When Apple announced Swift at the WWDC, the iOS developer community became excited about the opportunities to improve the way in which they build iOS apps. Swift is a user-friendly language with a smooth learning curve; it is safe, robust, and flexible, and it introduces new ways to solve old problems.

Swift by Example is a fast-paced, practical guide that shows you how to develop iOS apps using Swift. Through the development of six different apps, you’ll learn how to use either the right feature of the language or the right tool to solve a given problem.

By the end of the book you will be able to build well-designed apps, effectively use AutoLayout, and develop a video game.

Who this book is written for

This book is intended for those who want to learn to develop apps in Swift the right way. Whether you are an expert Objective-C programmer or new to this platform, you’ll learn quickly, grasping the code of real-world apps to use Swift effectively. Prior experience in development for Apple devices would be helpful, but is not mandatory.

What you will learn from this book

- Explore the features of Swift
- Connect to a server and parse JSON data
- Take advantage of CocoaPods to use third-party libraries
- Utilize a clean and effective architecture to decrease complexity and speed up development
- Work with the most useful parts of the iOS SDK
- Build video games with SpriteKit and SceneKit
- Develop apps from start to finish
- Implement a weather app using fake data

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'A Memory Game in Swift'
- A synopsis of the book’s content
- More information on Swift by Example
About the Author

Giordano Scalzo is a developer with 20 years of programming experience since the days of the ZX Spectrum. He has worked using C++, Java, .NET, Ruby, Python, and a ton of other languages whose names he has forgotten.

After several years of backend development, over the past 5 years, he has developed extensively for iOS, releasing more than 20 apps — apps that he wrote for clients, enterprises, or himself.

Currently, Giordano is a contractor in London, where he delivers code for iOS through his company, Effective Code (http://effectivecode.co.uk), aiming at quality and reliability. In his spare time, when he is not crafting retro game clones for iOS, he writes his thoughts at http://giordanoscalzo.com.
Preface

The introduction of Swift during the WWDC 2014 surprised the entire community of iOS developers, who were waiting for the new API brought by iOS 8 and not to be transformed into beginners.

Besides the surprise, most of them understood that this would be a great opportunity to create a new world of libraries, patterns, best practices, and so on. On the other hand, communities of programmers in different languages who were intimidated by the first (and rough) impact of Objective-C started getting attracted by Swift, which was less intimidating given its friendly syntax.

Whether you are part of the first or second group, Swift by Example will introduce the world of app development to you. Through simple step-by-step chapters, this book will teach you how to build both utility and game apps and, while building them, you’ll learn the basics of Swift and iOS.

What this book covers

Chapter 1, Welcome to the World of Swift, introduces the Swift syntax and the most important features brought forth by the language. To show you how to build a project with Xcode, a simple app is created.

Chapter 2, A Memory Game in Swift, shows the creation of a complete game, with images and animations, without using any game framework and using only the fundamental iOS libraries.

Chapter 3, A TodoList App in Swift, teaches you how to create a real-world utility app, handling library dependencies with CocoaPods.

Chapter 4, A Pretty Weather App, shows you how to create a nice-looking app that retrieves data from third-party servers.
Preface

Chapter 5, Flappy Swift, covers SpriteKit, the 2D iOS game engine, and the creation of a clone of the famous game Flappy Bird.

Chapter 6, Polishing Flappy Swift, completes the game by adding Game Center support and various "added value" touches.

Chapter 7, Cube Runner, covers SceneKit and 3D programming by implementing a 3D endless runner game with a space theme.

Chapter 8, Completing Cube Runner, demonstrates the addition of final touches and Game Center support to the game.
After learning the fundamental parts of the language, and getting a basic introduction to creating a simple app with Xcode, it's now time to build something more complex, but by using the basics from the previous chapter. This chapter aims to show you how to structure an app, creating clean and simple code, and how to make it appealing to the user with nice colors and smooth animations.

Compared to the previous chapter, this chapter is more advanced because I think the best way to learn new concepts is to see them in a real app. One of the many ways to show content in an iOS app is by using UICollectionView, which is a component that lays the subcomponents as a flow of cell. A good introduction to UICollectionView can be found at http://nshipster.com/uicollectionview/.

The app is...

