공식문서 읽기 : Properties

```
@propertyWrapper
struct SmallNumber {
private var number : Int
var projectedValue : Bool
init() {
self.number = 0
self.projectedValue = false
}
var wrappedValue: Int {
get {
return number
}
set {
if newValue > 12 {
number = 12
projectedValue = true
```

```
} else {
number = newValue
projectedValue = false
}
}
}
}
struct SomeStructure {
@SmallNumber var someNumber: Int
}
var someStructure = SomeStructure()
someStructure.someNumber = 4
print(someStructure.$someNumber)
// false
someStructure.someNumber = 55
print(someStructure.$someNumber)
// true
```

> Properties - The Swift Programming Language (Swift 5.5)
> Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and
> https://docs.swift.org/swift-book/LanguageGuide/Properties.html

프로퍼티(Property)는 클래스, 구조체, 열거형과 연관된 값이다. 저장 프로퍼티(stored property)는 인스턴스의 일부로 상수(let)나 변수(var)를 저장한다. 계산 프로퍼티(computed property)는 값을 계산하는 기능을 한다.

- 저장 프로퍼티는 클래스와 구조체에서만 사용 가능

- 계산 프로퍼티는 클래스와 구조체, 열거형에서 모두 사용 가능

프로퍼티 옵저버를 정의해 프로퍼티의 값 변화를 관찰할 수 있다. 옵저버는 저장된 프로퍼티, 또는 서브 클래스가 부모 클래스로부터 상속받은 프로퍼티에 더해질 수 있다.

프로퍼티의 getter와 setter 코드를 재사용하기 위해 프로퍼티 래퍼를 사용할 수도 있다.

1. Stored Properties (저장 프로퍼티)

가장 간단한 형식으로 저장된 속성은 특정 클래스, 구조의 인스턴스의 일부로 저장되는 상수(let), 변수(var) 저장 속성이다.

저장 프로퍼티를 정의할 때 기본값을 할당할 수 있다. 초기화 과정에서 저장 프로퍼티에 초기 값을 설정하거나 수정하는 것도 가능하다. 이는 상수 저장 프로퍼티에서도 허용된다.

```
struct FixedLengthRange {
  var firstValue: Int
  let length: Int
}

var rangeOfThreeItems = FixtedLengthRange(firstValue: 0, length: 3) // 0, 1, 2
rangeOfThreeItems.firstValue = 6 // 6, 7, 8
```

FixedLengthRange의 인스턴스는 변수 저장 프로퍼티인 firstValue와 상수 저장 프로퍼티인 length를 가지고 있다. 위의 예시에서 length는 새로운 인스턴스가 만들어질 때 초기화되고 이후에는 수정이 불가능하다.

Stored Properties of Constant Structure Instances (상수 구조체 인스턴스의 저장 프로퍼티)

구조체의 인스턴스를 상수에 할당하면, 변수 속성으로 선언된 경우에도 인스턴스의 속성을 수정할 수 없다.

```
let rangeOfFourItems = FixedLengthRange(firstVaelue: 0, length: 4)
rangeOfFourItems.fistValue = 6 // error
```

rangeOfFourItems은 상수로 선언됐으며, firstValue 프로퍼티가 변수로 선언되었음에도 바꿀 수 없다. 구조체는 값 타입이기 때문이다. 값 타입의 인스턴스가 상수로 지정되면 그 인스턴스의 모든 프로퍼티도 상수가 된다.

클래스는 참조 타입이기 때문에 구조체와 다르다. 참조 타입을 상수로 선언해도 인스턴스의 변수 프로퍼티는 변경 가능하다.

Lazy Stored Properties (지연 저장 프로퍼티)

Lazy Stored Properties는 최초 사용 전까지 초기값이 없는(계산되지 않은 상태) 프로퍼티이다. 선언부 앞에 lazy 키워드를 작성하여 지연 저장 속성을 만든다.

> 💡 지연 저장 속성은 반드시 변수(var)로 선언해야 한다. 인스턴스 초기화가 완료된 후 검색되기 때문이다. 상수(let)는 반드시 초기화 완료 전에 값을 가지고 있어야 하기 때문에 상수로 선언할 수 없다.

