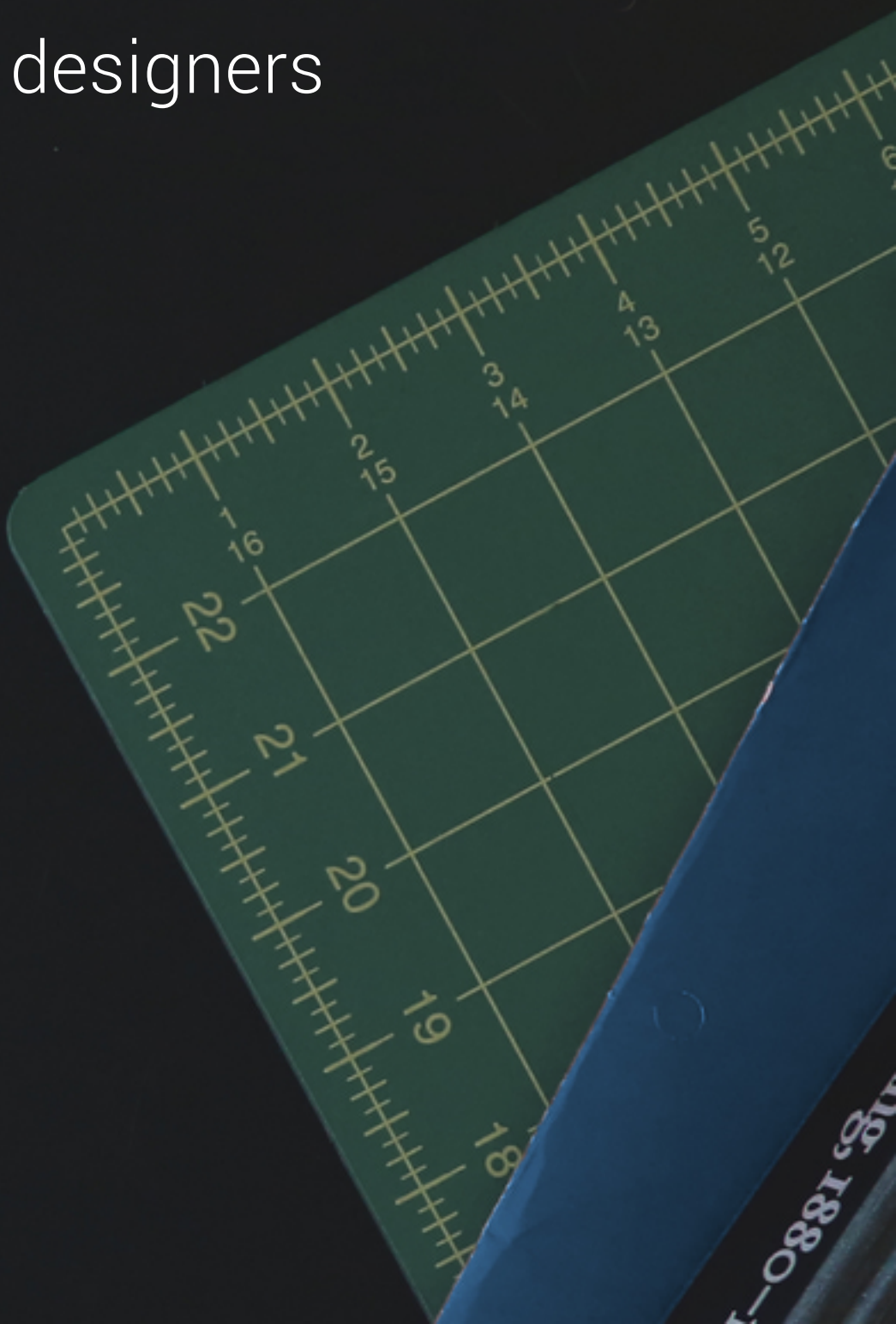
CoffeeScript for Framer.js

A guide for designers



Tessa Thornton

Contents

Introduction

What is CoffeeScript?

CoffeeScript is a relatively new programming language often used by front-end developers to create browser-based interfaces.

CoffeeScript is interesting in that it *compiles to JavaScript*. Compiles just means turns into or is transformed into. When we write CoffeeScript, we need to *compile* it into JavaScript before it can run in browsers. To understand more about CoffeeScript, you'll need to understand a bit about JavaScript.

What is JavaScript?

JavaScript is the programming language we use in browsers to do things like manipulate HTML and communicate with servers. JavaScript, along with

HTML and CSS, allows us to create rich and responsive user interfaces.

JavaScript is one of the most popular programming languages in the world, mainly because it's the *only* programming language that runs in browsers.

JavaScript was created by Brendan Eich at Netscape in 1995 to give web developers and designers an accessible way to manipualte web pages. It very quickly gained popularity because it allowed developers to add a lot more functionality to web pages and was relatively easy to learn.

Despite its popularity, JavaScript has always had many critics. Though many of its early flaws have been overcome in more recent releases of the language, many still consider it to be an "ugly" language with a lot of historical baggage.

Why CoffeeScript?

Though there have been various attempts to bring other programming languages to the browser, none have been successful, so we're still more or less "stuck" with JavaScript.

In the mid-2000s, another language called Ruby was gaining popularity as a server-side programming language (meaning it isn't run in the browser, it's run on the server that hosts a web site or application). Ruby, in contrast to languages like JavaScript, was designed to be easily human-readable and writeable, with an emphasis on developer productivity even enjoyability.

In 2009, Ruby developer Jeremy Ashkenas sought to bring some of the features he liked most about Ruby to client-side development (client-side means stuff that happens in the browser), and created CoffeeScript.

Since browsers can only understand JavaScript, code written in CoffeeScript first has to be *compiled* into JavaScript before it can run so that browsers can understand it. So if you were writing code in a file called `script.coffee`, you'd have to convert that file to `script.js` using the CoffeeScript compiler.

Many developers consider that extra step a worthwhile cost for the benefits of writing CoffeeScript, which they feel makes them more productive and

makes up for some of the shortcomings of JavaScript.

For example, the JavaScript code to output the numbers between 1 and 10 in reverse order looks like this:

```
var countdown, num;

countdown = (function() {
  var i, results;
  results = [];
  for (num = i = 10; i >= 1; num = --i) {
    results.push(num);
  }
  return results;
})();
```

The same code in CoffeeScript looks like this:

```
countdown = (num for num in [10..1])
```

The code is both shorter and easier to read and comprehend.

CoffeeScript and Framer.js

Framer.js is a JavaScript framework for prototyping user interfaces. If you wanted to, you could include framer.js in an HTML file, and then write that takes advantage of the framework in plain JavaScript.

Framer Studio is a companion Mac application that is based on Framer.js. Framer Studio makes your workflow much easier with features like a live preview panel and Sketch or Photoshop importers.

Framer Studio's editor allows you to write your code in CoffeeScript instead of JavaScript. Because Framer's target user base is designers, not developers, CoffeeScript offers a gentler learning curve for non-programmers and can be much faster to write, which is key when prototyping.

Framer Studio vs. Framer.js

Though I'd strongly recommend Framer Studio if you're going to be using Framer for a lot of prototyping, it is possible to take advantage of the library without using the app. The Github project includes instructions for setting up a JavaScript project with Framer.js, but it's fairly simple to set it up to use with CoffeeScript.

In this book, I'll be using Framer Studio for examples. You'll need either Framer Studio or a way of compiling CoffeeScript to follow along. To compile CoffeeScript without using the command line, you can use one of the following GUIs:

- Prepros (Mac/Windows/Linux). Indefinite free trial/$29
- Koala (Mac/Windows/Linux). Free
- Codekit (Mac). $32

Note: examples will make use of Framer Studio's built-in device templates. Not tested in regular browser environment.

Setup

All you'll need to follow along with this book is Framer Studio or Framer.js and CoffeeScript. For the earlier chapters, I recommend typing code into a browser-based console to observe the output yourself. I suggest CoffeeScript REPL, or if you're familliar with Chrome's web developer console, you can add a plugin that will let you run CoffeeScript, like CoffeeConsole or Scratch JS (go to settings and select "CoffeeScript" for the transformer).

