RAYWENDERLICH
Tutorials for iPhone / iOS Developers and Gamers

Learn to Code iOS Apps 3: Your First App

Mike Jaoudi on January 30, 2013

Learn how to make an iOS app.

Welcome to Part 3 of the Learn to Code iOS Apps series! This series introduces you to the basics of programming with the goal of creating an iOS app; no prior programming experience is required.

In the first two tutorials in the series, you learned the basics of programming in Objective-C. Specifically:

- In Part 1, you learned the basics of Objective-C programming and you created a simple command line number guessing game.
- In Part 2, you learned about objects and classes in Objective-C and you created a simple app to track people's names and ages.
- In Part 3 (You are Here!), the real fun begins! Now that you know the basics of programming, you will take all that you've learned and create a simple iPhone game of your own.
- In Part 4, you will take this app and make it beautiful, learning more about customizing the look and feel of iPhone apps.

You learned a lot in those tutorials, but you might have thought to yourself – "Wait a minute, I thought this was supposed to be about making iOS apps, but all I've made so far is command line apps!"

Well, good news – the long wait is over! You are now ready to make your first iOS app at long last!

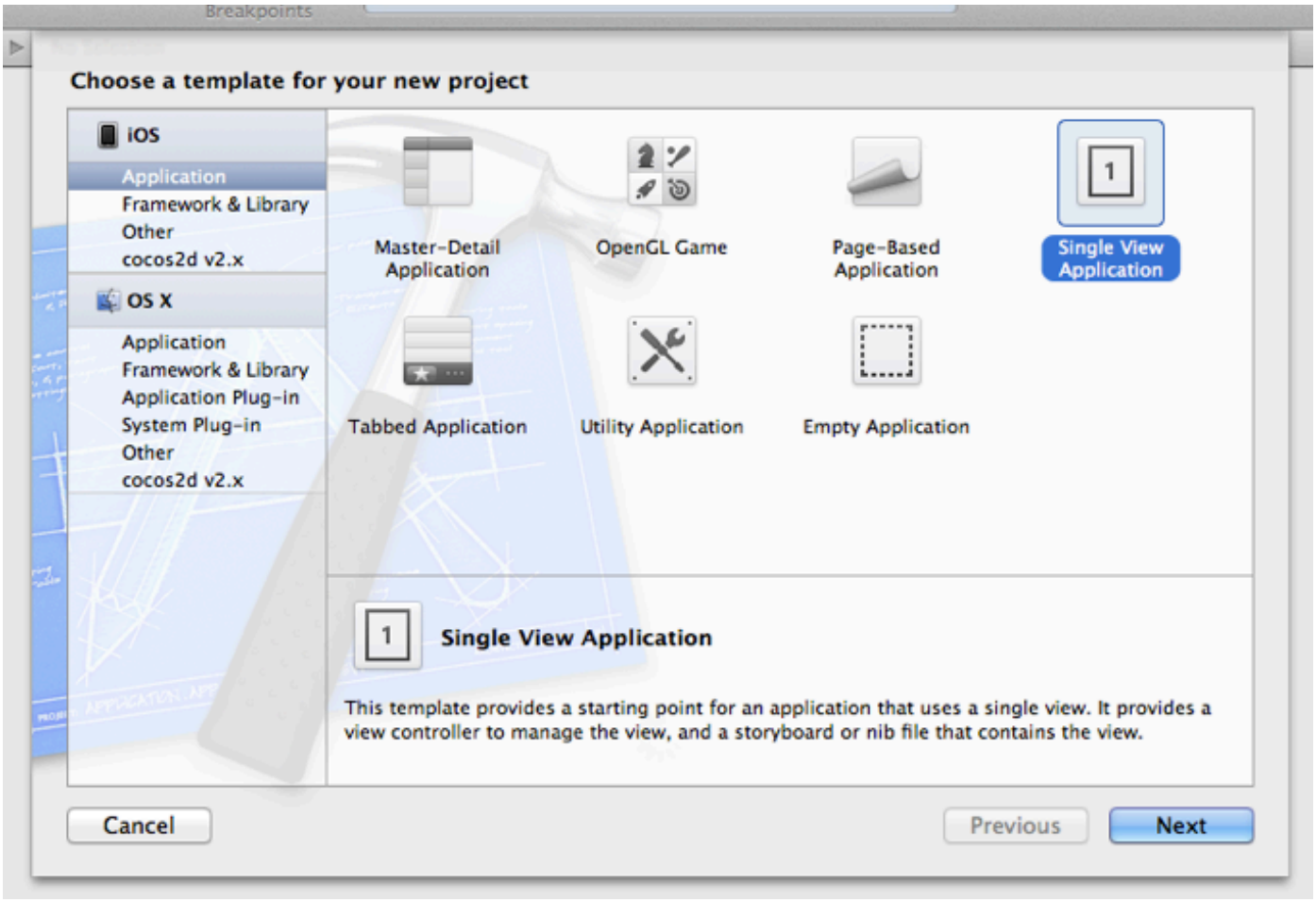
In this tutorial, you will create a simple iOS game where you have to tap a button as many times as you can in 30 seconds. Just don't get too excited and smash your screen by mistake! :]

In this first part, you'll create the basic app with all of the required functionality in place. In the second part of the tutorial, you will add custom images and sounds to your app to give it a more polished appearance!

Sound good? Then time to dive right in and get started!

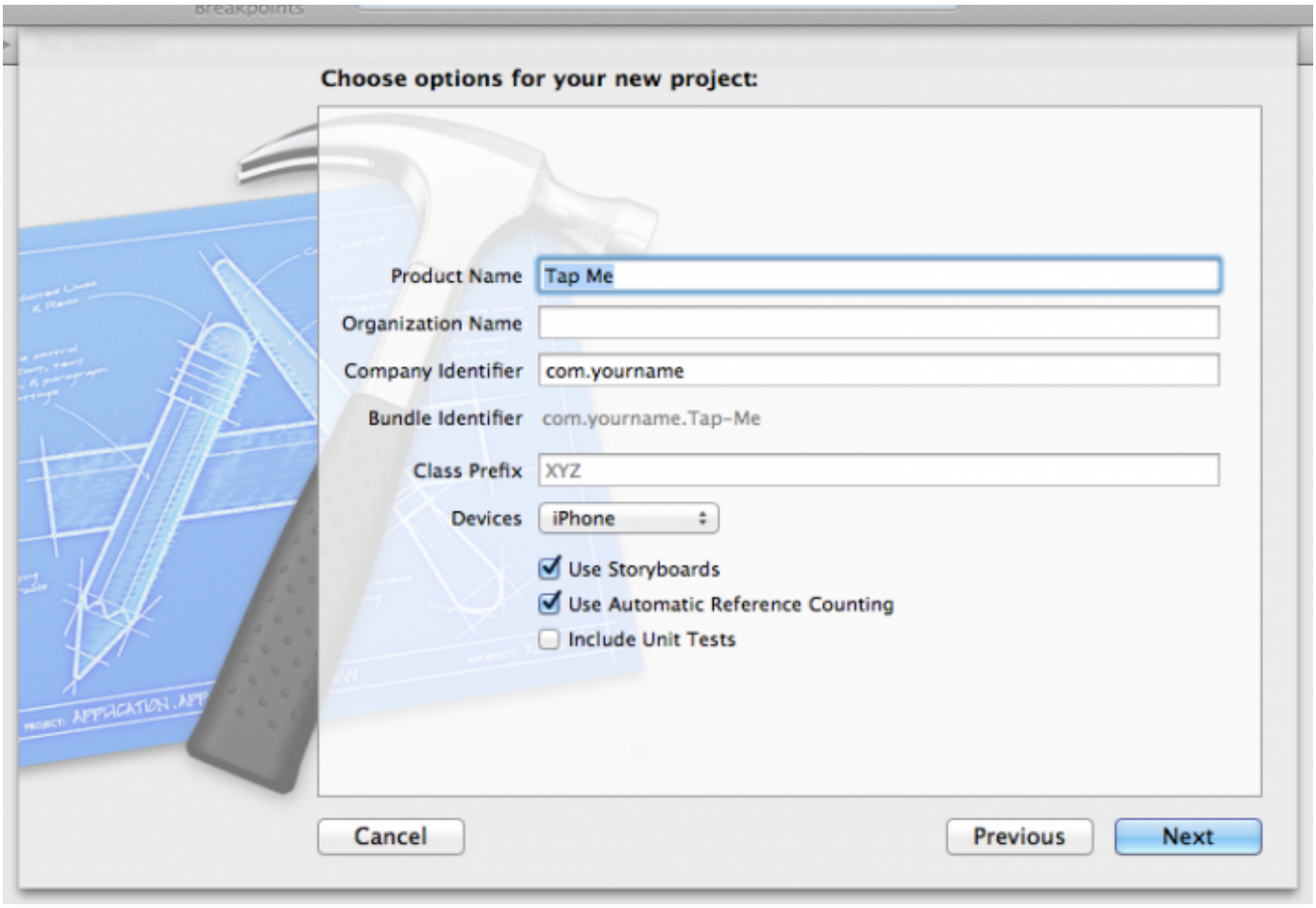
Getting Started

Start up **Xcode**, and select **File > New > Project**. In the left sidebar, select **Application** under iOS. Then double click **Single View Application**, as below:



Fill in the project options as follows:

- **Product Name**: **Tap Me**
- **Organization**: can be left blank.
- **Company Identifier**: use something like **com.yourname**, such as **com.janesmith**
- **Class Prefix**: can be left blank.
- **Devices**: select **iPhone**.
- Make sure that **Use Storyboard** and **Use Automatic Reference Counting** are the **only** two items checked.



Now choose the directory to where you want to save the project files, and click **Create**.

Off and Running!

Great! You now have everything setup in Xcode to start developing your iOS app; now is a great time to make sure that everything is configured correctly by running the app. Make sure that **iPhone Simulator** is selected in the upper left corner of the screen and then press **Run**, as below:

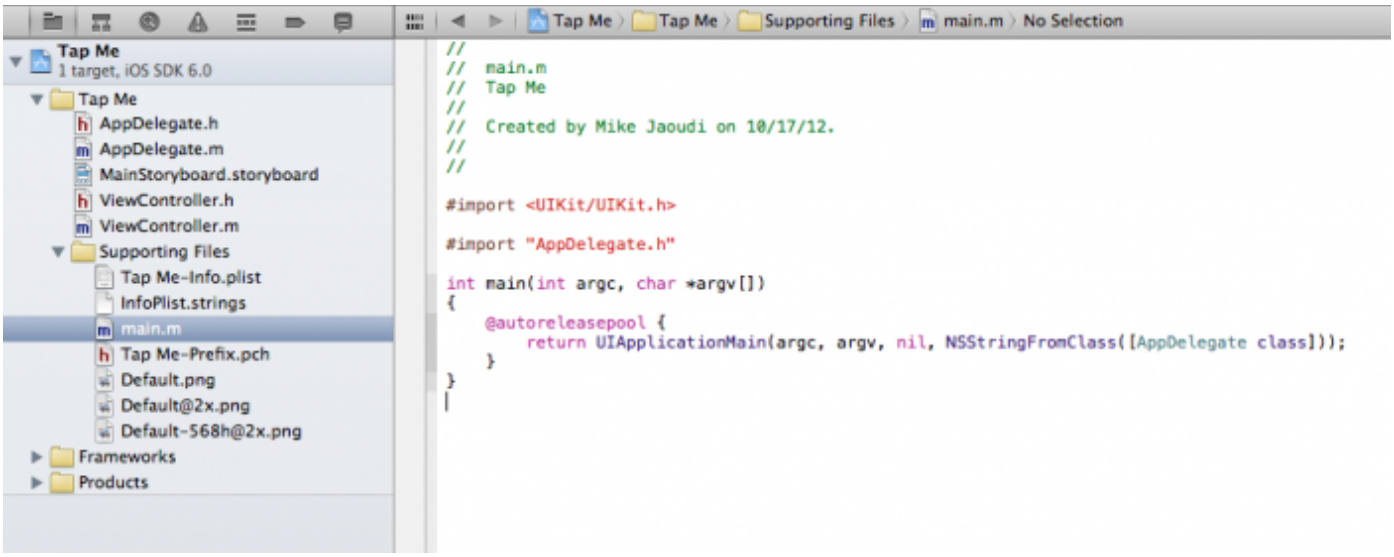
Xcode projects start off as fully-functional apps, but the starter project will do nothing but display a blank white screen in the simulator. Your stunning app should look like this:

Have you finished admiring your awesome app yet? :] Return to Xcode and press the **Stop** button to stop the app.

Take a look at the project navigator. This project looks quite a bit different than the project in the previous tutorials. There are a lot of new files inside this project that you didn't see with your previous Mac application.

For example, where is main.m? No worries; it's in the **Supporting Files** folder. But wait a minute — that's a little odd. Why is main.m in the Supporting Files folder, if all of your code needs to go inside that file?

Open up **main.m** and have a look at the code inside:



Hmm — in the section where you placed your code in the Mac application, there is a strange line of code:

```
return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
```

What on earth is this and what does it do? In simple terms, the UIApplication serves as the central brain of your app. It gives you a powerful app with graphical capabilities that allows your app to take advantage of all of the features of an iOS device.

While command-line applications will normally have more useful code in main.m, the typical iOS app just needs to boot up UIApplication and hand off control.

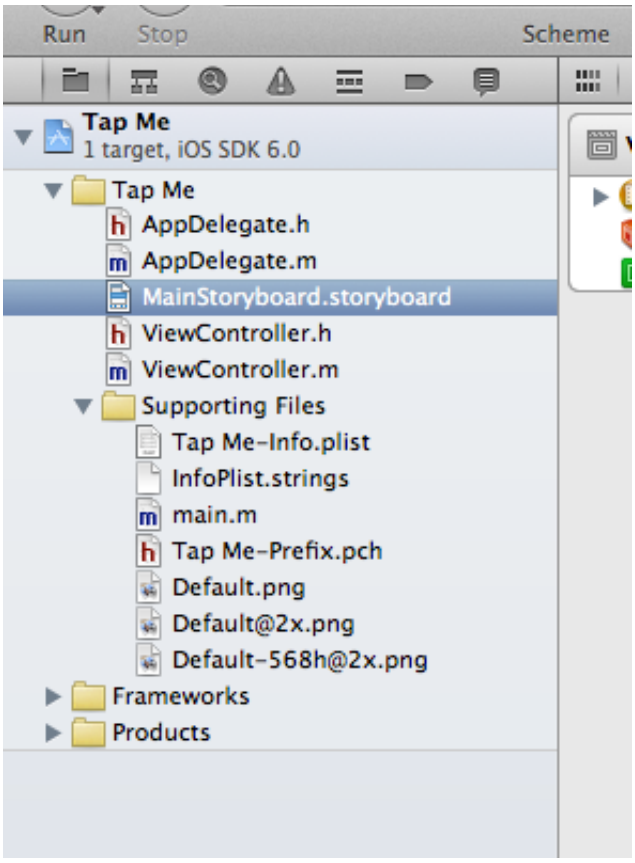
So the first code that runs in your application is still main.m, but now the sole purpose of this file is just to create a new UIApplication. In 99.99% of the iOS apps you'll encounter as a developer, you never need to edit main.m; therefore it is stored away in the Supporting Files folder.

Did you just breathe a sigh of relief? :] You'll leave main.m for now and head on into building the rest of your app!

