Comp41550/47390 – Assignment 2 – TipCalc
(revision 1.1)

Objectives: This assignment requires very little coding. You will use the current version of Xcode to create a project and the built-in Interface Builder tool to construct a simple user interface for a tip calculator. As you enter each digit of the bill, your app will update the rounded tip amount for a given % tip set by moving a slider.

Part 1 – Building the UI

Create new project.

Begin by creating a new iOS project using the Single View Application template. Specify the following settings in the Choose options for your new project sheet:

- Product Name use **TipCalc**,
- Company identifier use **ie.ucd.cs**
- Language: **Swift**
- Devices use **iPhone**.

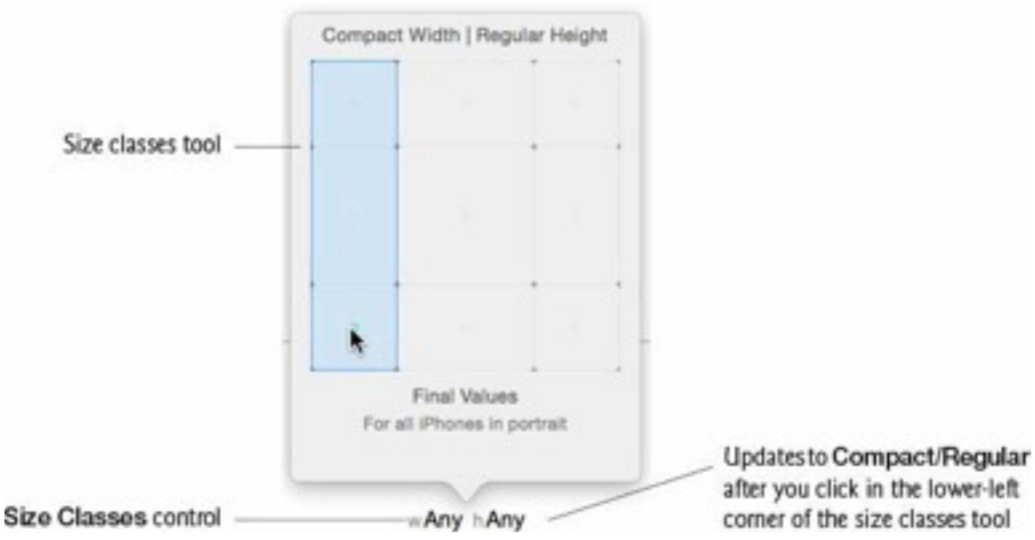
After specifying the settings, click Next, indicate where you'd like to save your project and click Create to create the project.

Configuring the App to Support only portrait orientation.

In Landscape orientation, the numeric keypad would obscure parts of the TipCalc's UI. For this reason, the app will only support portrait orientation. In the project settings' General tab that's displayed in the Xcode Editor area, scroll to the Deployment Info section, then, for Device Orientation, ensure that only Portrait is selected. Most iPhone apps should support portrait, landscape-left and landscape-right orientations, and most iPad apps should also support upside down orientation. You can learn more about this in Apple's Human Interface Guidelines.