The app we are going to implement is a UIKit implementation of a memory game—a solitaire version. A memory game, also known as Concentration, is a card game where the player must match all the cards, which start reversed, turning up two of them in each turn. If the cards match, they are removed from the table. Otherwise, they are turned down again and the score increases. The goal is to clear the table with the lowest score possible.

In our implementation, we are going to use only standard UIKit components, and to see another way of creating the interface in Xcode, we'll create all of our UI directly in code without using Interface Builder.

Let's start prototyping the screens. Despite this being an educational app, we want it to be a pretty and fun app, so we need at least one option to decide the difficulty, selecting the quantity of cards laid on the table.
A Memory Game in Swift

The following are the screens we'll implement for the app; the first is for selecting the difficulty — basically selecting the number of the cards in the deck:

![Screen 1: Difficulty Selection]

![Screen 2: Card Layout]

**Building the skeleton of the app**

As we have already seen in the previous chapter, we can create our app by going to File | New | Project, and then selecting Single View Application from the list of templates.

For simplifying the handling of different resolutions, our memory game is in landscape mode only, so when the creation of the template has been completed, uncheck Portrait as the allowed device orientation, as shown in the following screenshot:

| Device Orientation |  | Portrait |  | Upside Down | ✔️ Landscape Left | ✔️ Landscape Right |
The menu screen
Let's start implementing the first view, in which we can select the level of the game.

Implementing the basic menu screen
As we have planned to implement all of the UI in the code itself, we won't bother to touch the storyboard. We will proceed to open the ViewController class to implement the menu screen.

Looking at the mockup, we can see that we have three levels of difficulty: easy, medium, and hard. There are three buttons, each one used to select one of them. Also, the buttons are horizontally centered and vertically equidistant.

First of all, we define an enumeration to describe the difficulty.

Then, we implement the setup for the layout:

```swift
enum Difficulty {
    case Easy, Medium, Hard
}
```

Just for the sake of readability, we group the methods needed to implement a feature in a separated extension, leaving in the main class only the public functions and the variable definition. Although extensions were born for a different goal, which is to extend classes we don't have the source for, I've found that grouping together methods in an extension helps describe the goals of those methods. Matt Thompson, the creator of AFNetworking, the most used network library for iOS, used the same approach in Alamofire (https://github.com/Alamofire/Alamofire):

```swift
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        setup()
    }
}

private extension ViewController {
    func setup() {
        view.backgroundColor = UIColor.whiteColor()
        buildButtonWithCenter(CGPoint(x: view.center.x, y: view.center.y/2.0), title: "EASY", color: UIColor.greenColor(), action: "onEasyTapped:")
    }
}
```
A Memory Game in Swift

```swift
func buildButtonWithCenter(center: CGPoint, title: String, color: UIColor, action: Selector) {
    let button = UIButton()
    button.setTitle(title, forState: .Normal)
    button.setTitleColor(UIColor.blackColor(), forState: .Normal)
    button.frame = CGRect(origin: CGPoint(x: 0, y: 0), size: CGSize(width: 200, height: 50))
    button.center = center
    button.backgroundColor = color
    button.addTarget(self, action: action, forControlEvents: UIControlEvents.TouchUpInside)
    view.addSubview(button)
}
```

Again, we are not yet relying on AutoLayout to establish relations between the components, so we pass the coordinates of each button to the initializer method. In the same method, we also pass the text to be presented as a caption and the background color.

The last parameter, called action, contains the name of the method inside the ViewController that the button must call when pressed. The following implementation of buildButtonCenter() shows you how to create a button programmatically:

```swift
func buildButtonWithCenter(center: CGPoint, title: String, color: UIColor, action: Selector) {
    let button = UIButton()
    button.setTitle(title, forState: .Normal)
    button.setTitleColor(UIColor.blackColor(), forState: .Normal)
    button.frame = CGRect(origin: CGPoint(x: 0, y: 0), size: CGSize(width: 200, height: 50))
    button.center = center
    button.backgroundColor = color
    button.addTarget(self, action: action, forControlEvents: UIControlEvents.TouchUpInside)
    view.addSubview(button)
}
```
The last statement before adding the button to the view is the way to connect a callback to an event - the programmatic equivalent of creating a line connecting an event of the button to `@IBAction` using Interface Builder. This is a technique we saw in the previous chapter.

