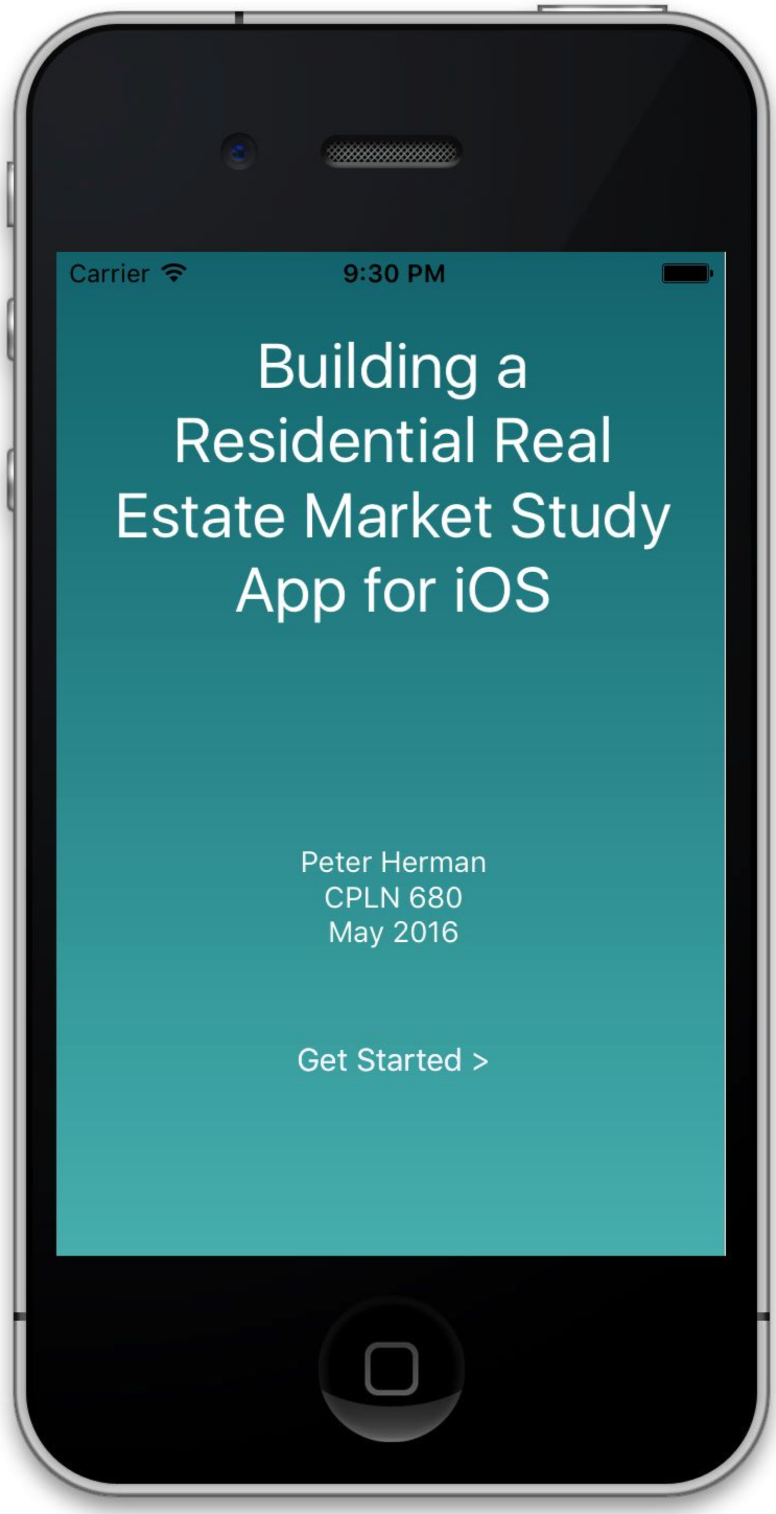


Building a Residential Real Estate Market Study App for iOS

Peter Herman
CPLN 680
May 2016

Abstract

Several iOS apps exist which use U.S. Census American Community Survey housing data, but none are geared towards housing development. The beginning of this paper justifies the creation of one such an app, named MarketStudy. The bulk of this paper is dedicated to explaining how MarketStudy was created, and detailing its functionality using examples of Swift code. Finally, major design choices and causes of common error are explained. This paper concludes suggestions for further directions MarketStudy could be taken in.

Introduction

The number one mistake amateur real estate developers make is paying too much for land. As with many economic inefficiencies, overpaying for land stems from the purchaser's lack of perfect information. In other words, the more the buyer in a real estate transaction can know about the market surrounding a parcel, the better deal he or she is likely to receive. This effect is not limited to real estate developers. A prospective renter touring several apartments is better off knowing the median rent of the area in which they are leasing. A small businessman looking to open up a storefront in a neighborhood would be well aided by knowing its average income. Most real estate transactions occur without the ability of a purchaser to conduct an extensive market study. Whether they know it or not, most buyers of real estate are looking to see if their prospective property follows Waldo Tobler's First Law of Geography. The First Law of Geography states that "everything is related to everything else, but near things are more related than distant things." Real estate pricing follows this law, as the hackneyed adage goes "location, location, location."

An app that could pull up American Community Survey (ACS) information surrounding a location would bring Tobler's Law right into the pocket of the purchaser. This dataset contains everything from what percentage of their income the average resident of a census block spends on rent, to the most common form of transportation they take to work. This is not to imply, however, that the ACS is the only useful dataset out there for real estate buyers. Say, for example, a prospective homebuyer is conscious of how close to a home is to local elementary schools, this data is unavailable to the census. That being said, Revenue from a property is derived from rents and vacancy, and the best indication of how much revenue a developer can expect to make from a project are the rents and vacancy of the surrounding area. By having the materials make a revenue projection on at their fingertips, hopefully the app will aid in land price negotiation for both sides. As the negotiated land price rises or falls, the developers could dynamically see how land price affects their projected revenue, and both sides could more easily to a mutually profitable agreement.

Objective

Produce an iOS application for use on the iPhone that gives users information about the housing and demographic characteristics of the area around a given address using data from the U.S. Census, and the ability to use this data to calculate land value.

Rationale

As was touched upon in the introduction, MarketStudy does not undertake any analysis aside from computing land value based on user inputs. The app only exists for convenience. Two scenarios for its use come to mind: if a developer is in the field and wants to do pencil out projections of whether or not a project will be profitable, and in a land purchase negotiation where a developer quickly needs to adjust their pencil out model to accommodate for fluctuating land price offers.While those may be limited use cases, the app fills a unique niche on iOS. The major differences between MarketStudy and other major apps which pull from the Census API are as follows:

First, the Census Bureau's own demographics app, **Dwellr**. Dwellr only offers information on the state, county, and municipal levels. MarketStudy, on the other hand, delivers information on the smallest available level, the census tract. The census tract is the superior geographical unit for an app built for comparing neighborhoods for two reasons. First, it controls for population. Every census tract contains an estimated 4,000 people. This allows for comparisons across different markets in a way that municipal level data does not. Comparing the rental apartment market of Philadelphia and Wilkes-Barre is almost meaningless due to the population discrepancy. Second, residential real estate is hyper-local. The vacancy rate for residential buildings across Philadelphia is 7.14%, but in Center City, it is 1.3%. The point is, municipal level data just doesn't cut it when it comes to making real estate decisions. Some third party apps like **Pocket Census** have the same issue.

Zillow and **Trulia** offer very localized real estate data, but do not offer information about how this data has changed over time. This is because those apps and MarketStudy are appealing to two different audiences. Apartment seekers are concerned with the variety of different rents in an area, while apartment developers are more concerned with the average rent of an area and if its increasing or decreasing over time. Moreover, the American Community Survey offers more indepth rent information: it draws from currently occupied apartments rather than ones for sale.

Sitewise, Census+, and Your Census Info are all amature which all share similar problems. None of them display income data, their user interfaces do not reflect thoughtful design, and like Zillow, they do not display demographic change over time.

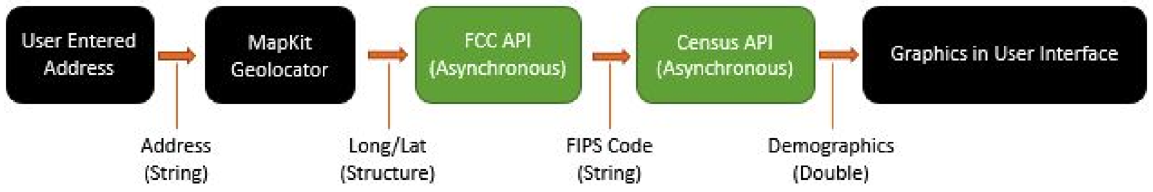
Methodology

Development

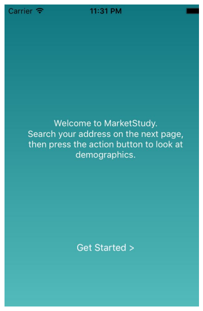
Swift was an entirely new language to me before started this semester. Not only was it a new language, it was a new type of language. The rules of the scripting languages I had learned no longer applied in the object oriented world. The major difference between the two is that Swift, being an object oriented programming language, contains all of its functionality within classes. Classes are models of the real, tactile elements of the project. Each page of the app has a class. A language like Javascript, on the other hand, operates with universal functions. Universal functions can be used to solve a problem in any part of the project. It is possible to do object oriented programming in Javascript of course, but it is *mandatory* to do it in Swift.

Switching to an object oriented language entailed looking at problems in an entirely new way. As mentioned above, classes correspond to real elements of the project. There was a class for the mapping page, a class for all of the graph pages, etc. Building the app, I proceeded page by page, or class by class, adding all the functionality I wanted on each one before moving on to the next. If I ran into any errors, they were contained on one page of my app. Interactivity between classes was confined to a single global variable. This is directly different from how one approaches problems in a procedural, or scripting, environment. In a scripting environment, one looks at the larger goal, and breaks it into smaller and smaller sub-goals, that eventually into such small components that they can be accomplished using the methods inherent to the language one is using. Changes to one of the small components of a procedurally written script changes everything , as everything interacts with everything else. Coding in a scripting environment forces one to look at their problem as a set of actions, in contrast, Swift forced me to break down my project based on its visual components, compartmentalizing my work for each page of the app.

I first began to learn Swift with the Stanford Introduction to Swift lectures on iTunes U, but quickly found I learned faster by getting my hands dirty. That is, by actually building the app, and solving the many errors inherent in amateaur coding. This method involved doing a tutorial for every piece of functionality I wanted to add to the app: geocoding, API Calls, fading elements in and out, etc. MarketStudy is thus a Frankenstein's monster of pieces of code from different tutorials that have been altered to fit my specific purposes. An outline of the workflow of my app is shown below:

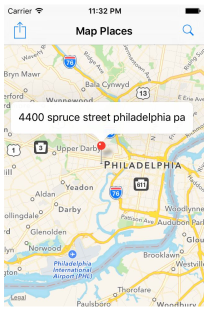


Each of the methods in the graphic above does not correspond to a page in the MarketStudy app. Rather, all of the methods take place in the Map Places Scene. To avoid any confusion, take a look at the graphic below. The user proceeds sequentially through each of these scenes, from left to right. Larger photos can be seen in the gallery section:

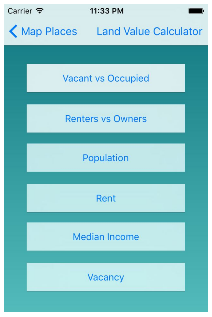


Welcome to MarketStudy.
Search your address on the next page, then press the action button to look at demographics.