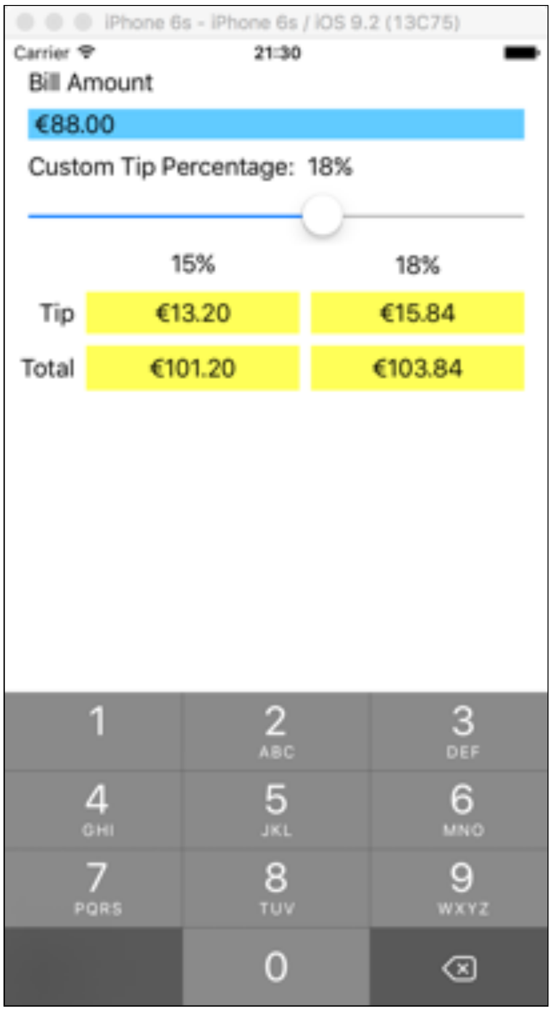
Configuring the Size Classes for Designing a Portrait Orientation iPhone App.

Until now, we designed a UI that supported both portrait and landscape orientations for any iOS device. For that purpose, we used the default size class *Any* for the design area's width and height. In this section, you will configure the design area (also called the canvas) for a tall narrow device, such as an iPhone or an iPod touch in portrait orientation. Select **Main.storyboard** to display the design area—also known as the canvas. At the bottom of the canvas, click the Size Classes control to display the size classes tool, then click in the lower-left corner to specify the size classes *Compact Width* and *Regular Height*.


Compact Width | Regular Height
Size classes tool
Final Values
For all iPhones in portrait
Updates to Compact/Regular after you click in the lower-left corner of the size classes tool
Size Classes control
wAny hAny

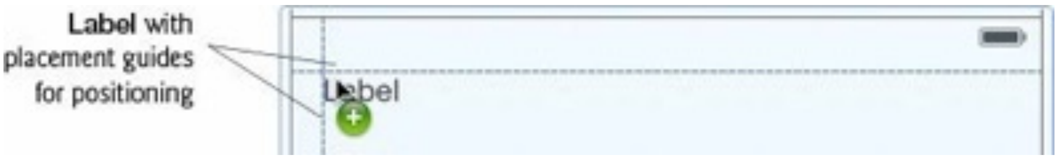
Adding UI Components.

The objective is to add and arrange UI components to create the basic design for your app as depicted in the Figure below.



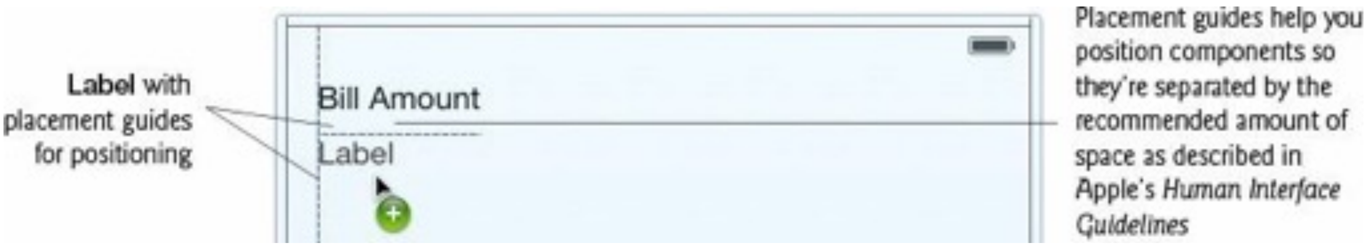
Step 1. Adding the "Bill Amount" Label

First, you will add the "Bill Amount" Label to the UI. Drag a Label from the Object library to the scene's upper-left corner, using the blue guide lines to position the Label at the recommended distance from the scene's top and left. Double click the Label, type *Bill Amount*, then press Enter to change its Text attribute.