Because all the actions are logically tied together, we create another extension to group them:

```swift
extension ViewController {
    func onEasyTapped(sender: UIButton) {
        newGameDifficulty(.Easy)
    }

    func onMediumTapped(sender: UIButton) {
        newGameDifficulty(.Medium)
    }

    func onHardTapped(sender: UIButton) {
        newGameDifficulty(.Hard)
    }

    func newGameDifficulty(difficulty: Difficulty) {
        switch difficulty {
        case .Easy:
            println("Easy")
        case .Medium:
            println("Medium")
        case .Hard:
            println("Hard")
        }
    }
}
```

If we run the app now by going to **Product | Run**, we can see that we have almost implemented the screen in the mockup, as you can see in the following screenshot:
Also, by tapping the buttons, we can verify that the button calls the correct function. We must see the correct message in the console when we press a button.

Although the screen is how we expected to implement it, it isn't very appealing, so before proceeding to implement the game ViewController, we customize the color palette to make the UI prettier.

**Creating a nice menu screen**

Because the flat design is very fashionable lately, we go to [http://flatuicolors.com](http://flatuicolors.com) to choose a few colors to decorate our components.

After choosing the color, we extend the UIColor class:

```swift
extension UIColor {
    class func greenSea() -> UIColor {
        return UIColor.colorComponents((22, 160, 133))
    }
    class func emerald() -> UIColor {
        return UIColor.colorComponents((46, 204, 113))
    }
    class func sunflower() -> UIColor {
        return UIColor.colorComponents((241, 196, 15))
    }
    class func alizarin() -> UIColor {
        return UIColor.colorComponents((231, 76, 60))
    }
}

private extension UIColor {
    class func colorComponents(components: (CGFloat, CGFloat, CGFloat)) -> UIColor {
        return UIColor(red: components.0/255, green: components.1/255, blue: components.2/255, alpha: 1)
    }
}
```

With this extended palette, we can change the setup of the ViewController:

```swift
func setup() {
    view.backgroundColor = UIColor.greenSea()

    buildButtonCenter(CGPoint(x: view.center.x, y: view.center.y/2.0),
        title: "EASY", color: UIColor.emerald(), action: "onEasyTapped:")
```

---

[46]
buildButtonCenter(CGPoint(x: view.center.x, y: view.center.y),
    title: "MEDIUM", color: UIColor.sunflower(), action:
    "onMediumTapped:")
buildButtonCenter(CGPoint(x: view.center.x, y: view.center.y*3.0/2.0),
    title: "HARD", color: UIColor.alizarin(), action:
    "onHardTapped:"}

The result, as shown in this screenshot, is definitely prettier, reminding us of a real card table:

![Image of card table with buttons for EASY, MEDIUM, HARD]

Now we can implement a proper `newGame()` function:

```swift
    func newGameDifficulty(difficulty: Difficulty) {
        let gameViewController = MemoryViewController(difficulty: difficulty)
        presentViewController(gameViewController, animated: true, completion: nil)
    }
```

You can find the code at [https://github.com/gscalzo/Memory/tree/menu](https://github.com/gscalzo/Memory/tree/menu).

The game screen
Before implementing the game, let's proceed to build the layout of the cards on the table.
The structure
After creating the MemoryViewController file, we add the class life cycle functions:

```swift
class MemoryViewController: UIViewController {
    private let difficulty: Difficulty

    init(difficulty: Difficulty) {
        self.difficulty = difficulty
        super.init(nibName: nil, bundle: nil)
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    deinit{
        println("deinit")
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        setup()
    }
}

// MARK: Setup
private extension MemoryViewController {
    func setup() {
        view.backgroundColor = UIColor.greenSea()
    }
}
```

Besides the initializer that accepts the chosen difficulty, although it's not used, we need to add the `required initializer` with NSCoder. Moreover, you should note that we need to call the parent initializer with `nibName` and the bundle, used when a UIViewController is built from an XIB file. If we called a plain `super.init()`, we would receive a runtime error, because the empty one is a `convenience initializer`—an initializer that calls a `required initializer` in the same class, which in our case is not implemented.

Although not mandatory, we have implemented the deinitializer as well, inserting just a debug log statement to verify that the class is correctly removed from the memory when dismissed. Thus, a retain cycle is avoided.

