HACKING WITH SWIFT

ADVANCED iOS

VOLUME THREE

COMPLETE TUTORIAL COURSE

Learn Siri shortcuts, ARKit, Create ML, and more real-world projects

Paul Hudson

Project 1
Brain Training

Get started with Create ML by recognizing handwritten digits.

Setting up

In this opening project we're going to build a game. Now, before you groan and put down the book, hear me out: this isn't like most games, for three reasons.

First, we're going to be using UIKit rather than SpriteKit – table views, image views, and so on.

Second, we're going to be using Apple's new Create ML system to build a machine learning model to recognize handwritten input from our player.

Third, it's about *mathematics* – it's more edutainment than entertainment.

Still here? Great! We're going to build a game called Brain Training, based on a Nintendo DS game I played over a decade ago called *Professor Kageyama's Maths Training*. More specifically, we're going to show users a series of basic arithmetic questions – 3 + 6, for example – and have them solve them by writing numbers directly onto the screen.

To make this work takes a number of steps: training a Core ML model using Create ML, designing a user interface, having our model recognize user handwriting, then connecting up the rest of the game. Hopefully this should give you a gentle introduction to Create ML while still producing a useful and fun project!

Start by creating a new iOS project using the Single View App template. Name it BrainTraining, then save it somewhere sensible.

Training a model using Create ML

The first step in this project will be to train a Core ML model using Create ML. If you've read any of my other books you'll realize this is an odd place for me to start – I usually prefer to start with the user interface because I think it helps shape the rest of the project when we know how things will be used.

This time it's different, because training a model takes *time*. Sure, you can train models quickly if you have relatively simple needs: you could train a model to distinguish between cats and dogs using just 30 pictures if you wanted.

However, in this project we're going to be distinguishing between handwritten numbers, and – to be blunt – the handwriting of most people is *terrible*. So, we need to use a significantly larger data set: about 70,000 items all in, and that's just for the numbers 0 through 9.

Helpfully, this data set already exists and is available for free through the MNIST database of handwritten digits. I've included it in the project files for this book, because even though it contains 70,000 images they are really small – just 28x28 pixels – so the package containing the images is only 17MB or so.

Note: because of the way files are stored on computers, the expanded picture collection will take around 300MB, but don't worry – the trained Core ML model will be much smaller.

Start by finding mnist-full.tar.gz in the project files that you should have received with this book. This is a tarball – like a zip file – so you should be able to double-click on it to unzip all the images.

Inside you'll find two directories: testing and training. And inside *those* you'll find directories numbered 0 through 9, representing the 10 digits inside this data set. Each of those numbered directories contains the same digit written again and again by different people – sometimes neatly, sometimes much less so, but it's important to feed our algorithm messy data as well as clean data.

I'm going to go into more detail on how the process works in a moment, but first I'd like to kick off the training process itself – it will take about an hour depending on your computer, so I have ample time to explain things once the process has begun!

First, create a new macOS playground in Xcode. Yes, a *macOS* playground: Create ML is a macOS-only framework, and if you try this in an iOS playground you'll hit trouble.

