DEVCON

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Table of Contents: Overview

Table of Contents: Extended

8: The Art of the Chart

When you're asked to include charts or graphs in an app, don't panic and reach for a third-party library. In this session you'll learn how to make your own fancy-looking data visualisations, with animations and color effects as a bonus!

22 The Art of the Chart: Demo 1

By Rich Turton

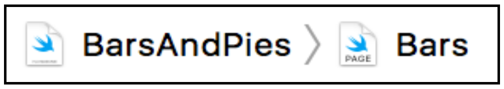
In this demo, you will learn how to make super-quick bar charts and pie charts!

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

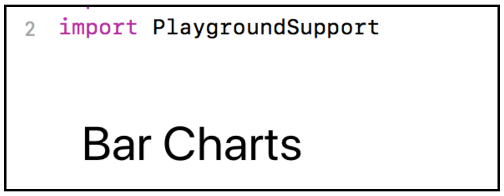
Note: Begin work with **BarsAndPies.playground** in **Demo1\starter**.

1) Check your playground

Make sure you are on the **Bars** page by using the jump bar:



Make sure that the playground looks its best. The top of the page should be in a large, bold font, not the standard code font:



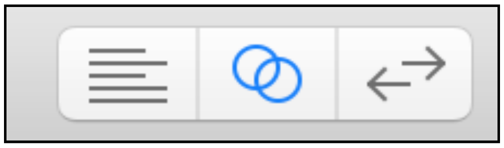
```
import PlaygroundSupport
```

Bar Charts

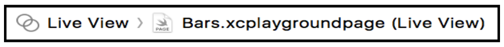
If it isn't, choose **Editor/Show Rendered Markup** from the menu bar.

You're going to be using live views in the playground and they can sometimes be a little tricky.

Make sure the assistant editor is open (command-option-enter will also do this):



The assistant editor should be showing the **Live View**, use the jump bar to change it if not:



Finally, you should see a boring-looking gray box in the live view. If you can see that, you're ready!

2) Add a stack view

The gray box is going to be the chart area you'll draw in.

You can see there is some code already in the playground - this is just setting up the views and creating some sample data, since that's a lot of boring typing that you don't have time to do. Your stack view will represent a simple struct, which has a numeric value and a color.

Find the comment that says `// Add a stack view` near the bottom of the playground. Underneath that, enter the following code:

```
let stack = UIStackView(frame: chart.bounds)
chart.addSubview(stack)
stack.axis = .horizontal
stack.alignment = .fill
stack.distribution = .fillEqually
stack.spacing = 5
```

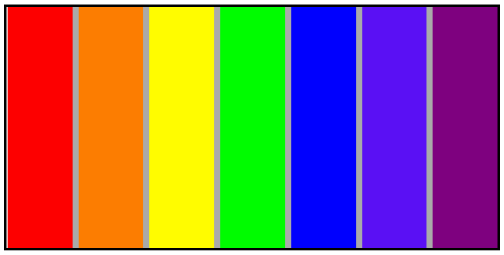
This creates a stack view that's the same size as the gray box and sets up some parameters for it.

3) Add some bars

Next, add the following:

```
for point in dataPoints {
  let bar = UIView()
  bar.translatesAutoresizingMaskIntoConstraints = false
  bar.backgroundColor = point.color
  stack.addArrangedSubview(bar)
}
```

This adds a view to the stack view to represent each value in the model. When the assistant editor updates you will see a lovely rainbow:



You can see that the stack view has taken care of the width of the bars and the spacing between them for you, but you still have some work to do on the height.

4) Size your bars

Go back to where you created the stack view and change the `alignment` to `bottom`:

```
stack.alignment = .bottom
```

Your bars will all disappear, because the stack view has no reason to give them anything other than zero height. You need to add some constraints.

Above the for loop, create an array for your constraints:

```
var heights = [NSLayoutConstraint]()
```

Inside the for loop, after you add the bar to the stack view, add the following code:

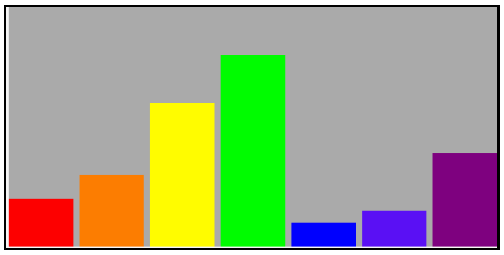
```
let proportion = point.value / chart.bounds.height
heights.append(
  bar.heightAnchor.constraint(equalTo: chart.heightAnchor,
  multiplier: proportion)
)
```

Nothing will happen; all you've done at this stage is create some constraints.

At the very end of the playground, add the following to activate them:

```
NSLayoutConstraint.activate(heights)
```

Hey presto, a bar chart!



5) Gratuitous animation

The fun part about building charts out of views like this is that you get to use all the features of views to make it look fancy. Let's bounce the values into existence. Add this code right at the end of the playground:

```
UIView.animate(
  withDuration: 1.5,
  delay: 1,
  usingSpringWithDamping: 0.5,
  initialSpringVelocity: 20,
  options: [],
  animations: { chart.layoutIfNeeded() },
  completion: nil
)
```

Boing! If you want to see your animation again (of *course* you want to see your animation again!), you can stop and re-start the playground using the control in the bottom toolbar.

6) Time for some pie

Using the jump bar, switch to the **Pies** page of the playground. Your assistant editor should go back to a boring gray square.

