

207: Swift Language Basics

Part 3: Lab 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.

# Tic Tac Swift

In this short lab and challenge, you will make a Tic Tac Toe game in Swift. You’ll build upon the things you learned in the playground demo and really put them to the test.

## A Starter for 10

In the resources for this session, you will find a starter project in the 2-Demo directory. Open it up and take a look around. There’s a single view controller and a view for the Tic Tac Toe board. In this lab, you’re going to add all the functionality for playing a game of Tic Tac Toe.

In the playground demo, you were introduced to the data structures that the game requires. They’re not in this app yet, so it’s time to add them. Click **File\New\File…**. Then select **iOS\Swift File** and click **Next**. Call the file **DataStructures.swift**, and save it in the project directory.

Next, open **DataStructures.swift** and add the following code:

struct Position: Hashable {

let column: Int

let row: Int

func asString() -> String {

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

}

var hashValue: Int {

return column \* 10 + row

}

}

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

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

}

This is the Position struct just as you saw in the playground demo. This will be used to indicate playing positions on the Tic Tac Toe board.

Now add the following code to the same file:

enum Player {

case X

case O

func opposite() -> Player {

switch self {

case .X:

return .O

case .O:

return .X

}

}

}

This is the Player enum from the playground demo, except for one change. Here there is a method in the enum. This is one way in which Swift differs from Objective-C. Enums in Objective-C were not much more than names for integers. There was little extra support over and above this.

However in Swift, enums are fully fledged types and behave in a similar way to structs. They can even have methods. In this enum, you’ve added a method to return the opposite of the current value. If you call opposite on X, you get O, and vice versa. Neat! :]

Next, open **Board.swift** and find the BoardDelegate protocol at the top of the file. Add the following methods to it:

func board(board: Board, didPressPosition position: Position)

func board(board: Board, playerAtPosition position: Position)

-> Player?

The first method tells the board view’s delegate in which position the player has tapped. The delegate can use this to add a move to that position if one doesn’t already exist.

The second method is how the board works out what to draw for each position on the board. The return type of this method is an optional, meaning that the delegate can either return either a Player enum value, or nil. This is to signify the absence of a play at that position.

This is a great example of where optionals are much superior over how this would be accomplished in Objective-C. In Objective-C, you would need another case in the enum to signify no value for example. Swift optionals make it explicit.

Now that you’ve added the data structures and protocol methods, there’s some admin to take care of in the board view.

Open **Board.swift** and find the two commented methods at the bottom of the file called drawPlayer(player:atPosition:) and touchesEnded(touches:withEvent:). Uncomment them. The first of these methods draws the X or O at the given position. The second handles touches and fires the delegate method to indicate which position was touched.

Now, find drawRect(rect:) and add the following code at the end of the method:

// Draw turns

for row in 0..<3 {

for col in 0..<3 {

let position = Position(column: col, row: row)

if let player = self.delegate?.board(

self,

playerAtPosition: position)

{

self.drawPlayer(player, atPosition: position)

}

}

}

This loops through all rows and for each row, all columns. For each of these positions it asks the delegate for the player at this position.

The inner loop uses optional chaining with the self.delegate? syntax. This means that if delegate is currently set to nil, then the board() method won’t be called and nil will be returned from the expression.

Now over to the view controller to wire everything up!

Open **ViewController.swift** and add the following properties to the class, just underneath the Interface Builder outlet for the board:

private var turns = [Position:Player]()

private var player = Player.X

This declares two properties. The first holds the turns as a dictionary mapping from board position to the player at that position. The second holds the current player whose turn it is.

Notice the use of private infront of the properties. This means that the property is only available inside this class. In Swift there are three levels of access, which can be applied to properties and methods:

* public – The item is available everywhere.
* internal – The item is available anywhere inside the package – i.e. the framework or app – that the item belongs to.
* private – The item is only available to the enclosing class.

Next, add the following code at the end of the file:

extension ViewController: BoardDelegate {

}

An extension in Swift is equivalent to a category in Objective-C, and this one declares conformance to the BoardDelegate protocol. But this isn’t enough, you need to implement the two delegate methods. Add the following code inside the extension you just added:

func board(board: Board, didPressPosition position: Position) {

println("Pressed \(position.asString())")

if let player = self.turns[position] {

println("Play already exists at this position!")