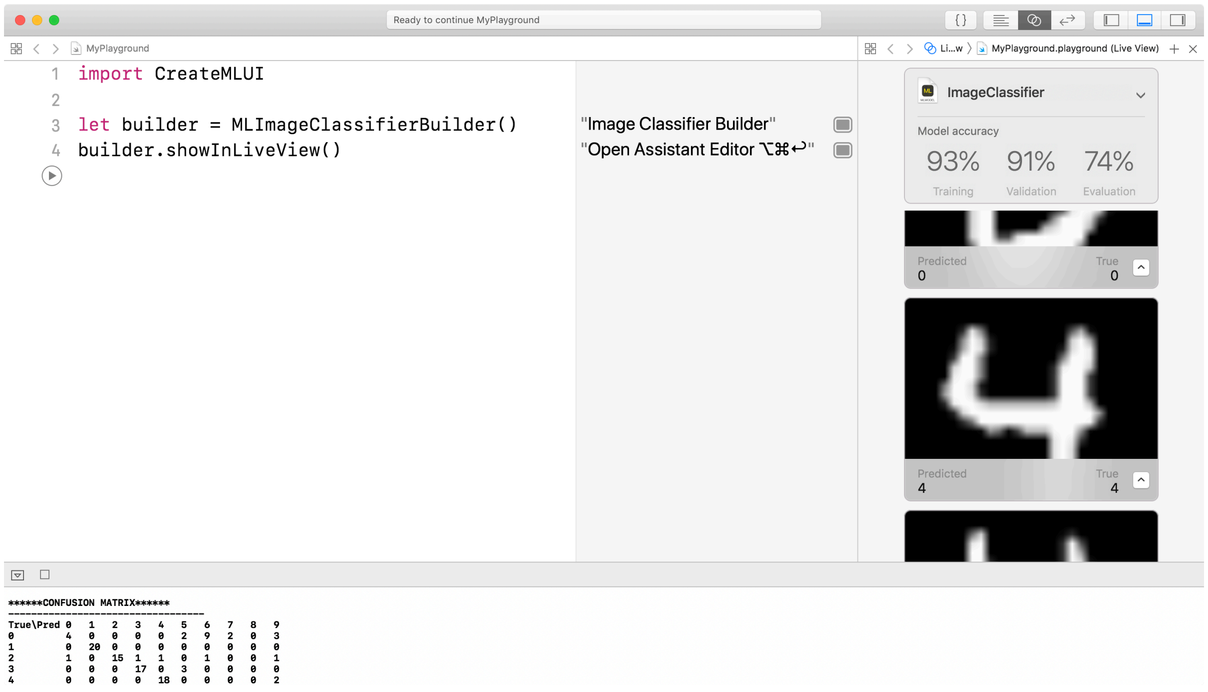
Give your playground the following code:

```
import CreateMLUI

let builder = MLImageClassifierBuilder()
builder.showInLiveView()
```

The first line brings in the CreateMLUI framework, the second creates a new image classifier, and the third tells it to show itself in Xcode's assistant editor. Press Play in your playground to get it moving, then activate Xcode's assistant editor so you can see its UI in action – go to the View menu and choose Assistant Editor > Show Assistant Editor.

Tip: In these early betas it's common to get errors when you press the Play button launch Create ML's UI. Sometimes you have to press it two or three times to get the UI to appear, but once it's visible you should be OK.



```swift
import CreateMLUI

let builder = MLImageClassifierBuilder()
builder.showInLiveView()
```

```
"Image Classifier Builder"
"Open Assistant Editor ⌥⌘↩"
```

```
******CONFUSION MATRIX******
----------------------------------
True\Pred 0   1   2   3   4   5   6   7   8   9
0         4   0   0   0   0   2   9   2   0   3
1         0   20  0   0   0   0   0   0   0   0
2         1   0   15  1   1   0   1   0   0   1
3         0   0   0   17  0   3   0   0   0   0
4         0   0   0   0   18  0   0   0   0   2
```

I'll explain more about the Create ML user interface in just a moment, but first let's get the training process started. You should see a large box saying Drop Images To Begin Training, and I'd like you to drag the whole Training directory into there. Not Testing, and certainly not the main "mnist" directory you extracted, but just the one marked Training.

Once it starts you'll see lots of image numbers flash up on the screen in Xcode, and Create ML will print regular updates about the process – how many of the images it has analyzed, how much time it has taken so far, and what kind of completion percentage it has reached.

This will take quite some time – it took my MacBook Pro about 35 minutes to complete. Normally I'd suggest you go and make coffee while waiting, but this long delay gives me a chance to explain a little more about Create ML.

First, our directory structures. We have Training and Testing directories, both of which contain subdirectories numbered 0 through 9. Training data is what we feed to Create ML to have it learn what all the digits look like. By placing all the handwritten images of 0 inside a directory called 0 we're telling Create ML what each image *means* – "this is a zero, this is another zero, this is a third zero, and so on."

Create ML will actually split that training data up into two parts: one large part will be used to train the model itself (the actual training data), and one will be used to validate the process (validation data).

We also have a third directory called Testing, which contains more images of digits from MNIST. We'll show these to our finished Core ML model in order to verify that it does a good enough job on images it hasn't seen before.