There is a `PieChart` class in the playground but the implementation is currently rather minimal.

You're going to implement the slices of pie by using `CAShapeLayers`. Each shape layer will represent a single slice of the pie.

Inside the `PieChart` class add a property to hold all the slices:

```
var slices = [CAShapeLayer]()
```

7) Slice it up

Now add a method that will take an array of `ChartData` and set up the layers:

```
func createSlicesFor(_ data: [ChartData]) {
  slices.forEach { $0.removeFromSuperlayer() }
  let total: CGFloat = data.reduce(0) { $0 + $1.value }
}
```

First you remove any existing slices, then you need to calculate the total value, since a pie chart is about representing fractions of a total. Still in `createSlicesFor(_:)`, Add the following code:

```
let center = CGPoint(x: bounds.midX, y: bounds.midY)
let radius = bounds.height * 0.25
let lineWidth = bounds.height * 0.5
```

This is the calculation mentioned earlier. Each slice will be represented by a fraction of a circle called an *arc*. The circle's center is the center of the pie. The radius of the circle is half of the radius of the pie, and the width used to draw the line will be the radius of the pie. This means the stroked arc will fill the whole circle.

> **Note:** You're about do do some calculations with angles. You're probably used to measuring angles in *degrees*, where there are 360 of them in a circle. Core graphics uses *radians*, which work much better with a lot of angle-based calculations. There are `2 * pi` radians in a circle. Pi is 3.14159 and a bit, but you don't need to worry about that. What's useful to remember is that you can think of your circle in multiples of Pi - Pi is half a circle, Pi / 2 is a quarter, and so on.

It's time to slice up your pie. Add the following code:

```
var runningTotal: CGFloat = 0
//1
slices = data.map {
  //2
  runningTotal += $0.value
  let fraction = runningTotal / total
  //3
  let endAngle = .pi * 2 * fraction
  //4
  let path = UIBezierPath(arcCenter: center, radius: radius, startAngle:
0, endAngle: endAngle, clockwise: true)
  //5
  let slice = CAShapeLayer()
  slice.frame = layer.bounds
  slice.strokeColor = $0.color.cgColor
  slice.fillColor = nil
  slice.lineWidth = lineWidth
  slice.path = path.cgPath
  //6
  layer.insertSublayer(slice, at: 0)
  return slice
}
```

Here's what's happening:

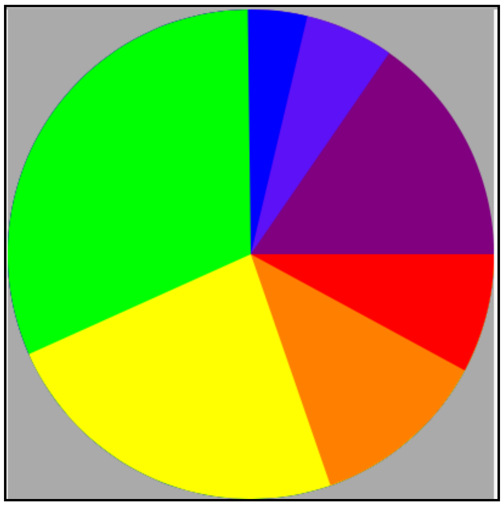
1. You want one slice per item in the data array, so `map` is the right function to use here.
2. It's simplest to make each slice start from the start and go all the way round to its end instead of calculating the start and end angle for each one, so you calculate a running total.

3. There are 2 pi radians in a full circle, so this line calculates the number of radians that the running total should take up.
4. Here you create a path which contains an arc starting from zero, using the center and radius you calculated earlier.
5. This code creates the shapelayer for the slice and configures it with the color from the data item and the path you just calculated.
6. Because the slice starts from zero, you insert it at the bottom so that it doesn't cover up the earlier slices.

Finally, add the following code towards the end of the playground, where it says "Populate your pie data here!":

```
pie.createSlicesFor(dataPoints)
```

Your pie will now look like this:



That's not quite right. Rainbows start at red, but the red slice is at East when it should be at North.

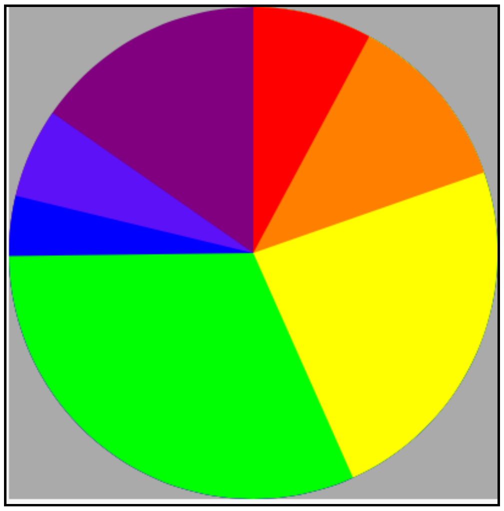
8) Straightening your pie

Your pie is sideways because `UIBezierPath` considers zero radians to be pointing due East. You could amend the code where you set the start and end angles, but doing that would make your code more complicated and harder to understand. I prefer to apply a transform to each slice instead. Layers can have transforms applied to them just like views. It's easier to understand if you try and keep your paths as close to the raw data values as possible, and then use transforms to shape them to your needs. You'll see more examples of this in later demos.