지연 저장 속성은 속성의 초기 값이 인스턴스 초기화 완료될 때까지 **알 수 없는 외부 요소**에 **의존적**일 때 유용하다. 복잡하거나 계산 비용이 높지만 필요해질 때까지 작동하지 않는 속성의 초기값을 다룰 때도 좋다.

```
class DataImporter {
  var filename = "data.txt"
}

class DatatManager {
  lazy var importer = DataImporter() // 지연 저장 속성
  var data = [String]()
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
```

DataManager 클래스는 빈 문자열 배열로 초기화되는 data 저장 프로퍼티를 가지고 있다.

DataManager의 목적은 문자열 데이터 배열에 대한 접근 또는 관리를 제공하는 것이다.

DataManager 클래스 기능의 일부는 파일로부터 데이터를 불러오는 것이다. 이 기능은 초기화하는 데 어느정도의 시간이 필요한 DataImporter 클래스로부터 제공받는다. DataImporter 인스턴스는 파일을 열고, 그것의 내용을 메모리에 읽기 때문에 인스턴스를 초기화할 때 시간이 소요된다.

DataManager 클래스는 파일로부터 데이터를 불러오지 않아도 데이터를 다룰 수 있다.

DataManager 인스턴스를 만들 때 DataImporter 인스턴스까지 만들 필요가 없다는 의미이다. 대신 DataImporter가 처음으로 사용될 때 이것을 만들게 하면 된다.

lazy 키워드가 사용됐기 때문에 importer 프로퍼티의 DataImporter 인스턴스는 프로퍼티에 처음 접근할 때만 만들어진다.

Stored Properties and Instance Variables (저장된 프로퍼티와 인스턴스 변수)

Objective-C는 값을 저장하거나 클래스 인스턴스를 참조하기 위한 두 가지 방법을 제공한다. 점 연산자나 set 연산을 사용한다. 뿐만 아니라 메모리 관리와 관련된 개념도 프로퍼티에 함께 명시한다. `@property (nonatomic, retain) NSString *propertyName`과 같은 형태이다

Swift는 이러한 개념을 하나의 프로퍼티 선언에 통합한다. Swift 프로퍼티는 해당 인스턴스 변수를 갖지 않고, 프로퍼티의 저장 공간에 직접적으로 접근하지 않는다. 이러한 접근법은 서로 다른 문맥에서 값에 접근하는 방식에 따른 혼란을 피하고, 프로퍼티의 선언을 하나의 일정한 구문으로 간소화한다. 프로퍼티에 대한 모든 정보(이름, 타입, 메모리 관리 특성을 포함한)는 일부분으로써 하나의 장소에 정의된다.

2. Computed Properties (계산 프로퍼티)

클래스, 구조체, 열거형에서 사용할 수 있다. 여기에는 실제 값이 저장되지 않는다. 대신 다른 프로퍼티와 값을 간접적으로 검색하고 설정할 수 있는 getter와 optional setter를 제공한다.

```
struct Point {
  var x = 0.0
  var y = 0.0
}

struct Size {
  var width = 0.0
  var height = 0.0
}
```

```
struct Rect {
  var origin = Point()
  var size = Size()
  var center: Point {
    get {
      let centerX = origin.x + (size.width / 2)
      let centerY = origin.y + (size.height / 2)
      return Point(x: centerX, y: centerY)
    }
    set (newCenter) {
      origin.x = newCenter.x - (size.width / 2)
      origin.y = newCenter.y - (size.height / 2)
    }
  }
}

var square = Rect(origin: Point(x: 0.0, y: 0.0), size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center

square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// "square.origin is now at (10.0, 10.0)"
```