So, that's all there is about the directory structure: it's just Training and Testing, then subdirectories based on how you want to categorize your data.

Where things get more interesting is how you *train* your data. I've given you the complete set of MNIST digits because it gives us the easiest way to generate Core ML models with suitable accuracy, but Create ML has a few other options you can experiment with by pressing the down arrow later on:

- Iterations: How closely Create ML should study your data. It might sound like going higher and higher is a good thing, but you get diminishing returns and if you go too high it might just effectively memorize your data rather than learning the general concepts.
- Validation data: If you have your own specific validation data you want to use, you can specify it here to get repeatable results.
- Data augmentation: this manipulates your input images (flipping, rotating, brightening/darkening, and shearing) in order to make it look more different to the image trainer.

Create ML defaults to using 10 iterations and no augmentation, but it's not uncommon to use a couple of hundred iterations and lots of augmentation depending on your data. Be careful, though: each augmentation you enable will 4x the amount of work Create ML has to do, so if you enable them all you might have to wait several days!

Now, this becomes more interesting because I've provided for you a second data set called mnist-partial.tar.gz. Rather than containing all 70,000 images, this only contains *1000*. Yes, that means it has substantially less diversity to train from, but on the flip side it's 70x faster and so will allow you to experiment with training options to see what works - it's very likely you'll be able to get better results by analyzing the smaller data set more effectively.

Keep in mind that this data set was built from people using real handwriting, which isn't a perfect fit for our needs – people touch a screen differently from how they hold a pen. So, once your initial training has finished and stashed away somewhere safe go ahead and experiment!

When your Mac eventually finishes the training process, you'll get a new box saying "Drop Images To Begin Testing". Drop the Testing directory there, and it will be scanned. This won't take long, but it gives us an idea of how effective the model is – ideally you'll see a score around 75-80%, but if you modify the training options described above you should be able to get that higher.

I'm not going to keep you waiting for training to finish – we have our user interface to build! – but before I move on I should at least tell you how to save the finished model once training completes. To save your model, open the disclosure indicator next to where it says "ImageClassifier" and fill in your name and a description of the model. Finally, click Save to save the model to your Documents directory.

At this point you'll now be getting an idea of just how long it takes to train lots of machine learning data – your Mac is likely to be only about 20% of the way through the process. Fortunately we still have all our user interface to design, so there's lots of work ahead.

Designing our interface

Leave your Xcode playground working on its training, then return to the BrainTraining project you created earlier and open Main.storyboard for editing.

Start by embedding the main view controller inside a navigation controller, which is always the easiest way to handle the safe area on things like the iPhone X.

Now drag out an image view, making it take up about the bottom half of the remaining view controller. This is going to be where the user draws their answer to each question, so I'd like you to give it a light gray background color to help it stand out – something like gray with 90% brightness ought to do it.

This image view should always take up most of the width of the screen, then resize itself vertically so that it remains square. Remember, the MNIST handwriting images are also square, so it's important our user's drawings match that shape.

So, give it the following constraints:

- Ctrl-drag from the image view to its parent view using the document outline, then select Leading, Trailing, and Bottom Space to Safe Area equal to 16.
- Ctrl-drag from the image view to itself to fix its aspect ratio, adjusting it to 1:1 – i.e., square.

Now make sure the User Interaction Enabled checkbox is checked so that our users can touch this to draw numbers.

