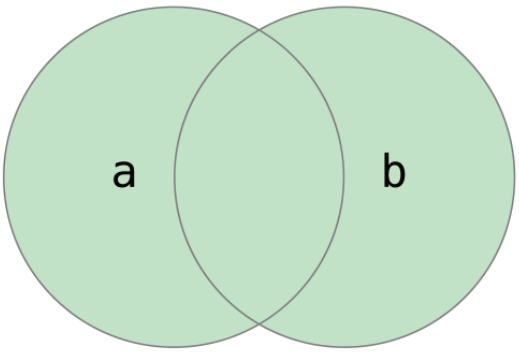
a.intersection(b)



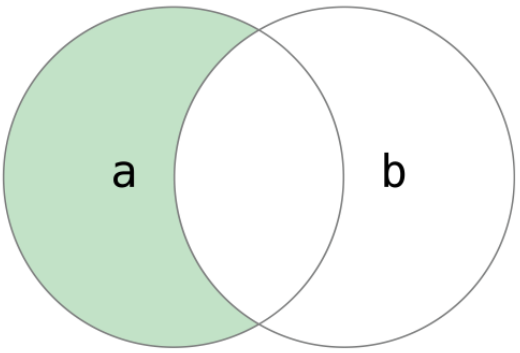

a.symmetricDifference(b)




a.union(b)




a.subtracting(b)




※ 타입 어노테이션이 없다면 배열로 선언되기 때문에 명시적으로 타임 어노테이션 Set을 붙여야함

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

### 3) 튜플(Tuple)

```
let tupleValue = ("a", "b", 1, 2.5, true)
tupleValue.0 // "a"
tupleValue.1 // "b"
tupleValue.2 // 1
tupleValue.3 // 2.5
tupleValue.4 // true

let tupleValue: (String, Character, Int, Float, Bool) = ("a",
"b", 1, 2.5, true)
let (a, b, c, d, e) = tupleValue

let tupleValue = (a: "a", b: "b", c: 1, d: 2.5, e: true)
tupleValue.a // "a"
tupleValue.b // "b"
tupleValue.c // 1
tupleValue.d // 2.5
tupleValue.e // true
```

### 4) 딕션너리(Dictionary)

```
정적 선언
let airports: Dictionary<String, String> = ["YYZ": "Toronto", "DUB": "Dublin"]
let airports: [String: String] = ["YYZ": "Toronto", "DUB": "Dublin"]

동적 선언
var airports: Dictionary<String, String> = Dictionary<String, String>()
var airports: [String: String] = [String: String]()
var airports: [String: String] = Dictionary()
var airports = [String: String]()
var airports: [String: String] = [:]
```

# 5. 옵셔널 (Optional)

- 스위프트에 도입된 새로운 개념으로 언어 차원에서 프로그램의 안전성을 높이기 위해 사용하는 개념
- nil을 사용할 수 있는 타입과 사용할 수 없는 타입을 구분
- nil이란 값이 없음을 의미하는 특수한 값

```
var optInt: Int?
optInt = 3
```

옵셔널 해제 및 옵셔널 바인딩

1) if 조건 사용

```
func intStr(str: String) {
    if let intFromStr = Int(str) {
        print("값이 변환되었습니다. 변환된 값은 \(intFromStr)입니다")
    } else {
        print("값 변환에 실패하였습니다")
    }
}
```

2) guard 구문 사용

```
func intStr(str: String) {
    guard let intFromStr = Int(str) else {
        print("값 변환에 실패하였습니다")
        return
    }
    print("값이 변환되었습니다. 변환된 값은 \(intFromStr)입니다")
}
```

3) ?? 사용

```
func intStr(str: String) {
    let intFromStr = Int(str) ?? 0
    print("값이 변환되었습니다. 변환된 값은 \(intFromStr)입니다")
}
```