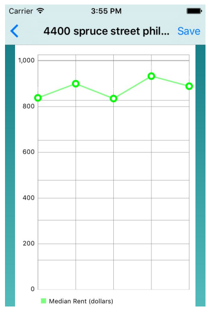
Intro Scene



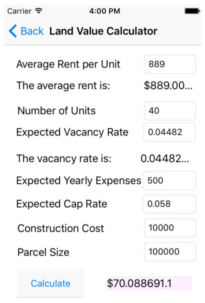
Map Places Scene



Graph Picker Scene



Graph Scenes

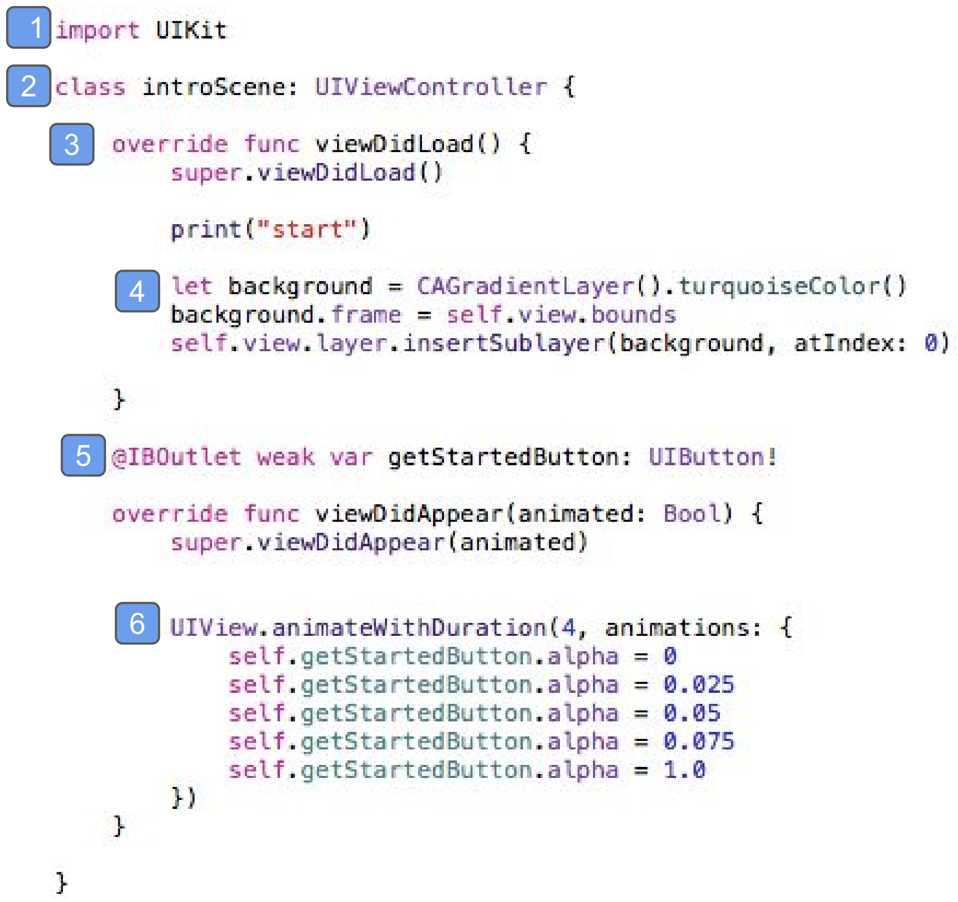


Average Rent per Unit	889
The average rent is:	$889.00...
Number of Units	40
Expected Vacancy Rate	0.04482
The vacancy rate is:	0.04482...
Expected Yearly Expenses	500
Expected Cap Rate	0.058
Construction Cost	10000
Parcel Size	100000
Calculate	$70.088691.1

Land Value Scene

The following section proceeds sequentially through the classes which control each of these scenes. The Swift code will be shown on the left, and points of interest will be explained on to the right and below. Only one example of the six graph scenes will be shown, as the code between them contains major overlap.

Intro Scene



```
1 import UIKit

2 class introScene: UIViewController {

3   override func viewDidLoad() {
        super.viewDidLoad()

        print("start")

4       let background = CAGradientLayer().turquoiseColor()
        background.frame = self.view.bounds
        self.view.layer.insertSublayer(background, atIndex: 0)

    }

5   @IBOutlet weak var getStartedButton: UIButton!

    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(animated)


6       UIView.animateWithDuration(4, animations: {
            self.getStartedButton.alpha = 0
            self.getStartedButton.alpha = 0.025
            self.getStartedButton.alpha = 0.05
            self.getStartedButton.alpha = 0.075
            self.getStartedButton.alpha = 1.0
        })
    }

}
```

1 UIKit is the main library (or framework, as Swift calls it) for iOS apps, it will be called in every scene. It handles all interactions with the elements seen on the phone: buttons, text, etc.

2 Classes are what Swift calls objects, there is one for every scene. The name of this class is introScene and it is of type UIViewController, which means it corresponds to one of the pages on the app. Swift is a very type specific language, so every definition includes a colon, then the type of what is being defined.

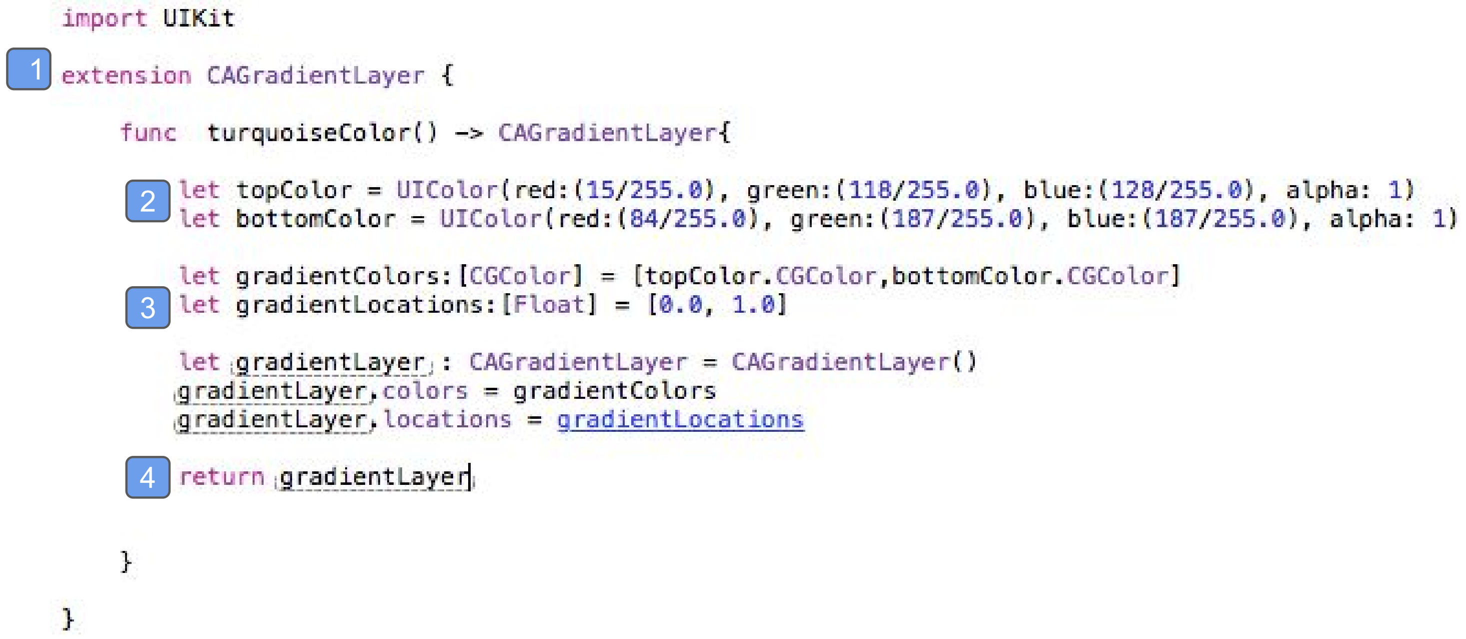
3 Anything that happens within the viewDidLoad function occurs when the scene loads. The override characteristic is attached to the function definition because viewDidLoad is defined in every scene, and Swift is very sensitive about instances with the same name.

4 These three lines call the background setting function, set up in another Swift document. The self.view.bounds call refers to the area of introScene that falls within the iPhone view. The background is set to index 0, meaning behind everything else.

5 This is the call of the button which leads the user to the next scene. There is no piece of code which corresponds to that! All one has to do is call the button (as a UIButton Object), and choose a setting in the Swift user interface, XCode.

6 This last function operates within getStartedButton, and fades it in. Every 4 sections, the getStartedButton's opacity, described as its alpha method, gets closer to 1, fully opaque.

Gradient Extension



```
import UIKit

extension CAGradientLayer {

    func  turquoiseColor() -> CAGradientLayer{

        let topColor = UIColor(red:(15/255.0), green:(118/255.0), blue:(128/255.0), alpha: 1)
        let bottomColor = UIColor(red:(84/255.0), green:(187/255.0), blue:(187/255.0), alpha: 1)

        let gradientColors:[CGColor] = [topColor.CGColor,bottomColor.CGColor]
        let gradientLocations:[Float] = [0.0, 1.0]

        let gradientLayer : CAGradientLayer = CAGradientLayer()
        gradientLayer.colors = gradientColors
        gradientLayer.locations = gradientLocations

        return gradientLayer


    }

}
```

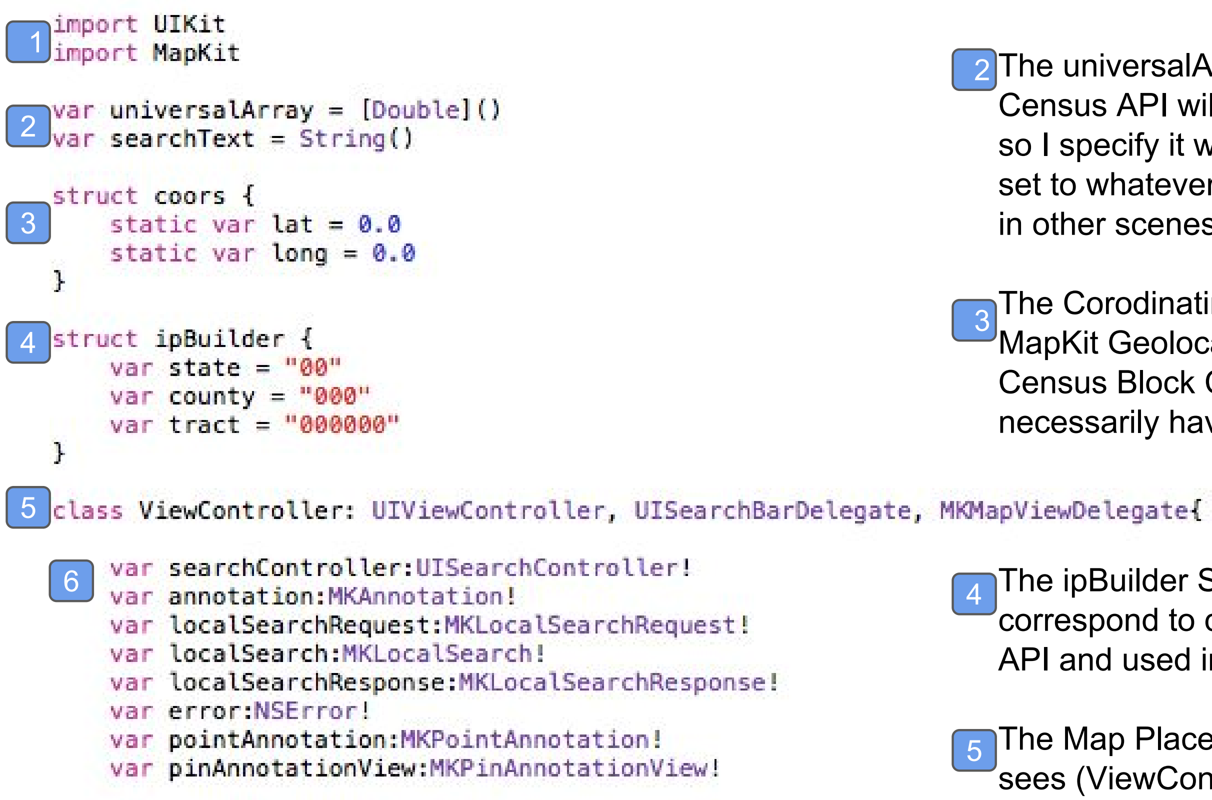
1 In Swift, extensions are places to add functionality to common types outside of a normal scene. The CAGradientLayer class is found within the UIKit library, and refers to a gradient image. In this extension, we are adding a new function for it to recognize.