The later examples require using image assets created for the projects, which came in the **Assets** folder you downloaded with this ebook.

1: CoffeeScript for beginners

Math

Let's start with some simple math.

CoffeeScript supports all your familiar math operators: `+ - * \` (add, subtract, multiply, divide), plus one you may be unfamiliar with: `%` or modulo.

A lot of what you'll be doing while prototyping interactions is just simple math.

Quick refresher on what you'll be up against:

Order of operations

So just like you learned in high school, BEDMAS still applies. If you want some addition and subtraction to happen before the multiplication and addition, put it in brackets.

Type into your CoffeeScript console of choice and observe:

```
10 + 20
# => 25

150 - 5 * 20
# => 50

(150 - 5) * 20
# => 2900
```

Numbers and math will behave more or less the way you remember from middle school. If you get stuck trying to do something like rounding a number, check out the MDN documentation for `Math` or just try a search.

Types of data

There are all different types of data you can program with, and some of them have special abilities and uses.

Numbers

Numbers are the simple ones. Numbers are numbers. `200`, `-10`, `4000` are all numbers. Don't include commas or spaces in your numbers, and you'll be ok. Numbers in CoffeeScript can have decimals and can be positive or negative.

Strings

When you're working with letters or words or punctuation, you're working with `strings`. Strings come in quotation marks. **Anything in quotation marks is a string**. You can use single quotes or double quotes, but there are fewer complications when you use double-quotes.

```
"This is a string"
'This is also a string'
```

You can squish strings together using a `+` sign. This is called *concatenating*.

```
"My name is " + "Tessa"
# => "My name is Tessa"
```

Doing things with numbers and strings Since anything in quotation marks is a string, you can end up with numbers that are actually strings (becasue they're in quotation marks). `"40"` is a string, `40` is a number.

Some strange things can happen if you treat strings like numbers:

```
"50" + "50"
# => "5050"
```

Instead of adding the numbers together mathemtically, the two strings were squished together. If one of the values is a string, and one is a number, we get the same result:

```
"50" + 50
# => "5050"
```

In some cases, mixing numbers and strings will work out okay, but it's best to avoid it when you're trying to do math.

Using variables

Other than some simple math, we can't do much of interest with just numbers and strings and mathematical operators. One of the most powerful tools we have for organizing our code is *variables*. Variables let you *assign* a value to an arbitrary symbol for later reference. A variable is a box you can put values into. Any kind of value, like a string or a number. Variables are assigned using the = operator.

```
name = "Tessa"

print name

# => "Tessa"

age = 26

print age

# => 26
```

Note The `print` command just outputs the result of our code to a console. In the browser, `print` doesn't exist, but you can use `console.log` for the same purpose.

A variable will retain the value you assigned to it until you change it. You can change a variable any time in your program (hence the name *variable*).

```
color = "green"

print "my favorite color is " + color
# => "my favorite color is green"

color = "red"

print "my new favorite color is " + color
# => "my new favorite color is red"
```

There are a couple rules about variable names:

- variable names can't contain spaces
- variable names can't start with numbers
- variable names can contain upper case and lower case characters
- variable names can't contain punctuation other than _

There are some common naming conventions and patterns for variable names. When a variable name is more than two words, you can combine the two words together using underscores "camel casing".

```
my_name = "Tessa"

myName = "Tessa"
```

String interpolation

The example `print "my favorite color is " + color` wasn't all that complicated, but combining variables with strings can easily get a bit messy. For example, if the variable comes in the middle of the string:

```
color = "green"

print "my favorite color is " + color + ", what's yours?"
```

We have to use a bunch of + signs, and remember where to put spaces and punctuation. There's an easier way, called *string interpolation*. We can embed the variable right in the string if we surround it with `#{}`:

```
color = "green"

print "my favorite color is #{color}, what's yours?"

# => "my favorite color is green, what's yours?"
```

String interpolation can make our code much easier to read.

Boolean values

Booleans are values that are either true or false. They're indicated with just the words `true` or `false` without quotation marks.

```
myBoolean = true
```

Boolean values are often the result of making comparisons:

```
10 > 9

# => true

9 < 8

# => false
```

You can assign the *result* of a comparison to a variable:

```
theTruth = 10 < 5

print theTruth
# => false
```

Comparing for equality

In regular math, you compare values using the = sign. As you hopefully recall, we're already using the = sign to assign variables (`myVar = 10`), so it would be confusing and error-prone for us to also use = for comparison.

In CoffeeScript, you can use the `is` operator to check to see if two values are the same.

Note: you may see code where == is used to compare values. In CoffeeScript `is` is a shortcut for ==. We'll use `is` because it's easier to read.

```
5 is 5

# => true

num = 5

num is 5

# => true

num is 10

# => false
```

To negate a condition you use the keyword `not`. In place of `is not` you can use the shortform `isnt`

```
num = 5

num isnt 10

# => true
```

Conditional statements

Comparing variables is only useful if we do something with the outcome of the comparison. That's what conditional statements are for. CoffeeScript uses simple `if/else` statements to run different code in different scenarios:

```
num = 14

if num >= 16
  print "you can learn to drive"
else
  print "you're too young to learn to drive"

# => "You're too young to learn to drive"
```

Note >= means "greater than or eaqual to" just as <= means "less than or equal to".

Indentation

If you've ever looked at other programming languages like Java or JavaScript before, you might have noticed that they have a lot of symbols like semi-colons and parentheses. CoffeeScript avoids using a lot of these symbols, which can make it much easier to read and write. To get away with this, in CoffeeScript we need to follow certain rules about indentation. In the above example, the indentation within the `if` and `else` statements is important. It indicates that the indented code "belongs" to the `if` statement, and so will only be run if the condition is true.

Comments

Sometimes you want to leave notes for yourself or others in your code. Maybe to explain what something does, or remind yourself to come back to something, or to help with organization. Comments don't get read by the computer when your code runs, so you can put whatever you want there.

In CoffeeScript, lines that start with a `#` will be treated as comments and ignored.

```
# this is a comment. It doesn't do anything. But it's nice
   to read.
```

Functions

Functions wrap up a bit of code for re-use. For example, the "age check" code that we wrote above can be wrapped up in a function so that we can re-use it on every young-looking driver we encounter.

When you make a function, you want to be able to *use* it somehow, so you have to have a way to reference it. We can do this by assigning our function to a variable:

```
checkAge =
```

To indicate that we're putting a function in this variable, we use the `->` arrow

```
checkAge = ->
  # we'll put the code for checking age here
```

The code above just says "checkAge is a function" but it doesn't do anything yet. To "call" our useless function (calling a function = using a function), we use parentheses.

```
checkAge = ->
  # doesn't do anything yet

checkAge()
```

The `()` part basically means "go". It tells the computer "run the function in the checkAge variable"

So let's make our `checkAge` function actually do something:

```
checkAge = ->
  if age >= 16
```

```
    print "Carry on"
  else
    print "Get out of the car please"
```

Note: the indentation is again significant. All the code wrapped in the checkAge function needs to be indented one level to indicate that it belongs to it.

Now we can call our function:

```
age = 16

checkAge()

# which will output "Carry on" because we set the age
   variable to 16
```

Lets try it with a couple young drivers:

```
age = 15

checkAge()

# => "Get out of the car please"

age = 18

checkAge()

# => "Carry on"
```

Functions with arguments

Functions can be even more useful if we can give them values to work with. These values are called *arguments*. If we give checkAge an age argument, we don't need to have a separate age variable.

Let's rewrite checkAge to accept an age *argument*:

In CoffeeScript, we can give a function the ability to accept arguments by adding parentheses containing the argument name before the -> sign.

```
checkAge = (age) ->
  if age >= 16
    print "Carry on"
  else
    print "Get out of the car please"
```

Once you've added the argument name to the parentheses, you'll be able to reference whatever `age` is using its name.