Finally, we come to this comment:

```
// MARK: Setup
```
This is a special comment that tells Xcode to present the sentence in the structure of a class, as shown in the following screenshot, to facilitate navigation to a different part of the class:

![Screenshot of Xcode showing class structure]

**Adding the collectionView class**

Let's move on to implementing the layout of the card. We'll use the UICollectionView class to lay the cards on the table. The UICollectionView class is a view that arranges the contained cells to follow a layout we set during the setup. In this case, we set a flow layout in which each card follows the previous one, and it creates a new row when the right border is reached.

We set the properties for the view and a model to fulfill the collection view:

```swift
private var collectionView: UICollectionView!
private var deck: Array<Int>!
```

Next, we write the function calls to set up everything in `viewDidLoad`, so that the functions are called when the view is loaded:

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    setup()
    start()
}
```

The `setup()` function basically creates and configures `CollectionView`:

```swift
// MARK: Setup
private extension MemoryViewController {
    func setup() {
        view.backgroundColor = UIColor.greenSea()
    }
}
```
let space: CGFloat = 5

let (covWidth, covHeight) = collectionViewSizeDifficulty(difficulty, space: space)
let layout = layoutCardSize(cardSizeDifficulty(difficulty, space: space), space: space)

collectionView = UICollectionView(frame: CGRect(x: 0, y: 0, width: covWidth, height: covHeight), collectionViewLayout: layout)
collectionView.center = view.center
collectionView.dataSource = self
collectionView.delegate = self
collectionView.scrollEnabled = false
collectionView.registerClass(UICollectionViewCell.self, forCellWithReuseIdentifier: "cardCell")
collectionView.backgroundColor = UIColor.clearColor()
self.view.addSubview(collectionView)

After setting the color of the table, we define a constant, space, to set the space between every two cards.

Next, we calculate the size of the collection view given the difficulty, and, hence, the number of rows and columns; then, the layout. Finally, we put everything together to build the collection view:

```swift
func collectionViewSizeDifficulty(difficulty: Difficulty, space: CGFloat) -> (CGFloat, CGFloat) {
    let (columns, rows) = sizeDifficulty(difficulty)
    let (cardWidth, cardHeight) = cardSizeDifficulty(difficulty, space: space)

    let covWidth = columns*(cardWidth + 2*space)
    let covHeight = rows*(cardHeight + space)
    return (covWidth, covHeight)
}
```

The cardSizeDifficulty() function calculates the size of the collection view by multiplying the size of each card by the number of rows or columns:

```swift
func cardSizeDifficulty(difficulty: Difficulty, space: CGFloat) -> (CGFloat, CGFloat) {
    let ratio: CGFloat = 1.452
    let
```
let (columns, rows) = sizeDifficulty(difficulty)
let cardHeight: CGFloat = view.frame.height/rows - 2*space
let cardWidth: CGFloat = cardHeight/ratio
return (cardWidth, cardHeight)
}

Sizing the components
As mentioned at the start of this chapter, we are not using AutoLayout, but we need
to handle the issue of different screen sizes somehow. Hence, using basic math, we
adapt the size of each card to the available size in the screen:

```swift
func layoutCardSize(cardSize: (cardWidth: CGFloat, cardHeight: CGFloat), space: CGFloat) -> UICollectionViewLayout {
    let layout: UICollectionViewFlowLayout = 
    UICollectionViewFlowLayout()
    layout.sectionInset = UIEdgeInsets(top: space, left: space, 
    bottom: space, right: space)
    layout.itemSize = CGSize(width: cardSize.cardWidth, height: 
    cardSize.cardHeight)
    layout.minimumLineSpacing = space
    return layout
}
```

As mentioned before, the UICollectionView class shows a series of cells
in its content view, but the way in which the cells are presented—as a grid
or a vertical pile, the space between them—is defined by an instance of
UICollectionViewFlowLayout.

Finally, we set up the layout, defining the size of each cell and how they are
separated and laid out.

We have seen that there is a connection between the difficulty setting and the
size of the grid of the cards, and this relation is simply implemented using
switch statements:

```swift
// MARK: Difficulty
private extension MemoryViewController {
    func sizeDifficulty(difficulty: Difficulty) -> (CGFloat, CGFloat) {
        switch difficulty {
        case .Easy:
            return (4,3)
        case .Medium:
            return (6,4)
        case .Hard:
A Memory Game in Swift

```swift
func numCardsNeededDifficulty(difficulty: Difficulty) -> Int {
    let (columns, rows) = sizeDifficulty(difficulty)
    return Int(columns * rows)
}
```

**Connecting the datasource and the delegate**

You have probably noticed that when we created `CollectionView`, we set the `ViewController` itself as `dataSource` and `delegate`:

```swift
collectionView.dataSource = self
collectionView.delegate = self
```

A common pattern found in Cocoa in many components is the delegate pattern, where part of the behavior is delegated to another object, and that object must implement a particular protocol.