2 The function created in this extension sets the background gradient to between blue and turquoise. Swift reads colors in RGB as well as hexadecimal format. Alpha refers to the opacity value.

3 The locations of these colors are 0.0, refering to the bottom of the screen, and 1.0 refering to the top. It's possible, but ugly, to set more than two colors in a gradient, at height values between 0.0 and 1.0.

4 Every function needs a return statement! Calling this function returns a CAGradientLayer, as defined in the "->" statement at the top.

Map Places Scene- Defining Variables



```
1 import UIKit
  import MapKit

2 var universalArray = [Double]()
  var searchText = String()

  struct coors {
3     static var lat = 0.0
      static var long = 0.0
  }

4 struct ipBuilder {
      var state = "00"
      var county = "000"
      var tract = "000000"
  }
5 class ViewController: UIViewController, UISearchBarDelegate, MKMapViewDelegate{

 6    var searchController:UISearchController!
      var annotation:MKAnnotation!
      var localSearchRequest:MKLocalSearchRequest!
      var localSearch:MKLocalSearch!
      var localSearchResponse:MKLocalSearchResponse!
      var error:NSError!
      var pointAnnotation:MKPointAnnotation!
      var pinAnnotationView:MKPinAnnotationView!
```

1 MapKit is the library used for Apple Maps in iOS Apps. I decided to use Apple Maps over Google Maps, because Google Maps uses a lot more battery power, and I thought it would be easier to use the map software native to the iOS environment.

2 The universalArray is the global variable where the values from the Census API will be stored. The graph library I used takes double values, so I specify it will be a list of doubles. The searchtext global string will be set to whatever the user searches, and will be used for labeling elements in other scenes.

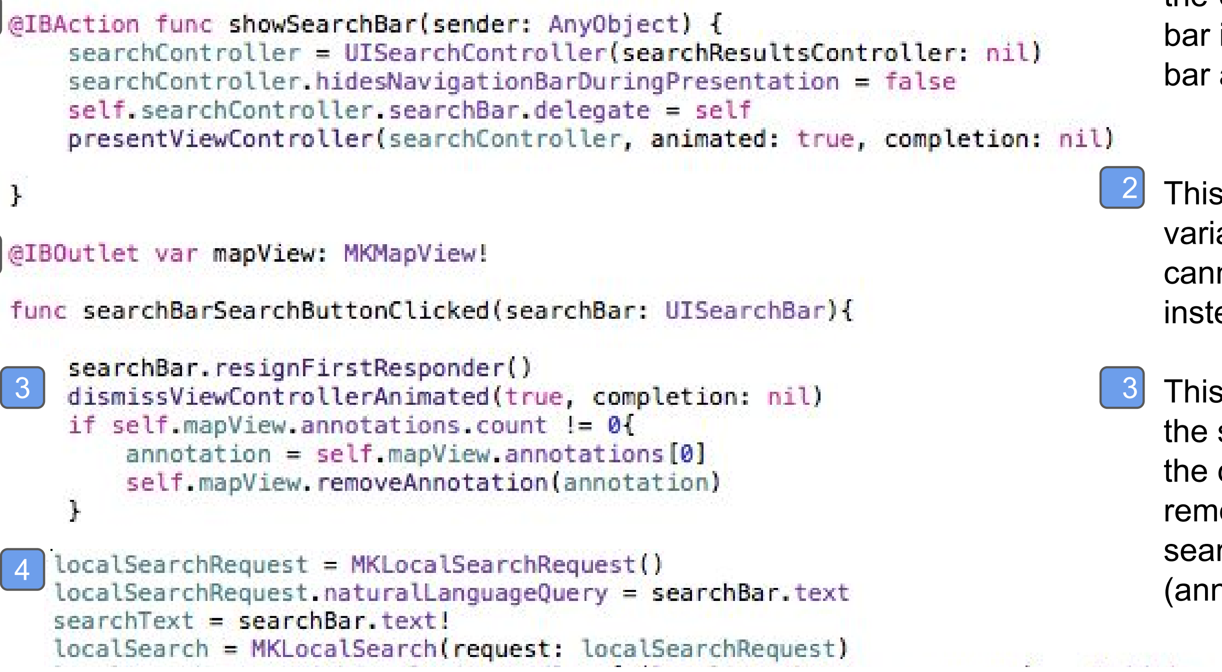
3 The Corodinatinates structure holds coordinates. It will be filled using the MapKit Geolocator, and then its values will be passed to the FCC Census Block Conversions API. This structure and the ipBuilder do not necessarily have to be a global variable since they'll be used in this scene only, but I thought they may be useful if I wanted to add functionality.

4 The ipBuilder Structure holds integer values (implicitly defined here), that correspond to different elements of the FIPS code, outputted by the FCC API and used in the call for the Census API.

5 The Map Places scene contains three classes that control what the user sees (ViewControllers). The view itself, defined on every page, the search bar, and the map.

6 The local variables within the classes correspond to different elements of the Scene. The searchController is the search bar, the annotation deals with any already existent pins on the map. The localSearchRequest contains the user inputted address, the localSearch carries out the search, and the response contains the outputted coordinates. The error term handles the error popup if a searched address isn't found. The pointAnnotation and pinAnnotationView reference the pin and its label, respectively.

Map Places Scene- The Search Bar



1
```
@IBAction func showSearchBar(sender: AnyObject) {
    searchController = UISearchController(searchResultsController: nil)
    searchController.hidesNavigationBarDuringPresentation = false
    self.searchController.searchBar.delegate = self
    presentViewController(searchController, animated: true, completion: nil)

}
```

2
```
@IBOutlet var mapView: MKMapView!
```

```
func searchBarSearchButtonClicked(searchBar: UISearchBar){
```

3
```
    searchBar.resignFirstResponder()
    dismissViewControllerAnimated(true, completion: nil)
    if self.mapView.annotations.count != 0{
        annotation = self.mapView.annotations[0]
        self.mapView.removeAnnotation(annotation)
    }
```

4
```
    localSearchRequest = MKLocalSearchRequest()
    localSearchRequest.naturalLanguageQuery = searchBar.text
    searchText = searchBar.text!
    localSearch = MKLocalSearch(request: localSearchRequest)
```

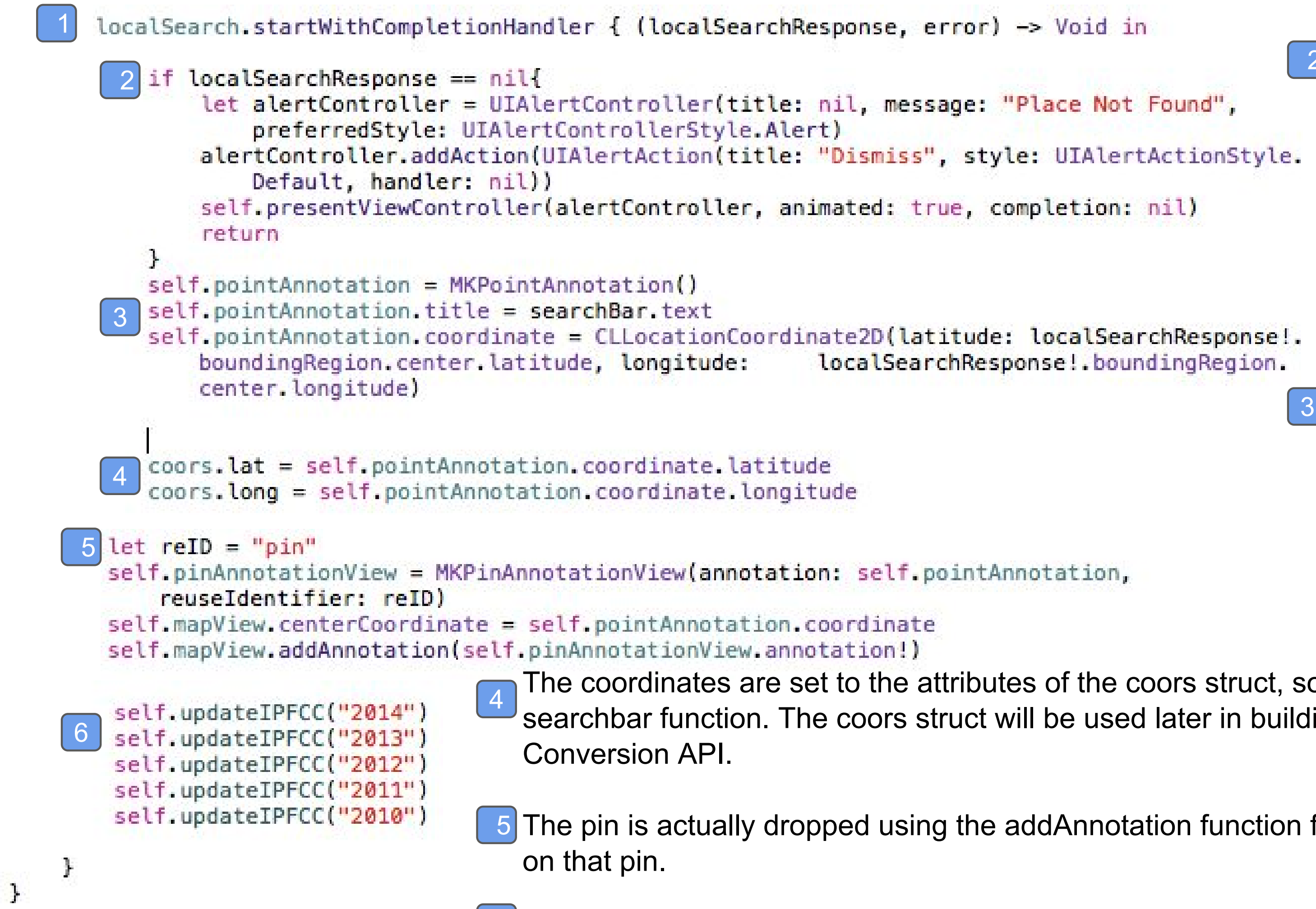
1 IBAction, as opposed to IBOutlet, defines an the function which operates upon clicking an element of the UIView. This function is tied to the top navigation bar in the Map Places Scene, and makes the search bar appear upon clicking it.

2 This is the map object. It's worth noting here that variables defined with exclamation points in Swift cannot be set to null. Variables defined with **let** instead of **var** are immutable.

3 This is the first of many blocks of code executed when the search bar's search button is clicked. It deals with the consequences of previous searches. First it removes the reference to them, then it makes the search bar disappear. Finally, if there are pins (annotations), it removes them.

4 The search bar text is turned into a natural language query. This means the MapKit search will not expect it to be a perfectly written address, but rather one that is incomplete. This also means that if the user types in a point of interest search, the map will be able to Geolocate that as well. When most users see a map in an application, they do not expect it to filter information for them. So on MarketStudy too, the user can enter "coffeeshop" and find the nearest coffeeshop. Without this functionality, users might think that the map is not working.

Map Places Scene- Error Handling and Drop Pin



```
1 localSearch.startWithCompletionHandler { (localSearchResponse, error) -> Void in

    2 if localSearchResponse == nil{
          let alertController = UIAlertController(title: nil, message: "Place Not Found",
              preferredStyle: UIAlertControllerStyle.Alert)
          alertController.addAction(UIAlertAction(title: "Dismiss", style: UIAlertActionStyle.
              Default, handler: nil))
          self.presentViewController(alertController, animated: true, completion: nil)
          return
      }
      self.pointAnnotation = MKPointAnnotation()
    3 self.pointAnnotation.title = searchBar.text
      self.pointAnnotation.coordinate = CLLocationCoordinate2D(latitude: localSearchResponse!.
          boundingRegion.center.latitude, longitude:     localSearchResponse!.boundingRegion.
          center.longitude)

      |
    4 coors.lat = self.pointAnnotation.coordinate.latitude
      coors.long = self.pointAnnotation.coordinate.longitude

  5 let reID = "pin"
    self.pinAnnotationView = MKPinAnnotationView(annotation: self.pointAnnotation,
        reuseIdentifier: reID)
    self.mapView.centerCoordinate = self.pointAnnotation.coordinate
    self.mapView.addAnnotation(self.pinAnnotationView.annotation!)

  6 self.updateIPFCC("2014")
    self.updateIPFCC("2013")
    self.updateIPFCC("2012")
    self.updateIPFCC("2011")
    self.updateIPFCC("2010")

  }
}
```

1 The start with completion handler line calls the geolocation.

2 The error function first checks if the completion handler for the geolocation function is empty. If it is, an alertController instance is created, and an animated popup is created telling the user the place was not found. An example of this can be seen in the gallery at the end of the paper.