For now, add the following code just before the line where you add the slice as a sublayer:

```
slice.setAffineTransform(CGAffineTransform(rotationAngle: -.pi * 0.5))
```

Remember you learned earlier that half of Pi is quarter of a circle. That should straighten out your pie:



9) More gratuitous animation

Shape layers have *great* animation capabilities. You can animate between different paths, which you'll look at later on, or you can animate the drawing of the path itself. In this section you're going to make your pie fill itself up.

Add a new method to `PieChart`:

```
func fill(to proportion: CGFloat, animated: Bool = false) {
  let animation = CABasicAnimation(keyPath: "strokeEnd")
  animation.duration = 0.5
  animation.toValue = proportion
  slices.forEach {
    slice in
    animation.fromValue = slice.strokeEnd

    if animated {
      slice.add(animation, forKey: nil)
    }
    slice.strokeEnd = proportion
  }
}
```

This method will take the proportion you pass in and use it to animate the `strokeEnd` property of each shape layer. This value can be between 0 and 1 and represents the proportion of the layer's path that will actually be drawn on screen.

At the end of the playground, add these two lines:

```
pie.fill(to: 0)
pie.fill(to: 1, animated: true)
```

Watch as your pie fills itself in! Because you've started each slice at the same point, the effect looks really nice. To see the effect again, stop and re-start the playground using the control in the bottom toolbar.

10) That's it!

Well done! You've learned how to draw two different types of chart with very little code. In the next section, you're going to look at line graphs.

The Art of the Chart: Demo 2

By Rich Turton

In this demo, you will learn how to calculate an axis and tick marks, and how to use that to create a line graph.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the **ChartSpace.playground** starter project in **Demo2\starter**.

1) Calculating ticks

Open the playground and make sure you are on the **Axes** page. The playground contains information about two `struct` types which form the basis of the axis model. A `Tick` represents a single tick mark, with a label value, a numeric value, and a relative position on the axis. An `Axis` is represented by a collection of ticks and a closed range which tells you the minimum and maximum values that the axis represents. Both of these structs are implemented in files in the **Sources** folder.

Any time you're dealing with passing round a min and max value, always use a range. It's one value instead of two, and there are several convenience methods you can define to make your charting work easier. Calculating the relevant ticks to cover a range is implemented as an extension on `ClosedRange<CGFloat>`.

We're not going to use a data set in this playground. Let's assume instead that you've already worked out the smallest and largest values you want your chart to cover. Represent this as a range by adding the following code to the playground:

```swift
let minValue: CGFloat = 0
let maxValue: CGFloat = 1000
let range = minValue...maxValue
```

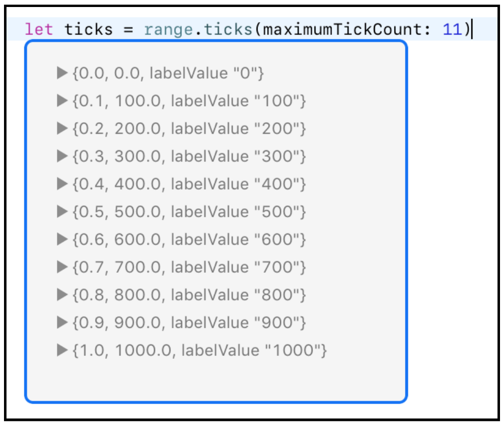
This is a *closed* range, so the lower bound is zero and the upper bound is 1000.

Now use the extension which will generate ticks from the range:

```
let ticks = range.ticks(maximumTickCount: 11)
```

Add a visualiser to see what that has given you; resize it to see more details.

Play around with the min and max values of the range and the tick count to see how different sets of ticks are calculated.



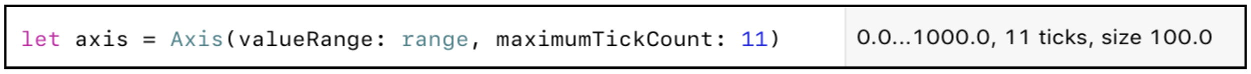
```
let ticks = range.ticks(maximumTickCount: 11)
```

```
▶ {0.0, 0.0, labelValue "0"}
▶ {0.1, 100.0, labelValue "100"}
▶ {0.2, 200.0, labelValue "200"}
▶ {0.3, 300.0, labelValue "300"}
▶ {0.4, 400.0, labelValue "400"}
▶ {0.5, 500.0, labelValue "500"}
▶ {0.6, 600.0, labelValue "600"}
▶ {0.7, 700.0, labelValue "700"}
▶ {0.8, 800.0, labelValue "800"}
▶ {0.9, 900.0, labelValue "900"}
▶ {1.0, 1000.0, labelValue "1000"}
```

2) Making an Axis

You're unlikely to make ticks without also making an axis, so an initializer for `Axis` has been added as well. Try it out by adding the following code to the bottom of the playground:

```
let axis = Axis(valueRange: range, maximumTickCount: 11)
```



```
let axis = Axis(valueRange: range, maximumTickCount: 11)
```

```
0.0...1000.0, 11 ticks, size 100.0
```

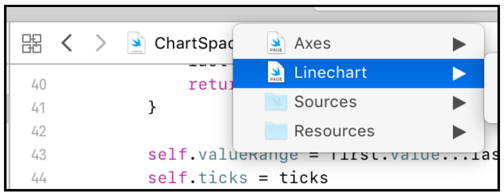
That's the model layer for your chart axes taken care of! Move to the **LineChart** page of the playground for the next section.

