Objects and Classes

Use `class` followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it's in the context of a class. Likewise, method and function declarations are written the same way.

```swift
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

EXPERIMENT

Add a constant property with `let`, and add another method that takes an argument.

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```swift
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

This version of the `Shape` class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

```swift
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

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

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned—either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).

Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is 할당해제하다 deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There's no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's 구현 implementation are marked with `override`—overriding a method by accident, without `override`, is detected by the compiler as an error. The compiler also detects methods with `override` that don't actually override any method in the superclass.

```swift
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() -> Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

EXPERIMENT

Make another subclass of `NamedShape` called `Circle` that takes a radius and a name as arguments to its initializer. Implement an `area()` and a `simpleDescription()` method on the `Circle` class.

게다가, 뿐만 아니라

In addition to simple properties that are stored, properties can have a getter and a setter.

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
      1. self.sideLength = sideLength
      2. super.init(name: name)
      3. numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \
  (sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
// Prints "9.3"
triangle.perimeter = 9.9
print(triangle.sideLength)
// Prints "3.3000000000000003"
```

In the setter for `perimeter`, the new value has the implicit name `newValue`. You can provide an explicit name in parentheses after `set`.

Notice that the initializer for the `EquilateralTriangle` class has three different steps:

1. Setting the value of properties that the subclass declares.
2. Calling the superclass's initializer.
3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that's run before and after setting a new value, use `willSet` and `didSet`. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
  shape")
print(triangleAndSquare.square.sideLength)
// Prints "10.0"
print(triangleAndSquare.triangle.sideLength)
// Prints "10.0"
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)
// Prints "50.0"
```

When working with optional values, you can write `?` before operations like methods, properties, and subscripting. If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

```swift
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
  square")
let sideLength = optionalSquare?.sideLength
```