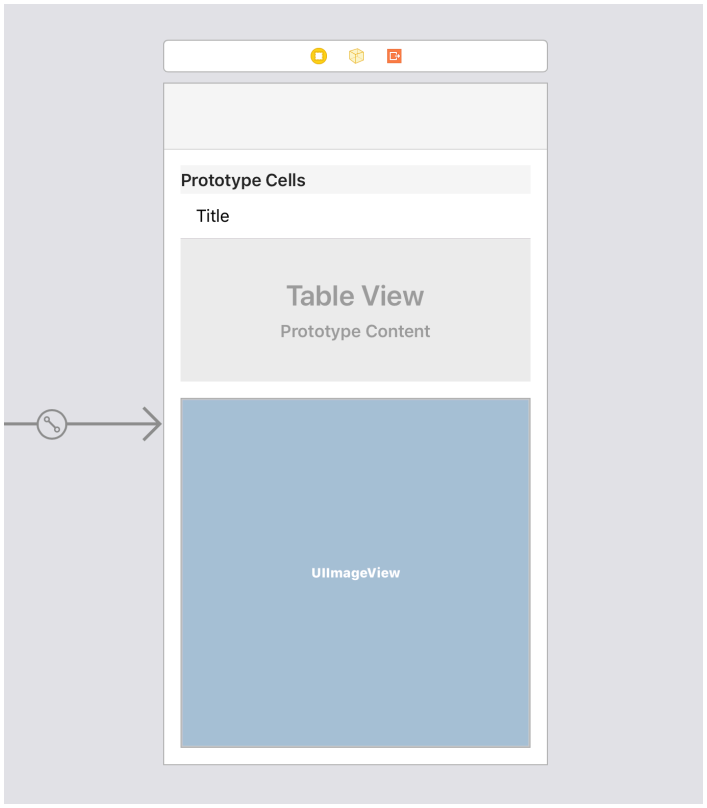
We're going to use a table view to fill the top half of the view, so drag one out above the image view. I'd normally ask you to make our view controller the delegate and data source for this table but please don't do that just yet. Instead, give it the following constraints:

- Ctrl-drag from the table to its parent view using the document outline, then select Leading, Trailing, and Top Space to Safe Area equal to 16.
- Ctrl-drag from the table down to the image view, then select Vertical Spacing equal to 16.

Using this approach, the image view will always remain square at the bottom of the user interface, while the table view resizes itself to fit the remaining space at the top.

Give the table view a single prototype cell, using the Basic style and identifier "Cell". Finally, change the selection style for the whole table view to be No Selection – this area is there to show users the questions to answer, and won't do anything when selected.

That's our user interface almost complete, but your training is probably still under 50% complete, so let's move on to the next part: drawing numbers.



Touch to draw

This game requires users to touch the screen to draw answers to various mathematics questions. I'll admit this isn't terribly easy to do in the simulator, so if you have a real device I suggest you try using that – it's much easier!

If were do any *serious* drawing we'd need to investigate all sorts of optimizations to make things more efficient. However, for this simple project we can just create our own custom **UIImageView** and draw some lines as the user moves their finger.

Press Cmd+N to create a new Cocoa Touch Class, calling it "DrawingImageView" and making it subclass from **UIImageView**.

To make this work we need to add two properties and five methods:

- A property to store our delegate, which is going to be the **ViewController** instance that owns this image view.
- A property to store the current touch position of the user. As they move their finger, we'll draw lines between the new touch position and the previous touch position.
- A **draw()** method to update the image as the user's finger moves.
- Implementations of **touchesBegan()**, **touchesMoved()**, and **touchesEnded()**.
- And finally a **numberDrawn()** method that we'll trigger once the user finishes drawing. This will send the finished image back to our view controller for processing.

First, the two properties. Add these two **DrawingImageView** now:

```
weak var delegate: ViewController?
var currentTouchPosition: CGPoint?
```

Second, the **draw()** method. This will be called with a start point (the previous touch position of the user) and an end point (the current touch position), and will add a line to our image. If you've done image rendering before this will be a cinch, but just in case here's a reminder:

1. We're going to use **UIGraphicsImageRenderer**, which is UIKit's preferred way of

rendering images.

2. We'll start by drawing our image view's *existing* image into the renderer, because we want new lines to be drawn alongside existing lines.
3. To make the line look good, we're going to make it 15 points wide, rounded on the edges, and black.
4. We can then move to the start point (wherever the user was previously), add a line to the destination point (wherever they are right now), and stroke that path.