3) Some handwaving

In this page of the playground there's quite a bit of code hidden in the **Sources** folder. Mostly it's a lot of autolayout stuff which isn't very fun to type and isn't directly pertinent to what I want to discuss, but feel free to look at it later.

4) Make some data

Switch over to the **LineChart** page by using the jump bar:



```
        retu
    }

    self.valueRange = first.value...las
    self.ticks = ticks
```

First of all, you can't plot a graph without some points. Add the following line:

```
let points = randomPoints(100)
```

This function will give you a set number of random points, starting at (0,0). X will be regular, and Y will wobble up and down, but never below zero. It's an array of `CGPoint` values which is something you could reasonably expect to generate from an API or data model.

Now convert these points into a path:

```
let path = CGMutablePath()
path.addLines(between: points)
```

You're using `CGMutablePath` here rather than `UIBezierPath` for two reasons - you'll be dealing with `CAShapeLayer`, so it's a natural companion, and there are some handy functions like the one above to generate a path from an array of points. This function is incredibly fast - I've used it with tens of thousands of data points and never seen any slowdown.

Now, you want to generate some axes from these points. You know that X goes from 0 to 100, but what about Y? It's random. Here's `CGPath` to the rescue again:

```
let maxY = path.boundingBox.maxY
```

`boundingBox` gives you a `CGRect` which encompasses the entire path, so the max Y is the largest value in the Y range. Now create your axes:

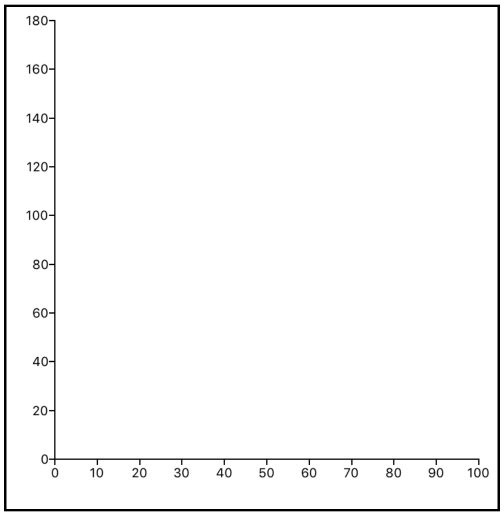
```
let xAxis = Axis(valueRange: 0...100, maximumTickCount: 11)!
let yAxis = Axis(valueRange: 0...maxY, maximumTickCount: 11)!
```

5) Make a chart

I've added a class called `ChartView` to the **Sources** folder for this playground. It's mainly here to help us visualise the important part of this talk, which is to convert between data space and chart space. All you need to know at this point is that it's a view, with axes and tick markers, and it has a subview which is used for plotting. Set one up like this:

```
let chart = ChartView(xAxis: xAxis, yAxis: yAxis)
chart.bounds = CGRect(x:0, y:0, width: 400, height: 400)
chart.layoutIfNeeded()
PlaygroundPage.current.liveView = chart
```

You should see an empty chart in the assistant editor. The axis labels will match up to the data you've generated - but the Y labels will change every time you update the code because random data is regenerated.

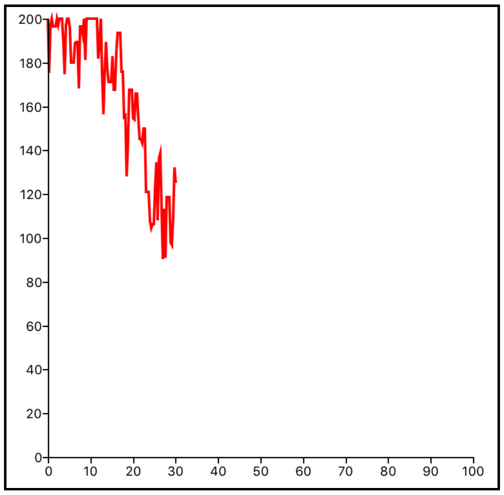


Get a reference to the plotting area of the chart. This is just a `UIView` in the area surrounded by the axes:

```
let plotArea = chart.plotArea
```

You'll be adding your data lines to this view. Create and add a shape layer:

```
let layer = CAShapeLayer()
layer.frame = plotArea.bounds
plotArea.layer.addSublayer(layer)
layer.strokeColor = UIColor.red.cgColor
layer.fillColor = nil
layer.lineWidth = 2
layer.path = path
```



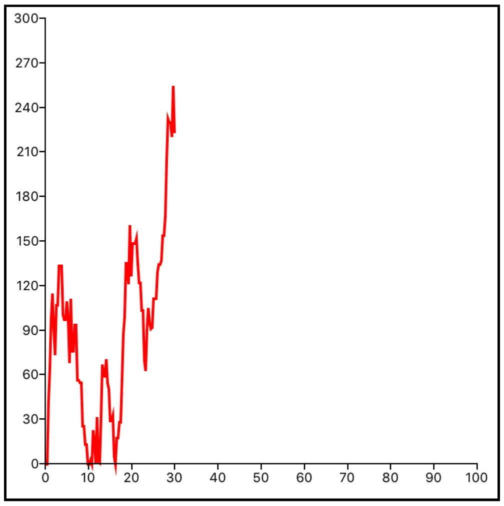
Well, that doesn't look right, does it? There are two problems:

1. The line is supposed to start at the lower-left corner, at zero-zero
2. The line doesn't fill the chart like you'd expect it to.