4) 강제 해제 사용

```
func intStr(str: String) {
    let intFromStr = Int(str)!
    print("값이 변환되었습니다. 변환된 값은 \(intFromStr)입니다")
}
```

# 6. 함수(Function)

스위프트는 함수형 프로그래밍 패러다임을 채택하고 있는 언어이므로 함수형 프로그래밍의 특성을 이해하는 것이 매우 중요

```
func 함수이름(매개변수1: 타입, 매개변수2: 타입, ...) -> 반환타입 {
    실행내용
    return 반환값
}
함수의 식별자 - 함수이름(매개변수1:매개변수2:)
```

함수의 호출

```
func increment(amount: Int, times: Int) {
    print(amount * times)
}
함수의 식별자 - increment(amount:times:)

increment(amount: 5, times: 2)
(Objective-C 메소드 호출 방식의 API를 수정 없이 그대로 사용하기 위해 최대한 형식을 맞춘 결과, 지금과 같은 독특한 문법이 만들어짐)
```

Objective-C 함수

```
Objective-c
- (void)incrementAmount:(NSInteger)amount
times:(NSInteger)times{}

[self incrementAmount: 5 times: 2];

Swift 변환환 모습
func increment(amount: Int, times: Int) { }
```

내부 매개변수명, 외부 매개변수명

```
func 함수이름(<외부 매개변수명> <내부 매개변수명>: 타입) {
   // 함수의 내용이 작성되는 곳
}
함수의 식별자 - 함수이름(외부 매개변수명:)

func increment(by amount: Int, time times: Int) {
    print(amount * times)
}
함수의 식별자 - increment(by:time:)

increment(by: 5, time: 2)

func increment(_ amount: Int, _ times: Int) {
    print(amount * times)
}
함수의 식별자 - increment(_:_:)

increment(5, 2)
```

가변인자

```
func 함수이름(매개변수명: 타입 ...)

func avg(score: Int...) -> Double {
    var total = 0
    for r in score {
        total += r
    }
    return Double(total) / Double(score.count)
}

print(avg(score: 10,20,30,40))
```

기본값을 갖는 매개변수

```
func 함수이름(매개변수: 타입 = 기본값)

func increment(amount: Int, times: Int = 5) {
    print(amount * times)
}
increment(amount: 1)
```

# 7. 일급 객체 함수(First-Class Function)의 특성

1. 객체가 런타임에도 생성이 가능해야 한다.
2. 인자값으로 객체를 전달 할 수 있어야 한다.
3. 반환값으로 객체를 사용할 수 있어야 한다.
4. 변수나 데이터 구조 안에 저장할 수 있어야 한다.
5. 할당에 사용된 이름과 관계없이 고유한 구별이 가능해야 한다.

변수나 상수에 함수를 대입

```
func increment(base: Int) {
    print("결과값은 \(base + 1)입니다.")
}

let inc1 = increment(base: 1) //단순한 결과값 대입
//결과값은 2입니다.

let inc2 = increment(base:) //함수 자체를 변수에 할당 (함수의 식별자)
//출력결과 없음
inc2(2)
//결과값은 3입니다.

let inc3 = increment //함수 자체를 변수에 할당 (함수의 이름, 중복이 없어 가능)
//출력결과 없음
inc3(3)
//결과값은 4입니다.

함수 타입(Function Type)
(인자 타입1, 인자 타입2) -> 반환 타입
(Int) -> ()  ⇒ (Int) -> Void

let inc4: (Int) -> () = increment
inc4(4)
```

함수의 반환 타입으로 함수를 사용할 수 있음

```
func desc() -> String {
    return "this is desc()"
}

func pass() -> () -> String {
    return desc
}

let p = pass()
p()
```

함수의 인자값으로 함수를 사용할 수 있음 - 콜백 함수와 같은 개념

```
func incr(param: Int) -> Int {
    return param + 1
}

func broker(base: Int, function fn: (Int) -> Int) -> Int {
    return fn(base)
}

broker(base: 3, function: incr)
```