In the case of `UICollectionView`, and likewise for `UITableView`, we have to delegate one to provide the content for the view, the datasource, and the other to react to events from the view itself. In this way, the presentation level is completely decoupled from the data and the business logic, which reside in two specialized objects.

So, we need to implement the required methods of the protocols:

```swift
// MARK: UICollectionViewDataSource
extension MemoryViewController: UICollectionViewDataSource {
    func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return deck.count
    }

    func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
        var cell = collectionView.dequeueReusableCellWithReuseIdentifier("cardCell", forIndexPath: indexPath) as! UICollectionViewCell
        cell.backgroundColor = UIColor.sunflower()
        return cell
    }
}
```
As you can notice, in the method called for the cell at a certain position, we are calling a method to reuse a cell, instead of creating a new one. This is a nifty feature of UICollectionView that saves all the cells in a cache and can reuse those outside the visible view. This not only saves a lot of memory, but is also really efficient, because creating new cells during scrolling could be CPU-consuming and affect performance.

Because we want to see just the flow of the card, we use the default empty cell as the view cell, changing the color of the background:

```swift
// MARK: UICollectionViewDelegate
extension MemoryViewController: UICollectionViewDelegate {
    func collectionView(collectionView: UICollectionView,
                         didSelectItemAtIndexPath indexPath: NSIndexPath) {
    }
}
```

For the delegate, we simply prepare ourselves to handle a touch on the card. Because we don't need a real deck of cards, an array of integers is enough as a model:

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    setup()
    start()
}

private func start() {
    deck = Array<Int>(count: numCardsNeededDifficulty(difficulty),
                      repeatedValue: 1)
    collectionView.reloadData()
}
```
A Memory Game in Swift

Upon running the app now and choosing a level, we will have all our empty cards laid out, like this:

![Simulator screenshot with empty cards]

Using a different simulator and the iPhone 5 or 4S, we can see that our table adapts its size smoothly.

The code for the app implemented so far can be downloaded from https://github.com/gscalzo/Memory/tree/card_layout.

Implementing a deck of cards

So far, we have implemented a pretty generic app that lays out views inside a bigger view. Let's proceed to implement the foundation of the game—a deck of cards.

What we are expecting

Before implementing the classes for a deck of cards, we must define the behavior we are expecting, whereby we implement the calls in MemoryViewController, assuming that the Deck object already exists. First of all, we change the type in the definition of the property:

```swift
private var deck: Deck!
```
Then, we change the implementation of the `start()` function:

```swift
private func start() {
    deck = createDeck(numCardsNeededDifficulty(difficulty))
    collectionView.reloadData()
}

private func createDeck(numCards: Int) -> Deck {
    let fullDeck = Deck.full().shuffled()
    let halfDeck = fullDeck.deckOfNumberOfCards(numCards/2)
    return (halfDeck + halfDeck).shuffled()
}
```

We are saying that we want a deck to be able to return a shuffled version of itself, and which can return a deck of a selected numbers of its cards. Also, it can be created using the plus operator (+) to join two decks. This is a lot of information, but it should help you learn a lot regarding structs.

**The Card entity**

There isn't anything regarding the entities inside `Deck` so far, but we can assume that it is a `Card` struct, and that it uses plain enumerations. A `Suit` and a `Rank` parameter define a card, so we can write this code:

```swift
class Card {
    enum Suit: Printable {
        case Spades, Hearts, Diamonds, Clubs
        var description: String {
            switch self {
            case .Spades:
                return "spades"
            case .Hearts:
                return "hearts"
            case .Diamonds:
                return "diamonds"
            case .Clubs:
                return "clubs"
            }
        }
    }

    enum Rank: Int, Printable {
        case Ace = 1
        case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
        case Jack, Queen, King
    }
}
```
A Memory Game in Swift