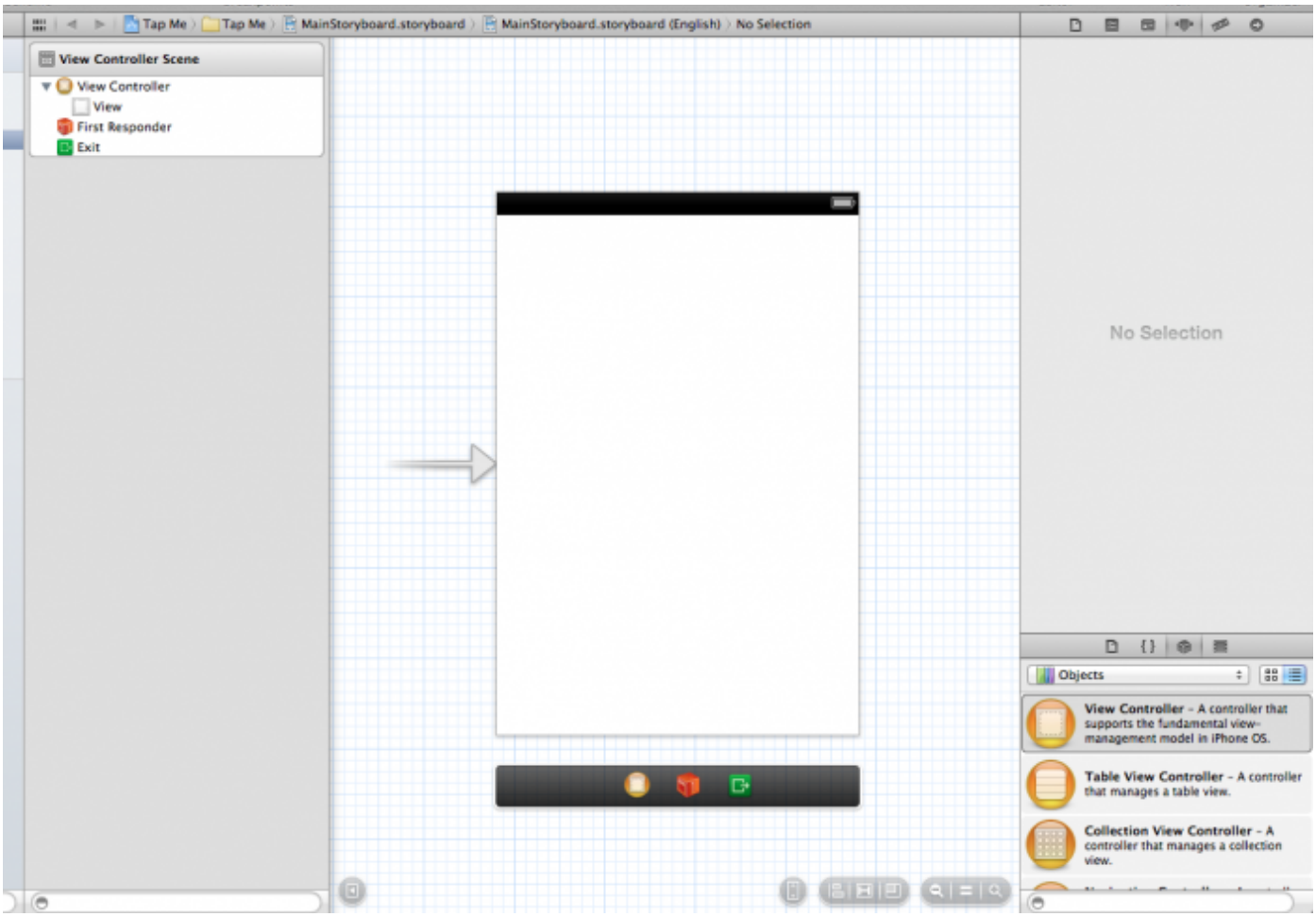
Two Sides to Every Story — Introducing Storyboards

Where do you start building the app if not in main.m? One great feature in iOS is called **Storyboards**. Storyboards are a method of designing the visual appearance and flow of your app, without having to write any code! This is sounding easier and easier, isn't it ? :]

The default project already has a Storyboard file in it called **MainStoryboard.storyboard**. Go ahead and open it up! You'll see the following:

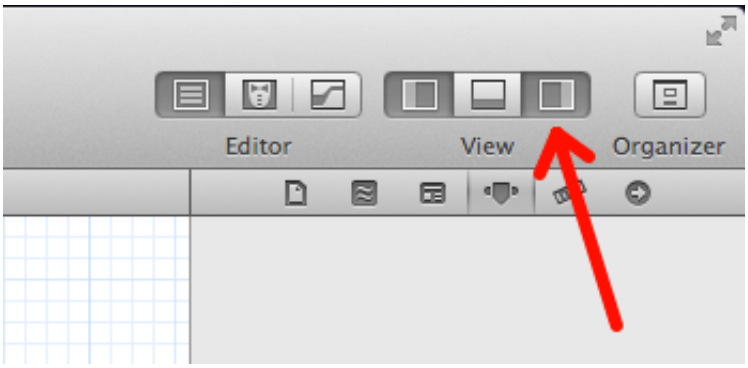


This is the basic storyboard file. So far, it is just displaying a blank white screen, just like the simulator showed when you ran the app earlier:

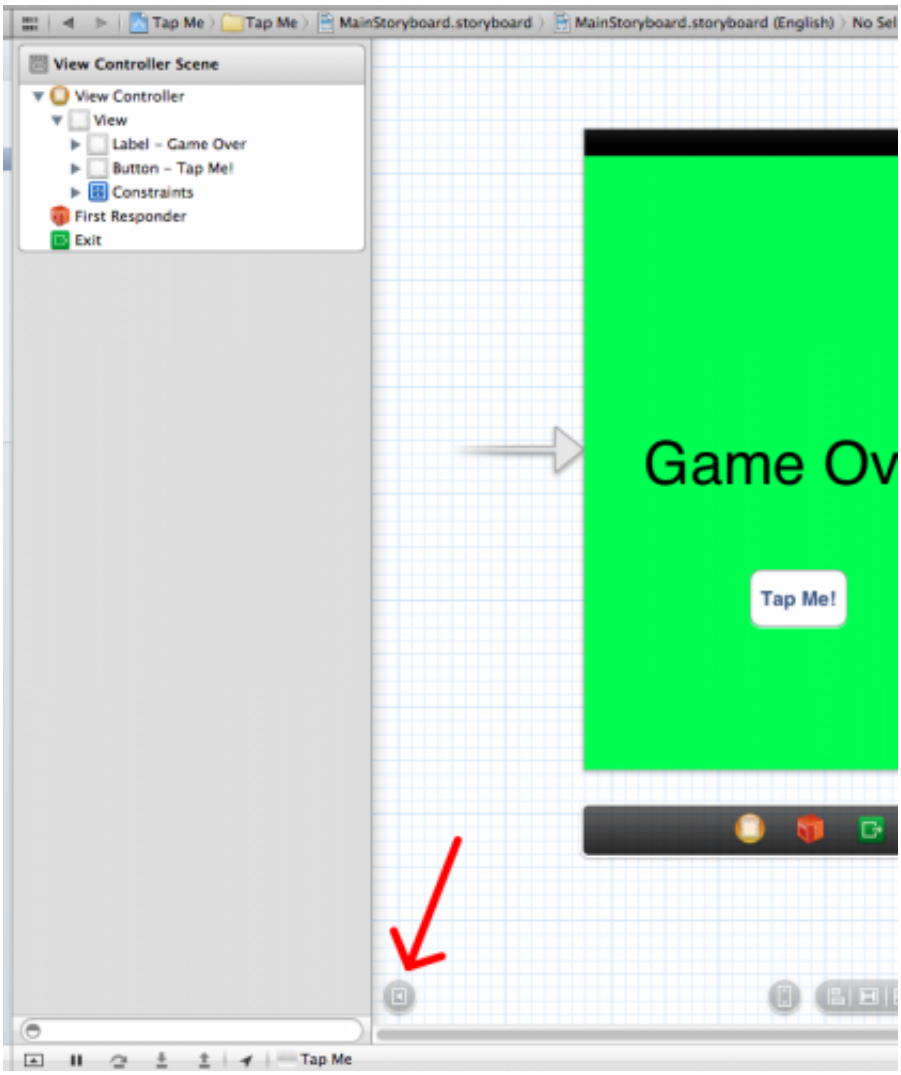


You'll notice there are sidebars on both sides of the storyboard view. Most of the time in this tutorial, you'll want to have both sidebars visible to make your life a little easier! :]

To toggle the **Utilities** sidebar on the right, click the button in the top-right of the Xcode toolbar in the **View** section, as below:



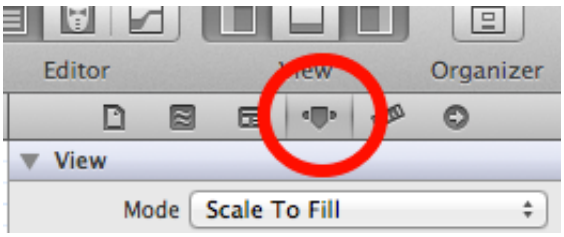
To toggle the **Document Outline** sidebar on the left, click the round arrow button in the lower-left corner of the storyboard. The left-pointing arrow will hide the sidebar; if it's hidden, it will turn into a right-pointing arrow to show the sidebar.