The first problem is to do with layer geometry. In views and layers in iOS, y = 0 is at the *top* of the view, and we want it to be at the bottom. Luckily, there's an easy fix for this. Add the following line:

```
plotArea.layer.isGeometryFlipped = true
```

That does exactly what you'd expect it to do, and you should see the results immediately - your line is now starting at the bottom left.

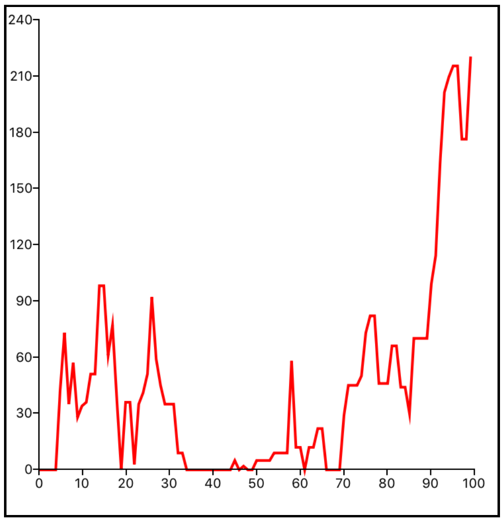


The second problem is to do with converting between data space and chart space. You need to work out a transform to move between the two coordinate systems. Add the following code just **before** you create the shape layer:

```
let xRatio = plotArea.bounds.width / xAxis.valueRange.size
let yRatio = plotArea.bounds.height / yAxis.valueRange.size
var scale = CGAffineTransform(scaleX: xRatio, y: yRatio)
```

This transform will convert between the two. To use the transform, you don't apply it to the layer, because that would also affect the drawing of the line, so it wouldn't be 2 points thick. You apply it to the *path*. Because the path is a useful model object in data space, you don't want to manipulate it. Amend the line where you assign the path to the layer:

```
layer.path = path.copy(using: &scale)
```



This scales the path at the point where you pass it off to the layer, giving you a neat boundary between data space and chart space. Again, you really don't need to worry about performance here - this thing is super fast.

6) Sanity checking

This is all very well, but you're looking at random data - how do you *know* it all lines up? Add some boring data as well. Back up to where you create `path`, and before you calculate the max Y, and add the following:

```
let boringPoints = uniformPoints(100)
let boringPath = CGMutablePath()
boringPath.addLines(between: boringPoints)
```

This function will just create a straight line where x = y the whole way. Boring, but useful.

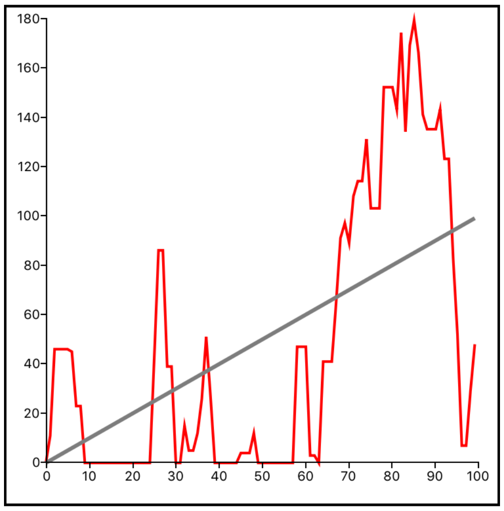
Amend the line where `maxY` is calculated:

```
let maxY = max(path.boundingBox.maxY, boringPath.boundingBox.maxY)
```

You do this because it's possible for the boring data set to be bigger than the random data set. Now move down to the end of the playground and add the boring data:

```
let boringLayer = CAShapeLayer()
boringLayer.frame = plotArea.bounds
plotArea.layer.addSublayer(boringLayer)
boringLayer.strokeColor = UIColor.gray.cgColor
boringLayer.fillColor = nil
boringLayer.lineWidth = 3
boringLayer.path = boringPath.copy(using: &scale)
```

You will see a line whose values you know, and it will be in the place you expect! The system works!!



7) That's it!

Congrats, at this time you should have a good understanding of the mental acrobatics needed to convert between data space and chart space! Take a break, and then it's time to move onto making things *fabulous!*

24 The Art of the Chart: Demo 3

By Rich Turton

In this demo, you will learn how to use different `CALayer` features to make your line graphs look great.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with **Fabulous.playground** in **Demo3\starter**.

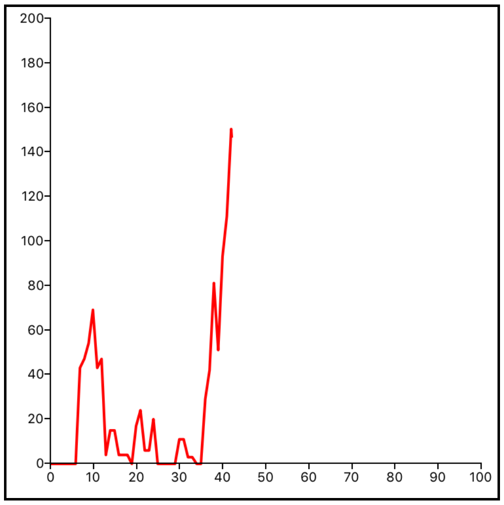
1) Animating lines

Make sure you are on the **Animate Lines** playground page, with the assistant editor open in live view mode. You should see a similar line graph to the ones you were making in the previous section. A lot of the code you worked on last time has been moved out of the way to make it easier to focus on the content for this demo.