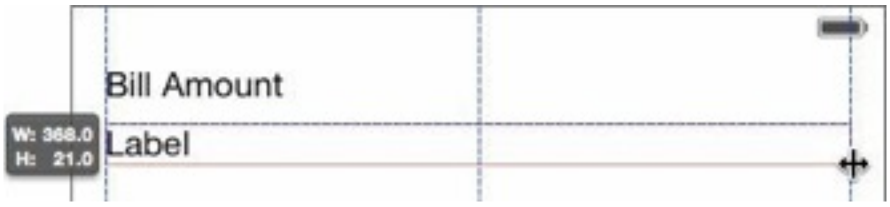


Step 2. Adding the Label That Displays the Formatted User Input

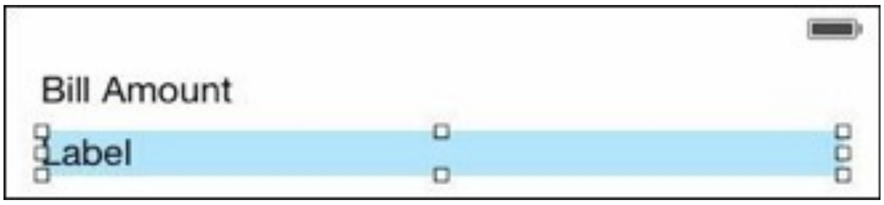
Next, you will add the blue Label that displays the formatted user input. Drag another Label below the "Bill Amount" Label, such that the placement guides appear as shown below.


Label with placement guides for positioning
Bill Amount
Label
Placement guides help you position components so they're separated by the recommended amount of space as described in Apple's Human Interface Guidelines

This is where the user input will be displayed. Drag the middle sizing handle at the new Label's right side until the blue guide line at the scene's right side appears. In the Attributes inspector, scroll to the View section and locate the Label's Background attribute.



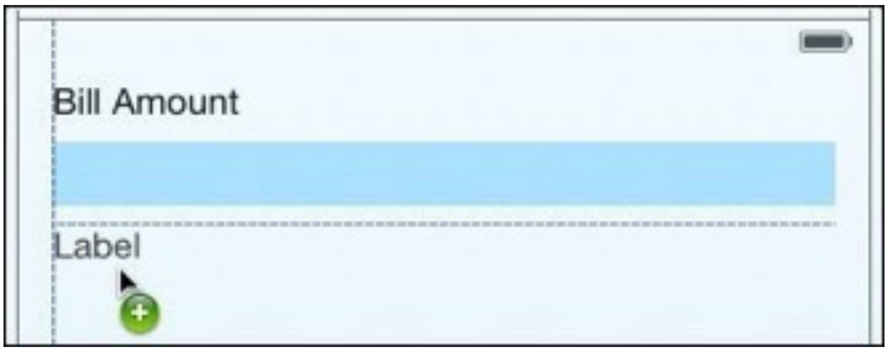
Click the attribute's value, then select Other... to display the Colours dialog. This dialog has five tabs at the top that allow you to select colours different ways. For this app, we used the Crayons tab. On the bottom row, select the Sky (blue) crayon as the colour, then set the Opacity to 50%—this allows the scene's white background to blend with the Label's color, resulting in a lighter blue color. The Label should now appear as shown below.



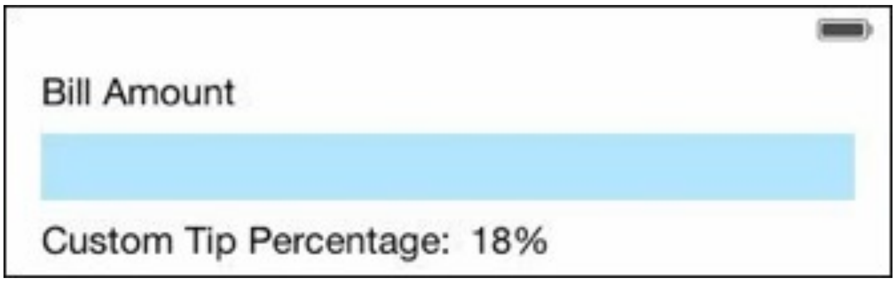
With the Label selected, delete the value for its Text property in the Attributes inspector. The Label should now be empty.