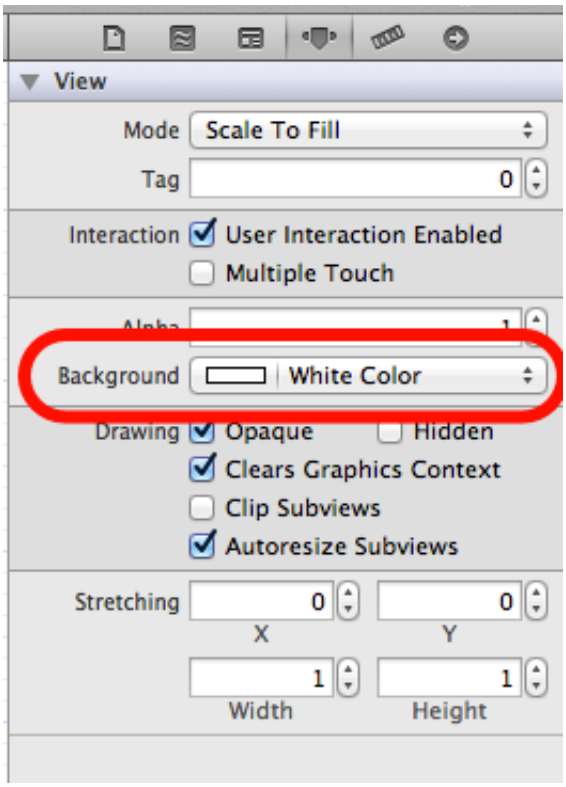
As exciting as the white background is, you'll probably want to change it to something a little snappier! :]

Coloring Inside the Lines

To change the background color, click anywhere inside the white view to select it. The Utilities sidebar on the right will show the details of whatever is currently selected. You're interested in the **Attributes** tab in the sidebar; it's the fourth one from the left and looks like a little slider:

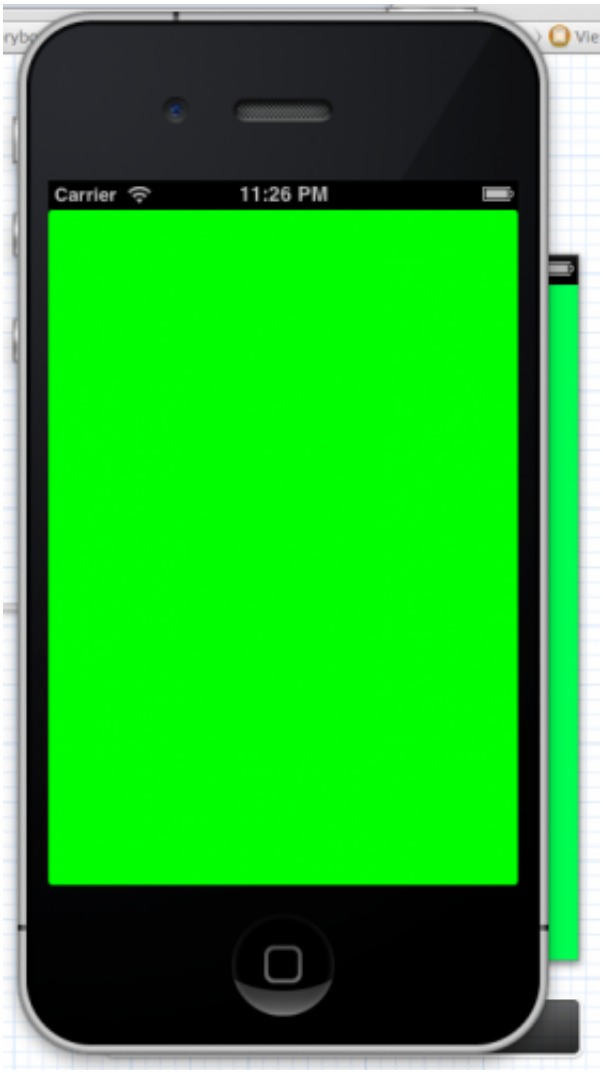


The sidebar now shows all of the attributes that can be changed for the view. Find the drop down menu labeled **Background**:



Click the small white rectangle to bring up the color palette, or click the words “White Color” and select “Other…” Choose a nice green color for your app.

Now click the **Run** button in the upper left corner of the Xcode toolbar to check out your app!



Amazing! Now your app is starting to look a little more interesting. However, you won't entertain your user for long by staring at a green screen. Time to add some user interface elements! :]

Tap Me!

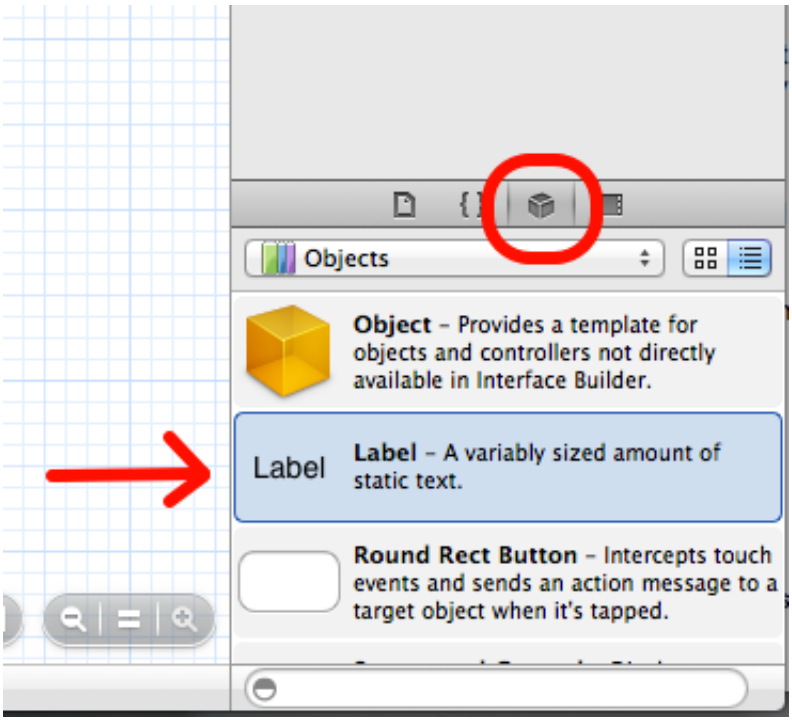
Think for a minute about the user interface elements you will need for your button-tapping game:

- Something to indicate how much time is left
- The player's current score
- A button to tap

For the first two elements, you'll use **Labels** which display text.

Return to the **MainStoryboard.storyboard** file in Xcode to add these two Labels to the view.

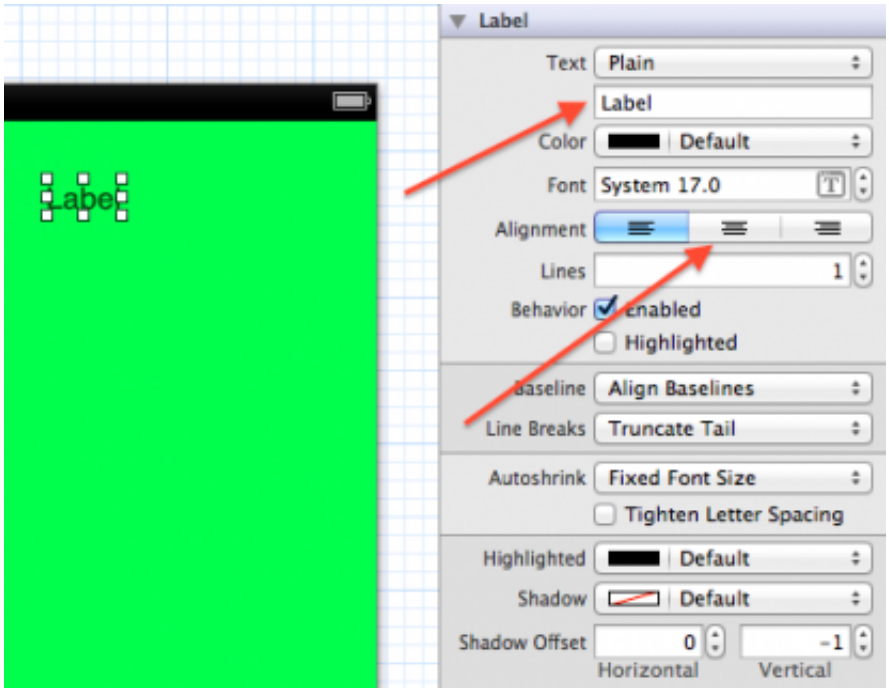
At the bottom of the Utilities sidebar, there is a Library of different objects. Scroll down until you find one called **Label**.