First, you're going to add an animation to make the line draw itself. Add the following code at the end of the playground, where it says **Start your demo coding here!**:

```
let strokeEnd = CABasicAnimation(keyPath: "strokeEnd")
strokeEnd.fromValue = 0
strokeEnd.toValue = 1
strokeEnd.duration = 1

layer.add(strokeEnd, forKey: nil)
```

This code animates the `CAShapeLayer` property `strokeEnd`, from a value of zero, meaning none of the path is drawn, to a value of 1, meaning the whole path is drawn.



That's kind of fun, but you can do better than that. Find the comment saying **Add an extra path here!**. After that line, add the following:

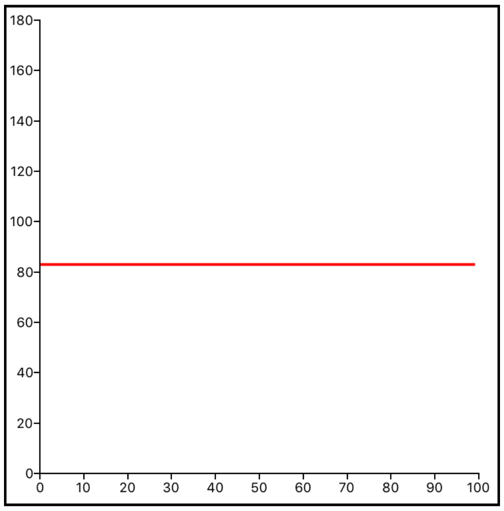
```
let midY = path.boundingBox.midY
let midPoints = points.map { CGPoint(x: $0.x, y: midY) }
let midPath = CGMutablePath()
midPath.addLines(between: midPoints)
```

Here, you're finding the middle Y value of your data, and then making a new path, with the same number of points as the data path, but with the same y value throughout.

Find the line where the `layer.path` is set, and change it to use `midPath` instead of `path`:

```
layer.path = midPath.copy(using: &scale)
```

The chart will now animate out along a straight line, which is the first stage of the animation you're aiming for.



When the stroke end animation completes, you want to add another animation. `CAAnimation` objects don't have a completion block, but you can wrap them in a `CATransaction`, which does.

Just above the line where the `strokeEnd` animation is added to the layer, add the following code to start a transaction:

```
CATransaction.begin()
```

This line begins a transaction. Everything that happens now until the transaction is committed will be part of the transaction. Add the following line at the end of the playground to commit the transaction:

```
CATransaction.commit()
```

So far, so good. The animation should still be looking exactly the same. Now, add a completion block. You have to add the completion block before you add any animations, because it will only wait for animations added to the transaction after it is set. Add the following, immediately *after* the line where the transaction is begun:

```
CATransaction.setCompletionBlock {
 print("Finished!")
}
```

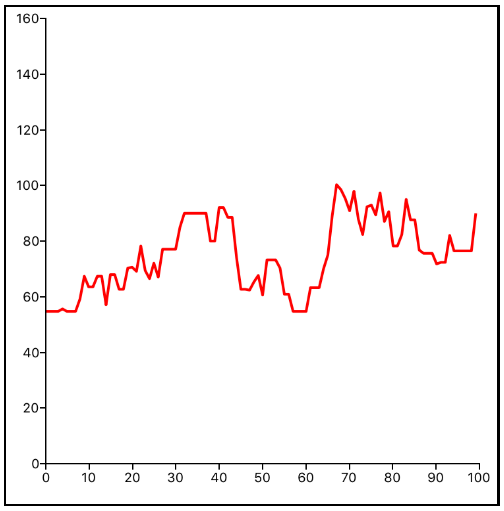
Run the playground again, and you will see that the "Finished" message doesn't appear in the results sidebar or the console until the stroke animation has completed.

Now you have a working completion block, you can do something useful in it. Add the following code after the `print` statement:

```
let newPath = CABasicAnimation(keyPath: "path")
let scaledPath = path.copy(using: &scale)
newPath.fromValue = layer.path
newPath.duration = 1
layer.add(newPath, forKey: nil)
layer.path = scaledPath
```

Here you create a new animation, setting the layer's path to the data path. You'll notice that you're adding an animation for the path, and also setting the actual path on the layer - setting the path like this causes it to be the `toValue` of the animation, and ensures that the model layer and presentation layer remain in sync. For more detail on this stuff, please refer to Marin Todorov's excellent iOS Animations by Tutorials, which I was lucky enough to work on with him.

Run the playground and behold your amazing morphing line.



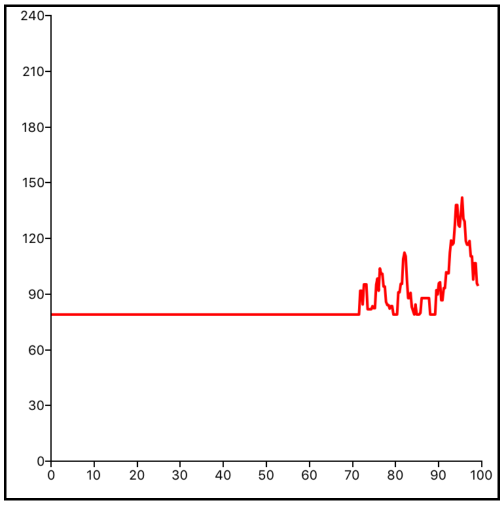
`CAShapeLayer` does all of the interpolation for you, but you have to give it a bit of a hint - that's why `midPath` had a point to match each point in the data path. If you'd just done a straight line with two points, the animation doesn't look nearly as good. To see what I mean, find the line where `midPath` is populated:

```
midPath.addLines(between: midPoints)
```

And temporarily change it to just use the start and end points:

```
midPath.addLines(between: [midPoints.first!, midPoints.last!])
```

You'll see that `CAShapeLayer` can't do as nice a job at interpolating between the two paths when they have such a different number of points:



Change the line back, and you're done with this section!