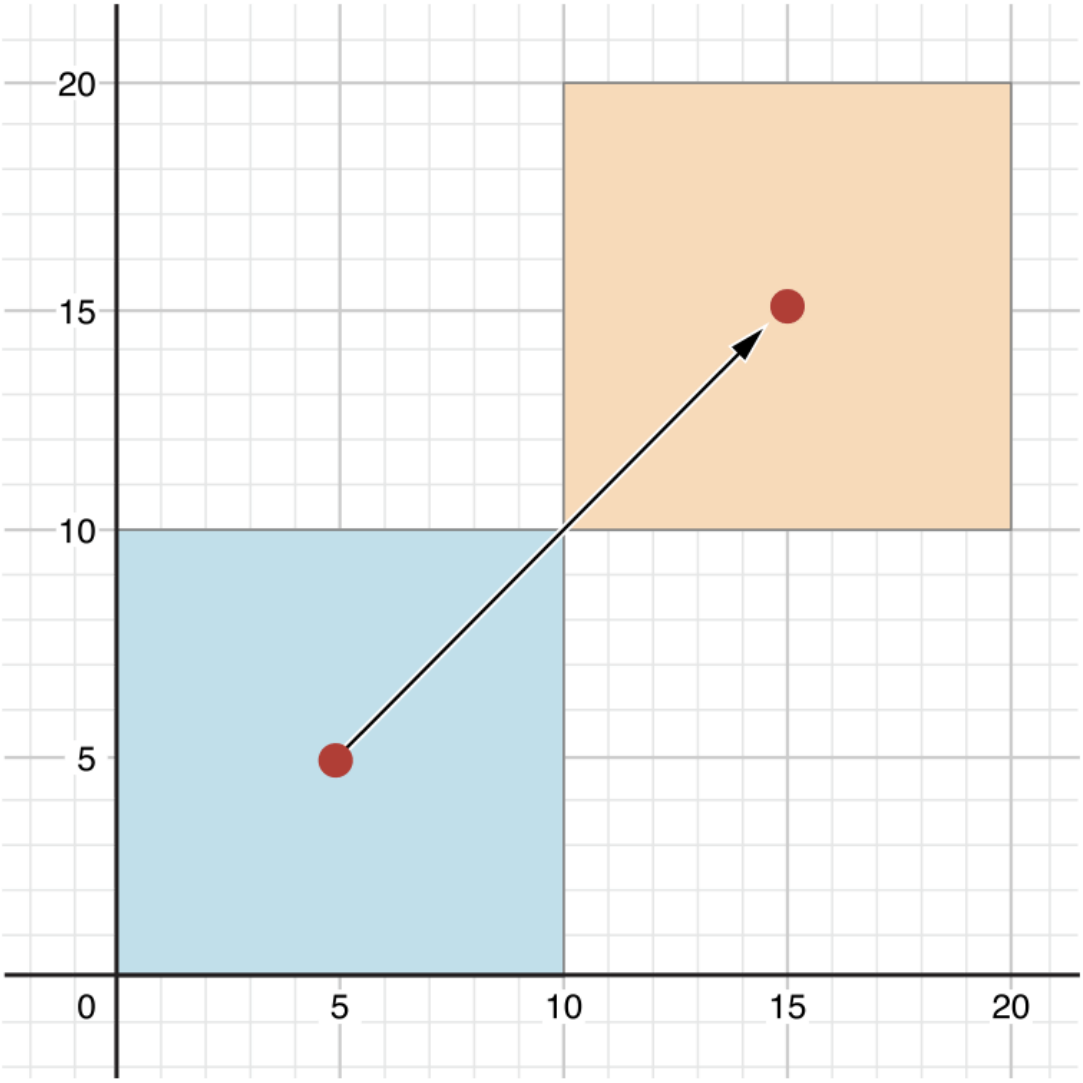
위 예시는 세 개의 구조체를 정의했다.

- Point는 x와 y 좌표
- Size는 너비와 높이
- Pect는 Origin Point와 Size를 통해 사각형을 정의한다.

Rect 구조체는 계산 프로퍼티인 center를 제공한다. 처음 가운데 위치는 origin과 size에 의해 결정된다. 때문에 가운데 위치를 명시적인 Point 값으로 저장할 필요가 없다. 대신 Rect는 center 계산 프로퍼티를 위한 getter, setter를 제공한다.