The first label you'll create will be your timer. It will start by saying "Time: 30" and then count down when the game starts.

Click and drag a label from the library and place it near the top of the view. The blue lines that show up on the view are guides to help you line up objects, but you don't have to worry about that yet.

Now that your label is in the view and is selected, you'll notice the Utilities sidebar now shows the attributes you can change for a label. Find the **Text** attribute and change it from "Label" to "**Time: 30**". Also, change the alignment to be center-aligned, as seen in the screenshot below:



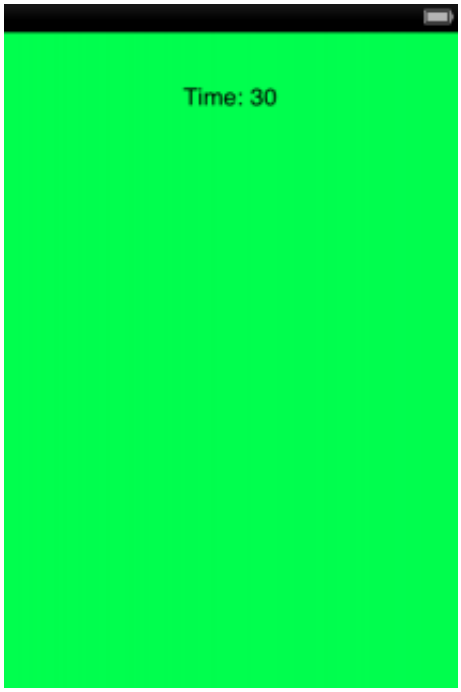
Whoops — it looks like the text doesn't quite fit inside the default size of the Label. You'll need to resize the label to make

it wider.

You don't have to make the Label exactly the width of your text; instead, give it a little extra width just to be sure that you'll always have enough room for your countdown.

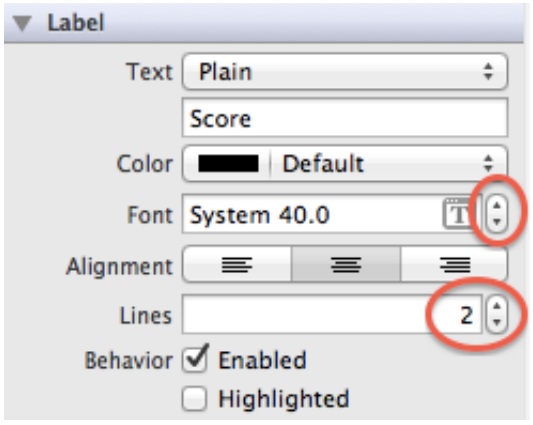
Once the label is wide enough, move it so it's centered in the view. When it's close to being in the center, Xcode will show a dotted blue line as a guide to help you position your objects.

At this point, your storyboard should look something like this:

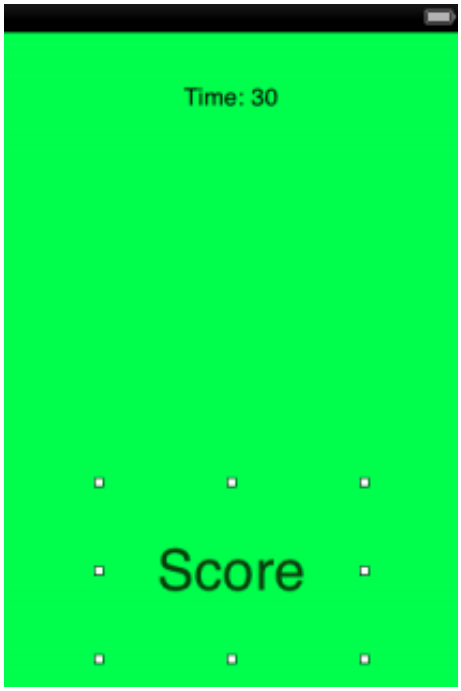


So far so good! Now, repeat the steps above to place one more label near the bottom of the view to show the current score; change the text to say "Score", and make it center-aligned.

You want the score to stand out on the screen, so make the font size of this Label bigger by clicking the small up-arrow next to the font size. Since you'll display the player's score on two lines, change the number of lines to 2 by clicking the small up-arrow for Lines.

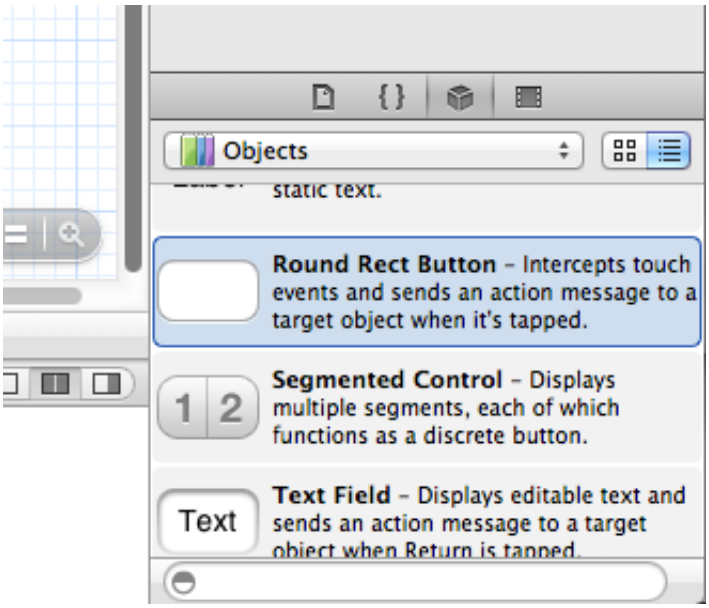


As before, the label is way too small to fit the two lines of large text you'll need! Resize the label so it's wide enough to fit the word "Score", and tall enough to fit two lines of text. At this point, your storyboard should look something like this:



Now what was that last user interface element that you needed? Oh yeah — a button for the user to click! :]

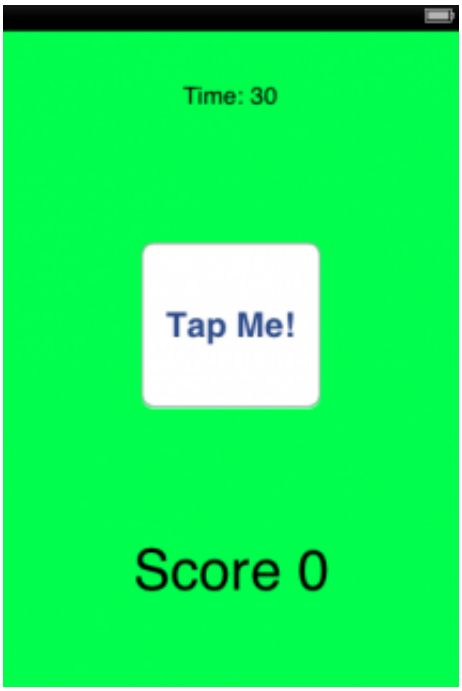
Scroll through the object library and find a **Round Rect Button**, as below:



Click and drag a Round Rect button from the library onto your view in between the two labels. Make sure the button is selected in your view, and have a look at its attributes in the sidebar.

Buttons should usually indicate the action the user should take, so change the title from boring **Button** to a more exciting **Tap Me!**. Since your Button is really the main event of your app, resize the button to make it bigger, and while you're at it, make the font size larger too!

Here's a final look at what your storyboard should look like:



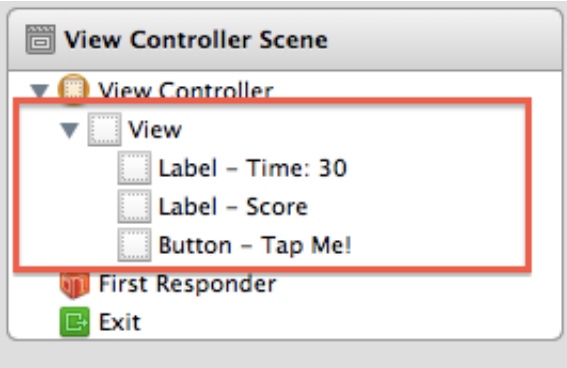
The basic layout of your simple button-tap game is now in place! It's starting to look like a real app, isn't it? :] Up until now, you haven't really written any code, which is pretty cool!

