

Swift Cheat Sheet

(Object-Oriented Programming)



Classes

```
class MyPointClass {  
}  
var ptA = MyPointClass()
```

Stored Properties

```
class MyPointClass {  
    var x = 0.0  
    var y = 0.0  
    let width = 2  
}  
var ptA = MyPointClass()  
ptA.x = 25.0  
ptA.y = 50.0  
  
print(ptA.x) //---25.0---  
print(ptA.y) //---50.0---  
print(ptA.width) //---2---
```

Lazy Stored Properties

```
class PointMath {  
    var someValue = 1.2345  
}  
class MyPointClass {  
    var x = 0.0  
    var y = 0.0  
    let width = 2  
    lazy var pointMath =  
        PointMath()  
}
```

Computed Properties

```
class Distance {  
    var miles = 0.0  
    var km: Double {  
        get {  
            return 1.60934 * miles  
        }  
        set (km) {  
            miles = km / 1.60934  
        }  
    }  
}  
var d = Distance()  
d.miles = 10.0  
print(d.km) //---16.0934---
```

Property Observers

```
class MyPointClass {  
    var x: Double = 0.0 {  
        willSet(newX) {  
            print("Before setting")  
        }  
        didSet {  
            print(  
                "Before: \(oldValue)"  
                "After: \(x)"  
            )  
            if x > 100 || x < (-100) {  
                x = oldValue  
            }  
        }  
    }  
}
```

Type Properties

```
class MyPointClass {  
    var x = 0.0  
    var y = 0.0  
    let width = 2  
    class var origin: (  
        Double, Double) {  
        get {  
            return (0,0)  
        }  
    }  
}  
print(MyPointClass.origin)
```

Initializers

```
class MyPointClass {  
    var x = 0.0  
    var y = 0.0  
    let width = 2  
    init() {  
        x = 5.0  
        y = 5.0  
    }  
    init(x: Double, y: Double) {  
        self.x = x  
        self.y = y  
    }  
}  
var ptC =  
    MyPointClass(x:7.0, y:8.0)  
print(ptC.x) //---7.0---  
print(ptC.y) //---8.0---  
print(ptC.width) //---2---
```

Identity Operator

```
var pt1 = MyPointClass()  
var pt2 = pt1  
var pt3 = MyPointClass()  
if pt1 === pt2 {  
    print("Identical")  
} else {  
    print("Not identical")  
} // Identical  
  
if pt1 === pt3 {  
    print("Identical")  
} else {  
    print("Not identical")  
} // Not identical
```

Operator Overloading

```
func == (ptA: MyPointClass,  
        ptB: MyPointClass) -> Bool  
{  
    return (ptA.x == ptB.x) &&  
        (ptA.y == ptB.y)  
}  
  
func != (ptA: MyPointClass,  
        ptB: MyPointClass) -> Bool  
{  
    return !(ptA == ptB)  
}
```

Instance Methods

```
class Car {  
    var speed = 0  
    func accelerate() {  
        ...  
    }  
    func decelerate() {  
        ...  
    }  
    func stop() {  
        ...  
    }  
    func printSpeed() {  
        ...  
    }  
}
```

Mutating Methods in Structures

```
struct Go {  
    var row: Int  
    var column: Int  
  
    mutating func move(  
        dRow: Int, dColumn: Int) {  
        row += dRow  
        column += dColumn  
    }  
}
```

Type Method

```
class Car {  
    var speed = 0  
    class func kilometersToMiles  
        (km: Int) -> Double {  
        return Double(km) / 1.60934  
    }  
}
```

Inheritance

```
class Shape {  
    //---stored properties---  
    var length: Double  
    var width: Double  
    init() {  
        length = 0  
        width = 0  
    }  
    func perimeter() -> Double {  
        return 2 * (length +  
            width)  
    }  
    func area() -> Double {  
        return length * width  
    }  
}  
class Rectangle: Shape {}
```

Overloading Methods

```
class Rectangle: Shape {  
    //---overload the init()---  
    init(length: Double,  
        width: Double) {  
        super.init()  
        self.length = length  
        self.width = width  
    }  
}
```

Overriding Methods

```
class Rectangle: Shape {  
    override init() {  
        super.init()  
        self.length = 5  
        self.width = 5  
    }  
    init(length: Double,  
        width: Double) {  
        super.init()  
        self.length = length  
        self.width = width  
    }  
}
```

Final Class

```
final class Square: Rectangle {  
    //---overload the init()---  
    init(length: Double) {  
        super.init()  
        self.length = length  
        self.width = self.length  
    }  
}
```

