struct Stack {
    private var myArray: [String] = []

    mutating func push(_ element: String) {
        myArray.append(element)
    }

    mutating func pop() -> String? {
        return myArray.popLast()
    }

    func peek() -> String {
        guard let topElement = myArray.last
        else { return "This stack is empty."}
        return topElement
    }
}
var stack = Stack()
stack.peek()
stack.push("Swift Arrays")
stack.push("Swift LinkedList")
stack.push("Swift Stack")
print(stack)
stack.peek()
stack.pop()
stack.pop()
stack.pop()
stack.pop()
stack.peek()
stack.pop()
Extension Syntax

Declare extensions with the `extension` keyword:

```swift
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```swift
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```
struct Size {
    var width = 0.0, height = 0.0
} //Size

struct Point {
    var x = 0.0, y = 0.0
} //Point

struct Rect {
    var origin = Point()
    var size = Size()
}

let defaultRect = Rect()

let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
                           size: Size(width: 5.0, height: 5.0))

extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
} //Rect
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))

// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
extension Int {
    func repetitions(task: () -> void) {
        for _ in 0..<self {
            task()
        }
    }
}

3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// Hello!
extension Int {
    mutating func square() {
        self = self * self
    }
}

var somelnt = 3
somelnt.square()
// somelnt is now 9
Subscripts

extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}

746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
Extensions can add new nested types to existing classes, structures, and enumerations:

```swift
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
            case 0:    return .zero
            case let x where x > 0: return .positive
            default:     return .negative
        }
    }
}
```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
            case .negative:   print("- ", terminator: "")
            case .zero:          print("0 ", terminator: "")
            case .positive: print("+ ", terminator: "")
        }
    }
    print(""
}

printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + -- 0 -- 0 + "

Computed Properties

extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
    print("Three feet is \(threeFeet) meters")
    // Prints "Three feet is 0.914399970739201 meters"
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
    // Prints "A marathon is 42195.0 meters long"
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, 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
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        } //set
    } //point
} //Rect
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))")
// Prints "square.origin is now at (10.0, 10.0)"
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        } //willSet
        didSet {
            if totalSteps > oldValue {
                print("Added \(totalSteps - oldValue) steps")
            }
        } //didSet
    } //totalsteps
} //class
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
Property Wrappers

A property wrapper adds a layer of separation between code that manages how a property is stored and the code that defines a property.

For example, if you have properties that provide thread-safety checks or store their underlying data in a database, you have to write that code on every property.

When you use a property wrapper, you write the management code once when you define the wrapper, and then reuse that management code by applying it to multiple properties.
@propertyWrapper
struct TwelveOrLess {
    private var number = 0
    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, 12) }
    }
} //TwelveOrLess
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"
Opaque Types

A function or method with an opaque return type hides its return value’s type information.

Instead of providing a concrete type as the function’s return type, the return value is described in terms of the protocols it supports.

```swift
protocol Animal {
    func isSibling(with animal: Self) -> Bool
}

class Dog: Animal {
    func isSibling(with animal: Dog) -> Bool {
        return true // doesn't really matter implementation of this
    }
}
```
func animalFromAnimalFamily() -> Animal {
    return myDog // myDog is just some random variable of type `Dog`
}

let animal1: Animal = animalFromAnimalFamily()
let animal2: Animal = animalFromAnimalFamily()

animal1.isSibling(animal2) // error

Swift doesn’t know if the animals are dogs, cats, or whatever.
How some T solves this problem

```swift
func animalFromAnimalFamily() -> some Animal {
    return randomDog
}

let animal1 = animalFromAnimalFamily()
let animal2 = animalFromAnimalFamily()

animal1.isSibling(animal2)

animal1 and animal2 are not Animal, but they are class that implements Animal.

What this lets you do now is when you call animal1.isSibling(animal2), Swift knows that animal1 and animal2 are the same type.