However, as you've correctly surmised, Xcode is doing a lot of work behind the scenes. Take a minute and have a look through the next section which explains the magic of the Views and View Controllers at use in your app!

Behind the Curtain — Views and View Controllers

Look at the Document Outline sidebar to the left of the storyboard. The top-level view (with the simple name "View") is the background layer and is where you set the background color earlier. Inside of it are the two labels and the "Tap Me" button that you added to the app.

In an iOS app, just about anything that you can see on the screen is a kind of View. That means your label and button are also types of views.



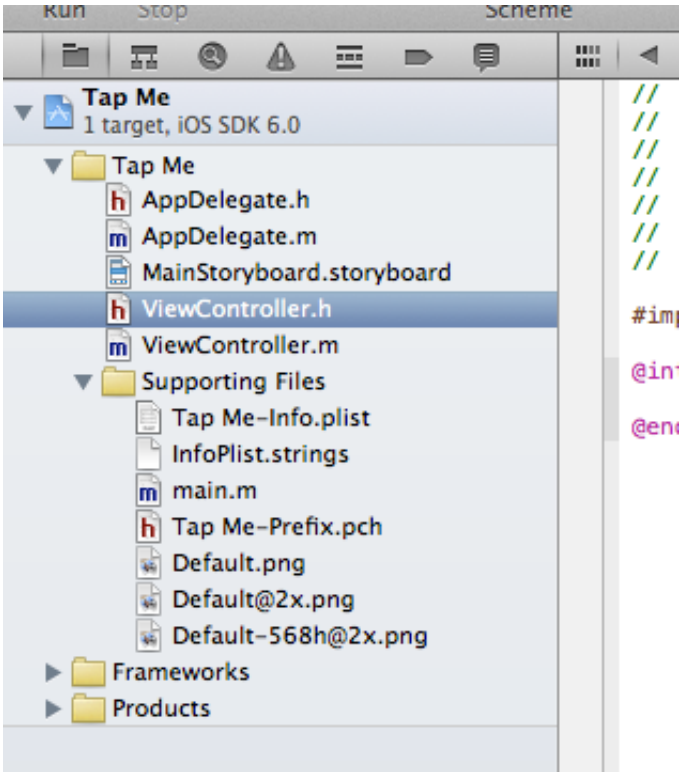
If you want to interact with views – such as change their attributes from code or check if the user tapped on them – you use something called a **Controller**. Look back at the left column, and you'll see that the the view, label and button are all inside something called **View Controller**.

The job of this View controller is to manage all of the views that are stored inside it. View controllers handle the behind-the-scenes jobs of your app, and are in charge of all of the actions that happen inside the views. For example, when the button is tapped, the view controller will take care of what to do.

Time to dive just a little deeper and check out what this View Controller does! :]

Controlling the View Controller

Xcode wrote some basic starter code for you, so start by opening the file **ViewController.h** to have a look at the code for the view controller:



Remember that a header file is used to **declare** different parts of the program. Declaring means to say that it exists, but doesn't give the actual implementation details.

You need some way to refer to the two labels you added earlier. This is called an **outlet** in Xcode, where some variable in the view controller refers to some user interface element in the storyboard view.

Near the top of the header file will be a line like this:

```
@interface ViewController : UIViewController
```

Change that line and add a few additional lines so it looks like this:

```
@interface ViewController : UIViewController {
  IBOutlet UILabel *scoreLabel;
  IBOutlet UILabel *timerLabel;
}
```

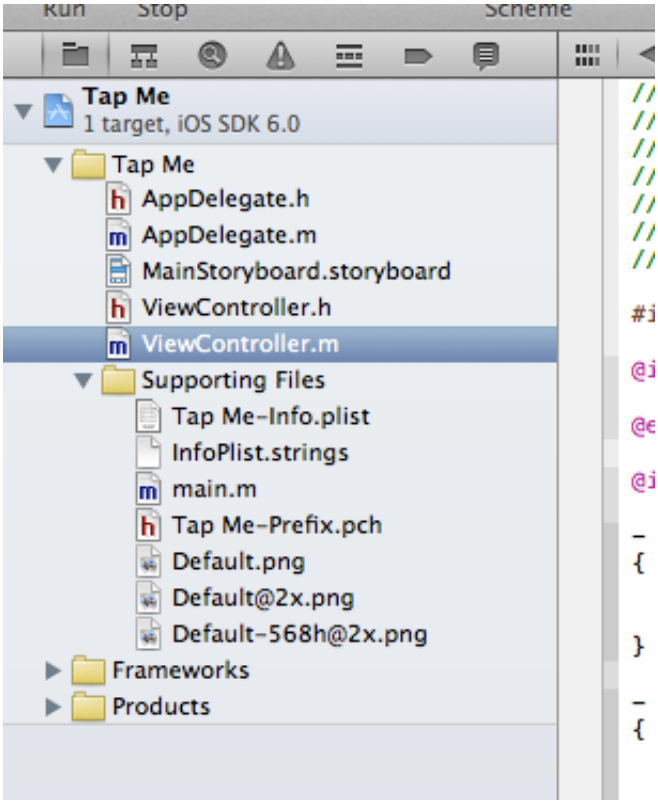
The code above will set up instance variables called **scoreLabel** and **timerLabel** that you can use later on to programatically change the label text. **IBOutlet** is a hint to Xcode that you want it to be an outlet and **UILabel** is the class name for a text label.

Next, add the declaration for a method called **buttonPressed**. This line should go just above the line that says **@end**:

```
- (IBAction)buttonPressed;
```

IBAction is another hint telling Xcode that this method will be connected to some action, such as a button press or a switch being toggled. After you fill in the implementation, you can connect the "Tap Me!" button to call the buttonPressed method.

Switch over to **ViewController.m** and have a look at the implementation code.



Xcode has stubbed out a few methods. One is called **viewDidLoad**, which gets called after the view has loaded and is ready to be displayed. The **didRecieveMemoryWarning** method gets called when the device is running low on memory; there's no need to worry about this method in your tutorial app!

It's time to add the implementation for the **buttonPressed** method. Add the following code toward the end of the file, before the line that says **@end**:

```
- (IBAction)buttonPressed {
  NSLog(@"Pressed!");
}
```

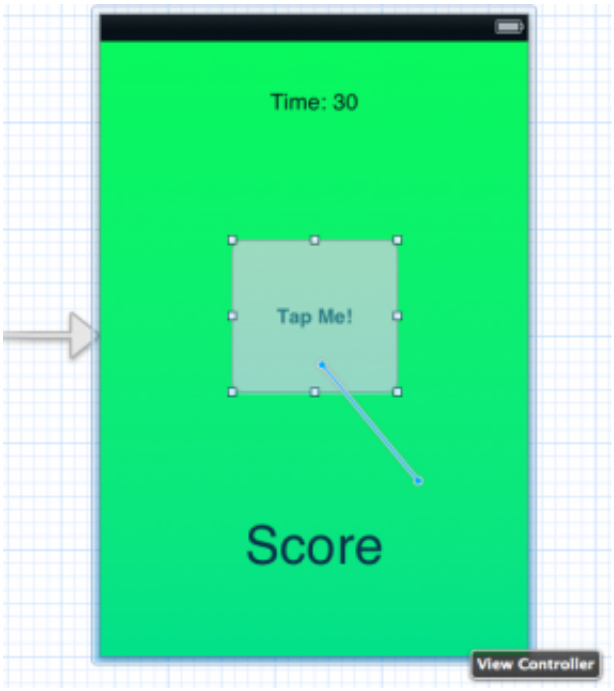
Perfect! The NSLog method will write out to the console when the method is called so you'll be able to tell that everything is working as expected when you run your app.

