TEB

iOS Programlama Eğitimi
6. Gün

Lazy Stored Properties

```
class DataImporter {
    var fileName = "data.txt"

    init() {
        println("Data Importer being initialized...")
    }
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
}
```

Lazy Stored Properties

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

// DataImporter instance has not been created yet
```

Lazy Stored Properties

```
println(manager.importer.fileName)
// Data Importer being initialized...
// data.txt
```

Any & AnyObject

- AnyObject
  - Bir class tipinde nesneyi ifade eder
- Any
  - Her hangi bir tipte nesneyi ifade eder
  - Fonksiyonlar dahil

Lazy Stored Properties

```
var things = [Any]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters"))
things.append({(name: String)->String in "Hello,\(name)"})
```

Lazy Stored Properties

```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
    case let someInt as Int:
        println("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        println("a positive double value of \(someDouble)")
    case is Double:
        println("some other double value")
    case let someString as String:
        println("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        println("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        println("a movie called '\(movie.name)'")
    case let stringConverter as String -> String:
        println(stringConverter("Michael"))
```

Nested Types

```
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case Spades = "♠", Hearts = "♡"
        case Diamonds = "♢", Clubs = "♣"
    }

    // BlackjackCard properties and methods
    let suit: Suit
    var description: String {
        var output = "Suit is \(suit.toRaw())"
        return output
    }
}
```

Nested Types

```
let theSpades = BlackjackCard(suit: .Spades)
println("Spades: \(theSpades.description)")
// prints "Spades: suit is ♠"
```

Nested Types

```
let heartsSymbol = BlackjackCard.Suit.Hearts.rawValue
// heartsSymbol is "♡"
```

Operator Functions

Operator Overloading olarak da bilinir.

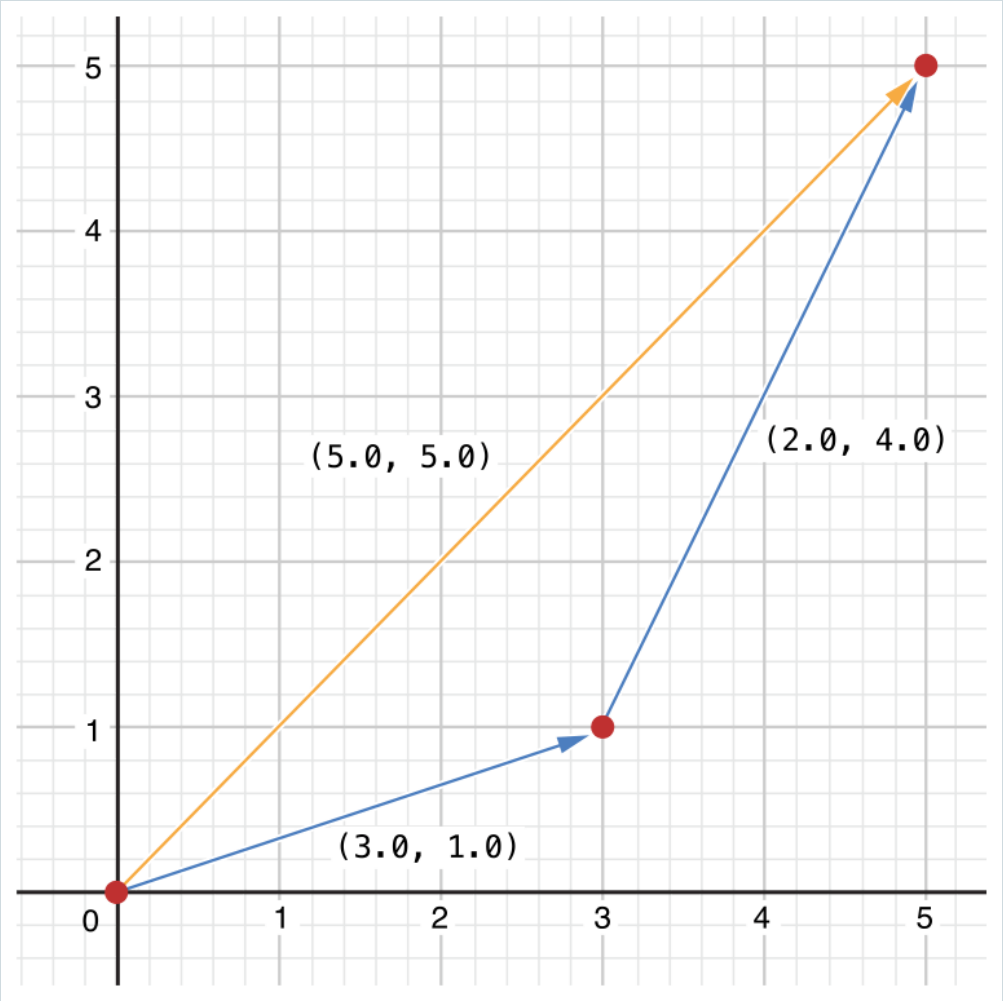
```
struct Vector2D {
    var x = 0.0, y = 0.0
}

func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x:left.x + right.x, y:left.y + right.y)
}
```

Operator Functions

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is Vector2D with values of (5.0, 5.0)
```

Operator Functions



Prefix and Postfix Operators

```
prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

Prefix and Postfix Operators

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is Vector2D with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is Vector2D with values of (3.0, 4.0)
```

Compound Assignment Operators

```
func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
```

Compound Assignment Operators

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

Compound Assignment Operators

```
prefix func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

Compound Assignment Operators

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement now has values of (4.0, 5.0)
// afterIncrement also has values of (4.0, 5.0)
```

Compound Assignment Operators

- = operatörü overload edilemez
- Ternary condition (a ? b : c) operatörü overload edilemez

Equivalence Operators

```
func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}

func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
```

Equivalence Operators

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("These two vectors are equivalent.")
}
// prints "These two vectors are equivalent."
```

Custom Operators

```
prefix operator +++ {}
```

Custom Operators

```
prefix func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

Custom Operators

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

More Operator Overloading

- http://nshipster.com/swift-operators/