# 8. 클로저(Clousure)

컴퓨터 언어에서 클로저(Closure)는 일급 객체 함수(first-class functions)의 개념을 이용하여 스코프(scope)에 묶인 변수를 바인딩 하기 위한 일종의 기술이다. 기능상으로, 클로저는 함수를 저장한 레코드(record)이며, 스코프(scope)의 인수(Factor)들은 클로저가 만들어질 때 정의(define)되며, 스코프 내의 영역이 소멸(remove)되었어도 그에 대한 접근(access)은 독립된 복사본인 클로저를 통해 이루어질 수 있다.

※ 참조
https://ko.wikipedia.org/wiki/%ED%81%B4%EB%A1%9C%EC%A0%80_(%EC%BB%B4%ED%93%A8%ED%84%B0_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)

스위프트에서 클로저는 일회용 함수를 작성할 수 있는 구문

클로저 표현식

```
//일반 함수의 선언 형식에서 func 키워드와 함수명을 제외한 나머지 부분만 작성
{ (매개변수) -> 반환 타입 in
   실행할 구문
}

let f = { () -> Void in
    print("클로저가 실행됩니다.")
}
f()

let c = { (s1: Int, s2: String) -> Void in
    print("s1:\(s1), s2:\(s2)")
}
c(1, "clousre")
```

### 클로저 표현식 경량화

```
var value = [1, 9, 5, 7, 3, 2]
func order(s1: Int, s2: Int) -> Bool {
    return s1 > s2
}
value.sort(by: order)

// 함수 order를 클로저 표현식으로 작성
value.sort(by: { (s1: Int, s2: Int) -> Bool in
    return s1 > s2
})

// 반환값 표현 생략
value.sort(by: { (s1: Int, s2: Int) in return s1 > s2 })

// 매개변수의 타입 생략
value.sort(by: { s1, s2 in return s1 > s2 })

// 매개변수 생략
value.sort(by: { return $0 > $1 })

// return 구문 생략
value.sort(by: { $0 > $1 })
```

### 트레일링 클로저(Trailing Closure)

```
//함수의 마지막 인자값이 클로저일 때, 이를 인자값 형식으로 작성하는 대신
함수의 뒤에 꼬리처럼 붙일 수 있는 문법을 의미 (중괄호 블록 불필요)

value.sort { (s1, s2) -> Bool in
    return s1 > s2
}
```

@escaping

```
//인자값으로 전달된 클로저를 저장해 두었다가, 나중에 다른 곳에서도 실행 할 수 있도록 허용해주는 속성

//스위프트에서 함수의 인자값으로 전달된 클로저는 기본적으로
탈출불가(non-escape)의 성격을 가진다.
//즉, 클로저를 함수 내에서 직접 실행을 위해서만 사용해야 한다. (컴파일러가 코드 최적화 과정에서의 성능 향상 때문)

func callback(fn: @escaping () -> Void) {
    let f = fn
    f()
}

callback {
    print("Closuer가 실행되었습니다.")
}
```

@autoclosure

```
//@autoclosure 속성이 붙은 인자값은 컴파일러에 의해 '()' 일반 형식을 '{}' 클로저 형식으로 자동 래핑된다.

func condition(stmt: @autoclosure () -> Bool) {
    if stmt() == true {
        print("참")
    } else {
        print("거짓")
    }
}

condition(stmt: 4 > 2)
```

# 9. 구조체와 클래스

## 1) 구조체VS클래스

### 공통점

| | |
|---|---|
| 프로퍼티 | 변수나 상수를 사용하여 값을 저장하는 프로퍼티를 정의할 수 있다. |
| 메소드 | 함수를 사용하여 기능을 제공하는 메소드를 정의할 수 있다. |
| 서브스크립트 | 속성값에 접근할 수 있는 방법을 제공하는 서브스크립트를 정의할 수 있다. |
| 초기화 블록 | 객체를 원하는 초기 상태로 설정해주는 초기화 블록을 정의할 수 있다. |
| 확장 | 객체에 함수적 기능을 추가하는 확장(extends) 구문을 사용할 수 있다. |
| 프로토콜 | 특정 형식의 함수적 표준을 제공하기 위한 프로토콜을 구현할 수 있다. |