```swift
var description: String {
    switch self {
    case .Ace:
        return "ace"
    case .Jack:
        return "jack"
    case .Queen:
        return "queen"
    case .King:
        return "king"
    default:
        return String(self.rawValue)
    }
}
```

Note that we have used an integer as a type in Rank, but not in Suit. That's because we want the possibility to create a Rank from an integer—its raw value—but not for Suit. This will soon become clearer.

We have implemented the Printable protocol in order to be able to print the enumeration. The Card parameter is nothing more than a pair of Rank and Suit:

```swift
struct Card: Printable, Equatable {
    private let rank: Rank
    private let suit: Suit

    var description: String {
        return "\(rank.description)_of_\(suit.description)"
    }
}
```

We have also implemented the Equatable protocol to be able to check whether two cards are of the same value.

### Crafting the deck

Now we can implement the constructor of a full deck, iterating through all the values of the Rank and Suit enumerations:

```swift
struct Deck {
    private var cards = [Card]()
static func full() -> Deck {
    var deck = Deck()
    for i in Rank.Ace.rawValue...Rank.King.rawValue {
            let card = Card(rank: Rank(rawValue: i)!,
                            suit: suit)

            deck.cards.append(card)
        }
    }
    return deck
}

Shuffling the deck

The next function we will implement is shuffled():

    // Fisher-Yates (fast and uniform) shuffle
    func shuffled() -> Deck {
        var list = cards
        for i in 0..<(list.count - 1) {
            let j = Int(arc4random_uniform(UInt32(list.count - i))) + i
            swap(&list[i], &list[j])
        }
        return Deck(cards: list)
    }

The usual way to shuffle a deck of cards in a computer program is to use the Fisher-Yates algorithm. Starting from the first card, we iterate until the very end, each turn swapping the current card with a random card in the set with index greater than the current one. A complete explanation of this can be found on Wikipedia at http://en.wikipedia.org/wiki/Fisher–Yates_shuffle.

If you look carefully at the swap() function, you will see an ampersand (&) symbol before the parameters. It means that the parameters are inout, and that they can be changed inside functions. We can consider inout parameters as shared variables between the caller and the called.
A Memory Game in Swift

Finishing the deck
We are almost done with the expected behavior of Deck; we just need to add the creation of a subset of Deck:

```swift
func deckOfNumberOfCards(num: Int) -> Deck {
    return Deck(cards: Array(cards[0..<num]))
}
```

Note that by using the notation for the range `[..<]`, the upper bound is not included in the range, whereas by using `[..]`, the upper bound is included. We can create that by exploiting the splicing feature of the Swift `Array`. Using the same trick, we create the sum operator:

```swift
func +(deck1: Deck, deck2: Deck) -> Deck {
    return Deck(cards: deck1.cards + deck2.cards)
}
```

The last function left is the `count` property, which we implement by using a computed property:

```swift
var count: Int {
    get {
        return cards.count
    }
}
```

Before moving on to implement the remainder of the game, we want to check whether everything works fine, so we add a log after creating the deck, like this:

```swift
deck = createDeck(numCardsNeededDifficulty(difficulty))
for i in 0..<deck.count {
    println("The card at index \(i) is \(deck[i].description)"
}
```

Unfortunately, the compiler complains that it doesn't know how to retrieve the element at a specified index.

For the purpose of mimicking the accessor of an array, Swift provides a special computed property to add to the definition of our struct—`subscript`. Implementing the subscript just involves forwarding the request to the private property cards:

```swift
subscript(index: Int) -> Card {
    get {
        return cards[index]
    }
}
```
Now the app compiles. If we run it, we get a console output like this:

The card at index [0] is [8_of_clubs]
The card at index [1] is [ace_of_spades]
The card at index [2] is [ace_of_clubs]
The card at index [3] is [ace_of_hearts]
The card at index [4] is [9_of_hearts]
The card at index [5] is [ace_of_hearts]
The card at index [6] is [queen_of_clubs]
The card at index [7] is [ace_of_clubs]
The card at index [8] is [ace_of_spades]
The card at index [9] is [queen_of_clubs]
The card at index [10] is [9_of_hearts]

The source code for this block can be downloaded from https://
github.com/gscalzo/Memory/tree/foundation_for_cards.