`age` gets its value when the `checkAge` function is called. To give `checkAge` an `age` argument, we put the value in the parentheses:

```
checkAge(17)

# => "Carry on"
```

Functions can take multiple arguments:

```
patrol = (age, speed) ->
  if speed > 60
    if age >= 16
      print "Happy speeding ticket"
    else
      print "Get out of the car, kid"
```

In this example, the `patrol` function also takes a `speed` argument. Now we only do the age check if the speed is greater than 60. Budget cuts.

Note how all the code nested under the `if speed > 60` statement is indented an additional level.

To use our new `patrol` function, we now have to put two values in the parentheses: the first one is the age, the second one is the speed:

```
patrol(17, 70)

# => "Happy speeding ticket"
```

```
patrol(17, 40)

# => ... (nothing happens)

patrol(15, 90)

# => "Get out of the car, kid"
```

If we forget to add the the `speed` argument:

```
patrol(16)

# => undefined
```

Using pre-written functions

When you're prototyping animations and interactions, there's a good chance you won't have to write a whole lot of functions yourself. You will however be *using* quite a few functions, most of them are provided by the framer.js library.

That's where functions become really useful: when you can share them around. The folks beheind Framer figured out how to do all sorts of useful things related to manipulating pixels on a screen, so they wrapped up all that useful code into functions that you can use.

Let's say we're using a library that gives us a `licensePlateCheck` function. It takes one argument, which is a license plate. It does all sorts of complicated things to associate that license number with a person, find out whether that person has a criminal record, if there are any warrants for their arrest, or if the car is stolen. To use this function, you don't need to know any of that. All you have to know is that it takes one argument, and that that argument needs to be a license plate. For any given license plate, the `licensePlateCheck` function will tell you `true` if the plate is associated with criminal activity, and `false` if it's not.

We also need to know what type of argument to provide. In this case, we need to know that the `licensePlate` argument is a string. Which makes sense, since it's a mix of numbers and letters.

```
licensePlateCheck("BAD455")

# => true
```

And that's all we need to do to find out that the license plate is associated with trouble.

Other types of data

Strings, numbers, and booleans are the simplest kinds of values in CoffeeScript, but we can do more with more complex data types, like arrays and objects.

Arrays

Arrays are lists or collections of multiple items. Say we wanted to keep track of a list of fruits:

```
fruits = ["apples", "oranges", "bananas"]
```

Arrays have some built in *methods* (functions that they can use) for finding out information about them, like `length`:

```
fruits.length

# => 3
```

You can access elements in an array by their *index*. The index is the element's position within the array.

Arrays are zero-indexed. This means that the first item in the array is item 0, and the second item is item 1. This can be a little confusing at first, but you'll get used to it.

To access an element in an array, we use square brackets containing the index of the element we're looking for. For example, if we want go get "oranges" from the fruits array:

```
print fruits[1]

# => "oranges"
```

Since "oranges" is item 1 in the array (the second item).

Adding to arrays

You can add new items to an array with the method `push`. `push` adds the item you specify to the end of the array.

```
fruits.push("kiwis")

print fruits

# => ["apples", "oranges", "bananas", "kiwis"]
```

Looping through arrays

Arrays can be very powerful in CoffeeScript, because they let you repeat certain functionality over and over with different items in an array.

One of the most common ways to control your program is by using *loops*. Loops let you do something for every item in an array.

`toUpperCase` is a method that you can use on strings to capitalize them

If we wanted to print out each element in our array of fruits in capital letters, we could do:

```
fruits[0].toUpperCase()
# => "APPLES"

fruits[1].toUpperCase()
```

```
# => "ORANGES"

fruits[2].toUpperCase()
# => "BANANAS"
```

But since we have an array containing our list of fruits, we can do this a lot more efficiently with a `for` loop:

```
for fruit in fruits
  fruit.toUpperCase()

# => "APPLES"
# => "ORANGES"
# => "BANANAS"
```

In english, we read that as "for every fruit in our list of fruits, capitalize that fruit"

If we break it down, the `for` loop does two things: it executes our code once for each item in the array, *and* it lets you refer to the currently "active" element by whatever name you like.

The code `for fruit in fruits` tells us that each time through the array, we'll have a variable called `fruit`. That variable `fruit` will refer to the item we're working with each time through. So the first time through the array, `fruit` will refer to "apples", the second time it will refer to "oranges", etc.

What if we had a lineup of cars to run license checks on?

```
plates = ["BRR010", "BUU888", "NNB001", "MBB991"]
```

Let's check each plate for criminal activity with a `for` loop:

```
for plate in plates
  licensePlateCheck(plate)

# => false
# => true
# => false
```

Again, the indentation is significant. The indentation of the second line means that code gets executed only inside the `for` loop.

Objects

Objects are one of the most useful data types in CoffeeScript. An object is a collection of *properties*. In programming, a *property* is an association between a *name* and a *value*. A name-value pair could be something like "price: $10", where *price* is the name, and *$10* is the value, or "age: 30", where *age* is the name, nad *30* is the value.

If something has properties, you can store them in an object. For example, a "book" object might have a `title` property, an `author` property, and a `genre` property. In CoffeeScript, we would code that as:

```
book =
  title: "Slaughterhouse Five"
  author: "Kurt Vonnegut"
  genre: "Science Fiction"
```

You can store any kind of data in an object's properties, including arrays and booleans:

```
book =
  pages: 256
  genres: ["Science Fiction", "Satire"]
  fiction: true
```

You can even store another object inside a property:

```
book =
  author:
    name: "Kurt Vonnegut"
    born: 1922
    died: 2007
```

Note that we use `=` and `:` in the above example. Note the difference: `=` assigns a varibale name (book) to an object, and `:` matches up names and values. `author` is the name, and the object containing `name`, `born`, and `died` is the value.

To access properties in an object, we use dots:

```
print book.title

# => "Slaugherhouse Five"

print book.pages

# => 256
```

We can keep going with the dots to access objects within objects:

```
print book.author.born

# => 1922
```

And to access the elements of the array inside the object, we combine the square bracket and the dot syntax:

```
print book.genres[1]

# => "Satire"
```

Methods Since we can store any kind of data in an object's properties, we can also store functions as properties. When a function is the property of an object, it is called a *method*.

```
book =
  title: "Slaugherhouse Five"
  read: ->
    print "All this happened, more or less."
```

The `read` property is a method that belongs to our `book` object. We can call it like we would any function:

```
book.read()

# => "All this happened, more or less."
```

This is how the `.length` method and the `toUpperCase` method work. They are methods that belong to all string objects.

Configuration objects In framer.js, one of the most common uses of objects will be to configure animations and elements:

```
box =
  width: 120
  height: 120
  x: 0
  y: 0

animation =
  duration: 300
  easing: "ease-in"
```

Objects are ideal for configuring animations and elements since they all have many properties of different types (some are numbers, some are strings, etc).

2: Simple Animations

Let's open up Framer Studio and try to create some simple animations.

Creating layers

We're going to work with a small square for a while.

To create a new element on the screen, we use `new Layer`. This is a special kind of function that creates an object. If we assign a variable to the result of `new Layer`, we can maintain a reference to that element that we can manipulate.

```
square = new Layer()
```

You should see a blue square on the screen.