} else {

self.turns[position] = self.player

self.player = self.player.opposite()

self.board.setNeedsDisplay()

}

}

This implements the first method of the protocol. It handles what happens when the player taps on a position on the board. If there is already a play at that position, then nothing happens apart from logging this. If a play doesn’t already exist, then a play is added, the current player toggled and the board refreshed.

Now add the following code to the extension:

func board(board: Board, playerAtPosition position: Position)

-> Player?

{

return self.turns[position]

}

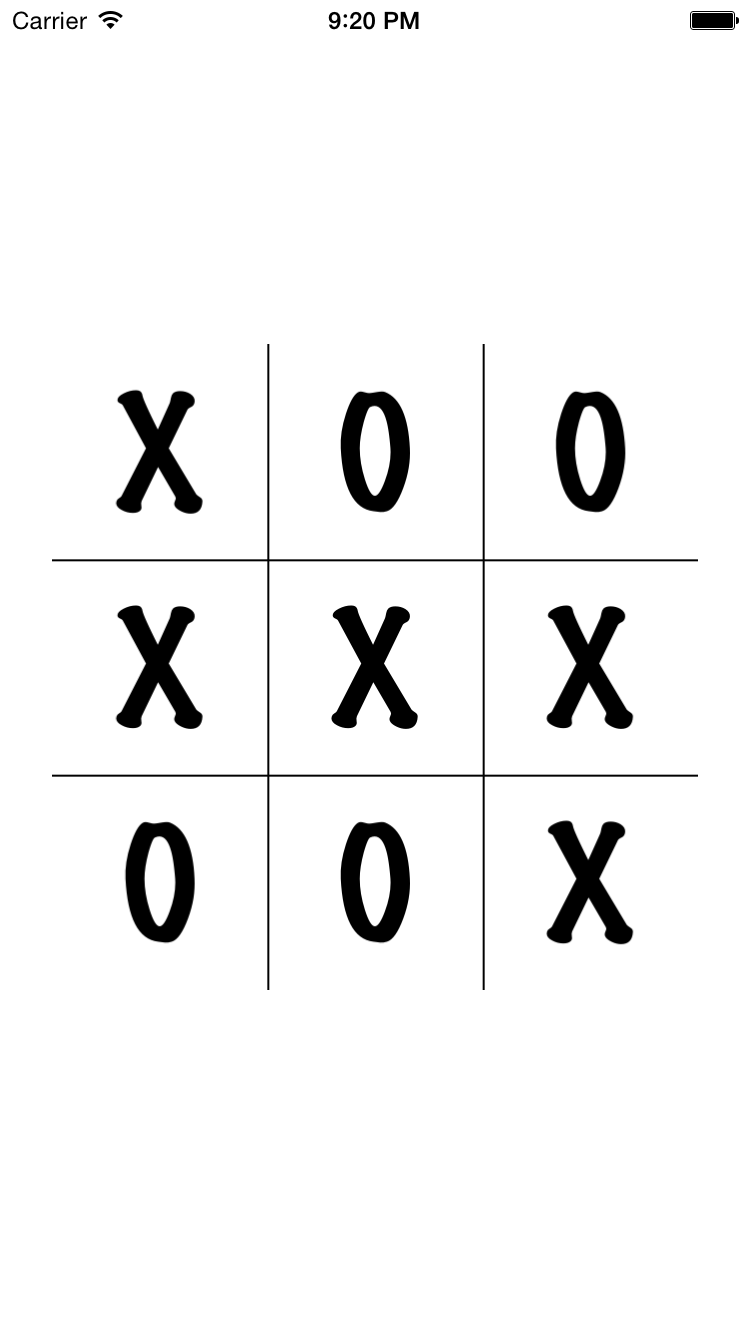
This returns the player for the given position. It reads out of the dictionary. The dictionary returns an optional, because there might not be a value for the given key.

Finally, add the following line to the end of viewDidLoad():

self.board.delegate = self

This sets the view controller as the delegate of the board so that it will be notified of touches and will be responsible for telling the board which player is at each position.

That’s it! You’ve built the game! Build & run the app and then tap away. You’ll see something like this:



Woop! But you’ll notice there’s no way to start a new game. You’ll fix that in the challenge.