Connecting the Dots

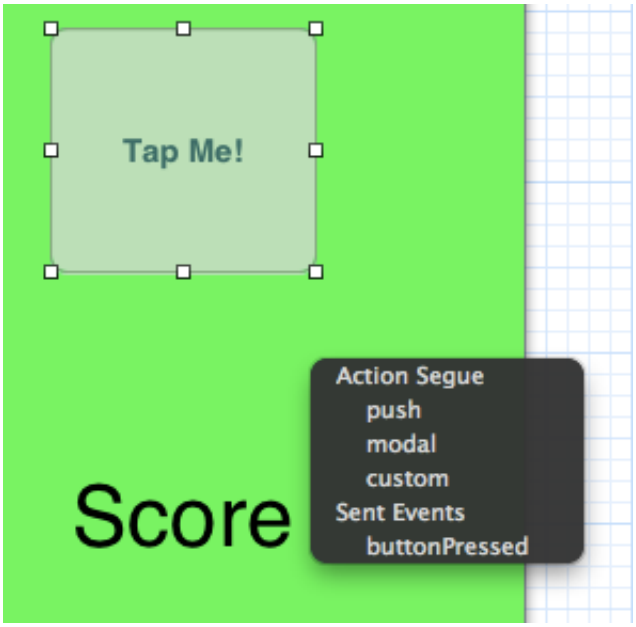
You've set up the look of the app in the storyboard and set up the view controller code. Now it's time to connect the action and outlets to get the storyboard and code talking to each other! :]

Open up **MainStoryboard.storyboard**. With the cursor over the "Tap Me" button, hold down the **Control** key, then click and drag to anywhere in the green background. You will see a blue line connecting the button to wherever you drag your

cursor:



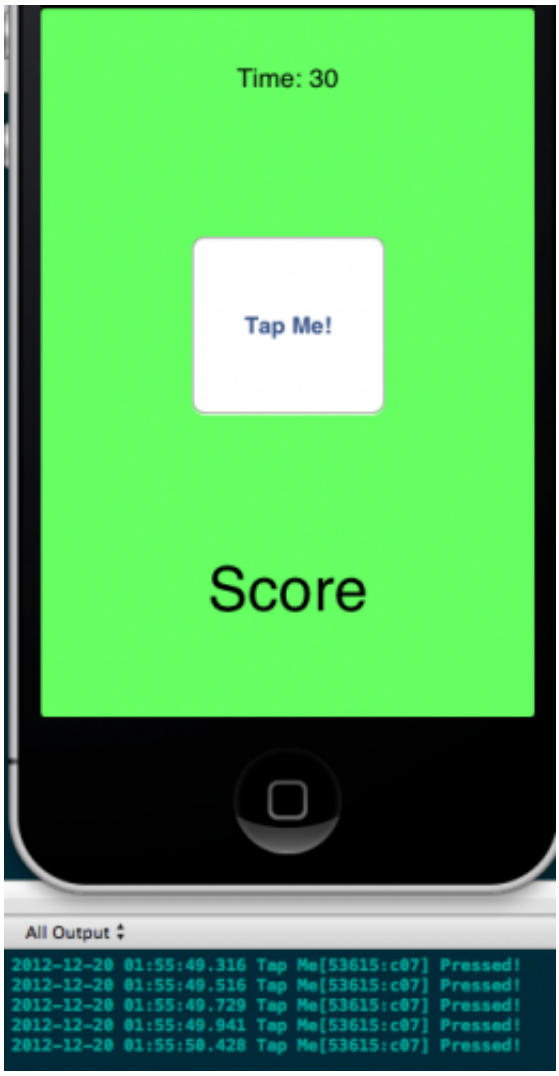
When you release the mouse or trackpad button, a little window will pop up. Hey, cool — there's the **buttonPressed** method that was created earlier! Select it now to hook the two together:



1. Right click on the button
2. Touch Down - add pressedButton

Because you declared buttonPressed with the **IBAction** keyword, Xcode offers it as a choice here. If you don't see **buttonPressed** in the window, go back one step and check your code in the **ViewController.h** header file.

Time to test out the button! :] Run the app and tap the button a few times. Your **buttonPressed** method should be getting called and you will see some debugging output, as below:

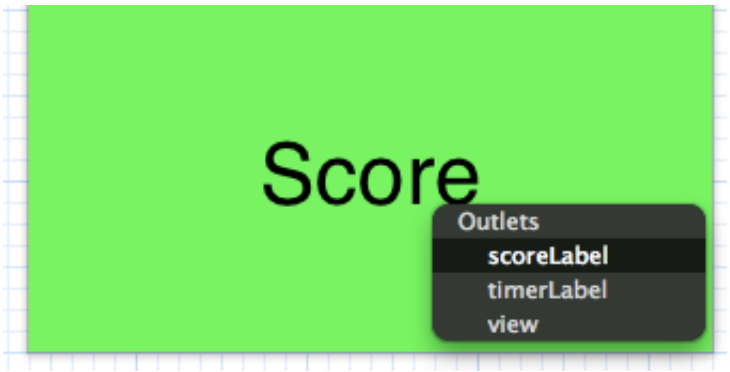


Great, a working button! You're closing in on your completed app!

Note that to connect an action you held down the control key, then clicked and dragged from the button (the view) to the view controller (in the background area).

However, to connect an outlet to the label, you need to do the opposite: hold down the control key, then click and drag from the view controller (represented by the round yellow icon at the bottom) to the "Score" label , just like below:

Yup, there's **scoreLabel**, which you created in your code earlier! Select it now:



Now the label is connected and accessible from the code! Do the same thing to connect the outlet to **timerLabel**. You need to control-click and drag from the view controller to the "Time: 30" label this time.

Okay, you have things connected and some console debugging code setup in the app. But you still need the countdown to happen, and the score to change when the player taps the button. Head on in to the next section to do this now! :]

Labeling Labels

You've already made sure the button works, so it's time to do the same with changing the label's text. In **ViewController.m**, replace what you previously wrote for the **buttonPressed** method with this code:

```
- (IBAction)buttonPressed {
  scoreLabel.text = @"Pressed!";
}
```

The **scoreLabel** variable is the outlet connected to the UILabel in the storyboard. Now when the button is pressed, the label text should update.

Build and Run the app to see for yourself that the buttonPressed gets called and properly updates the label! Neat, eh? :]

You're setting the label text from code, but there are many more interesting attributes such as text color, size, alignment, and others that you can access programatically. The iOS Developer documentation is a great way to learn more about all the things your UILabels and UIButtons can do.

Testing, Testing, 1, 2, 3

Now that the interface is set up and the code is running, pause for a moment and think about the game mechanics.

There are a few things you'll need to keep track of:

- The number of times the button was pressed
- The number of seconds remaining
- Some way to keep track of time

Time to add a few more variables! Add the code below to **ViewController.h**:

```
@interface ViewController : UIViewController {
    IBOutlet UILabel *label;
    IBOutlet UILabel *timerLabel;

    // Add the next three lines
    NSInteger count;
    NSInteger seconds;
    NSTimer *timer;
}
```

In the code above, the "count" variable will hold the number of button taps, and "seconds" will hold the number of seconds remaining. The **NSInteger** type makes the most sense since it can store integers (whole numbers) from zero up to over two billion. Here's hoping that no one who tries your app can tap that many times in 30 seconds! :] The "timer" variable will be dealt with in the next section.

When the player taps on the button, you need to increase the tap count and update the label on screen with the new count. Change the **buttonPressed** method as below:

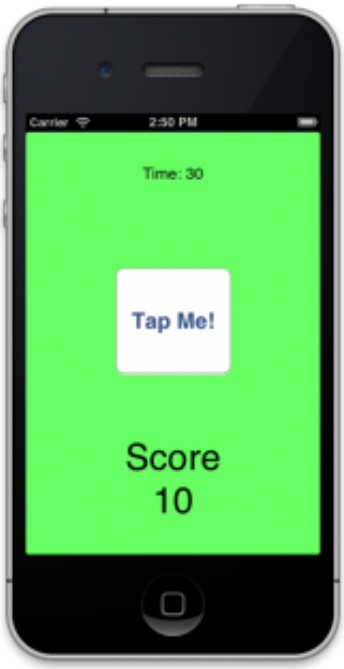
```
- (IBAction)buttonPressed {
  count++;

  scoreLabel.text = [NSString stringWithFormat:@"Score\n%i", count];
}
```

The code above probably looks a little obtuse, so break it down piece by piece:

- "Score" – The plain old word "Score"
- "\n" – A line break. Everything else after this will be on the next line
- "%i" – A placeholder for an integer. In this case, the "%i" will be replaced by your variable **count**

Build and run the app! Tap the button a few times, and you should see your score increase as you tap!



Well, you can't give your player unlimited time to tap the button, can you? :] Looks like you need a countdown timer!