Here's that in code:

```
func draw(from start: CGPoint, to end: CGPoint) {
   let renderer = UIGraphicsImageRenderer(size: bounds.size)

   image = renderer.image { ctx in
      image?.draw(in: bounds)

      UIColor.black.setStroke()
      ctx.cgContext.setLineCap(.round)
      ctx.cgContext.setLineWidth(15)

      ctx.cgContext.move(to: start)
      ctx.cgContext.addLine(to: end)
      ctx.cgContext.strokePath()
   }
}
```

Next we need to implement **touchesBegan()**. This is triggered when the user touches down on the screen, so it's our chance to set the **currentTouchPosition** property to the location of their touch:

```
override public func touchesBegan(_ touches: Set<UITouch>, with
event: UIEvent?) {
   super.touchesBegan(touches, with: event)
```

```
    currentTouchPosition = touches.first?.location(in: self)
}
```

The third method is **touchesMoved()**, which is called when the user moves their finger across the screen. This will read the new touch position, unwrap the previous touch position, then pass them both to the **draw()** method we just wrote. Once that's done, it will update **currentTouchPosition** to be the new touch position so that further movement works correctly:

```
override public func touchesMoved(_ touches: Set<UITouch>, with
event: UIEvent?) {
    super.touchesMoved(touches, with: event)

    guard let newTouchPoint = touches.first?.location(in: self)
else { return }
    guard let previousTouchPoint = currentTouchPosition else
{ return }

    draw(from: previousTouchPoint, to: newTouchPoint)
    currentTouchPosition = newTouchPoint
}
```

Third, we need to implement **touchesEnded()**. This will set our **currentTouchPosition** property back to **nil** so that fresh touches are reset, but it also needs to do something more important: after a short pause, we need to make it call the final method for this class, **numberDrawn()**.

You see, some numbers – 4 and 5 for most people, 7 for some – require *two* finger movements to draw. If we submitted the user's answer as soon as they lifted their finger then we would often get a wrong answer because they hadn't finished drawing. So, instead we're going to submit their answer after 0.3 seconds: long enough to give them enough time to draw some more, but short enough that it isn't too annoying.

Add this method:

```
override public func touchesEnded(_ touches: Set<UITouch>, with
event: UIEvent?) {
    super.touchesEnded(touches, with: event)
    currentTouchPosition = nil

    perform(#selector(numberDrawn), with: nil, afterDelay: 0.3)
}
```

Tip: That code won't compile just yet, because we haven't written **numberDrawn()**.

As a compliment to that, we're going to add one further line to **touchesBegan()** to *cancel* the call to **numberDrawn()** if the user touches the screen again. This will allow them to finish drawing their second line fully before we evaluate the finished image.

So, put this extra line at the end of **touchesBegan()**:

```
NSObject.cancelPreviousPerformRequests(withTarget: self)
```

That will cancel the impending call to **numberDrawn()**, but we'll reschedule it as soon as **touchesEnded()** is called again.

The final method to implement is **numberDrawn()**, and this takes a little thought because we need to make the user's drawing as close as possible to the training data we gave to Core ML. This means scaling it down to 28x28 pixels in size, placing it onto a white background, then inverting the whole thing so that their drawing is white and the background is black.

Let's write this method step by step, starting with the method stub:

```
@objc func numberDrawn() {
}
```

The first step is to make sure we have a valid image in our image view, so start with this line:

```
guard let image = image else { return }
```

Next we need to create another **UIGraphicsImageRenderer**, this time with the size 28x28. UIKit will automatically try to make this at whatever drawing scale your device uses, but we need exactly 28x28 so we need to specifically ask for a renderer format that uses 1x scale.

Add this code:

```
let drawRect = CGRect(x: 0, y: 0, width: 28, height: 28)

let format = UIGraphicsImageRendererFormat()
format.scale = 1

let renderer = UIGraphicsImageRenderer(bounds: drawRect,
format: format)
```

Third, we need to fill that new image with a white background, then render our user's drawing on top:

```
let imageWithBackground = renderer.image { ctx in
   UIColor.white.setFill()
   ctx.fill(bounds)
   image.draw(in: drawRect)
}
```

And finally we need to use Core Image to invert that drawing so that the user's black number is white, and the white background is black – just like the MNIST source data we used.