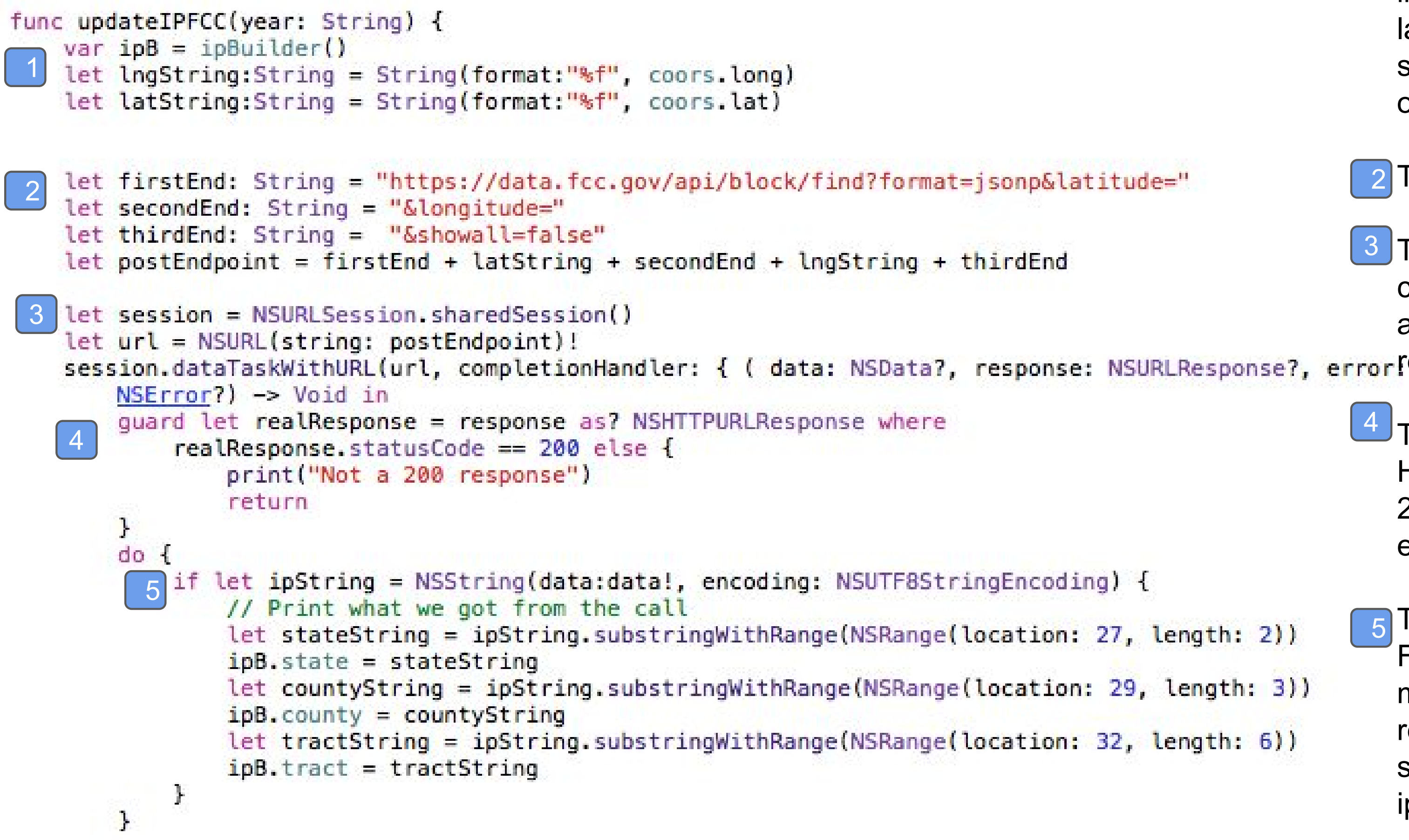
3 If the geolocation has been performed successfully, and the coordinates have been passed from the address search to the pin variable via the The localSearchResponse class. The pin's label is set to the text of the address.

4 The coordinates are set to the attributes of the coors struct, so they can be used outside of the searchbar function. The coors struct will be used later in building the call for the FCC Census Block Conversion API.

5 The pin is actually dropped using the addAnnotation function from MapKit, and the map is centered on that pin.

6 The updateIPFCC function preforms our API Calls, one for every year of the census we call

Map Places Scene- FCC API Call



```swift
func updateIPFCC(year: String) {
    var ipB = ipBuilder()
    let lngString:String = String(format:"%f", coors.long)
    let latString:String = String(format:"%f", coors.lat)


    let firstEnd: String = "https://data.fcc.gov/api/block/find?format=jsonp&latitude="
    let secondEnd: String = "&longitude="
    let thirdEnd: String =  "&showall=false"
    let postEndpoint = firstEnd + latString + secondEnd + lngString + thirdEnd

    let session = NSURLSession.sharedSession()
    let url = NSURL(string: postEndpoint)!
    session.dataTaskWithURL(url, completionHandler: { ( data: NSData?, response: NSURLResponse?, error:
        NSError?) -> Void in
        guard let realResponse = response as? NSHTTPURLResponse where
            realResponse.statusCode == 200 else {
                print("Not a 200 response")
                return
        }
        do {
            if let ipString = NSString(data:data!, encoding: NSUTF8StringEncoding) {
                // Print what we got from the call
                let stateString = ipString.substringWithRange(NSRange(location: 27, length: 2))
                ipB.state = stateString
                let countyString = ipString.substringWithRange(NSRange(location: 29, length: 3))
                ipB.county = countyString
                let tractString = ipString.substringWithRange(NSRange(location: 32, length: 6))
                ipB.tract = tractString
            }
        }
```

1 The global structure ipBuilder is initiated, and the longitude and latitude doubles from the coors structure are converted into strings in order to build the API Call.

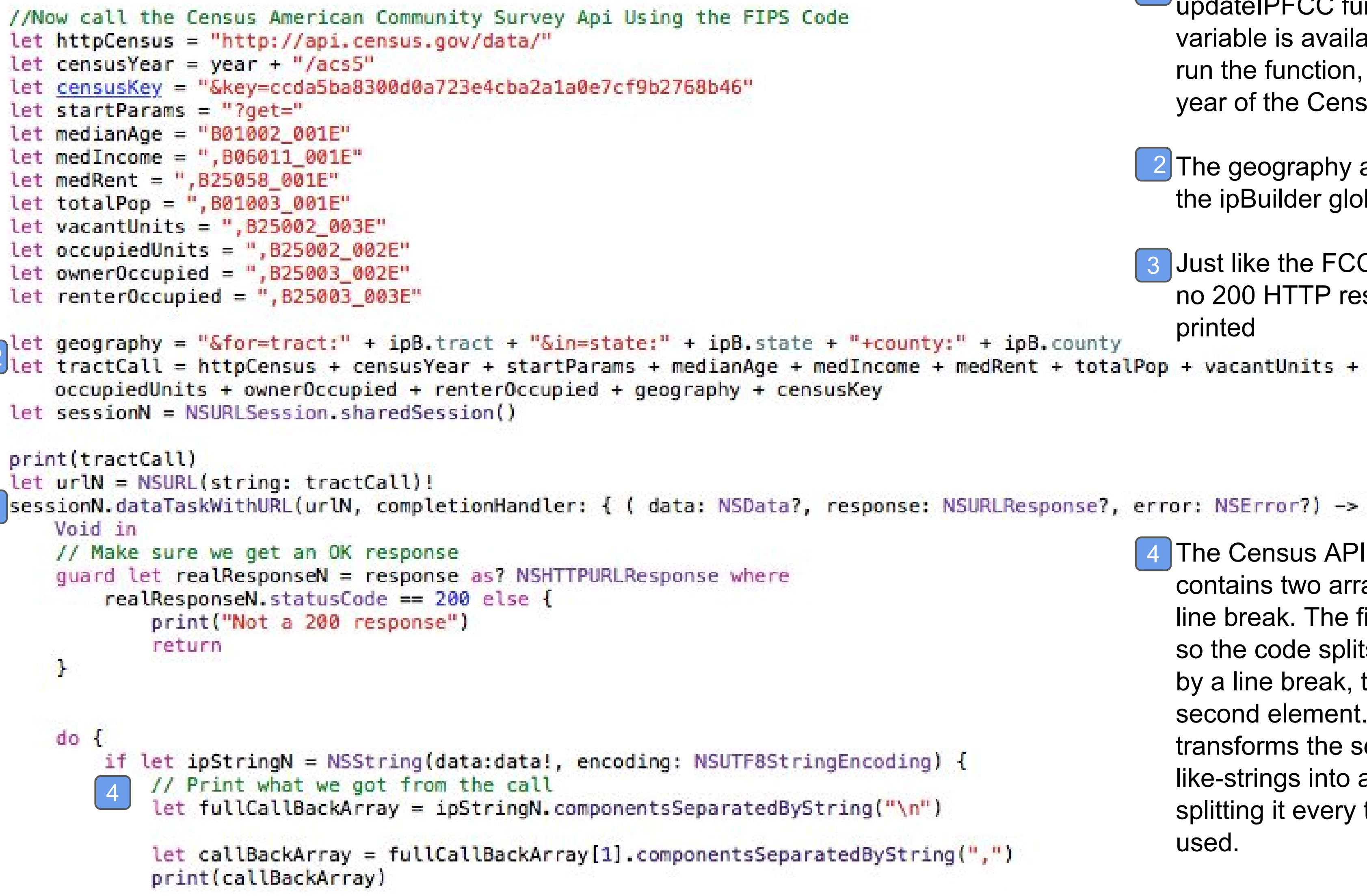
2 The API Call is built.

3 The NSURLSession class is initiated, called. By putting a question mark after NSData, Swift is prepared to receive a null value.

4 The guard statement checks if the HTTP request is successful, we get a 200 response. If not, we print the error statement in the console.

5 The code captures the data from the FCC API as a String if the call was made successfully. Finally, the response FIPS code string is parsed so that the correct components of the ipBuilder are put in the right place.

Map Places Scene- Census API Call



```
  //Now call the Census American Community Survey Api Using the FIPS Code
  let httpCensus = "http://api.census.gov/data/"
  let censusYear = year + "/acs5"
1 let censusKey = "&key=ccda5ba8300d0a723e4cba2a1a0e7cf9b2768b46"
  let startParams = "?get="
  let medianAge = "B01002_001E"
  let medIncome = ",B06011_001E"
  let medRent = ",B25058_001E"
  let totalPop = ",B01003_001E"
  let vacantUnits = ",B25002_003E"
  let occupiedUnits = ",B25002_002E"
  let ownerOccupied = ",B25003_002E"
  let renterOccupied = ",B25003_003E"

  let geography = "&for=tract:" + ipB.tract + "&in=state:" + ipB.state + "+county:" + ipB.county
2 let tractCall = httpCensus + censusYear + startParams + medianAge + medIncome + medRent + totalPop + vacantUnits +
      occupiedUnits + ownerOccupied + renterOccupied + geography + censusKey
  let sessionN = NSURLSession.sharedSession()

  print(tractCall)
  let urlN = NSURL(string: tractCall)!
3 sessionN.dataTaskWithURL(urlN, completionHandler: { ( data: NSData?, response: NSURLResponse?, error: NSError?) ->
      Void in
      // Make sure we get an OK response
      guard let realResponseN = response as? NSHTTPURLResponse where
          realResponseN.statusCode == 200 else {
              print("Not a 200 response")
              return
      }


      do {
          if let ipStringN = NSString(data:data!, encoding: NSUTF8StringEncoding) {
4             // Print what we got from the call
              let fullCallBackArray = ipStringN.componentsSeparatedByString("\n")

              let callBackArray = fullCallBackArray[1].componentsSeparatedByString(",")
              print(callBackArray)
```

1 We are still operating in the updateIPFCC function, so the year variable is available. Every time we run the function, we call a different year of the Census API.

