

207: Swift Language Basics

Part 2: Demo Instructions

Copyright © 2014 Razeware LLC.

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written per- mission of the copyright owner.

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connec- tion with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this book are the property of their respective owners.

# Playground

## Introduction

Open Xcode and select **File\New\Playground…**. Call it **ASwiftIntro.playground** and save it somewhere.

Delete everything except:

import UIKit

## Step 1 – Variables, strings and string interpolation

Type:

var conference = "RWDevCon"

<This is a variable definition.>

Type:

println("The best conference is \(conference)")

<This is printing out a line (NSLog) and this is using string interpolation.>

Add the following line above the println:

conference = "WWDC"

<We’ve changed the variable>

Change the top line to:

let conference = "RWDevCon"

<There’s now an error, because you can’t reassign ‘let’ variables.>

Remove:

conference = "WWDC"

## Step 2 – Numbers, control flow

Type:

let attendees = 1\_000

let cakes = 850

<Notice that \_ can be used in numbers to make them more readable.>

Type:

let cakesPerAttendee = cakes / attendees

<This shows 'cakesPerAttendee' as '0'. Makes sense, but not what we want.>

Change last line to:

let cakesPerAttendee = Double(cakes) / Double(attendees)

<Casting the values to Double means that we now get what we want.>

Type:

if cakesPerAttendee > 0.5 {

println("At least half the attendees will get a cake.")

} else {

println("All hell will break loose!")

}

<Explain this is what an if-statement looks like. Notice no brackets around the expression. While, for, etc loops are like this too.>

## Step 3 – Collections, optionals, tuples

Type:

var animals = ["Dog", "Cat", "Mouse", "Horse"]

<This is an array. It's declared as 'var', so we can add to it.>

Type:

animals.append("Cow")

<See, we added to the array. If you change 'var' to 'let' then this line will show an error.>

Type:

animals.append(123)

<The error comes from the fact that the array is an array of Strings, but we're trying to add an Int. That doesn't work. Collections are typed in Swift.>

Type:

for animal in animals {

println(animal)

}

<This is fast enumeration in Swift.>

<Turn on assistant editor - View\Assistant Editor\Show Assistant Editor - to see the output from the printing.>

Type:

let animalCounts = ["Dog":2, "Cat":0, "Cow":100]

<This is a Dictionary in Swift. This is a Dictionary from String to Int.>

Type:

let animal = "Dog"

let count = animalCounts[animal]

<Here we're accessing the Dictionary through its subscript to obtain the value for a given key.>

Type:

println("I have \(count) \(animal)s")

<This prints "I have Optional(2) Dogs" because the value returned from the subscript is an optional.>

<Talk about optionals.>

Change previous line to:

println("I have \(count!) \(animal)s")

<This is forced unwrapping of the optional. This means that the optional's value is taken instead of the optional itself.>

Change the animal variable to:

let animal = "Mouse"

<All hell breaks loose because the optional is now its 'nil' case, and it tries to unwrap that and we get a runtime crash. So this sort of unwrapping is unsafe - do it only when you know the optional can't be nil.>

Wrap the println like so:

if count != nil {

println("I have \(count!) \(animal)s")

}

<This fixes the problem, but we can do better. We can use optional binding to make things a lot cleaner.>

Type:

if let count = animalCounts[animal] {

println("I have \(count) \(animal)s")

}

<This sets a variable called 'count' local to the if-statement. The if-statement will only enter if 'count' is non-nil. Inside the if-statement, 'count' can be used as the unwrapped optional - i.e. it's not an optional in here. It's the concrete value.>

Type:

for (key, value) in animalCounts {

println("\(key) => \(value)")

}

<This iterates over the dictionary, just like we did with the Array. But each 'value' is a tuple of (key, value). Tuples are multi-value values.>

## Step 4 – Introducing Tic Tac Toe

Type:

enum Player {

case X

case O

}

<This is an Enum in Swift. It's much like NS\_ENUM from Objective-C, but much more powerful as we'll see.>

Type:

struct Position: Hashable {

var column: Int

var row: Int

}

<This is a Struct in Swift. It's like a class, in that it has members, but it is a value type.>

Type:

var pos1 = Position(column: 0, row: 0)

var pos2 = pos1

<This sets up two "Position" variables.>

Type:

pos1.column = 1

println(pos1.column)

println(pos2.column)

<Only "pos1" has "column" set to 1. This is because Structs are value types. When you assign a struct variable it takes a copy. If this were a class, then pos1.column would equal pos2.column.>

Delete the last 5 lines.

Change var to let in the Position struct:

struct Position: Hashable {

let column: Int

let row: Int

}

Add the following method to Position:

func asString() -> String {

return "\(column):\(row)"

}

<Structs can have methods just like classes can.>

Type:

let pos1 = Position(column: 0, row: 0)

let pos2 = Position(column: 0, row: 1)

println(pos1.asString())

<We're using the method we just added to the struct.>

<We're going to want to use this struct as the key in a Dictionary. To do this, it needs to be "Hashable". Hashable is a protocol in Swift. Structs can implement protocols too.>

Change Position definitions to:

struct Position: Hashable

<It tells us we need to implement a few methods. The first is 'hashValue', a computed property.>

Add the following method to Position:

var hashValue: Int {

return row \* column

}

<This is just like 'hash' from NSObject in Objective-C.>

<One more thing needs adding for the Hashable protocol conformance. We need to know when two Position objects are equal. Hashable extends Equatable. To do this we need to use operator overloading.>

Type:

func ==(lhs: Position, rhs: Position) -> Bool {

return lhs.row == rhs.row && lhs.column == rhs.column

}

<This is a top-level function. It's called '==' and takes two values - the left hand side and the right hand side of the operator, and returns a Boolean. It's equal if the left hand side row and column equal the right hand side row and column.>

Type:

if pos1 == pos2 {

println("They're equal")

} else {

println("They're not equal")

}

<It's worked. Now let's use the struct in a dictionary.>

Type:

var turns: [Position:Player] = [:]

<This created an empty Dictionary. We've had to specify the type because type inference can't work out what types we want if the dictionary is empty.>

Type:

turns[pos1] = .X

turns[pos2] = .O

<We've used the struct as the key of the dictionary. The value is a Player enum.>

<This makes the start of the Tic Tac Toe game. You'll build on this in the lab and challenge. The Position struct is the position on the game board and the Player enum is X or O.>