Put the cards on the table
Finally, let's add the card images and implement the entire game.

Adding the assets
Now that everything works, let's create a nice UI again.

First of all, let's import all the assets into the project. There are plenty of free cards assets on the Internet, but if you are lazy, I've prepared for you a complete deck of images ready for this game, and you can download them from https://github.com/gscalzo/Memory/blob/master/Assets/CardImages.zip?raw=true.

The archive contains an image for the back of the cards, and all the front images for the others. To include them in the app, select the image assets file from the project structure view, as shown in this screenshot:
A Memory Game in Swift

After selecting the catalogue, the images can be dragged into Xcode, as in the following screenshot:

In this operation, you must pay attention and ensure that you move all the images from 1x to 2x. Otherwise, when you run the app, you will see them pixelate.

The CardCell structure

Let's go ahead and implement our CardCell. Again, we pretend that we already have the class, so we register that class during the setup of CollectionView:

```swift
func setup() {
//...
collectionView.registerClass(CardCell.self,
   forCellWithReuseIdentifier: "cardCell")
//...
}
```

Then, we implement the rendering of the class when the datasource protocol asks for a cell, given an index:

```swift
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
var cell = collectionView.dequeueReusableCellWithIdentifier("cardCell",
   forIndexPath: indexPath) as! CardCell
let card = deck[indexPath.row]
cell.renderCardName(card.description, backImageName: "back")
return cell
}
```
We are trying to push as much presentation code as we can into the new class to decouple the responsibilities between `Cell` and the controller, which holds the model. `CardCell` contains only `UIImageView` to present the card images, and two properties to hold the names of the front and back images:

```swift
class CardCell: UICollectionViewCell {
    private let frontImageView: UIImageView!
    private var cardImageName: String!
    private var backImageName: String!

    override init(frame: CGRect) {
        frontImageView = UIImageView(frame: CGRect(x: 0, y: 0,
            width: frame.size.width,
            height: frame.size.height))
        super.init(frame: frame)
        contentView.addSubview(frontImageView)
        contentView.backgroundColor = UIColor.clearColor()
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func renderCardName(cardImageName: String, backImageName: String){
        self.cardImageName = cardImageName
        self.backImageName = backImageName
        frontImageView.image = UIImage(named: self.backImageName)
    }
}
```

If you run the app now, you should see some nice cards face down.

**Handling touches**

Now, let's get them face up!

```swift
func collectionView(collectionView: UICollectionView,
    didSelectItemAtIndexPath indexPath: NSIndexPath) {
    let cell = collectionView.cellForItemAtIndexPath(indexPath)
        as CardCell
    cell.upturn()
}
```
This code is pretty clear, and now we only need to implement the `upturn()` function inside `CardCell`:

```swift
func upturn() {
    UIView.transitionWithView(contentView,
        duration: 1,
        options: .TransitionFlipFromRight,
        animations: {
            self.frontImageView.image =
                UIImage(named: self.cardImageName)
        },
        completion: nil)
}
```

By leveraging a handy function inside the `UIView` class, we have created a nice transition from the back image to the front image, simulating the flip of a card.

To complete the functions required to manage the card from a visual point of view, we implement the `downturn()` function in a similar way:

```swift
func downturn() {
    UIView.transitionWithView(contentView,
        duration: 1,
        options: .TransitionFlipFromLeft,
        animations: {
            self.frontImageView.image =
                UIImage(named: self.backImageName)
        },
        completion: nil)
}
```

To test them, we turn down the card 2 seconds after we turned it up. To run a delayed function, we use the `dispatch_after` function, but to remove the boilerplate call, we wrap it in a smaller common function:

```swift
extension UIViewController {
    func execAfter(delay: Double, block: () -> Void) {
        dispatch_after(
            dispatch_time(
                DISPATCH_TIME_NOW,
                Int64(delay * Double(NSEC_PER_SEC))
            ),
            dispatch_get_main_queue(), block)
    }
}
```
So, after having the card turned up, we turn it down using this newly implemented function:

```swift
func collectionView(collectionView: UICollectionView,
didSelectItemAtIndexPath indexPath: NSIndexPath) {
    //...
    cell.upturn()
    execAfter(2) {
        cell.downturn()
    }
}
```

By running the app, we now see the cards turning up and down, with smooth and nice animation.