Note: The morphing animation looks super cool if you use a

`CASpringAnimation`, but unfortunately these don't seem to work very well with playgrounds.

2) Fills

`CAShapeLayer` can't fill the path as it stands, because the path isn't closed. If you close the path, then the data line won't look right. The solution is to add *another* shape layer, which will draw the fill. Remember, shape layer performance is super fast, and adding multiple layers is not going to hurt your app.

Open the **Plain Fill** page of the playground, and you'll be greeted with the familiar line graph, with no animation this time.

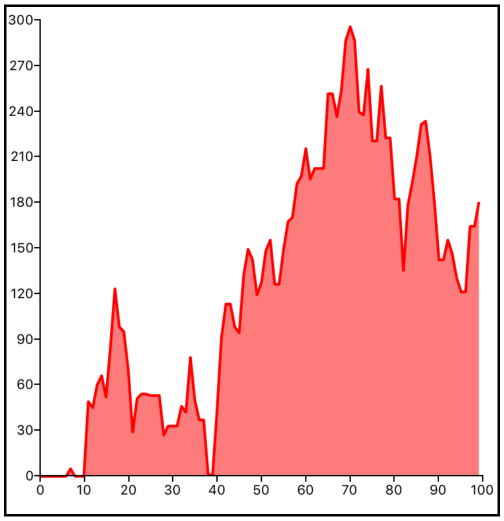
Find the line marked **Start your demo coding here!**. `createShapeLayer` is a convenience method which creates, adds and returns a new shape layer fitting the plot area, so the layers will be in the order that you are calling `createShapeLayer`. You want the fill to be underneath the line for the best effect, so add the following code just above the `createShapeLayer` line:

```
if let fillPath = path.mutableCopy() {

  let current = fillPath.currentPoint
  fillPath.addLine(to: CGPoint(x: current.x, y: 0))
  fillPath.addLine(to: .zero)
  fillPath.closeSubpath()

  let fill = chart.createShapeLayer(fillColor:
    UIColor.red.withAlphaComponent(0.5))
  fill.path = fillPath.copy(using: &scale)
}
```

Here you create a copy of the data path, then close it off by adding a line section straight down, another straight back to (0, 0), then closing off the path (which will add a straight section from the current point to the start point).

You then create a shape layer, give it a fill color, and give it the closed path.

Run the playground and you'll see that your line graph is now filled in beneath the data!



Your graph is looking good, but it's a bit... boring. Your users deserve more interesting data representations than this!

3) Fancy Fills

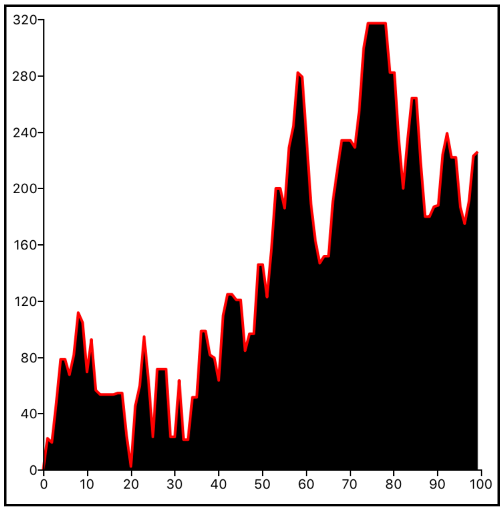
In this section you're going to use a layer mask to make an even better fill effect. A layer mask is like a stencil or cutout that you add to a layer so that only part of its contents are visible. Happily, a layer's mask is just another layer, and you're already an expert at making those.

Open the **Gradient Fill** playground page, and you'll see the familiar boring graph. The code to make the fill path is already there, to save you having to type it out again, but it's going to work a little differently this time.

After the **Start your demo coding here!** marker, add the following code:

```
let fill = chart.createShapeLayer(fillColor: .black)
fill.path = fillPath.copy(using: &scale)
```

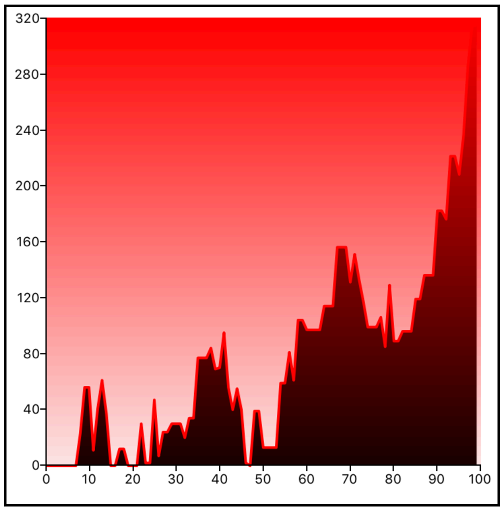
This is similar to the code you used before, except you're using a plain black fill. If you run the playground now, you'll see it, but it doesn't look great.