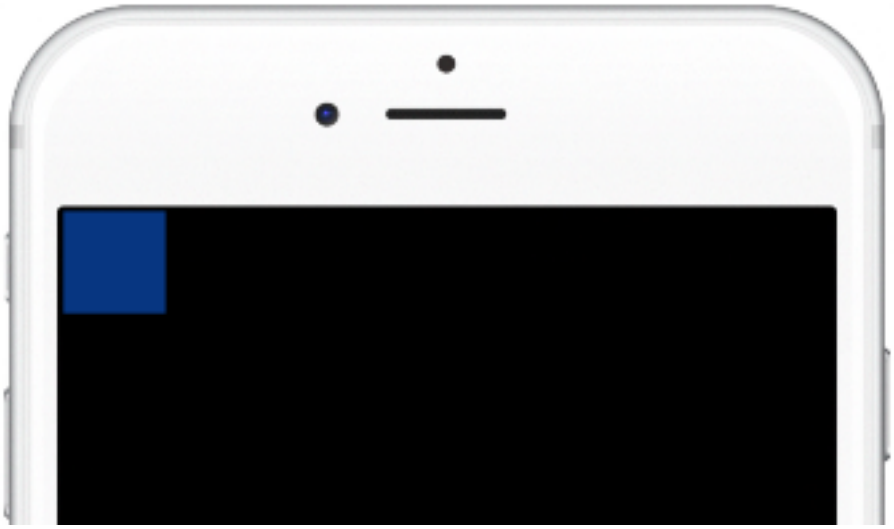


Figure 1: New Layer

To customize the square so it's not just a plain blue square, we can pass one argument to `new Layer()`. This argument is a configuration object, you can configure the element using properties for various attributes such as width, height, position, and appearance.

```
square = new Layer(
  width: 200
  height: 200
  x: 100
  y: 100
)
```

To make that code a bit easier to read, we can get rid of the parentheses.

```
square = new Layer
  width: 200
  height: 200
```

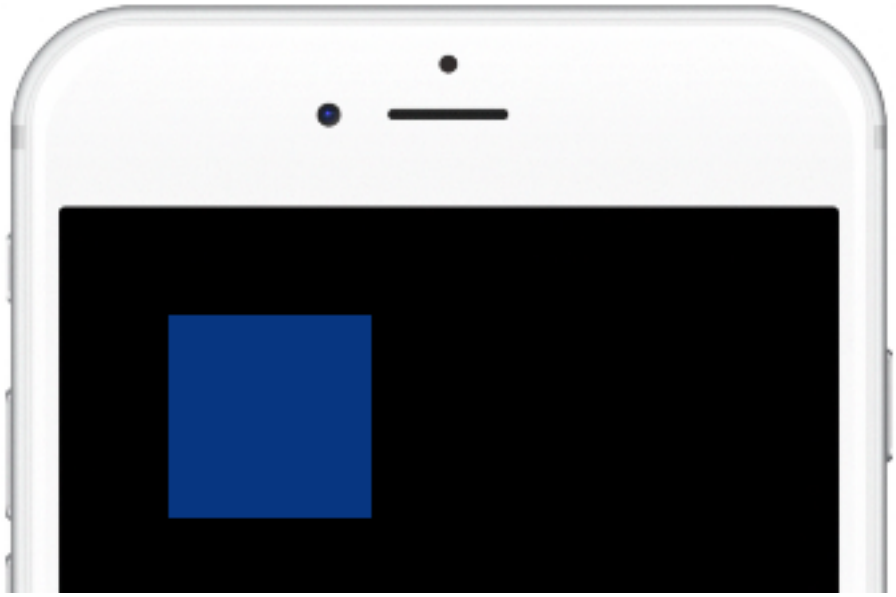


Figure 2: Layer with properties

There are a lot of cases in CoffeeScript where we can remove parentheses like this, and it can sometimes make the code look a bit cleaner.

Manipulating layers

Now that we've assigned the variable `square` to our layer, we can continue to manipulate it. After we've created our layer, we can modify it at any time:

```
square.backgroundColor = "red"
square.x = 200
```

Note the dot syntax: this is because `square` is an object. It has a `backgroundColor` property and a lot of other properties that we can manipulate.

You can modify a layer's coordinates (x, y), as well as appearance by using camel-cased (camelCased) versions of most CSS properties, as well as some built-in framer properties. For example, `color` is just `color` but `border-radius`

becomes `borderRadius`. If you want to rotate an element, you can modify the `rotation` property framer provides. Framer Studio will autocomplete most of these property names for you, but you can also look them up in the docs.

Animating layers

Let's add our first animation. To animate a layer, we use the `animate` method.

```
square.animate()
```

By itself, `.animate()` isn't goint to do anything. We need to pass it a *configuration object* to tell it what to animate, and how to animate it. A configuration object will follow this format:

```
configObject =
  properties:
    property: value
    property: value
  time: 1
  curve: "ease"
  delay: 2
```

The `time`, `curve` and `delay` properties are all optional, but you need to specify one or more properties to animate nested in the `properties` object (it's an object nested inside an object).

For example, if we want to fade out a square, we would animate the `opacity` property. By default, the opacity is set to 1, so we animate it to 0.

```
square.animate
  properties:
    opacity: 0
```

We can easily transition multiple properties at once:

```
square.animate
  properties:
    opacity: 0
```

```
    x: 400
    y: 400
    rotation: 180
```

Configuring animations

Time-related properties (delay and time), are specified in seconds. By default, animations take 1s. Speed up the animation:

```
square.animate
  properties:
    opacity: 0
  time: .2
```

Note: again whitespace is significant here. `opacity` is a property, so it's indented one level further than `properties` to indicate the relationship. Move back out one level to specify `time`, because that's a property of the animation, not a property of the object animating.

To make your animations more dynamic, you can specify the curve. To learn more about curves, check out easings.net and the framer docs. You can use a built-in easing string like `ease-in`, `ease-out` or `ease-in-out`, or use one of the more advanced functions described in the docs, like `bezier-curve` or `spring-dho`.

```
square.animate
  properties:
    x: 500
  curve: "ease-in"
```

Other animation options You can set an animation to repeat any number of times with `repeat`, and delay it with `delay`, specifying the delay in seconds.

```
square.animate
  properties:
```

```
    opacity: 0
  repeat: 4
  delay: 2
```

3: Events

When prototyping interactions, you're often going to want to react to user input. This is done by using the `on` method to "listen" for events triggered by the user.

The code for doing something "on" an event might look a bit weird at first, but we'll break it down step by step.

```
button = new Layer

button.on Events.Click, ->
  doFunAnimation() // do animating here
```

This is actually a method being called with two arguments, though it may not look like it. The format for the `on` method is `on(eventName, function)`, where the `function` is the code that gets run when the event is triggered.

When you use a function as an argument like this, it's called a "callback". In plain english, calling the `on` method with a callback is like saying "listen for and do", and the two arguments are the thing that you're listening for, and the thing that you should then do.

If you remember, in CoffeeScript, we indicate a function with the following syntax:

```
functionName = ->
```

When we give the `on` method a function as an argument, it doesn't need a name or an equals sign, so we just need the `->` part. The comma is the separation between the first argument (the name of the event), and the second (the function).

It might make more sense if we leave on the parentheses:

```
button.on(Events.Click, ->)
```

Framer gives us a bunch of events to listen for, all in the format `Events.Name`. Some of the more common events are `Events.Click` and `Events.TouchStart`.

Let's try it out:

```
button = new Layer

button.on Events.Click, ->
  print "clicked!"
```

Animations and events

If we combine what we know about events and animations, we can begin to prototype interactions. Let's slide our layer right on click:

```
button = new Layer

button.on Events.Click, ->
  button.animate
    properties:
      x: 500
```

Working with screen dimensions

Many of the interactions in web and mobile interfaces require calculations based on the dimensions of the screen itself. For example, if we wanted to slide our box from the left edge of the screen to the right edge, we'll need to know where the right edge is.

We can access the properties of the screen we're working with by accessing the `Framer.Device.screen.width` and `Framer.Device.screen.height` properties.

```
width = Framer.Device.screen.width
```

```
button = new Layer

button.on Events.Click, ->
  button.animate
    properties:
      x: width
```

This will cause the square to animate off the right edge of the screen. The `x` coordinate of our box is calculated from the top left of the screen, so by setting the `x` value to the width of the screen, we've set it just off the edge of the screen. To animate the box so that it stays on the screen, we can subtract the box's width from the screen's width to get the `x` value. By default, all layers are 100px wide.

```
button.on Events.Click, ->
  button.animate
    properties:
      x: width - 100
```

4: Simple interactions

Example 1: Dismiss modal window

View finished animation.

Import the "example1_popup" psd or Sketch file into Framer Studio. Set the device type to iPhone 6 for best arrangement.