예시에서는 새로운 Rect 변수인 square를 만들었다. 이 변수는 (0, 0)의 origin point와 10의 width, height로 초기화된다. 이 정사각형은 아래의 파란색 사각형이다.

center 프로퍼티가 새로운 (15, 15)가 저장되었을 때는 아래의 주황색 사각형이다. center 프로퍼티를 세팅하는 것은 center의 setter를 호출하고, origin 프로퍼티에 저장된 x, y 값을 수정한다. 정사각형은 새로운 위치로 옮겨진다.



Shorthand Setter Declaration (세터 선언 축약)

계산 프로퍼티의 세터가 새로운 값의 이름을 정의하지 않으면 newValue라는 이름이 기본적으로 사용된다.

```
struct Rect {
  var origin = Point()
  var size = Size()
  var center: Point {
    get {
      let centerX = origin.x + (size.width / 2)
      let centerY = origin.y + (size.height / 2)
      return Point(x: centerX, y: centerY)
```

```
    }
    set (newCenter) {
      origin.x = newValue.x - (size.width / 2)
      origin.y = newValue.y - (size.height / 2)
    }
  }
}
```

Shorthand Getter Declaration (게터 선언 축약)

게터의 전체 바디가 한줄로 표현된다면 그 표현을 암시적으로 반환할 수 있다.

```
struct Rect {
  var origin = Point()
  var size = Size()
  var center: Point {
    get {
      Point(x: origin.x + (size.width / 2), y: origin.y + (size.height / 2))
    }
    set (newCenter) {
      origin.x = newValue.x - (size.width / 2)
      origin.y = newValue.y - (size.height / 2)
    }
  }
}
```

Read-Only Computed Properties (읽기 전용 계산 프로퍼티)

게터는 있지만 세터는 없는 계산 프로퍼티이다. 읽기 전용 프로퍼티는 점 문법으로 접근할 수 있고, 항상 값을 반환하지만 새로운 값을 설정하는 것은 불가능하다.

읽기 전용 계산 프로퍼티를 포함한 모든 계산 프로퍼티는 반드시 변수로 선언되어야 한다. 이러한 값은 고정되어 있지 않기 때문이다. let 키워드는 오직 상수 프로퍼티에서만 사용되며, 인스턴스 초기화 시에 한번 정해지면 더 이상 값을 바꿀 수 없다.

get 키워드를 제거함으로써 읽기 전용 계산 프로퍼티를 간단하게 선언할 수 있다.

```
struct Cuboid {
  var width = 0.0, height = 0.0, depth = 0.0
  var volume: Double {
    return width * height * depth
  }
}

let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
fourByFiveByTwo.volume // 40.0
```

3. Property Observers (프로퍼티 옵저버)

프로퍼티의 값을 관찰하고 변화에 반응한다. 프로퍼티 옵저버는 값이 설정될 때마다 호출된다. 현재 값과 똑같은 값이 들어오더라도 마찬가지로 반응한다.

지연 저장속성을 제외한 모든 저장속성에 프로퍼티 옵저버를 더할 수 있다. 부모클래스로부터 오버라이딩한 모든 프로퍼티에도 가능하다. 오버라이드되지 않은 계산 프로퍼티를 위한 프로퍼티 옵저버를 정의할 필요는 없다. 계산 프로퍼티의 세터가 값의 변화를 이미 관찰하고 있기 때문이다.

두 가지 옵션이 있다.

- willSet : 값이 저장되기 이전에 호출
- didSet : 새로운 값이 저장된 후 즉시 호출

willSet 옵저버를 구현했다면 새 값의 매개변수 이름을 지정할 수 있는데, 지정하지 않을 경우 기본적으로 newValue를 제공한다. didSet 옵저버를 구현한다면 바뀌기 전의 매개변수 이름을 정할 수 있다. 지정하지 않으면 oldValue를 사용한다.