Add the following code:

```
let gradient = CAGradientLayer()
gradient.frame = fill.frame
gradient.colors = [
  UIColor.red.withAlphaComponent(0.1).cgColor,
  UIColor.red.cgColor
]
chart.plotArea.layer.addSublayer(gradient)
```

Run the playground now and you'll see your gradient fill, but it's everywhere!

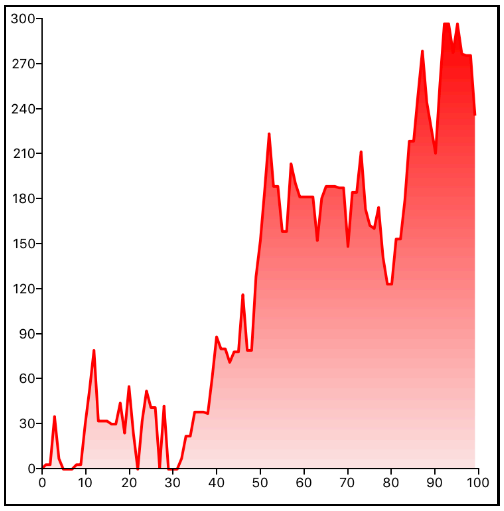


This is where the layer mask comes in. You only want to see the gradient in the area where the black fill currently is.

Add the following code:

```
fill.removeFromSuperlayer()
gradient.mask = fill
```

This removes the black fill from the chart area, then adds it as a mask to the gradient. Layer masks work by only showing the layer's content when there is content in the mask - so you only see the gradient where the fill is present. Doesn't that look good?



4) Frivolously Fancy Fills

We've featured lots of rainbows in this session, and I see no reason to stop that now. You're going to push the bounds of data visualisation to the limits by rainbowing up your chart fill.

Just above the line:

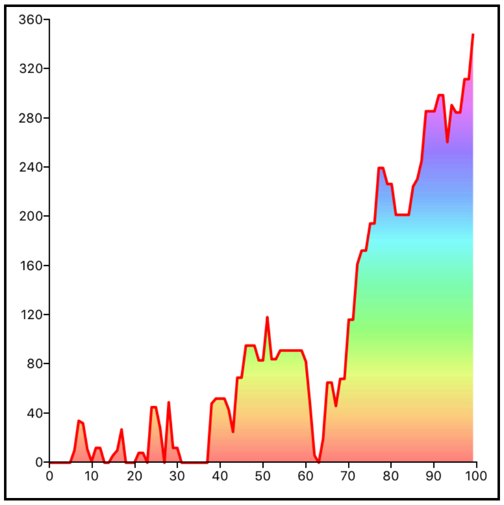
```
chart.plotArea.layer.addSublayer(gradient)
```

Delete the code where you set `gradient.colors` and add the following:

```
let rainbow: [CGColor] = (0...10).map {
  UIColor(hue: CGFloat($0) / 10,
          saturation: 1,
          brightness: 1,
          alpha: 0.5).cgColor
}
gradient.colors = rainbow
```

This removes the mundane, two-color gradient from the gradient layer and replaces it with a sprinkle of magic instead!

Now *that's* a fabulous line graph!



5) Sorry, not fabulous enough

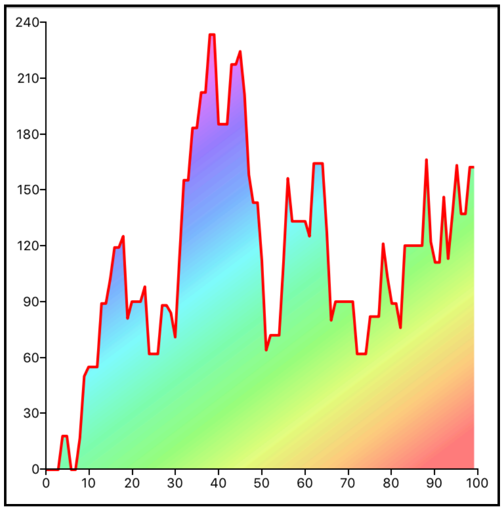
It's a `CAGradientLayer`, and the `A` stands for Animation. What you're about to do might not be the best data visualisation, but it does look cool - and you can definitely think of more practical applications for the effect, I'm sure.

Immediately after the line where you set the gradient mask, add the following code:

```
let group = CAAnimationGroup()
group.duration = 1
group.repeatCount = .greatestFiniteMagnitude
group.autoreverses = true

let startPoint = CABasicAnimation(keyPath: "startPoint")
startPoint.fromValue = NSValue(cgPoint: CGPoint(x: 0, y: 0))
startPoint.toValue = NSValue(cgPoint: CGPoint(x: 1, y: 0))

let endPoint = CABasicAnimation(keyPath: "endPoint")
endPoint.fromValue = NSValue(cgPoint: CGPoint(x: 1, y: 1))
endPoint.toValue = NSValue(cgPoint: CGPoint(x: 0, y: 1))
group.animations = [startPoint, endPoint]

gradient.add(group, forKey: nil)
```

What you're doing here is animating the start and end points of the axis of the gradient. First, you create an animation group - this will control the timing of the animations you add to it, to save repeating yourself.

Then, you create two separate animations, one for the gradient's start point, and one for the end point, add them to the group, then add the group to the layer.



If you're not feeling nauseous enough, try playing about with the point values to see what other effects you can make!

6) That's it!

Congrats, at this time you should have a good understanding of how to use various CALayer features to make your line graphs much more interesting! You've hopefully also learned how not to take things too far...