The first thing we're going to prototype is the dismissal of this popup when the user clicks on the "x". The "x" layer group is called "close", so we access it by name (it is a property of the imported `psd` object). We'll add a click event handler to the close layer:

```
file = Framer.Importer.load "imported/example1_popup"

file.close.on Events.Click, ->
```

Figure 3: Modal window

To start, we'll just fade out the popup on click. The popup layer is called "popup" so we access it with `file.popup`:

```
file.close.on Events.Click, ->
  file.popup.animate
    properties:
      opacity: 0
```

That's a bit too slow, so let's adjust the `time` property:

```
file = Framer.Importer.load "imported/Popup"

file.close.on Events.Click, ->
  file.popup.animate
    properties:
      opacity: 0
    time: 0.4
```

It's a bit of a dull animation, so let's slide it off the screen upwards. To do this, we'll need to animate the layer's `y` property.

We'll want the layer's end position to be above the top of the screen, and to be all the way off the top of the screen, we'll need to send it above the top edge of the screen by the height of the layer itself.

To do that, we'll need to get the height of the layer. We can get that with `file.Popup.height`. Since the y position of the top of the screen is 0, we'll want to subtract the height from 0:

```
layerHeight = file.Popup.height

file.close.on Events.Click, ->
  file.popup.animate
    properties:
      opacity: 0
      y: 0 - layerHeight
    time: 0.4
```

We've saved the layer's height in a `layerHeight` variable so that the code is a bit easier to read.

To make the animation a bit more dynamic, we can add an "ease-in" curve to it.

```
file.close.on Events.Click, ->
  file.popup.animate
    properties:
      opacity: 0
      y: 0 - layerHeight
    time: 0.4
    curve: "ease-in"
```

Multiple animations

Once the modal is dismissed, our prototype currently just shows a mustache badge. It would be cool if that badge popped up from nowhere after you'd dismissed the mdoal.

To do an animation *after* another one, we have to "listen" for the end of the first animation. We can attach an `AnimationEnd` event listener to the popup layer (the one that is animating), and then do something else once it's finished animating:

```
(...same code as before)

file.popup.on Events.AnimationEnd, ->
  print "animation ended"
```

Now lets select the `mustache` layer and animate its size using the `scale` property:

```
file.popup.on Events.AnimationEnd, ->
  file.mustache.animate
    properties:
      scale: 2
```

Okay, but maybe we wanted the badge to appear from nothing. To do that, we have to initially set the badge to be teeny, and then animate it to a visible size:

```
file.mustache.scale = 0

file.popup.on Events.AnimationEnd, ->
  file.mustache.animate
    properties:
      scale: 1
```

To add a bit more life to this animation, we're going to make it look bouncy. To achieve a bounce effect on our animation, we can use one of the custom curve functions Framer comes with. The `spring()` function takes 4 arguments: tension, friction, velocity, and tolerance. Explaining all these properties is beyond the scope of this book, but we'll use a simple bounce using the settings `200, 15, 0`.

```
file.popup.on Events.AnimationEnd, ->
  file.mustache.animate
    properties:
      scale: 1
    curve:"spring(200,15,0)"
```

Example 2: Toggling between states

We're going to toggle a menu between closed and open states when an icon is clicked:

View finished animation

Import "example2_dropdown" psd or Sketch file into Framer.

Let's start off by hiding the menu content by default:

```
file = Framer.Importer.load "imported/example2_dropdown"

file.menu_content.opacity = 0
```

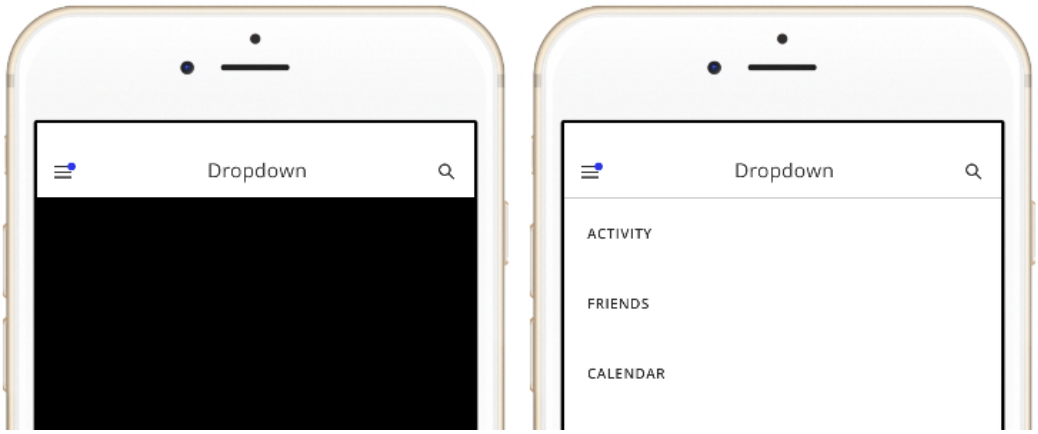


Figure 4: Dropdown menu

Now we'll add a click event listener on the menu icon, and then switch the opacity to 1:

```
file.menu_icon.on Events.Click, ->
  file.menu_content.opacity = 1
```

There's a problem though: we want the menu to close again when the menu icon is clicked again. Unfortunately, the menu won't know whether to open or close on each click unless we somehow keep track of what state it's already in.

To do this, we'll create a variable `is_open` that will be false if the menu is closed, and true if it's open. It'll default to false.

```
is_open = false
```

Next, we'll set the opacity in the click event based on our `is_open` variable:

```
file.menu_icon.on Events.Click, ->
  if is_open
    file.menu_content.opacity = 0
  else
    file.menu_content.opacity = 1
```

For this to work, we'll need to toggle `is_open` between `false` and `true` when the user clicks the icon. To toggle a value between true and false, we can

re-assign the variable to it's opposite. True and false are opposite of each other, so not true = false and not false = true. In CoffeeScript, we use the `!` symbol to mean "not": `true != false`.

To set a value to its opposite, we do `= !` or "set the value to *not* whatever it currently is"

```
is_open = !is_open
```

Let's put it all together:

```
is_open = false

file.menu_icon.on Events.Click, ->
  is_open = !is_open
  if is_open
    file.menu_content.opacity = 0
  else
    file.menu_content.opacity = 1
```

And now our menu toggles open when we click it.

Easier interactions with states

Framer gives us an easier way to transition between different states, called "states" turns out. Basically, you give a layer a set of named states which specify what it should look like when it is in that state. For example, our menu will have an "open" state where the opacity is 1, and a "closed" state where the opacity is 0.

To add states to our layer, we use the `states.add` method. Each state consists of a name and property pair, where the property contains the various options for the appearance:

```
file.menu_content.states.add
  open:
    opacity: 1
  closed:
```

```
    opacity: 0
```

Now we can switch between the two states in a few different ways. The easiest way to go back and forth between the two states is just by using `states.next()`. We can take out the `if else` statement now, as well as the `is_open` variable.

```
file.menu_content.states.add
  open:
    opacity: 1
  closed:
    opacity: 0

file.menu_icon.on Events.Click, ->
  file.menu_content.states.next()
```

By default, `states.next` animates between the two states. To customize this animation, we need to add and configure `states.animationOptions`:

```
file.menu_content.states.animationOptions =
  time: 0.2
```

One of the nice things about states is that it makes it easy to customize our animation and make it more complex. Instead of fading in, lets have our menu expand out from the top left.

To do that, we first need to set the menu's default width and height to 0.

```
file.menu_content.width = 0
file.menu_content.height = 0
```

And then update the states so that `open` resets the height and width to the original values, and `closed` sets them to 0. To get the original height and width values of our menu, we need to save those values as variables before we set them to 0.

```
original_width = file.menu_content.width
original_height = file.menu_content.height

```

```
file.menu_content.width = 0
file.menu_content.height = 0
```

And then use those variables in `states.add`:

```
file.menu_content.states.add
  open:
    width: original_width
    height: original_height
  closed:
    width: 0
    height: 0
```

This animation would look even better with some easing:

```
file.menu_content.states.animationOptions =
  time: 0.2
  curve: "ease-out"
```

Example 3: Touch interactions

Framer comes with a lot of useful utilites for easily prototyping touch-based interactions. We're going to prototype a swipe-based dismissal, like you'd have in a list view on a mobile app.