### Finishing the game

In this section, we will finally be able to play the game.

#### Implementing the game logic

After having all the required functions in place, it's now straightforward to complete the game. First of all, we add the instance variables to hold the number of the pairs already made, the current score, and the list of selected cards turned up:

```swift
private var selectedIndexes = Array<NSIndexPath>()
private var numberOfPairs = 0
private var score = 0
```

Then, we put the logic when a card is selected:

```swift
func collectionView(collectionView: UICollectionView,
didSelectItemAtIndexPath indexPath: NSIndexPath) {
    if selectedIndexes.count == 2 ||
        contains(selectedIndexes, indexPath) {
        return
    }
    selectedIndexes.append(indexPath)

    let cell = collectionView.cellForItemAtIndexPath(indexPath)
        as CardCell
    cell.upturn()

    if selectedIndexes.count < 2 {
        return
    }
```
We first check whether we have touched an already turned-up card or if we have two cards turned up. If not, we save the index. Then, we check whether we have flipped the first card, and, if not, we proceed to check the values of the cards.

The pattern of checking a condition and leaving the current function if the condition is true is called Guard. It helps make the code more readable by avoiding the use of the else clause and the nesting of curly braces.

We got a pair

As in the previous part of the source, we implement the missing actions in a private extension:

```swift
// MARK: Actions
private extension MemoryViewController {
    func checkIfFinished() {
    }
    func removeCards() {
    }
    func turnCardsFaceDown() {
    }
}
```

The first one checks whether we have completed all the pairs, and if so, it presents a popup with the score and returns to the main menu:

```swift
func checkIfFinished() {
    if numberOfPairs == deck.count/2 {
        showFinalPopUp()
    }
}
```
func showFinalPopUp() {
    var alert = UIAlertController(title: "Great!",
                                message: "You won with score: \(score)!",
                                preferredStyle: UIAlertControllerStyle.Alert)
    alert.addAction(UIAlertAction(title: "Ok", style: .Default,
                                  handler: { action in
                                      self.dismissViewControllerAnimated(true, completion: nil)
                                      return
                                   })))

    self.presentViewController(alert, animated: true, completion: nil)
}

Note that in iOS 8, UIAlertController is slightly different from that in the previous version. In our case, a simple dialog box with an Ok button is enough.

If the cards are equal, we need to remove them:

    func removeCards(){
        execAfter(1.0) {
            self.removeCardsAtPlaces(self.selectedIndexes)
            self.selectedIndexes = Array<NSIndexPath>()
        }
    }

    func removeCardsAtPlaces(places: Array<NSIndexPath>){
        for index in selectedIndexes {
            let cardCell = collectionView.
cellForItemAtIndexPath(index)
                as CardCell
            cardCell.remove()
        }
    }

The remove() function in CardCell is similar to turnUp() and turnDown(), but instead of making a transition, it just performs an animation before hiding the cell:

    func remove() {
        UIView.animateWithDuration(1,
                    animations: {
                        self.alpha = 0
                    },
                    completion: { completed in
                    self.hidden = true
                })
    }
We played a wrong turn
Finally, if the cards are different, we need to turn them down:

```swift
func turnCardsFaceDown()
{
    execAfter(2.0) {
        self.downturnCardsAtPlaces(self.selectedIndexes)
        self.selectedIndexes = Array<NSIndexPath>()
    }
}

func downturnCardsAtPlaces(places: Array<NSIndexPath>){
    for index in selectedIndexes {
        let cardCell = collectionView.
        cellForItemAtIndexPath(index)
            as CardCell
        cardCell.downturn()
    }
}
```

Et voilà! The game is completed
As you can see in the following screenshot, the game presents smooth animation and nice images:
Summary

In this chapter, we implemented our first complete app, beginning with using basic components, and then moving on to use more advanced techniques to create smooth animation, without relying on game frameworks such as Cocos2d or SpriteKit.

We saw when, and how, to use structs in an effective way, and how to split responsibilities among different classes. Moreover, we experimented and saw how to separate different parts of the same class using extensions, and how to design an interface of a class or struct, pretending we have already implemented it.

You have learned a few things about puzzle games, and it's now time to move on to something different, but more similar to a normal app we'll have a chance to work on—a TodoList app.
Where to buy this book

You can buy Swift by Example from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.