### 클래스만 가지고 있는 기능들

| | |
|---|---|
| 상속 | 클래스의 특성을 다른 클래스에서 물려줄 수 있다. |
| 타입 캐스팅 | 실행 시 컴파일러가 클래스 인스턴스의 타입을 미리 파악하고 검사할 수 있다. |
| 소멸화 구문 | 인스턴스가 소멸되기 직전에 처리해야 할 구문을 미리 등록해 놓을 수 있다. |
| 참조에 의한 전달 | 클래스 인스턴스가 전달될 때에는 참조 형식으로 제공되며, 이때 참조가 가능한 개수는 제약이 없다. |

## 2) 정의 구문

구조체의 정의 형식

```
struct 구조체_이름 {
    // 구조체 정의 내용이 들어갈 부분
}
```

클래스의 정의 형식

```
class 클래스_이름 {
    // 클래스 정의 내용이 들어갈 부분
}
```

카멜(Camel) 표기법

```
① 구주체와 클래스 이름의 첫 글자는 대문자로, 나머지는 글자는 소문자로 작성한다.
② 2개 이상의 복합 단어는 단어별로 끊어 첫 글자는 대문자로, 나머지는 소문자로 작성한다.
③ 이미 축약된 약어는 모두 대문자로 작성 가능하다.(ex. JSON, NS, HTTP 등)
④ 프로퍼티나 메소드는 선언할 때는 소문자로 시작한다.
⑤ 언더바로 단어를 연결하는 방식은 지양한다.
```

## 3) 메소드와 프로퍼티

프로퍼티(Property) (또는 속성) - 구조체와 클래스 내부에서 정의된 변수나 상수
메소드(Method) - 구조체와 클래스 내부에서 정의된 함수

```
struct Resolution {
    var width = 0
    var height = 0

    func desc() -> String {
        return "Resolution 구조체"
    }
}
```

```
class VideoMode {
    var interlaced = false
    var frameRate = 0.0
    var name: String?

    var res = Resolution()

    func desc() -> String {
        return "VideoMode 클래스"
    }
}
```

## 4) 인스턴스(Instance)

타입의 설계도를 사용하여 메모리 공간을 할당받은 것

```
// Resolution 구조체에 대한 인스턴스를 생성하고 상수 insRes에 할당
let insRes = Resolution()

// VideoMode 클래스에 대한 인스턴스를 생성하고 상수 insVMode에 할당
let insVMode = VideoMode()
```

프로퍼티 접근

```
프로퍼티에 접근할려면 반드시 인스턴스를 먼저 생성해야 한다.
프로퍼티에 접근할 때는 점 문법(Dot Syntax)을 이용하여 인스턴스의 하위
객체에 접근 할 수 있다.

<인스턴스 이름>.<프로퍼티 이름>

let width = insRes.width
```

서브 프로퍼티 접근

```
<인스턴스 이름>.<프로퍼티 이름>.<프로퍼티의 서브 프로퍼티 이름>

let width = insVMode.res.width

이러한 방식을 사슬이 계속 연결되는 방식과 비슷하다 하여 체인(Chain)이라
한다.
```

## 5) 초기화

스위프트에서 옵셔널 타입으로 선언되지 않은 모든 프로퍼티는 명시적으로 초기화해 주어야 한다.