💡 자식클래스에서 특정 프로퍼티의 값을 설정했을 때, 부모클래스의 initializer가 호출된 후 willSet, didSet 프로퍼티 옵저버가 실행된다. 부모 클래스에서 프로퍼티를 변경하는 것도 마찬가지로 부모클래스의 initializer가 호출된 후 옵저버가 실행된다.

```
class StepCounter {
  var totalSteps: Int = 0 {
    willSet(newTotalSteps) {
      print("About to set totalSteps to \(newTotalSteps)")
    }
    didSet {
      if totalSteps > oldValue {
        print("Added \(TotalSetps - 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
```

💡 만약 함수의 인아웃 매개변수에 프로퍼티를 넘기면 willSet, didSet은 항상 호출된다. 인아웃 매개변수에서는 프로퍼티가 항상 복사되기 때문이다. 함수가 끝날 때 프로퍼티는 원래 값에 새 값을 덮어쓰게 된다.

4. Property Wrappers (프로퍼티 래퍼)

프로퍼티가 저장되는 방식을 다루는 코드와 프로퍼티를 정의하는 코드를 분리하는 단계를 더한다. 예를 들어, 스레드 안전 체크를 하거나, 데이터베이스에 있는 데이터를 저장하는 프로퍼티가 있다면, 모든 프로퍼티에 그러한 코드를 작성해야한다. 프로퍼티 래퍼를 사용하면 래퍼를 정의할 때 관리 기능을 하는 코드는 한 번만 작성하면 된다. 그리고 여러 프로퍼티에 그 코드를 적용함으로써 재사용할 수 있다.

wrappedValue 프로퍼티를 정의하여 프로퍼티 래퍼를 만든다. 아래의 코드에서, TwelveOrLess 구조체는 그것이 감싸는 값이 항상 12 이하의 숫자를 포함한다는 것을 보장한다.

```
@propertyWrapper
struct TwelveOfLess {
  private var number: Int
  init() { self.number = 0 }
  var wrappedValue : Int {
    get { return number }
    set { number = min(newValue, 12) }
  }
}
```


위의 예에서 number는 private로 선언되었다. number 프로퍼티는 오직 TweleveOrLess의 구현에서만 사용될 수 있다는 의미이다.

프로퍼티 래퍼 적용 : 래퍼의 이름을 프로퍼티 전에 속성처럼 표기한다.

```
struct SmallRectangle {
  @TwelveOrLess var height: Int
  @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"

rectangle.height = 10
print(rectangle.height)
// Prints "10"

rectangle.height = 24
print(rectangle.height)
// Prints "12"
```

height와 width 프로퍼티는 TwelveOfLess로부터 초기값을 갖게 된다. 숫자 10을 rectangle.height에 저장하는 것은 성공하지만, 24를 저장하려고 하면 세터의 규칙에 의해 12가 대신 저장된다.

프로퍼티에 래퍼를 적용할 때, 컴파일러는 래퍼를 제공하는 저장 공간과 래퍼를 통해 프로퍼티에 접근하는 코드를 합성한다. 프로퍼티 래퍼의 행위를 특별한 속성 문법의 장점을 취하지 않고 사용하여 코드를 작성할 수도 있다.

말이 어려워 보이는데, 아래의 예제를 살펴보면 이해할 수 있다.

```
struct SmallRectangle {
  private var _height = TwelveOrLess()
  private var _width = TwelveOrLess()
  var height: Int {
    get { return _height.wrappedValue }
    set { _height.wrappedValue = newValue }
  }
  var width: Int {
    get { return _width.wrappedValue }
    set { _width.wrappedValue = newValue }
  }
}
```

_height 와 _width 프로퍼티는 TwelveOrLess 프로퍼티 래퍼 인스턴스를 저장한다. height와 width에 대한 게터와 세터로 wrappedValue에 접근할 수 있다.