Add this code to finish the method:

```
// convert our UIImage into a CIImage; the force unwrap is safe
here because the CGImage is only nil if the UIImage was created
from a CIImage
let ciImage = CIImage(cgImage: imageWithBackground.cgImage!)
```

```
// attempt to create a color inversion filter
if let filter = CIFilter(name: "CIColorInvert") {
   // give it our input CIImage
   filter.setValue(ciImage, forKey: kCIInputImageKey)

   // create a context so we can perform conversion
   let context = CIContext(options: nil)

   // attempt to read the output CIImage
   if let outputImage = filter.outputImage {
      // attempt to convert that to a CGImage
      if let imageRef = context.createCGImage(outputImage,
from: ciImage.extent) {
         // attempt to convert *that* to a UIImage
         let finalImage = UIImage(cgImage: imageRef)

         // and finally pass the finished image to our delegate
         delegate?.numberDrawn(finalImage)
      }
   }
}
```

We haven't written the **numberDrawn()** method on our view controller just yet, so let's fill that in with an empty placeholder just so you can see your progress so far. Add this to ViewController.swift now:

```
func numberDrawn(_ image: UIImage) {
}
```

Before you build: I know you're probably keen to try this out, but we need to head back to our user interface before you can make this all work.

Earlier we created an image view and a table view in Main.storyboard. We've just defined a custom **UIImageView** subclass for our drawing, so we need to make sure Interface Builder uses *that* rather than the standard **UIImageView**.

So, select the image view in Main.storyboard, then use the identity inspector to change its class to be **DrawingImageView**. It should also automatically check the "Inherit Module From Target" box, but if it doesn't please check it yourself.

The last step is to create two outlets using the assistant editor: create one for the table view called **tableView**, and one for the image view called **drawView**. If you changed the latter's class name correctly, you should see it linked in your code as a **DrawingImageView**.

Hopefully you can now build and run your code, and you should be able to scribble in the gray image view using your finger. It won't be recognized by Core ML yet, but all being well your Create ML process should be approaching completion.

Connecting our game

In this game we're going to show arithmetic problems to our users in the table at the top, and have them write answers below. The current question will be shown in a larger font, but as they answer we'll show previous questions (and their answers) in the table view cells below.

So, while your Create ML training finishes up, let's make the rest of the user interface come to life.

First, we need a custom data type to hold one question: the sum we want users to answer, the correct answer, and their actual answer. You can place this in a separate Question.swift file if you want, or just add this directly above **class ViewController** in ViewController.swift:

```
struct Question {
   var text: String
   var correct: Int
   var actual: Int?
}
```

Next we need two pieces of state for our game, both to be added to the **ViewController** class. The first will be an array of our new **Question** type, where new questions are always inserted at the front so they appear at the top, and the second will be a **score** property that tracks how many they got correct.

So, please add these two properties to the **ViewController** class now:

```
var questions = [Question]()
var score = 0
```

I already said that the current question the user is answering will be shown at the top of the table, and will have a larger font to make it clear that's the one they are answering. While working on this project I came across a rather annoying **UITableView** bug that triggers ugly animations when you insert a table view cell while also reloading one other cell, so as a workaround we're going to have to adjust cells by hand.

To make the workaround less ugly, the smart thing to do is to create a **setText()** method like this one:

```
func setText(for cell: UITableViewCell, at indexPath:
IndexPath, to question: Question) {
   if indexPath.row == 0 {
      cell.textLabel?.font = UIFont.boldSystemFont(ofSize: 48)
   } else {
      cell.textLabel?.font = UIFont.systemFont(ofSize: 17)
   }

   if let actual = question.actual {
      cell.textLabel?.text = "\(question.text) = \(actual)"
   } else {
      cell.textLabel?.text = "\(question.text) = ?"
   }
}
```

That accepts a table view cell, the index path where the cell will be shown, and a question to place inside the cell. It then uses a large font if it's the first question or a small font for subsequent questions, then formats the cell's label to show the user's answer if there is one.

With that in place, we can write the **cellForRowAt** method easily because it mostly just calls the new **setText()** method:

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
   let cell = tableView.dequeueReusableCell(withIdentifier:
"Cell", for: indexPath)
   let currentQuestion = questions[indexPath.row]
   setText(for: cell, at: indexPath, to: currentQuestion)
```

```
    return cell
}
```

You'll also need to adjust the **numberOfRowsInSection** method so that it returns as many rows as we have questions:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    return questions.count
}
```

Now for two important parts of our game: a method that creates a single question, and a method that displays that question in our user interface. Neither of these are difficult, but it's worth breaking them down just to be sure.