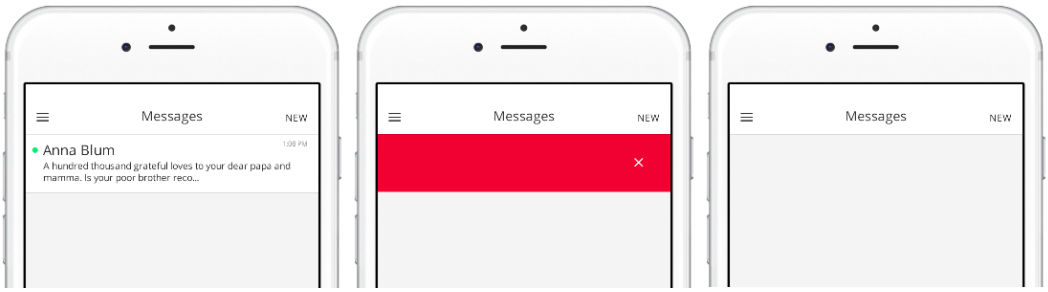


Figure 5: Swipe interaction

View completed animation

Import "swipe.psd" into framer.

One of the really convenient things Framer includes for touch interactivity is the ability to make a layer "draggable". To do this, we set `draggable.enabled` to true.

```
file = Framer.Importer.load "imported/example3_swipe"

file.message.draggable.enabled = true
```

We can now drag the layer all over the screen!

In our example, however, we want to restrict dragging to the x-axis. To do this, we set `draggable.speedY` to 0.

```
file.message.draggable.speedY = 0
```

Now we don't want the user to have to drag the message all the way off the screen, so we'll take over and animate the message off the screen if it's past a certain point.

A lot of the code for prototyping touch is going to be similar to this: doing actions based on how much an element has moved. We can do this by comparing the `x` property of the layer to either it's previous value or some absolute value based on the screen dimensions. We can listen for a number of different events, including `TouchStart`, `TouchMove` and `TouchEnd`. When you're working with draggable elements, you can use `DragStart`, `DragMove` and `DragEnd`.

In our message-dismissal example we're going to listen for the `DragEnd` event and then decide what to do.

```
file.message.on Events.DragEnd, ->
```

At this point, we have to come up with some rules for how to animate the message. In our case, the default result will be that the message snaps back to its starting position. If the message has been moved more than halfway off the left side of the screen, we want to animate it off the screen. There are a couple ways we can check for this, but I think the most intuitive is "when the midpoint of the message reaches the left edge of the screen." This is

easy to represent in code, since Framer gives us a convenient `midX` (and `midY`) property which returns the center point of the element.

```
if file.message.midX <= 0
```

So in this case, we want to animate the `x` property of our message to be all the way off the screen. To ensure it's all the way off the screen, we'll set the `x` value to 0 minus the width of the layer.

```
w = file.message.width

file.message.on Events.DragEnd, ->
  if file.message.midX <= 0
    file.message.animate
      properties:
        x: 0 - w
```

In other cases, we want to animate the message back to its original `x` position, which was 0.

```
file.message.on Events.DragEnd, ->
  if file.message.midX < 0
    file.message.animate
      properties:
        x: 0 - w
  else
    file.message.animate
      properties:
        x: 0
```

We can make the animations look a lot nicer with some easing and timing:

```
file.message.on Events.DragEnd, ->
  if file.message.midX < 0
    file.message.animate
      properties:
        x: 0 - w
      time: 0.1
      curve: "ease-in"
```

```
  else
    file.message.animate
      properties:
        x: 0
      time: 0.2
```

Bonus: animating the red "delete" bar after the message is dismissed.

This is pretty much the same as the code for animating in the badge after the popup is dismissed:

```
file.message.on Events.AnimationEnd, ->
  if file.message.midX < 0
    file.delete.animate
      properties:
        scale: .8
        opacity: 0
      time: 0.2
      curve: "ease-in"
```

Example 4: Generating elements with loops

If we want to deal with multiple elements of the same type, we can very quickly end up dealing with a lot of repetition. For example, if we wanted a series of squares in a row, we might do:

```
new Layer
  width: 100
  x: 0
new Layer
  width: 100
  x: 110
new Layer
  width: 100
  x: 220
new Layer
  width: 100
```

```
  x: 330
```

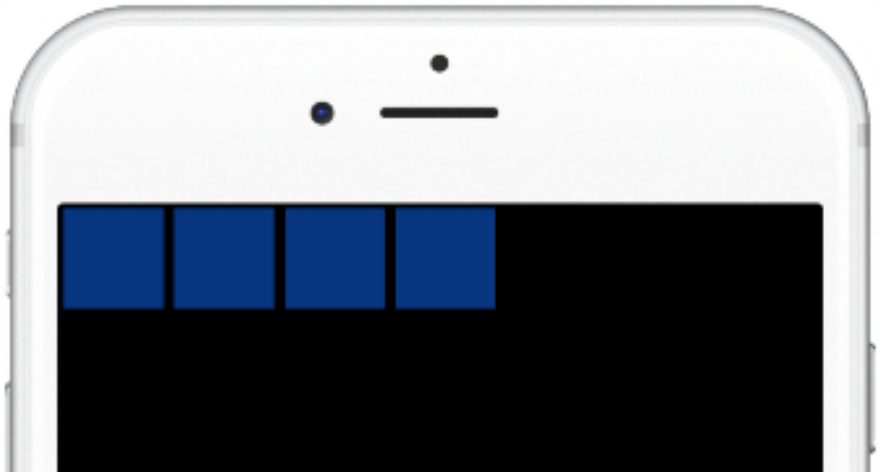


Figure 6: Multiple layers

There's a lot of repetition there, but there's an easy way to do this without the repeated code using *loops*.

Remember when we looped through all the fruits in an array?

```
fruits = ["apples", "oranges", "bananas"]

for fruit in fruits
  fruit.toUpperCase()

# => "APPLES"
# => "ORANGES"
# => "BANANAS"
```

One of the convenient things about CoffeeScript is that we can make a new array and loop through it all in one line:

```
for fruit in ["apples", "oranges", "bananas"]
  print fruit

# => "apples"
```

```
# => "oranges"
# => "bananas"
```

If we don't really need an array of *things* but just want to do something *x* number of times, we can use a shortcut to make an array of x items:

```
print [1..5]

# => [1,2,3,4,5]
```

So if we want to just do something 5 times:

```
for i in [1..5]
  print i

# => 1
# => 2
# => 3
# => 4
# => 5
```

This is a fairly common pattern when you're prototyping lots of elements in CoffeeScript. It's a bit of a convention to use `i` for the variable that gets re-assigned for each time through the loop (like in `for fruit in fruits`, `fruit` got re-assigned to "apple", "orange", "banana" each time through the loop). `i` as in "iterator" or "index".

So again, if we wanted to create 4 squares, we can do it much more easily with a loop:

```
for i in [0..3]
  new Layer
    width: 100
```

This will just stack all 4 squares on top of each other:

We want to set the `x` values for the squares to 0, 110, 220, 330, respectively. Conveniently, these are all multiples of our index (0 * 110, 1 * 110, 2 * 110, 3 * 110).