Designated Initializers

```
class Contact {
    var firstName:String
    var lastName:String
    var email:String
    var group:Int
    init(firstName: String,
        lastName:String,
        email:String,
        group: Int) {
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.group = group
    }
}
```

Convenience Initializers

```
class Contact {
    var firstName:String
    var lastName:String
    var email:String
    var group:Int

    //---designated initializer---
    init(firstName: String,
        lastName:String,
        email:String,
        group: Int) {
        ...
    }

    //---convenience initializer;
    // delegate to the designated
    // one---
    convenience init(
        firstName: String,
        lastName:String,
        email:String) {
        self.init(
            firstName: firstName,
            lastName: lastName,
            email: email,
            group: 0)
    }
}
```

Extensions

```
extension String {
    func getLatLng(
        splitter:String) ->
        (Double, Double) {
        ...
    }
}
var str = "1.23456,103.345678"
var latlng =
    str.getLatLng(splitter:",")
print(latlng.0)
print(latlng.1)
```

Closure

```
let numbers = [5,2,8,7,9,4,3,1]
var sortedNumbers = numbers.sorted()

{
    (num1:Int, num2:Int) ->
        Bool in
        return num1<num2
}
```

Map Function

```
let prices =
    [12.0,45.0,23.5,78.9,12.5]
var pricesIn$ = prices.map(
    {
        (price:Double) -> String in
        return "$\(price)"
    }
)
```

Filter Function

```
let prices =
    [12.0,45.0,23.5,78.9,12.5]
var pricesAbove20 =
    prices.filter(
    {
        (price:Double) -> Bool in
        price>20
    }
)
```

Reduce Function

```
let prices =
    [12.0,45.0,23.5,78.9,12.5]
var totalPrice = prices.reduce(
    0.0,
    combine:{
        (subTotal: Double,
        price: Double) -> Double in
        return subTotal + price
    }
)
```

Using Closure

```
func bubbleSort(
    items:inout [Int],
    compareFunction:(Int, Int)
    ->Bool) {

    for j in 0 ..< items.count-1 {
        var swapped = false
        for i in
            0 ..< items.count-1-j {
            if compareFunction(
                items[i],items[i+1])
                {
                    let temp=items[i+1]
                    items[i+1]=items[i]
                    items[i]=temp
                    swapped=true
                }
            if !swapped {
                break
            }
        }
    }

    var numbers = [5,2,8,7,9,4,3,1]
    bubbleSort(items: &numbers,
        compareFunction:
        {
            (num1:Int, num2:Int) ->
                Bool in
                return num1 > num2
        }
    )
}
```

Protocols

```
@objc protocol CarProtocol {
    func accelerate()
    func decelerate()
    @objc optional func
        accelerateBy(amount:Int)
}
```

Delegate

```
@objc protocol CarProtocol {
    func accelerate()
    func decelerate()
    @objc optional func
        accelerateBy(amount:Int)
}

@objc protocol CarDelegate {
    func reachedMaxSpeed()
    @objc optional func
        accelerating()
}
```

```
class Car: CarProtocol {
    var delegate: CarDelegate?
    var speed = 0
    @objc func accelerate() {
        speed += 10
        if speed > 50 {
            speed = 50
        }
        delegate?.reachedMaxSpeed()
    }
}
```

```
    } else {
        delegate?.accelerating?()
    }
}

@objc func decelerate() {
    ...
}

class CarStatus: CarDelegate {
    @objc func reachedMaxSpeed() {
        print("Max speed!")
    }

    //===optional method===
    @objc func accelerating() {
        print("Accelerating...")
    }
}

var c1 = Car()
c1.delegate = CarStatus()
```

Generics

```
func swapItems<T>(
    item1:inout T, item2:inout T) {
    let temp = item1
    item1 = item2
    item2 = temp
}
```

Specifying Constraints

```
func sortItems<T: Comparable>
    ( items:inout [T]) {
    for _ in 0 ..< items.count-1 {
        var swapped = false
        for i in 0 ..< items.count-1
            {
                if items[i]>items[i+1] {
                    swapItems(item1:
                        &items[i],
                        item2:
                            &items[i+1])
                    swapped = true
                }
            }
        if !swapped {
            break
        }
    }
}
```

Generic Class

```
class MyStack<T> {
    var elements = [T]()
    func push(item:T) {
        elements.append(item)
    }

    func pop() -> T! {
        if elements.count>0 {
            return
                elements.removeLast()
        } else {
            return nil
        }
    }
}

var myStringStack =
    MyStack<String>()
myStringStack.push(item:
    "Programming")
myStringStack.push(item: "Swift")
```