First, it's the job of **createQuestion()** to create one **Question** instance to ask our player. The MNIST data set is able to recognize the digits 0 through 9, so this method needs to be able to make adding and subtracting arithmetic where the number is in the range of 0 through 9, inclusive on both sides.

There are a few ways you can do this, but given that the mathematics is so simple we're going to take a brute force approach:

1. Enter an infinite loop using **while true**.
2. Generate two random numbers between 0 and 9 using **Int.random(in:)**.
3. If **Bool.random()** returns true then we'll add the two numbers together. If the result is less than 10, make that our question and exit the loop.
4. If **Bool.random()** returns false then we'll subtract the second number from the first. If the result is greater than or equal to 0, make that our question and exit the loop.
5. If we didn't get a good question the first time, the infinite loop will go around again and again until we do.
6. Eventually the loop will break, and we'll wrap up the question and answer inside a new

Question instance, and return that to our caller.

That’s the entire method, so please go ahead and add this now:

```
func createQuestion() -> Question {
   var question = ""
   var correctAnswer = 0

   while true {
      let firstNumber = Int.random(in: 0...9)
      let secondNumber = Int.random(in: 0...9)

      if Bool.random() == true {
         let result = firstNumber + secondNumber

         if result < 10 {
            question = "\(firstNumber) + \(secondNumber)"
            correctAnswer = result
            break
         }
      } else {
         let result = firstNumber - secondNumber

         if result >= 0 {
            question = "\(firstNumber) - \(secondNumber)"
            correctAnswer = result
            break
         }
      }
   }

   return Question(text: question, correct: correctAnswer,
actual: nil)
}
```

So, that creates a single question ready to be asked. The next step is to *ask* it – i.e. to clear the user's drawing, and insert the new question into the table view.

This is where my **setText()** workaround becomes important: the top table view cell has a larger font than the others, so if we animate adding a cell normally you'll get a strange animation glitch – you'll see the top cell slide down behind the others and carry on going.

To work around this, we're going to call **setText()** directly on the second cell to make its font size change. This isn't ideal – really we should be calling **reloadRows()** on our table view to reload the second cell – but when it's called just after inserting a cell UIKit is unhappy.

Go ahead and add this second method now:

```
func askQuestion() {
   // clear any previous iamge
   drawView.image = nil

   // create a question and insert it into our array so that it
appears at the top of the table
   questions.insert(createQuestion(), at: 0)

   let newIndexPath = IndexPath(row: 0, section: 0)
   tableView.insertRows(at: [newIndexPath], with: .right)

   // try to find the second cell in our table; this was the
top cell a moment ago, and needs to be changed
   let secondIndexPath = IndexPath(row: 1, section: 0)
   if let cell = tableView.cellForRow(at: secondIndexPath) {
      // update this cell so that it shows the user's answer in
the correct font
      setText(for: cell, at: secondIndexPath, to: questions[1])
```

```
    }
}
```

The last step in getting our app working is to call **askQuestion()** for the first time inside **viewDidLoad()**. While we're there we can also put a title in the navigation bar, and add a light stroke around the table to make the whole thing look better:

```
title = "Brain Training"
tableView.layer.borderColor = UIColor.lightGray.cgColor
tableView.layer.borderWidth = 1
drawView.delegate = self
askQuestion()
```

That completes all the configuration for our table view, so now it's time to head back to the storyboard and make our view controller the data source and delegate for the table. So, open Main.storyboard, then Ctrl-drag from the table view to its view controller, selecting data source then delegate.

If you build and run the app now you should be able to see the first question appear in our table view. If you'd like to see how the animation looks, try adding this dummy line of code to the **numberDrawn()** method in ViewController.swift:

```
askQuestion()
```

That will trigger **askQuestion()** every time you draw anything in the image view, so you should see how the 0.3 second delay works.

Note: Using a large font for the first cell will cause **UITableView** to assign large cell heights to all our cells, which doesn't look great. So, you might want to add these two methods to squeeze them a bit tighter:

```
func tableView(_ tableView: UITableView,
estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat {
    return 56
}

func tableView(_ tableView: UITableView, heightForRowAt
indexPath: IndexPath) -> CGFloat {
    return 56
}
```