Setting Initial Values for Wrapped Properties (Wrapped Properties에 초기값 설정하기)

위 예시는 TwelveOrLess의 정의에서 number에 초기값을 설정했다. 이런 프로퍼티 래퍼를 사용하는 코드는 TwelveOrLess에 의해 포장된 프로퍼티에 서로 다른 초기값을 특정할 수 없다. 예를 들어, SmallRectangle은 height, width에 초기값을 전달하지 못한다. 초기값과 다른 커스터마이징을 위해 프로퍼티 래퍼는 initializer를 추가할 필요가 있다.

```
@propertyWrapper
struct SmallNumber {
  private var maximum: Int
  private var number: Int

  var wrappedValue: Int {
    get { return number }
```

```
    set { number = min(newValue, maximum) }
  }

  init() {
    maximum = 12
    number = 0
  }
  init(wrappedValue: Int) {
    maximum = 12
    number = min(wrappedValue, maximum)
  }
  init(wrappedValue: Int, maximum: Int) {
    self.maximum = maximum
    number = min(wrappedValue, maximum)
  }
}
```

초기값 없이 프로퍼티에 래퍼를 적용할 때, Swift는 init()를 사용한다.

```
struct ZeroRectangle {
  @SmallNumber var height: Int
  @SmallNumber var width: Int
}

var zeroRectangle = ZeroRectangle()
print(zeroRectangle.height, zeroRectangle.width) // 0, 0
```

프로퍼티에 초기값을 지정한다면 Swift는 init(wrappedValue:)를 사용한다.

```
struct UnitRectangle {
  @SmallNumber var height: Int = 1
  @SmallNumber var width: Int = 1
}

var unitRectangle = UnitRectangle()
print(unitRectangle.height, unitRectangle.width) // 1, 1
```

커스텀 속성 소괄호 안에 인자를 전달하면 Swift는 이 인자를 받아들이는 initializer를 사용해 래퍼를 설정한다. 예를 들어 초기값과 maximum 값을 준다면 Swift는 init(wrappedValue:maximum:)를 사용한다.

```
struct NarrowRectangle {
  @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
  @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
}

var narrowRectangle = NarrowRectangle()
print(narrowRectangle.height, narrowRectangle.width)
// 2, 3

narrowRectangle.height = 100
narrowRectangle. width = 100
print(narrowRectangle.height, narrowRectangle.width)
// 5, 4
```

프로퍼티 래퍼에 인자를 포함해 래퍼의 초기값을 설정하고 그것이 생성될 때 래퍼에 다른 옵션을 넣을 수 있다. 이 문법은 프로퍼티 래퍼를 사용하는 가장 일반적인 문법이다.

프로퍼티 래퍼 인자를 포함할 때 할당연산자를 사용해 초기값을 지정할 수 있다. Swift는 할당 연산을 wrappedValue 인자와 같이 다루며, 포함한 인자를 수용하는 initializer를 사용한다.

```
struct MixedRectangle {
  @SmallNumber var height: Int = 1
  @SmallNumber(maximum: 9) var width: Int = 2
}

var mixedRectangle = MixedRectangle()
print(mixedRectangle.height)
// 1

mixedRectangle.height = 20
print(mixedRectangle.height)
// 12
```

height를 wrap하는 SmallNumber의 인스턴스는 최대값이 12인 SmallNumber(wrappedValue:1)을 호출하며, width를 wrap하는 인스턴스는 SmallNumber(wrappedValue: 2, maximum: 9)를 호출한다.

Projecting a Value From a Property Wrapper (프로퍼티 래퍼로부터 값 예상하기)

프로퍼티 래퍼는 projected value를 정의함으로써 새로운 기능을 드러낸다. 예를 들어, 데이터베이스에 대한 접근을 관리하는 프로퍼티 래퍼는 그것의 projected value에 flushDatabaseConnection() 메서드를 노출할 수 있다. projected value의 이름은 달러 사인($)으로 시작한다는 점을 제외하면 wrapped value와 같다. 개발자가 정의하는 코드에서 $로 시작하는 프로퍼티를 정의할 수 없기 때문에, 다른 프로퍼티에 영향을 주지 않는다.

```
@propertyWrapper
struct SmallNumber {
  private var number : Int
  var projectedValue : Bool

  init() {
    self.number = 0
    self.projectedValue = false
  }

  var wrappedValue: Int {
    get {
      return number
    }
    set {
      if newValue > 12 {
          number = 12
          projectedValue = true
      } else {
          number = newValue
          projectedValue = false
      }
    }
  }
}

struct SomeStructure {
  @SmallNumber var someNumber: Int
}

var someStructure = SomeStructure()

someStructure.someNumber = 4
print(someStructure.$someNumber)
// false
someStructure.someNumber = 55
print(someStructure.$someNumber)
// true
```