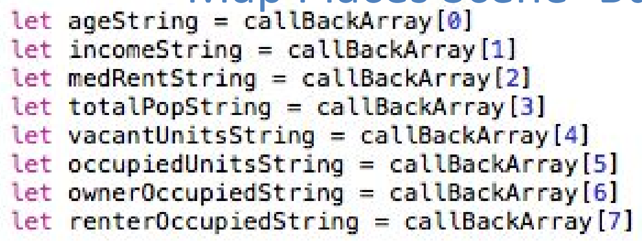
2 The geography attribute is built using the ipBuilder global structure.

3 Just like the FCC API Call, if there is no 200 HTTP response, an error is printed

4 The Census API returns a string that contains two arrays seperated by a line break. The first array is useless, so the code splits the call back string by a line break, then grabs the second element. Next, the code transforms the second of the array like-strings into a real array by splitting it every time a comma is used.

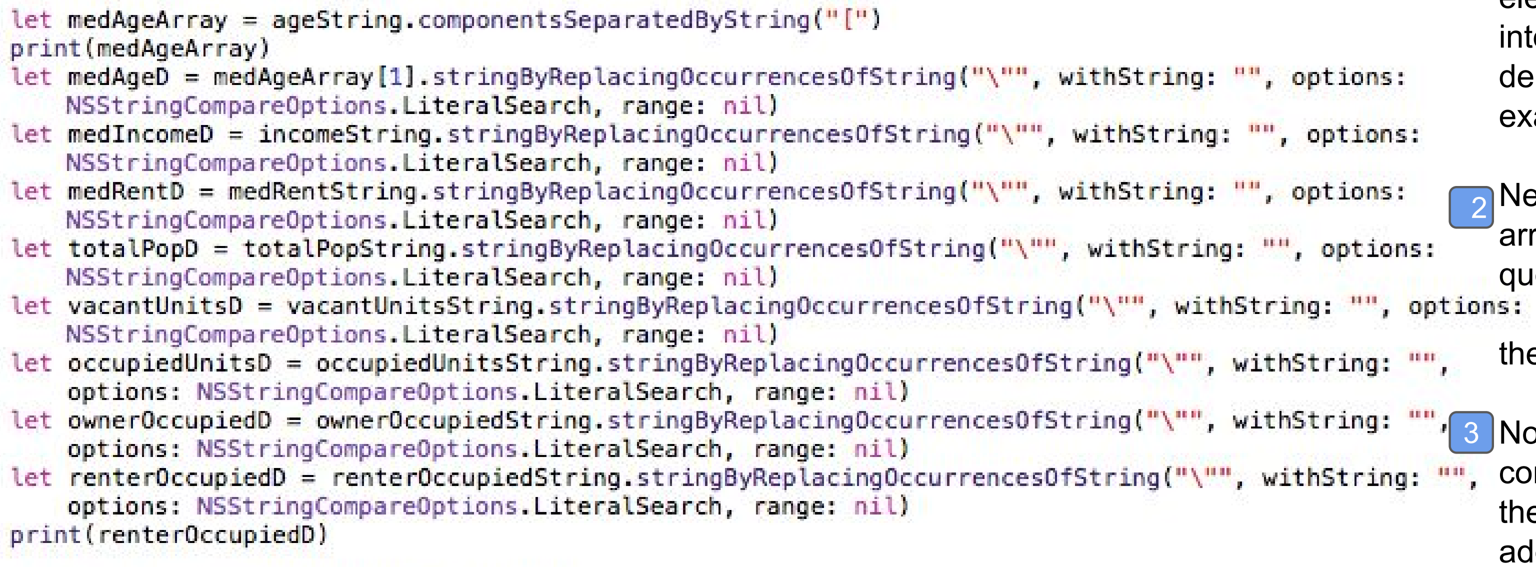
Map Places Scene- Building the Global Array

1



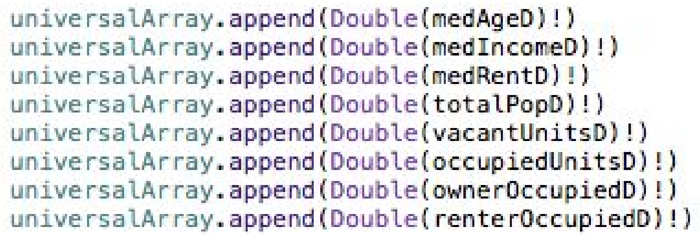
```
let ageString = callBackArray[0]
let incomeString = callBackArray[1]
let medRentString = callBackArray[2]
let totalPopString = callBackArray[3]
let vacantUnitsString = callBackArray[4]
let occupiedUnitsString = callBackArray[5]
let ownerOccupiedString = callBackArray[6]
let renterOccupiedString = callBackArray[7]
```

2



```
let medAgeArray = ageString.componentsSeparatedByString("[")
print(medAgeArray)
let medAgeD = medAgeArray[1].stringByReplacingOccurrencesOfString("\"", withString: "", options:
    NSStringCompareOptions.LiteralSearch, range: nil)
let medIncomeD = incomeString.stringByReplacingOccurrencesOfString("\"", withString: "", options:
    NSStringCompareOptions.LiteralSearch, range: nil)
let medRentD = medRentString.stringByReplacingOccurrencesOfString("\"", withString: "", options:
    NSStringCompareOptions.LiteralSearch, range: nil)
let totalPopD = totalPopString.stringByReplacingOccurrencesOfString("\"", withString: "", options:
    NSStringCompareOptions.LiteralSearch, range: nil)
let vacantUnitsD = vacantUnitsString.stringByReplacingOccurrencesOfString("\"", withString: "", options:
    NSStringCompareOptions.LiteralSearch, range: nil)
let occupiedUnitsD = occupiedUnitsString.stringByReplacingOccurrencesOfString("\"", withString: "",
    options: NSStringCompareOptions.LiteralSearch, range: nil)
let ownerOccupiedD = ownerOccupiedString.stringByReplacingOccurrencesOfString("\"", withString: "",
    options: NSStringCompareOptions.LiteralSearch, range: nil)
let renterOccupiedD = renterOccupiedString.stringByReplacingOccurrencesOfString("\"", withString: "",
    options: NSStringCompareOptions.LiteralSearch, range: nil)
print(renterOccupiedD)
```

3



```
universalArray.append(Double(medAgeD)!)
universalArray.append(Double(medIncomeD)!)
universalArray.append(Double(medRentD)!)
universalArray.append(Double(totalPopD)!)
universalArray.append(Double(vacantUnitsD)!)
universalArray.append(Double(occupiedUnitsD)!)
universalArray.append(Double(ownerOccupiedD)!)
universalArray.append(Double(renterOccupiedD)!)
```

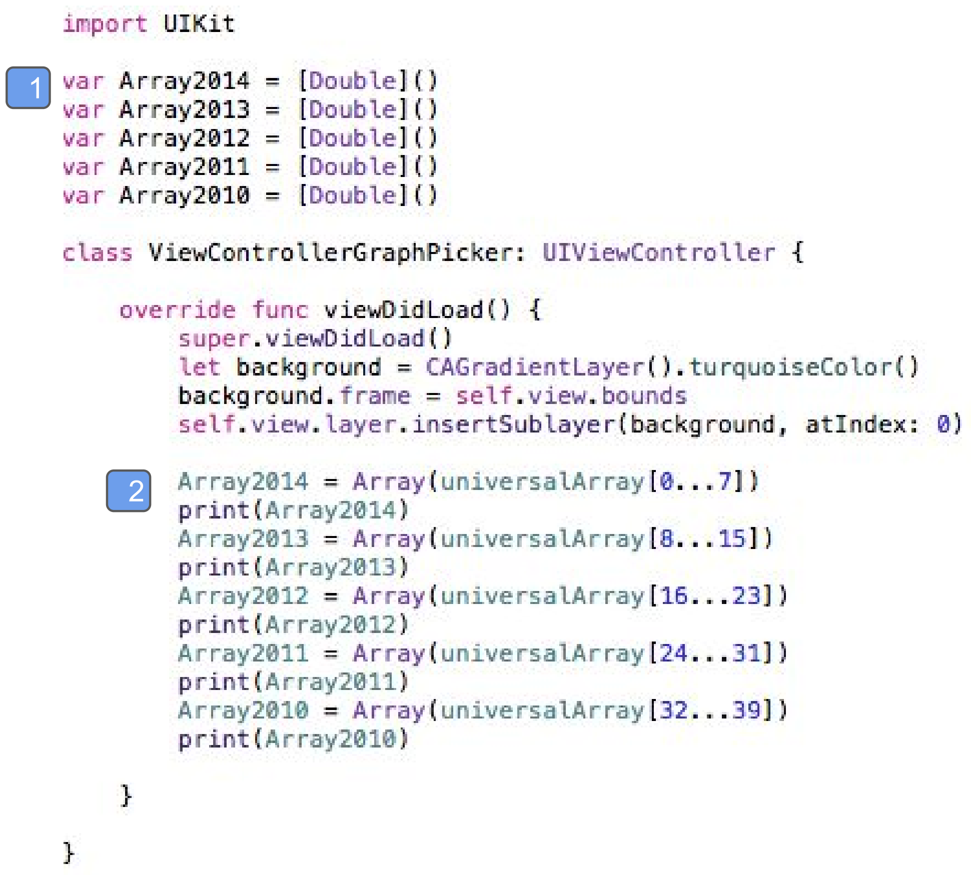
1 An example of an output from the last block of code is as follows: **["[\"28.7\"", "\"9933\"", "\"536\"", "\"3131\"", "\"194\"", "\"927\"", "\"540\"", "\"387\"", "\"42\"", "\"101\"", "\"038300\"]]"]**

Clearly it's very ugly! First the elements of the array were sorted into variables named after the demographic they represent. For example, incomeString = **"\"9933\""**

2 Next, the elements of the call back array are escaped from their double quotes. The first and last elements of the callback array also have the brackets removed from them.

3 Now that there are variables which contian the demographics as strings, they are converted to Doubles and added to the universalArray, so they can be accessed in other scenes. The universal array ends up being 40 elements long, 8 for each year, as it is not reset between API Calls. The universalArray is only reset when a new address is searched.

Graph Picker Scene



```
import UIKit
```

1

```
var Array2014 = [Double]()
var Array2013 = [Double]()
var Array2012 = [Double]()
var Array2011 = [Double]()
var Array2010 = [Double]()

class ViewControllerGraphPicker: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let background = CAGradientLayer().turquoiseColor()
        background.frame = self.view.bounds
        self.view.layer.insertSublayer(background, atIndex: 0)
```

2

```
        Array2014 = Array(universalArray[0...7])
        print(Array2014)
        Array2013 = Array(universalArray[8...15])
        print(Array2013)
        Array2012 = Array(universalArray[16...23])
        print(Array2012)
        Array2011 = Array(universalArray[24...31])
        print(Array2011)
        Array2010 = Array(universalArray[32...39])
        print(Array2010)

    }

}
```