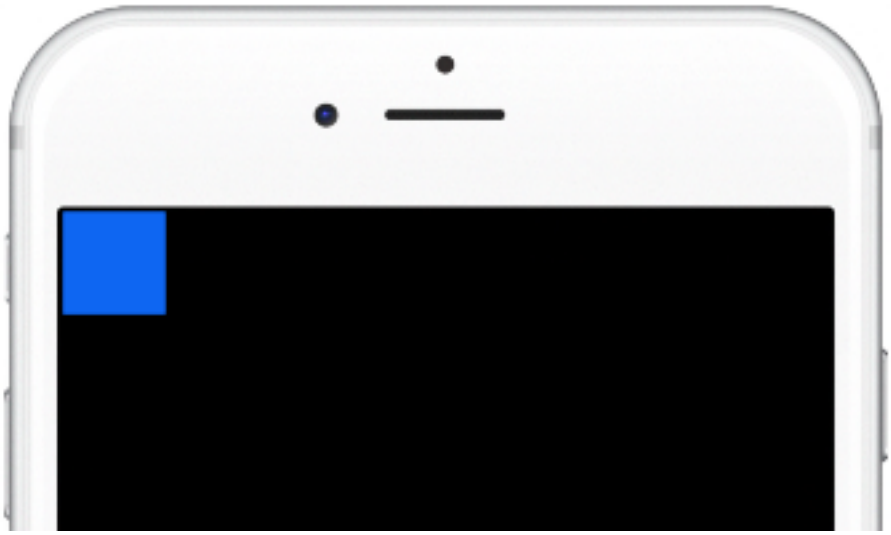


Figure 7: Stacked layers

```
for i in [0..3]
  new Layer
    width: 100
    x: i * 110
```

.

Fun with loops

Let's do something more fun with loops. We'll make something similar to the way the cards stack in the iOS Passbook app:

Figure 8: Stacked cards

And we'll have the cards animate in nicely. View the example of the finished prototype.

```
for i in [0..4]
  layer = new Layer
    width: Framer.Device.screen.width
```

```
    height: Framer.Device.screen.height
    y: 100 * i
    borderRadius: 50
```

We've set each layer to be the full width and height of the screen, and offset them by 100px from the top (to get them to stack like this, we multiply 100 * the index of the loop).

If we want the cards to animate in from the bottom of the screen, we have to start with the cards being way down below the bottom of the screen. Let's add the height of the screen to the `y` offset of each layer:

```
for i in [0..4]
  layer = new Layer
    width: Framer.Device.screen.width
    height: Framer.Device.screen.height
    y: 100 * i + Framer.Device.screen.height
```

Now we'll animate the `y` property to what it was originally:

```
for i in [0..4]
  layer = new Layer
    width: Framer.Device.screen.width
    height: Framer.Device.screen.height
    y: 100 * i + Framer.Device.screen.height
    borderRadius: 50

  layer.animate
    properties:
      y: 100 * i
```

Note that we're still indented in a level, so that we're still inside the loop, where `layer` is assigned to the current layer each time through the loop.

Now all the cards animate in at the same time, which isn't quite what we want. If we set a delay on the animation, they'll all slide in together after the delay. We have to increase the delay for each time through the loop:

```
layer.animate
```

```
  properties:
    y: 100 * i
  delay: i
```

Now the delay will be 0, the first time through the loop, 1 second the second time, 2 the third etc. To reduce it, we can multiply by `i` by whatever we want the delay to be between each item:

```
layer.animate
  properties:
    y: 100 * i
  delay: i * 0.2
```

And then match the length of the animation to that delay:

```
layer.animate
  properties:
    y: 100 * i
  delay: i * 0.2
  time: 0.2
```

It still doesn't look great, but adding a spring curve will make a *huge* difference:

```
layer.animate
  properties:
    y: 100 * i
  delay: i * 0.2
  time: .2
  curve:"spring(200,30)"
```

Mapping values to arrays

Right now, our cards are all the same colour (slightly transparent blue), they just look like different shades because they're stacked on top of each other. If we wanted to make them different colors, we can store a bunch of colors in an array, and then use those colors for the background colors of the cards.

Here's a nice array of nice colors:

```
colors = ["#f1c40f", "#2ecc71", "#1abc9c", "#3498db", "#9
   b59b6"]
```

And then we can access the elements in the array one at a time using `i`:

```
for i in [0..4]
  layer = new Layer
    backgroundColor: colors[i]
```

So the first time through the loop, we'll be setting `backgroundColor` to `colors[0]`, which is the first item, which is `#f1c40f` (yellow). Second time through the array `i` is 1, so we'll be grabbing `colors[1]`, which is the greenish color. And so on.

Example 5: Multi-part animations

Let's try something a little more complicated: a multi-step animation with some interactivity and multiple moving parts.

We're going to prototype a push notification on the Apple Watch, like you would get from a calendar notification.

View completed animation

Import the example4_icon.png (or make your own icon) file into Framer, then example4_bg.png. Set the device to Apple watch in the 42mm size.

```
bg = new Layer
  x:0, y:0, width:312, height:366, image:"images/example4_bg
   .png"

icon = new Layer
  x:0, y:0, width:196, height:196, image:"images/
   example4_icon.png"
```

We're going to need to reference the width and height of the device, so we'll save those in variables called `w` and `h`.

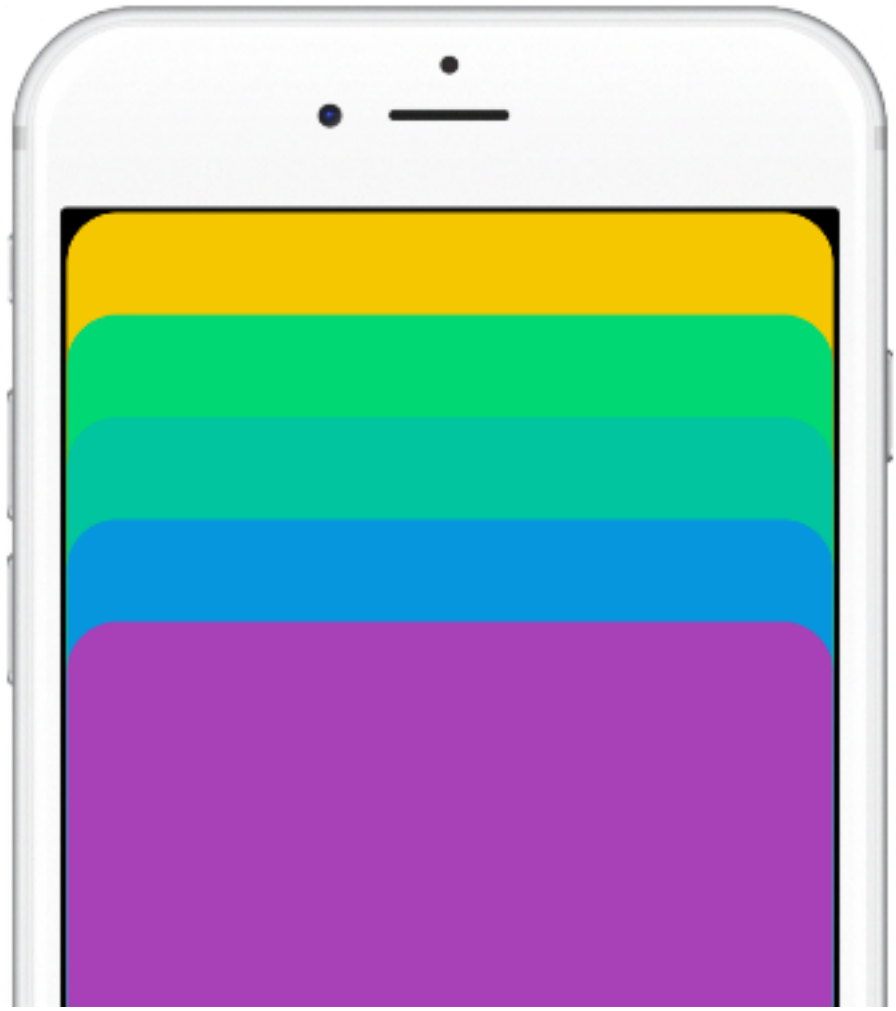


Figure 9: Colored cards

Figure 10: Apple watch notification

```
w = Framer.Device.screen.width
h = Framer.Device.screen.height
```

We're going to set the initial state of the icon as being horizontally centered and positioned just below the bottom of the screen:

```
icon.centerX()
icon.y = h
```

`centerX()` is a convenient method from Framer that horizontally centers our layer. There's also `centerY()` and just `center()`.