프로퍼티 래퍼는 어떤 타입의 projected value든 반환할 수 있다. 예제에서는 프로퍼티 래퍼가 정보 하나만을 노출하므로 Bool Type을 projected value로 사용했다. 더 많은 정보를 드러내야할 때는 다른 데이터타입의 인스턴스를 반환할 수 있다. 또는 자기 자신의 projected value로서 self를 사용해 래퍼의 인스턴스를 노출할 수 있다.

```
enum Size {
  case small, large
}
struct SizedRectangle {
  @SmallNumber var height: Int
  @SmallNumber var width: Int

  mutating func resize(to size: Size) -> Bool {
    switch size {
    case .small:
        height = 10
        width = 20
    case .large:
        height = 100
        width = 100
    }
    return $height || $width
  }
}

var rec = SizedRectangle()
rec.resize(to: .small)
print(rec.$height) // false ???
print(rec.$width) // true ???
```

프로퍼티 래퍼는 게터와 세터를 위한 쉽고 편리한 문법이기 때문에, 다른 프로퍼티와 동일한 방법으로 height, width에 접근할 수 있다.

5. Global and Local Variables (지역변수와 전역변수)

계산 프로퍼티와 프로퍼티 옵저버의 기능은 전역변수와 지역변수 모두에 사용 가능하다. 전역변수는 함수, 메소드, 클로저, 타입 컨텍스트 밖에서 정의된 변수이고, 지역변수는 그 안에서 정의된 변수이다.

저장변수는 저장 프로퍼티와 마찬가지로 특정 타입의 값에 대한 저장공간을 제공하고, 값을 설정하거나 검색할 수 있도록 한다.

계산된 변수(computed variable)나 저장된 변수를 위한 옵저버(observer)도 전역/지역 변수로 정의할 수 있다. 계산된 변수는 값을 저장하기보다는 계산한다. 계산 프로퍼티와 같

은 방식으로 작성된다.

전역 상수, 변수는 지연저장속성과 비슷하게 지연계산된다. 하지만 지연저장속성과 다르게 lazy 키워드를 붙일 필요가 없다.

지역 상수, 변수는 지연계산되지 않는다.

6. Type Properties (타입 프로퍼티)

타입의 인스턴스 하나가 아니라 타입 자체에 속한 프로퍼티를 정의할 수 있다. 그 타입은 해당되는 단 하나의 프로퍼티만 생성된다. 이러한 프로퍼티를 type property라 부른다.

타입 프로퍼티는 모든 인스턴스가 사용하는 상수 프로퍼티처럼 특정 타입의 인스턴스에 공통되는 값을 정의할 때 유용하다. 또는 그 타입의 모든 인스턴스에 공통되는 값을 저장하는 변수 프로퍼티를 정의할 때도 좋다.

저장 타입 프로퍼티는 변수, 상수 모두 사용 가능하지만 계산 프로퍼티는 변수만 사용할 수 있다.

저장 인스턴스 프로퍼티와 달리 저장 타입 프로퍼티에는 반드시 기본 값을 주어야 한다. 타입 자신은 초기화 시간에 저장 타입 프로퍼티에 값을 할당할 수 있는 initializer를 가질 수 없기 때문이다.

저장 타입 프로퍼티는 최초 접근시 지연 초기화된다. 여러 스레드가 동시에 접근해도 한 번만 초기화 되는 것을 보장한다. 또한 lazy 키워드가 필요 없다.