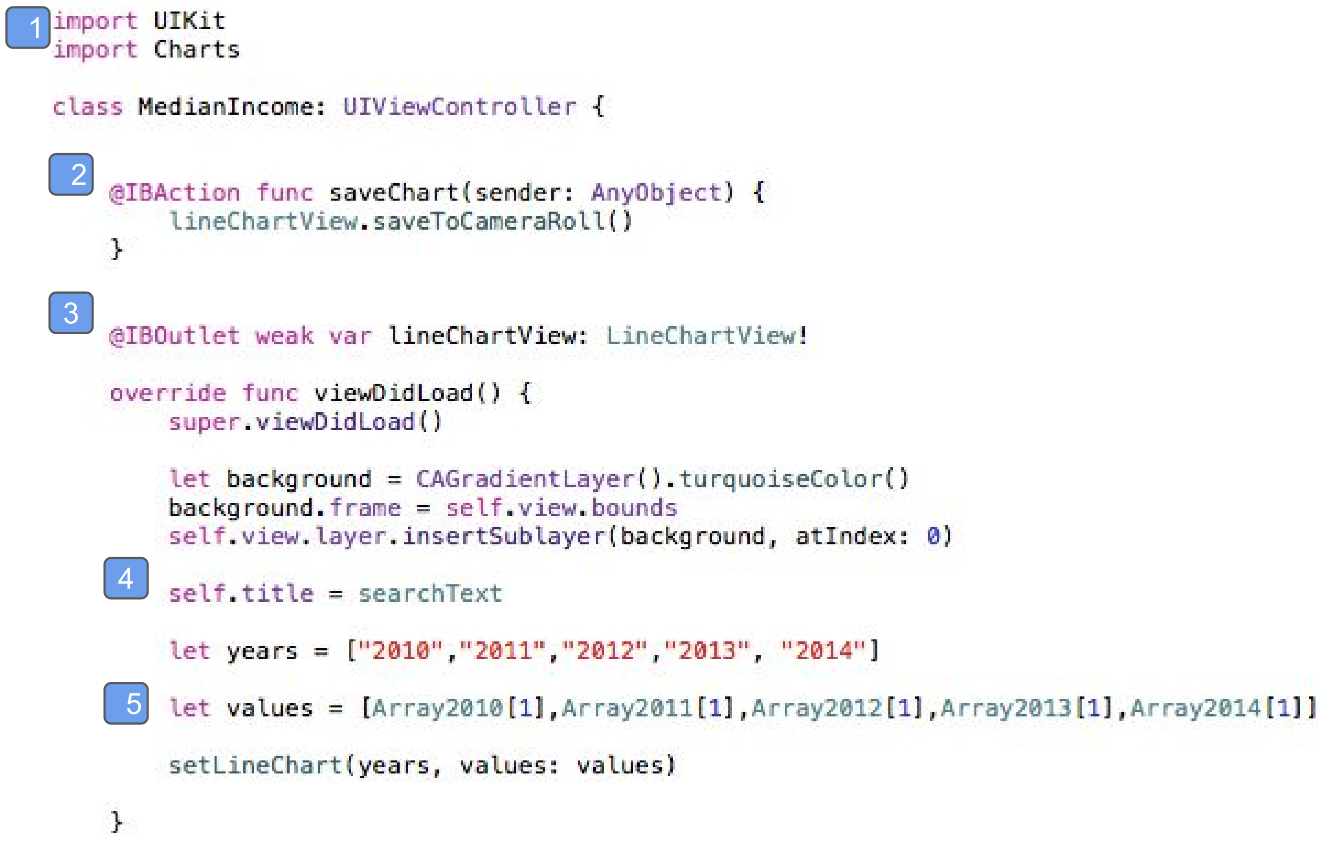
1 New global variables are set up so that the universalArray can be broken into 5 smaller arrays based on year.

2 After the view loads and the gradient background is set, the universalArray is broken up. Oddly enough, an array slice is a different type of object than an array, so the Array() function must be called to reconvert the universalArray slice into an array again.

As one can see, the universalArray is split sequentially in the order the updateIPFCC function was called in the Map Places Scene.

The year arrays contain demographic information at the following indicies:
0.Median Age
1.Median Income
2.Median Rent
3.Total Population
4.Vacant Units
5.Occupied Units
6.Owner Occupied Units
7.Renter Occupied Units

Median Income Graph Scene - Local Variables



```
1 import UIKit
  import Charts

  class MedianIncome: UIViewController {

    2
      @IBAction func saveChart(sender: AnyObject) {
          lineChartView.saveToCameraRoll()
      }

    3
      @IBOutlet weak var lineChartView: LineChartView!

      override func viewDidLoad() {
          super.viewDidLoad()

          let background = CAGradientLayer().turquoiseColor()
          background.frame = self.view.bounds
          self.view.layer.insertSublayer(background, atIndex: 0)

        4 self.title = searchText

          let years = ["2010","2011","2012","2013", "2014"]

        5 let values = [Array2010[1],Array2011[1],Array2012[1],Array2013[1],Array2014[1]]

          setLineChart(years, values: values)

      }
```

To avoid being overly repetitive, only one graph scene will be discussed.

1 The Charts library contains classes for line charts, bar charts, donut charts, etc. It was downloaded from GitHub.

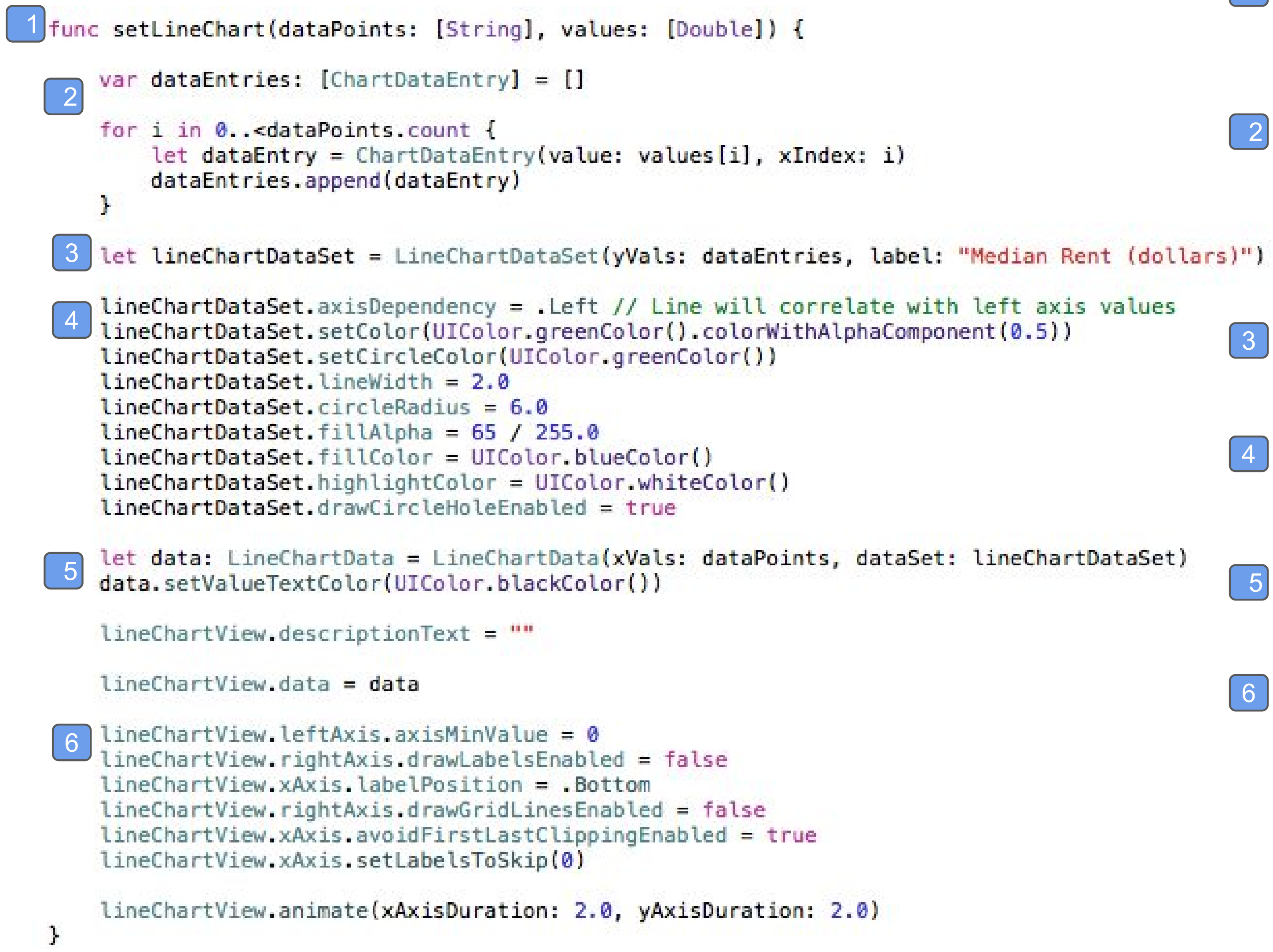
2 In the upper right corner of every graph scene is a button for saving the graph to the camera roll. Swift accomplishes this in just one line of code.

3 The line chart class in the UIView is referenced.

4 The title displayed on the top navigation bar is set to the address searched in the Map Places Scene. This is so that the user knows which address the graph corresponds to, if they save the graph to their camera roll.

5 The values array is a collection of the second elements in the demographics array of each year. As noted on the previous slide, this corresponds to Median Income. setLineChart creates the chart, and will be discussed on the next page.

Median Income Graph Scene - Set Graph Function



1
```
func setLineChart(dataPoints: [String], values: [Double]) {
```

2
```
    var dataEntries: [ChartDataEntry] = []

    for i in 0..<dataPoints.count {
        let dataEntry = ChartDataEntry(value: values[i], xIndex: i)
        dataEntries.append(dataEntry)
    }
```

3
```
    let lineChartDataSet = LineChartDataSet(yVals: dataEntries, label: "Median Rent (dollars)")
```

4
```
    lineChartDataSet.axisDependency = .Left // Line will correlate with left axis values
    lineChartDataSet.setColor(UIColor.greenColor().colorWithAlphaComponent(0.5))
    lineChartDataSet.setCircleColor(UIColor.greenColor())
    lineChartDataSet.lineWidth = 2.0
    lineChartDataSet.circleRadius = 6.0
    lineChartDataSet.fillAlpha = 65 / 255.0
    lineChartDataSet.fillColor = UIColor.blueColor()
    lineChartDataSet.highlightColor = UIColor.whiteColor()
    lineChartDataSet.drawCircleHoleEnabled = true
```

5
```
    let data: LineChartData = LineChartData(xVals: dataPoints, dataSet: lineChartDataSet)
    data.setValueTextColor(UIColor.blackColor())

    lineChartView.descriptionText = ""

    lineChartView.data = data
```

6
```
    lineChartView.leftAxis.axisMinValue = 0
    lineChartView.rightAxis.drawLabelsEnabled = false
    lineChartView.xAxis.labelPosition = .Bottom
    lineChartView.rightAxis.drawGridLinesEnabled = false
    lineChartView.xAxis.avoidFirstLastClippingEnabled = true
    lineChartView.xAxis.setLabelsToSkip(0)

    lineChartView.animate(xAxisDuration: 2.0, yAxisDuration: 2.0)
}
```

1 setLineChart takes two values, our labels corresponding to each year, and the median income values.

2 First, the function creates an empty array full of a type unique to the Charts library, ChartDataEntry. Then, the function loops through every median income value, and adds them to the dataEntries array.

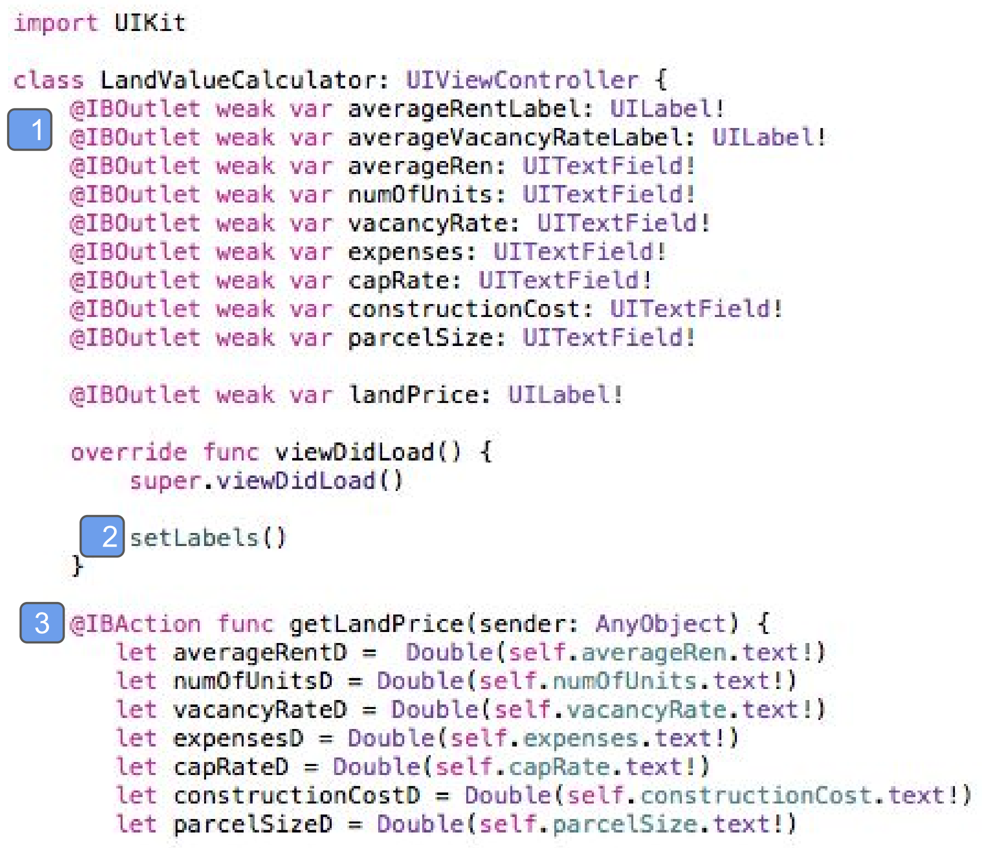
3 Next, the y values of the line chart are set to the dataEntries Array.