Time For a Timer

Now that game works and keeps score, you'll need to set up the timer to count down from 30 seconds to zero, when the player's turn will end.

First, set up a method that will initialize the game state. Add this method to **ViewController.m**:

```
- (void)setupGame {
    // 1
    seconds = 30;
    count = 0;

    // 2
    timerLabel.text = [NSString stringWithFormat:@"Time: %i", seconds];
    scoreLabel.text = [NSString stringWithFormat:@"Score\n%i", count];

    // 3
    timer = [NSTimer scheduledTimerWithTimeInterval:1.0f
                                             target:self
                                           selector:@selector(subtractTime)
                                           userInfo:nil
                                            repeats:YES];
}
```

Part 1 of the method resets the clock to 30 seconds and the tap count to 0. This will be the initial state for the game.

Part 2 of the method resets the on screen display. The stringWithFormat call and the %i should already be familiar to you!

Part 3 is one dense-looking bit of code. Here, you're setting up a **NSTimer** object that will send you a message every second. That way, you'll be able to update the number of seconds remaining and end the game after 30 seconds.

If you break down the various parts of the NSTimer call, it isn't too scary:

- **Time Interval** is simply how often you want the timer to go off (1 second, in this case)
- **Target** is which instance to send a message to every second. You're in the view controller now and you want the message to go to the view controller, so the target is **self**.
- **Selector** is what method you want to call. It isn't enough to write the method, you also need to put @selector around the method name.
- **User Info** is any extra info you want stored with the timer. You won't need anything for this timer so you say nil to represent nothing.
- **Repeats** says if the timer should repeat or just fire off once. You want the timer to go off every second until you say stop, so this is set to **YES**.

You've asked the timer to call the **subtractTime** method every second but you haven't defined it yet! :]

Add the following code to **ViewController.m**:

```
- (void)subtractTime {
    // 1
    seconds--;
    timerLabel.text = [NSString stringWithFormat:@"Time: %i",seconds];

    // 2
    if (seconds == 0) {
        [timer invalidate];
    }
}
```

At this point I had [libd] thread issues for an hour. What I did was removing all the yellow warning sign by following the recommendation

Part 1 of the method should be familiar looking. Here, you're decreasing the number of seconds and then updating the label on screen with the new time.

When the number of seconds hits 0, you'll want the timer to stop and end the game. When you tell the timer to invalidate, it will stop the timer from calling subtractTime again.

There's just a few last-minute things to add to your app until it's done! :]

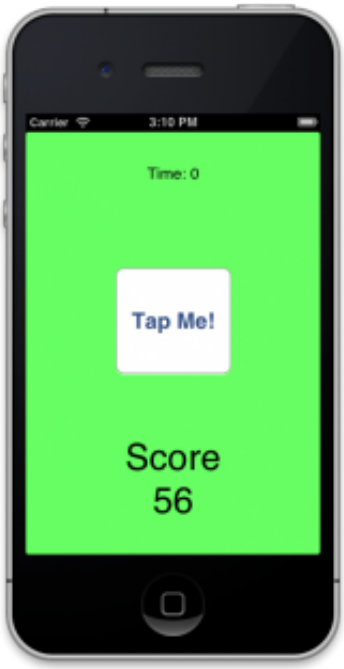
Starts and Stops

Lots of pieces are now in place! To keep things simple, you can just start the game, and the timer, when the app launches. Update the viewDidLoad method in **ViewController.m**:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    [self setupGame];
}
```

Remember, iOS will call viewDidLoad automatically for you when the storyboard and view are loaded. The call to the setupGame method you just wrote will then start things off.

Launch the app and see what kind of high score you can reach!



Game Over Weirdness

Did you notice a few odd things about how your app handles the end-of-game scenario? :]

When the timer hits zero, you can still tap the "Tap Me" button, so you'll need some way to prevent the user from clicking it! As well, there's no way to restart the game and try again. Hmm, that could be a problem! :]

An alert sounds like a good fit here. When the game is over, you can add an alert that will pop up showing the score and displaying a button to the user that that lets you play the game again. Since you want the alert to show when the game is over, add the code for it inside the subtractTime method:

```
    // 2
    if (seconds == 0) {
        [timer invalidate];

        // new code is here!
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Time is up!"
            message:[NSString stringWithFormat:@"You scored %i points", count]
            delegate:self
            cancelButtonTitle:@"Play Again"
            otherButtonTitles:nil];

        [alert show];
    }
```

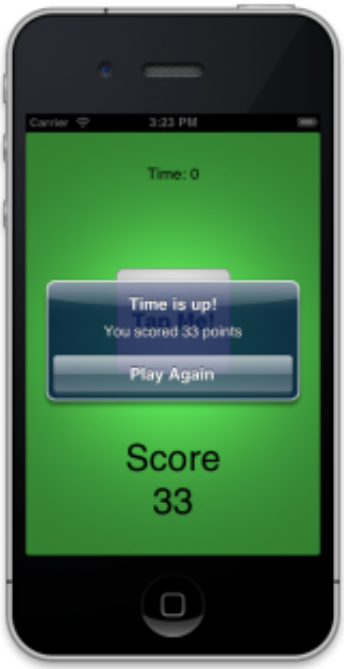
A UIAlertView has a title, message, and one or more buttons. The first part of the new code sets up the alert, and then sending it the **show** message will display it to the user.

Break down the various pieces of the alert call, and you'll see the following:

- **Title** – the title that will appear at the top of the alert.
- **Message** – the message in the center of the alert. In this alert, you want to display the score.

- **Delegate** will be set to self, just like the timer. Delegates will be explained in the next section.
- **Cancel Button Title** – the title of the button. For this app, it is Play Again.
- **Other Button Titles** – a list of button titles. You only need a play again button so set this to nil.

Now, build and run the project and see how your new, end-game code works! :]



Awesome! So the alert now pops up and shows the high score. And pressing "play again"…uh, doesn't do anything! What gives?

In your Storyboards, you connected an IBAction to the button to handle the tap events, but there's nothing like that for alert buttons. Instead, you need to use something called a **delegate**.

Dealing With Delegates

A **delegate** is a way for an object to find out about interesting things that are happening. For example, you might set your alarm clock to wake you up at 7 in the morning or you might ask your friend to call you when she gets home. In these examples, you are the delegate (the thing being told) and the alarm clock and your friend are the things that will tell you something (such as, "it is time to wake up" or "I am now at home").

Going back to the app, your alert has something interesting to tell – when the "Play Again" button is pressed. Your view controller would love to know when that happens, so it should become the alert's **delegate**.

So you need to state that the View Controller is a delegate for UIAlertView. Near the top of **ViewController.h**, change the line starting with "@interface" like this:

```
@interface ViewController : UIViewController<UIAlertViewDelegate> {
```

Just as the alarm clock has a certain signal it can send (playing music or an alarm sound), UIAlertView has a set of messages it can send to its delegate. You can check out the documentation for a full list, but the one you're concerned with here is called "alertView:clickedButtonAtIndex:". This will send a message when a button is clicked.

Now you just have to put that method into the View Controller!

Add the code below to **ViewController.m**:

```
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    [self setupGame];
}
```

To summarize: when the game ends, a UIAlertView will be displayed with the score and a button. When the button is tapped, a message will be sent to the delegate — your view controller — which will then call the **setupGame** method you wrote a while back. To refresh your memory, **setupGame** resets the time and score and restarts the game.

What are you waiting for? Build and run your app and test your skills in a button tapping frenzy!

Congratulations!! You have now made your first iOS app – be sure to show it off to your family and friends! :]

Where To Go From Here?

Here is an example project with all of the code you've developed in this tutorial so far.

This is a great little app to play around with to get a feel for how UI elements, Views and View Controllers work. Feel free to play around with different parts of the app. See if you can change the look of the app or try adding additional features!

Check out the next tutorial in this series, where you will learn how to add artwork and sounds to the app to make it more polished and to give the user a better playing experience.

If you have any question or comments, I would love to hear them in the forums!

Mike Jaoudi

Mike Jaoudi is a Computer Science major at New York University. For the past couple of years, Mike also worked as an Instructor and Curriculum Developer at iD Tech Camps for iOS App Design and iOS Game Design. While at iD Tech, Mike discovered an enjoyment for teaching, as he's found that "when you teach you learn twice". He is a fan of the BBC show "Sherlock" and NCAA fencer. Check out his multiplayer snake game, Snakez. You can find Mike on Twitter .