명시적 초기화

```
1. 프로퍼티를 선언하면서 동시에 초기값을 지정하는 경우
2. 초기화 메서드 내에서 프로퍼티의 초기값을 지정하는 경우
```

구조체 초기화

```
Resolution() //기본 초기화 구문. 내부적으로 프로퍼티를 초기화하지 않음
Resolution(width: Int, height: Int) //모든 프로퍼티의 초기값을 입력받는 멤버와이즈 초기화 구문(Memberwise Initializer). 내부적으로 모든 프로퍼티를 초기화함
```

## 6) 값 전달 방식

구조체의 값 전달 방식 : 복사에 의한 전달

클래스의 값 전달 방식 : 참조에 의한 전달

```
동일 인스턴스인지 비교할 때 : ===
동일 인스턴스가 아닌지 비교할 때 : !==
```

어떤 경우 구조체를 사용할까?

```
1. 서로 연관된 몇 개의 기본 데이터 타입들을 캡슐화하여 묶는 것이 목적일 때
2. 캡슐화된 데이터에 상속이 필요하지 않을 때
3. 캡슐화된 데이터를 전달하거나 할당하는 과정에서 참조 방식보다는 값이 복사되는 것이 합리적일 때
4. 캡슐화된 원본 데이터를 보존해야 할 때
```

## 7) 프로퍼티(Property)

클래스나 구조체에서 프로퍼티가 하는 정확한 역할은 값을 제공하는 것이다.

저장 프로퍼티(Stored Property)

```
입력된 값을 저장하거나 저장된 값을 제공하는 역할
상수 및 변수를 사용해서 정의 가능
클래스와 구조체에서는 사용이 가능하지만, 열거형에서는 사용할 수 없음
```

① 초기값 할당

```
입력된 값을 저장하거나 저장된 값을 제공하는 역할
상수 및 변수를 사용해서 정의 가능
클래스와 구조체에서는 사용이 가능하지만, 열거형에서는 사용할 수 없음

초기값 할당
1. 초기화 구문을 작성하고, 그 안에서 초기값을 할당
class User {
    var name: String

    init() {
        self.name = ""
    }
}


2. 프로퍼티를 옵셔널 타입으로 설정
class User {
    var name: String?
}

class User {
    var name: String! //묵시적 옵셔널 해제 타입 (지양)
}

3. 프로퍼티에 초기값을 할당
class User {
    var name: String = ""
}
```

② 저장 프로퍼티의 분류

```
var : 변수형 저장 프로퍼티(멤버 변수)
let : 상수형 저장 프로퍼티(멤버 상수)

구조체 - 저장 프로퍼티의 값이 바뀌면 상수에 할당된 인스턴스 전체가 변경
클래스 - 저장 프로퍼티의 값이 바뀌더라도 상수에 할당된 인스턴스의 레퍼런스는
변경 안됨
```

③ 지연 저장 프로퍼티

```
lazy 키워드가 붙음
호출되기 전에는 선언만 된 상태로 대기하다가 실제로 호출되는 시점에서
초기화가 이루어진다.
```

④ 클로저를 이용한 저장 프로퍼티 초기화

```
연산이나 로직 처리를 통해 얻어진 값을 이용하여 초기화 할 경우 사용

let/var 프로퍼티명: 타입 = {
    정의 내용
    return 반환값
}()

초기값을 반환하고, 이후로는 재실행되지 않음.


lazy 키워드를 붙여서 참조되는 시점에 초기화되기 때문에 메모리 낭비를 줄일 수 있어 여러 용도로 활용된다.
```

### 연산 프로퍼티(Computed Property)

```
특정 연산을 통해 값을 만들어 제공하는 역할
변수만 사용해서 정의 가능
클래스, 구조체, 열거형 모두에서 사용 가능

class/struct/enum 객체명 {
    …
    var 프로퍼티명: 타입 {
        get { // 필수
            필요한 연산 과정
            return 반환값
        }
        set(매개변수명(생략시 newValue기본 사용됨)) { // 생략 가능
            필요한 연산구문
        }
    }
}

// set 생략시 - 외부에서 값 할당 불가능, 읽기 전용 프로퍼티(read-only or get-only)
```