4 These methods set the appearence of the line chart graph: which axis is labeled, what color to set the line, etc.

5 Set the x-values to years, and the color of the labels.

6 Finally, set the minimum value on the y-axis scale to 0, put the x-axis labels on the bottom, and animate the creation of the line chart upon loading the scene.

Land Value Scene - Set Variables



```
import UIKit

class LandValueCalculator: UIViewController {
    @IBOutlet weak var averageRentLabel: UILabel!
1   @IBOutlet weak var averageVacancyRateLabel: UILabel!
    @IBOutlet weak var averageRen: UITextField!
    @IBOutlet weak var numOfUnits: UITextField!
    @IBOutlet weak var vacancyRate: UITextField!
    @IBOutlet weak var expenses: UITextField!
    @IBOutlet weak var capRate: UITextField!
    @IBOutlet weak var constructionCost: UITextField!
    @IBOutlet weak var parcelSize: UITextField!

    @IBOutlet weak var landPrice: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

      2 setLabels()
    }

3   @IBAction func getLandPrice(sender: AnyObject) {
        let averageRentD =  Double(self.averageRen.text!)
        let numOfUnitsD = Double(self.numOfUnits.text!)
        let vacancyRateD = Double(self.vacancyRate.text!)
        let expensesD = Double(self.expenses.text!)
        let capRateD = Double(self.capRate.text!)
        let constructionCostD = Double(self.constructionCost.text!)
        let parcelSizeD = Double(self.parcelSize.text!)
```

1 The Calculate Land value scene is intended to aid the user in land price negotiation. If the ultimate output of the land value calculator is less than the asking price for the land, then the deal is not profitable.

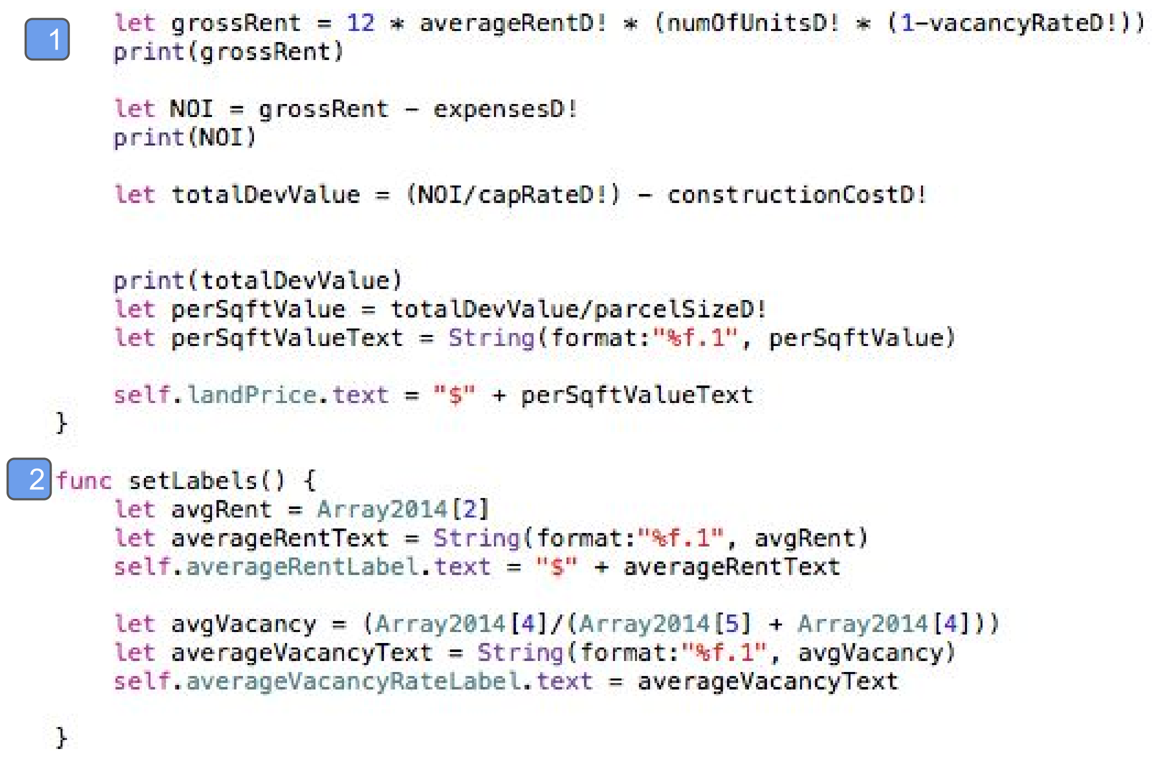
The inputs into the land value calculator are doubles corresponding to:

1. Rent
2. Vacancy Rate
3. Number of Units
4. Yearly Expenses (things like lighting in common areas)
5. Cap Rate
6. Construction
7. Parcel Size in Square feet

2 MarketStudy does not automatically give the user the inputs into the land calculator, but where there are overlaps between the American Community Survey demographic information, suggested inputs are portrayed. This is what the setLabels function does.

3 Upon clicking the calculate button, all of the text input strings are converted to doubles.

Land Value Scene - Calculate Land Value and Set Labels



1

```
        let grossRent = 12 * averageRentD! * (numOfUnitsD! * (1-vacancyRateD!))
        print(grossRent)

        let NOI = grossRent - expensesD!
        print(NOI)

        let totalDevValue = (NOI/capRateD!) - constructionCostD!


        print(totalDevValue)
        let perSqftValue = totalDevValue/parcelSizeD!
        let perSqftValueText = String(format:"%f.1", perSqftValue)

        self.landPrice.text = "$" + perSqftValueText
    }
```

2

```
    func setLabels() {
        let avgRent = Array2014[2]
        let averageRentText = String(format:"%f.1", avgRent)
        self.averageRentLabel.text = "$" + averageRentText

        let avgVacancy = (Array2014[4]/(Array2014[5] + Array2014[4]))
        let averageVacancyText = String(format:"%f.1", avgVacancy)
        self.averageVacancyRateLabel.text = averageVacancyText

    }
```

1 The land value calculation is borrowed from John Landis's Introduction to Property Development class, and is based on the net rent derived from a property one year after construction is completed.

To calculate the gross rent, multiply the inputted per unit monthly rent multiplied by 12, then multiply again by the number of units taking into account what percentage of them will be vacant. Then, subtract expenses.

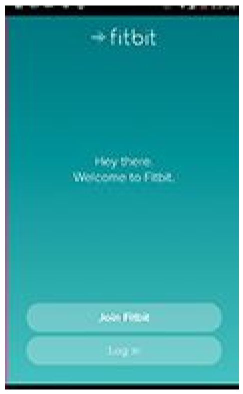
Next, the divide the NOI (net operating income) by the cap rate, and subtract construction costs from this value. Finally, divide this value by the size of the parcel to get the per square foot land value.

Again, if this land value is less than the owner's asking price, it's a bad deal!

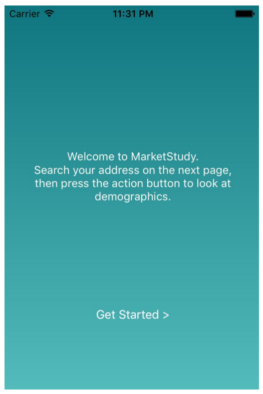
2 Two of the land value calculator inputs overlap with the Census API information, average rent and vacancy rate within the tract where the address falls. Both are displayed in the Land Value Scene as suggestions for inputs.

Data Presentation

The major stylistic inspirations for the design of MarketStudy are the FitBit app and censusreporter.org. FitBit's minimalist introductory scene alerts the user that the app is both professional, and likely to run well. MarketStudy borrows FitBit's blue and turquoise gradient color scheme, and initial welcoming dialogue. A comparison between the introduction scenes of both the apps is shown below:



Hey there.
Welcome to Fitbit.



Welcome to MarketStudy.
Search your address on the next page, then press the action button to look at demographics.

The graphs scenes, on the other hand, were inspired by censusreporter.org. Their muted color scheme informed the graph colors in MarketStudy. Unlike MarketStudy, however, censusreporter.org does not contain any line graphs, however, as it only reports on one year of the census at a time.

Testing

This being the first time I have ever programmed in Swift, or any Object Oriented language, most of my time was spent dealing with errors. Most of these had to do with wrapping variables, and setting their type. Recall that wrapped variables can be set to null (called nil in Swift) while unwrapped variables cannot.If an operation did not go through, and a variable ended up with a nil value, Swift would throw an error that a value was unwrapped, instead of telling you where its value failed to be assigned. As mentioned earlier, Swift is an incredibly type specific language. Doubles and integers cannot be used in the same operation, for example.

The API calls were another major source of error. If there was a problem in the built call, there was no way to print the http error response in the console. Adjusting Swift security permissions was also a major source of pain. It is for this reason the Zillow API was not called, as was originally planned. Finally, the API's built the universalArray asynchronously. The FCC and Census APIs are fast enough to get this done before the Graph Picker scene loads 90% of the time, but occasionally an error is thrown where the universalArray failed to build.

Finally, as far as the user experience is concerned, a few decisions were made based on user testing. I had some friends play with my app (on my computer) in the study room, and took their feedback to improve the MarketStudy UI. The main page was criticized for looking simple to the point of being obviously a novice's work. Thus, the text fade in was added. User feedback helped most with the graph scenes. Previously the graphs only showed point labels upon touch, but this was deemed unintuitive. A major source of frustration was my inability to work out the aesthetics of the Land Use Calculator scene, which everyone agreed was too busy yet also too unstyled.

Conclusion

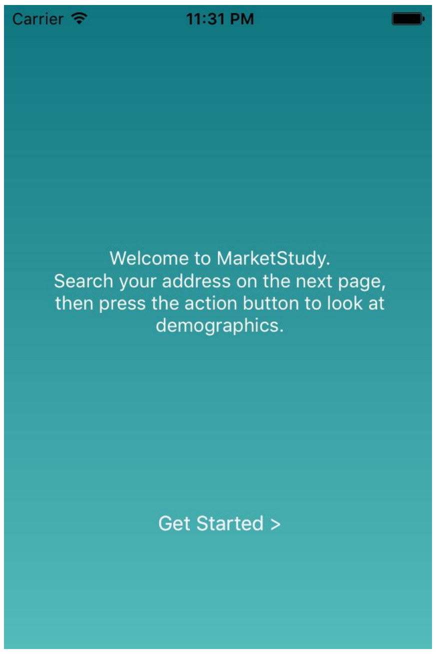
I still have a few more minor adjustments to make before MarketStudy is ready for the App Store. Most of these are visual. I never was entirely happy with the look of the Land Value Calculator Page. There is so much information on screen that a gradient background looks busy, but a white background looks amateurish. In the months to come, I will reformat it to fit user entry forms in a table. The Charts package for iOS is worth further exploration. Currently the charts look very neat, but uninteresting. The labeling of the line graph points is also slightly out of bounds on the left. Further error handling is also necessary. Currently, if the user searches for an address located in another country, the Graph Picker Scene simply does not load. Similarly, the user can enter negative and zero values in the land value calculator, and still calculate (nonsensical) results. There should be a customized error popup for these events, this is not particularly difficult code to write, but was cut due to time constraints.