Multipart Animation step 1

The first step of our animation involves two transitions:

1. sliding the icon up over the background
2. blurring the background.

Let's start animating the icon to the center of the screen:

```
icon.animate
  properties:
    midY: h / 2
```

We're setting the layer's `midY` property to half the height of the device, because if we just set `y` property, it would position the top edge of the layer. That would put our icon on the bottom half of the screen instead of at the midpoint. Each layer also has a `midX` property for setting the horizontal center of a layer.

At the same time as we animate the icon into position, we're going to both blur and fade the background a bit:

```
bg.animate
  properties:
    blur: 15
    opacity: 0.6
```

The `blur` property is set in pixels, so you can copy it right out of the Gaussian blur in Photoshop or Sketch.

Let's speed up the animation and add an Apple-style springy curve:

```
icon.animate
  properties:
    midY: h / 2
  time: 0.5
  curve: "spring(120,18,0)"

bg.animate
  properties:
    blur: 15
    opacity: 0.6
  time: 0.5
```

Multipart Animation step 2

The next step involves another two transitions:

1. move the icon to the top left of the screen and shrink it
2. slide in the content of the notification

Figure 11: Apple watch icon

To initiate this set of animations *after* the first set have finished, we'll listen for the `AnimationEnd` event on the icon:

```
icon.on Events.AnimationEnd, ->
```

Let's start with shrinking the icon:

```
icon.on Events.AnimationEnd, ->
  icon.animate
    properties:
      scale: .5
```

At the same time, we'll move it to the top left of the screen. Unfortunately, we can't just move the icon to `x: 0` and `y: 0`, because when we used `scale` to shrink the icon, the icon's bounding box didn't shrink at the same time, so the icon would be positioned too far from the edges. We'll have to adjust for that by subtracting one-half the icon's *new* width from the `x` and `y` values.

The icon was originally 196px, so it's 50% scaled size is 98px, so we'll offset x and y by 49px.

```
icon.on Events.AnimationEnd, ->
  icon.animate
    properties:
      scale: .5
      x: -49
      y: -49
```

At the same time, let's animate in the notification content. Import "watchapp.psd" at the top of the file:

```
bg = new Layer
  x:0, y:0, width:312, height:366, image:"images/example4_bg
   .png"

watch_file = Framer.Importer.load "imported/watchapp"

icon = new Layer
```

```
  x:0, y:0, width:196, height:196, image:"images/
   example4_icon.png"
```

Note the order in which we're importing the files: this layers the files in the correct order. We could explicitly set the `z-index` values of each layer using the `index` property, but this is simpler.

Let's set the notification layer's initial position below the bottom of the screen:

```
notification = watch_file.notification
notification.y = h
```

Now we'll animate it into the scene at the same time as we move the icon to the top right and shrink it:

```
icon.on Events.AnimationEnd, ->
  icon.animate
    # code from earlier goes here
  notification.animate
    properties:
      y: 49
```

We're setting `y` to 49 because it will align with the midpoint of the icon. Now that the notification is aligned with the icon, it looks like the icon is a bit too close to the edge of the screen. Let's push it over by 20px:

```
icon.on Events.AnimationEnd, ->
  icon.animate
    properties:
      scale: .5
      x: -29
      y: -49
  notification.animate
    properties:
      y: 49
```

And speed up the animation and add a spring curve:

```
  icon.animate
    properties:
      scale: .5
      x: -29
      y: -49
    time: .3
    curve: "spring(320,26,0)"
  notification.animate
    properties:
      y: 49
    time: .3
    curve: "spring(320,26,0)"
```

Multipart animation part 3

The last step is to dismiss the notification panel when the button is pressed. We'll find the `button` layer and add a `Click` event listener:

```
button = watch_file.button

button.on Events.Click, ->
```

There are a couple things we need to do now to get back to our initial state:

1. fade out the notification content
2. fade out the icon
3. un-blur and un-fade the background

This is mostly code we've seen before by this point:

```
button.on Events.Click, ->
  notification.animate
    properties:
      opacity: 0
  icon.animate
```

Figure 12: Apple watch notification

```
    properties:
      opacity: 0
  bg.animate
    properties:
      opacity: 1
      blur: 0
```

And then we'll speed all the animations up:

```
button.on Events.Click, ->
  notification.animate
    properties:
      opacity: 0
    time: .3
  icon.animate
    properties:
      opacity: 0
    time: .3
  bg.animate
    properties:
      opacity: 1
      blur: 0
    time: .3
```

And now we're back to where we started.

Next Steps

I think that with the building blocks from the last few examples you should be able to prototype a pretty wide variety of interactions and animations using Framer. I've intentionally stayed away from discussing topics such as code organization and best practices because I find these concepts unnecessary for beginners and in the context of building prototypes to communicate experiences.

If you're taking prototyping with CoffeeScript seriously, I'd strongly recommend learning some JavaScript fundamentals. Unfortunately, there are few CoffeeScript resources aimed at beginner developers, since most CoffeeScript developers come from the background of already knowing at least some JavaScript.

For JavaScript (and general programming) fundamentals, I suggest working through the Codeacademy JavaScript track. It's an interactive set of tutorials that let you program in the browser and provides feedback on your code, and it's totally free.

If you're looking for some more in-depth resources, Codeschool has excellent JavaScript and CoffeeScript courses which feature excellent video tutorials and interactive challenges.

If you're enjoying programming and want to dig a bit deeper into the fundamentals (using CoffeeScript), read Reginald Braithwaite's CoffeeScript Ristretto, which starts at the very beginning of programming with functions and gets into some pretty advanced concepts, using CoffeeScript for all code examples.

If you'd like to take your prototyping skills out of the Framer environment so you can make prototypes or production code for any website, I'd still recommend leveraging a library to help out with the animations. Some suggestions:

- jQuery takes a lot of the pain out of interacting with the native browser environment. Outside of Framer, you might find that working with elements on a page can be a bit complex and verbose. jQuery can help ease that pain, and has a large plugin ecosystem that can give you a lot of extra functionality with very little code. You can accomplish a lot with some basic JavaScript knowledge and jQuery.
- Move.js is a small and easy-to-use library for making CSS-based animations simpler. CSS-based animations are quickly becoming the standard because of their flexibility and performance, and you likely already know a lot of the syntax if you know some CSS.

- AnimateCSS also leverages CSS for animations, and lets you write minimal JavaScript for your animations by moving it all to pre-written CSS.
- Snap.svg is a great library for animating vector graphics in the browser, which are resolution-independent and more flexible than the boxes and circles you can make with regular browser elements.

Getting help

First of all, if you work with developers, they should be the first people you ask for help. You'll find that a lot of developers love sharing their knowledge and are generally excited when other members of their team take an interest in code. It doesn't matter if your coworkers have never seen CoffeeScript before; the concepts are similar accross all programming languages and they'll probably be able to help anyways.

If you're going it alone, there is a great and growing community behind Framer, primarily congregating on the Facebook page, where people share tips and resources and ask and answer questions.

For more programming-related questions, any developer will tell you that Stack Overflow is one of the most valuable resources out there. Checkout the #CoffeeScript tag to see if your question has been asked before, or ask a new question. You'll probably get an answer quickly, especially if your question is specific, clear, and includes a code example.

When you're Googling around for answers to your questions, seek out answers from reputable sources like the Mozilla Developer Network.

Acknowledgements

Thanks to ___ and ____ for technical review and proofreading assistance.

If you find a typo or mistake

Please report an issue on the Github repo or shoot me a memo on Twitter.

About the Author

Tessa Thornton is a JavaScript and CoffeeScript developer at Shopify in Toronto, Canada. She is an experienced developer, technical writer, and has been using CoffeeScript since before it got cool and then not cool and then cool again.

She has a BA in Philosophy and Anthropology, which she puts to good use getting offended by strangers on the internet, and occasionally even writing essays.

Contact

Twitter: tessthornton

Github: tessalt