CS2048—Homework & Reading List 2

Reading

Read the following sections of the *language guide*

Section	Special attention to
The Basics	Tuples
Operators	Nil Coalescing Operator
Functions	Functions with Multiple Return Values, Optional Tuple Return Types Function Parameter Names
Closures	Trailing Closures, Capturing Values, Closures are Reference Types, Nonescaping Closures, Auto Closures
Enumerations	Raw Values, Recursive Enumerations
Protocols	Entire chapter
Type Casting	Entire chapter
Error Handling	We are not going to use this as much in this course but we did have a quick example of it in the graphing calculator lab.

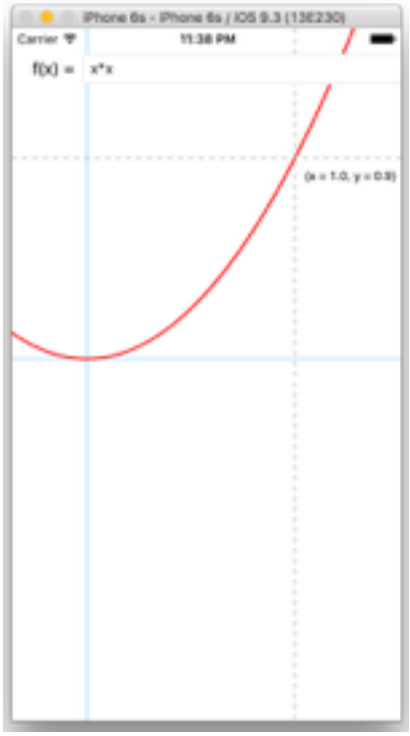
Read the following sections from the *Cocoa Core Competencies* guide: *introspection*, *delegates*, *MVC*

Read the following sections from the *Event Handling Guide* guide: *Gesture Recognizers*

Programming Assignment (due Sep 18th)

Continue the graphing calculator app we started in class.

TODO



- You calculator must be able to plot the following functions: log, sin, cos, tan. This is fairly simple to achieve and we have done some of it during class, you just need to create a variable in JavaScript that points to the function in the Math package (e.g., to expose the sin function, simply do `sin = Math.sin` in the JavaScript code).
- Handle discontinuous functions and regions where a function is not defined (e.g., it should only try to draw lines to or from points whose y value `.isNormal` or `.isZero`). Some examples are:
  - 1/x: discontinuous at 0
  - log(x): not defined for x < 0
  - sin(1/x): very fast oscillations around 0
  - tan(2 * x)
  - It is beyond the scope of this class to write a fully robust plotting function. Our approach to creating the plot is to sample the function at regular intervals on the x axis, and that has its limitations: a discontinuity can always hide in between samples. For that reason we will only test the simple case of `log(x)`.
- Your graphing view must support the following four gestures:
  - **Tap:** puts crosshair on function and displays the x and y position (see screenshot on the right). Note that the crosshair's y location should be the intersection of the crosshairs's x location with the function, so if the user taps at the position (x, y), the crosshair should actually show up at the position (x, f(x)).
  - **Panning:** moves the entire graph, including the axes. This should be done in a smooth, continuous manner (don't update the plot only after the gesture ends).
  - **Long Tap:** dismisses the crosshair.
  - **Pinch:** zoom in and out
- Extra points: draw the function derivative at the crosshair.

Tips

In the code we wrote in class we handled the fact that the plot and the screen had different coordinate systems and scales by multiplying the x and y coordinates each point in the function curve by a scale and then shifted them, like this

```
p.x *=  scale
p.y *= -scale
p.x +=  rect.midX
p.y +=  rect.midY
```

This type of transformation can be expressed with what is called an Affine Transform (which can also handle other transforms, such as rotation and sheer), implemented in iOS by the class `CGAffineTransform`. The code above can be re-implemented using `CGAffineTransform`:

```
// Create the transform
var T = CGAffineTransform.identity
T = T.translatedBy(x: rect.midX, y: rect.midY)
T = T.scaledBy(x: scale, y: -scale)

// Apply it to a point
p = p.applying(T)
```

While you can apply the transform to every point in `drawRect`, a cleaner way to organize your application is to apply the transform to the path itself. That way, you can draw in the coordinate system of the function and forget about applying the transform to every point.

```
path.apply(T)
```

this would apply the transform T to all the point in the path.

Another benefit of using the affine transform is that it makes it easy to get the inverse transform, this will be useful when implementing the tap gesture.

To handle user interactions you might want to have a pair of transforms: one that accumulates transforms from user interactions, another one that reflects the current transform (i.e., while the user is panning or zooming you would use the second one to give immediate visual feedback to the user, once the interaction is done, you would accumulated in the first transform). When drawing the function and the axis you would concatenate the transforms using the method `concatenating`.

Another complication of dealing with zooming and panning is that the x range of the plot changes after each pan or zoom operation. Ideally, we should only worry about plotting the range that is visible on the screen, so you will need to recompute it every time the user changes coordinate transform by either zooming or panning. In order to compute the new range we would need the inverse of the transform that we use to compute the screen coordinates given the function coordinates (as given above). That is, we would want to find the `xmin` and `xmax` in function space that corresponds to the left and right edges of the screen. You can obtain the inverse of an affine transform with the method `inverted`.

In order to learn more about CGAffineTransform you can read the API documentation *here*.

What to Hand In

Submit, via CMS, the entire project directory. If you project is called `MyProject`, Xcode will create a directory called `MyProject`, with a file called `MyProject.xcodeproj` and a subdirectory `MyProject`, please submit the **top level** `MyProject`.

Honesty and Integrity Policy

Projects are to be done individually. You may collaborate on the whiteboard, but each student's code must be their own.