As far as further directions are concerned, two possibilities exist. One is to go farther down the real estate rabbit hole. Try again to get the Zillow API to work, and look into pulling other demographics from the Census API. The Census API is rich in information, even continuing demographics about how residents commute to work. However, calling too much from the API could slow down the app a lot. Even the neighborhood Walkscores could be of interest. Before 2010 there are no Census API's with housing information, but updating the time graphs to go until 2015 would be prudent.

The other direction for further possibilities is to use the API calling and mobile mapping skills I've learned to shift to create an app that parses and maps JSON or GeoJSON files. This idea was abandoned because the Census API outputs an array of two arrays, rather than a javascript object. However, there are currently no apps that offer a way of mapping user uploaded JSON, which is a large niche to fill. This would involve using the DropBox API to share a JSON file into the iOS app. While the GeoJSON app project sounds difficult, the greatest joy I gained from building MarketStudy is that I the ability to take an idea for an app, and begin to think of the concrete steps I would need to create it.

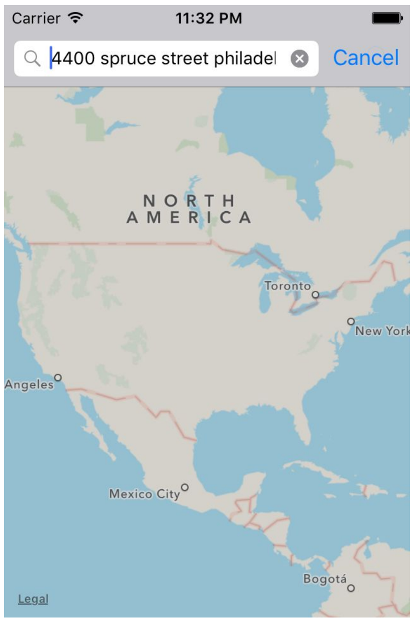
Gallery

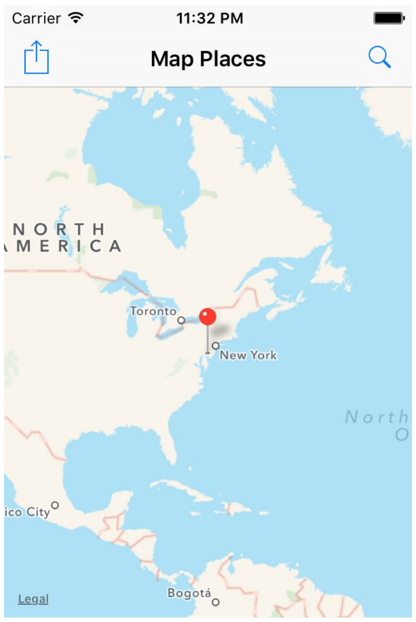
Intro Scene

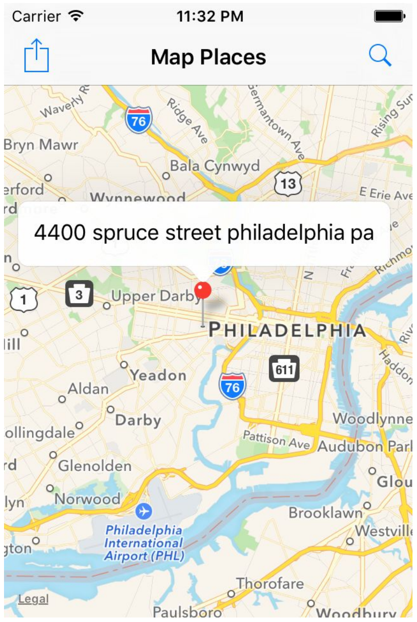


Welcome to MarketStudy.
Search your address on the next page, then press the action button to look at demographics.

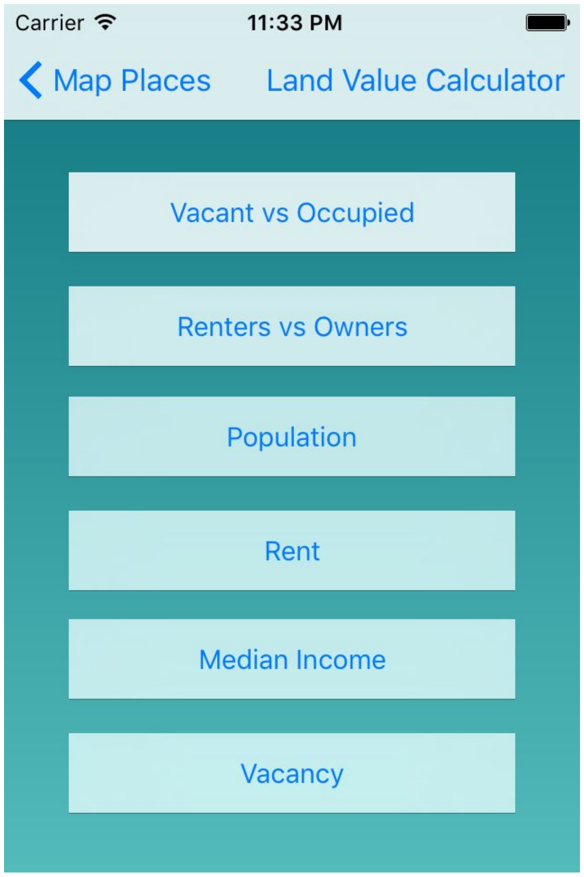
Map Places Scene



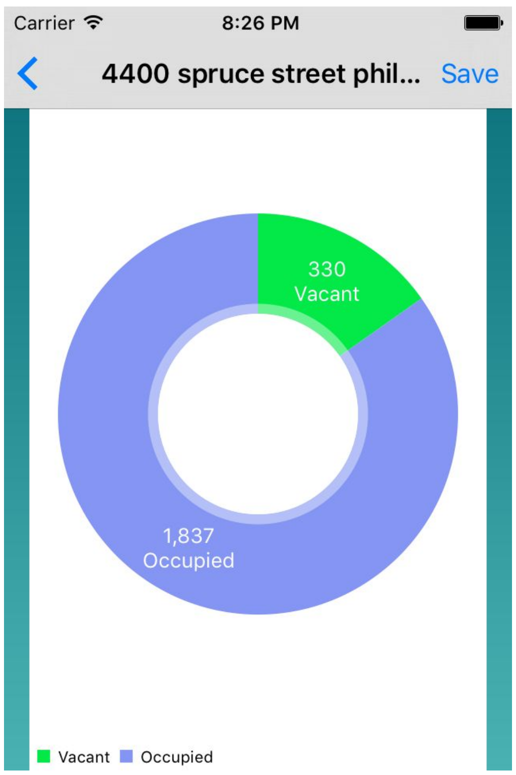


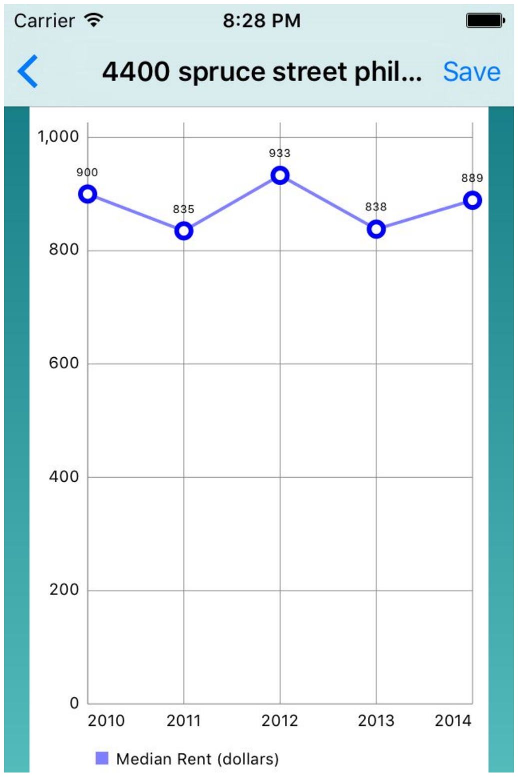


Graph Picker Scene

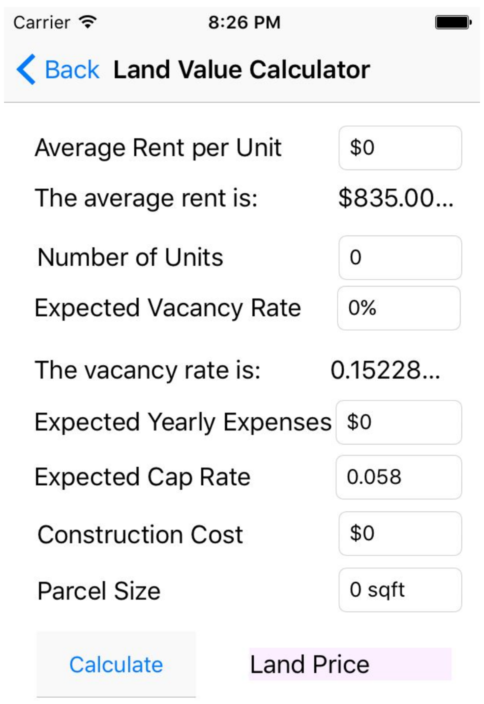


Graph Scene



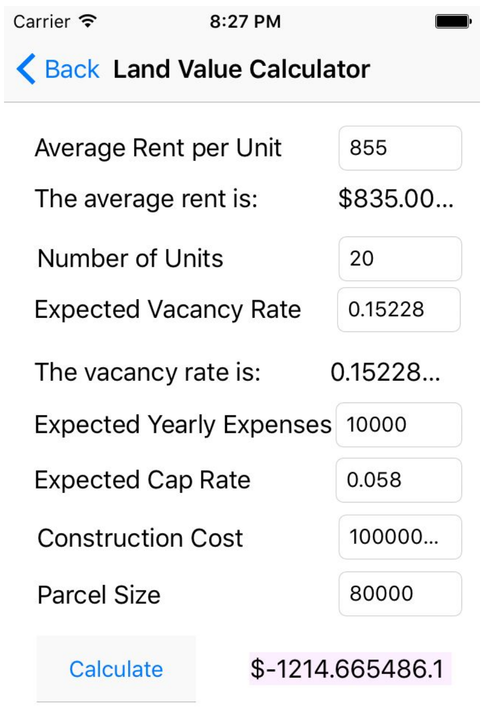


Land Value Scene



Land Value Calculator

Average Rent per Unit	$0
The average rent is:	$835.00...
Number of Units	0
Expected Vacancy Rate	0%
The vacancy rate is:	0.15228...
Expected Yearly Expenses	$0
Expected Cap Rate	0.058
Construction Cost	$0
Parcel Size	0 sqft
Calculate	Land Price



Land Value Calculator

Average Rent per Unit	855
The average rent is:	$835.00...
Number of Units	20
Expected Vacancy Rate	0.15228
The vacancy rate is:	0.15228...
Expected Yearly Expenses	10000
Expected Cap Rate	0.058
Construction Cost	100000...
Parcel Size	80000
Calculate	$-1214.665486.1

Resources

1. Introduction to Swift, iTunes U: https://itunes.apple.com/us/course/developing-ios-8-apps-swift/id961180099

2. Introduction to Swift, Apple Conference: https://www.youtube.com/watch?v=A0C6L4XmrZM

3. Charts Tutorial AppCoda: http://www.appcoda.com/ios-charts-api-tutorial/

4. Charts Library GitHub Repository: https://github.com/danielgindi/Charts

5. Charts Tutorial RayWenderlich: https://www.raywenderlich.com/90693/modern-core-graphics-with-swift-part-2

6. Making Rest Calls with Swift Tutorial: http://www.deegeu.com/ios-swift-rest-json-tutorial/

7. Rest Calls with Swift Example Code: https://github.com/deege/deegeu-swift-rest-example

8. Adding Gradient Backgrounds in Swift VideoTurorial: https://www.youtube.com/watch?v=pabNgxzEaRk

9. MapKit Tutorial by SweetTutos: http://sweettutos.com/2015/04/24/swift-mapkit-tutorial-series-how-to-search-a-place-address-or-poi-in-the-map/

10. Federal Communicaitions Comission Census Block Conversions API: https://www.fcc.gov/general/census-block-conversions-api

11. Census Bureau American Community Survey API: https://www.census.gov/data/developers/data-sets/acs-survey-5-year-data.html