Type Property Syntax (타입 프로퍼티 문법)

Swift에서의 타입 프로퍼티는 타입 정의의 일부분으로써 작성된다. 각각의 타입 프로퍼티는 명시적으로 그 타입이 지원하는 범위가 정해져 있다. static 키워드를 사용해 타입 프로퍼티

를 정의한다. 클래스 타입의 계산 타입 프로퍼티에서는 자식클래스가 부모클래스의 구현을 오버라이딩을 허용하는 대신 class 키워드를 사용할 수 있다.

```
struct SomeStructure {
  static var storedTypeProperty = "Some Value."
  static var computedProperty: Int {
    return 1
  }
}

enum SomeEnumeration {
  static var storedTypeProperty = "Some Value."
  static var computedProperty: Int {
    return 6
  }
}

class SomeClass {
  static var storedTypeProperty = "Some Value."
  static var computedProperty: Int {
    return 27
  }
  class var overrideableComputedTypeProperty: Int {
    return 107
  }
}
```

Querying and Setting Type Properties (타입 프로퍼티의 질의와 설정)

점 문법을 통해 질의받고 설정될 수 있다는 점은 타입 프로퍼티와 인스턴스 프로퍼티의 공통점이다. 그러나 타입 프로퍼티는 타입에 대해서만 질의받고 설정된다. 타입의 인스턴스가 아니다

```
print(SomeStructure.storedTypeProperty)
// "Some Value."

SomeStructure.storedTypeProperty = "Another Value."
print(SomeStructure.storedTypeProperty)
// "Another Value."

print(SomeEnumeration.computedTypeProperty)
// "6"
print(SomeClass.computedTypeProperty)
// "27"
```

아래의 예시는 오디오 채널을 모델링하는 구조체의 일부로, 두 개의 저장 타입 프로퍼티를 사용한다. 각각의 채널은 0 에서 10 사이의 오디오 레벨을 갖는다.

```
struct AudioChannel {
  static let thresholdLevel = 10
  static var maxInputLevelForAllChannels = 0

  var currentLevel: Int = 0 {
    didSet {
      if currentLevel > AudioChannel.thresholdLevel {
          currentLevel = AudioChannel.thresholdLevel
      }
      if currentLevel > AudioChannel.maxInputLevelForAllChannels {
          AudioChannel.maxInputLevelForAllChannels = currentLevel
      }
    }
  }
}
```

AudioChannel 구조체는 두 개의 저장 프로퍼티를 정의한다. 첫번째로 thresholdLevel은 오디오 레벨이 가질 수 있는 최대 레벨값을 정의한다. 이것은 모든 AudioChannel 인스턴스에 대해 10의 상수값을 갖는다. 만약 오디오 신호로 10보다 더 큰 값이 들어온다면, 제한 값으로 수정된다.

currentLevel 프로퍼티는 값의 변화를 확인하는 didSet 프로퍼티 옵저버를 갖는다. 이 옵저버는 두가지를 체크한다.

- 만약 새로운 currentLevel 값이 허용된 thresholdLevel보다 크면, 프로퍼티 옵저버는 currentLevel을 thresholdLevel로 맞춘다.
- 만약 새로운 currentLevel 값이 아무 AudioChannel 인스턴스로부터 받은 값보다 크다면, 프로퍼티 옵저버는 currentLevel을 maxInputLevelForAllChannels 타입 프로퍼티에 저장한다.

처음 두 타입 프로퍼티를 체크할 때 didSet 옵저버는 currentLevel에 기본값을 저장한다.

AudioChannel 구조체를 새로운 두 오디오 채널을 만드는 데 사용할 수 있다.

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

만약 leftChannel의 currentLevel을 7로 설정한다면, maxInputLevelForAllChannels의 타입 프로퍼티는 7로 수정된다.

```
leftChannel.currentLevel = 7
print(leftChannel.currentLevel) // 7
print(AudioChannel.maxInputLevelForAllChannels) // 7
```

만약 currentLevel에 11을 넣으려 한다면, rightChannel의 currentLevel 프로퍼티는 최대값인 10으로 정해진다. maxInputLevelForAllChannels 프로퍼티 역시 10으로 수정된다.

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel) // 10
print(AudioChannel.maxInputLevelForAllChannels) // 10
```