Step 3. Adding the "Custom Tip Percentage:" Label and a Label to Display the Current Custom Tip Percentage.

Next, you will add the Labels in the UI's third row. Drag another Label onto the scene and position it below the blue Label as shown below.

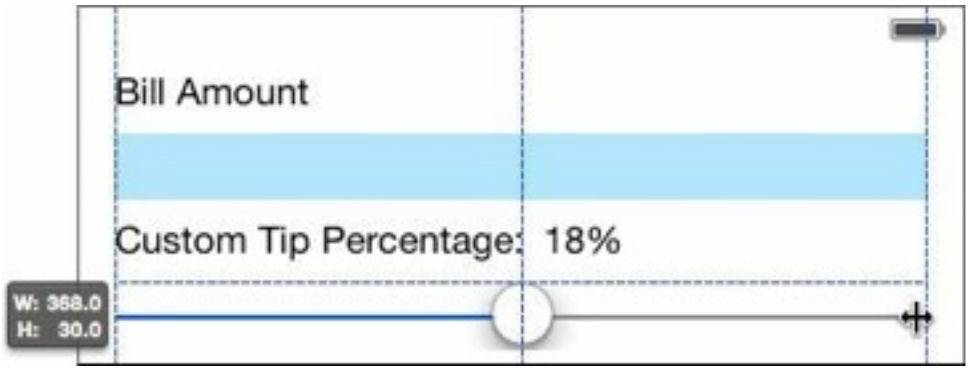


Double click the Label and set its text to *Custom Tip Percentage*:. Drag another Label onto the scene and position it to the right of the "Custom Tip Percentage:" then set its text to *18%*—the initial custom tip percentage we chose in this app, which the app will update when the user moves the Slider's thumb. The UI should now appear as shown below.



Step 4. Creating the Custom Tip Percentage Slider

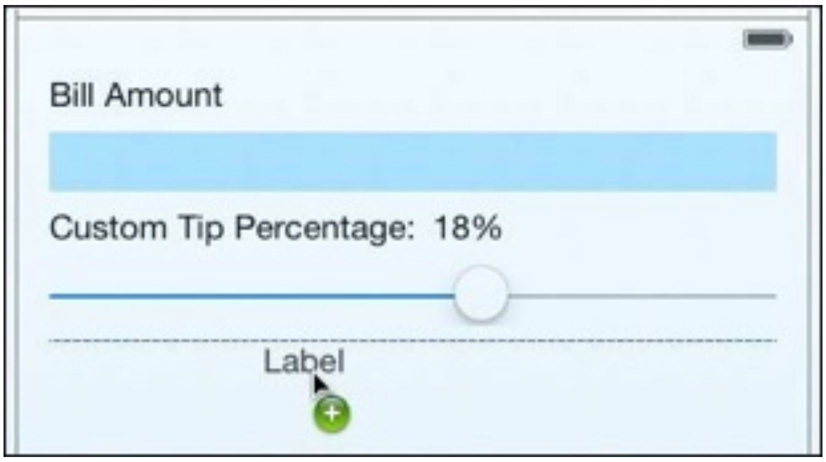
You will now create the Slider for selecting the custom tip percentage. Drag a Slider from the Object library onto the scene so that it is located at the recommended distance from the "Custom Tip Percentage:" Label, then size and position it as shown in below. Use the Attributes inspector to set the Slider's Minimum value to 0 (the default), Maximum value to 30 and Current value to 18.



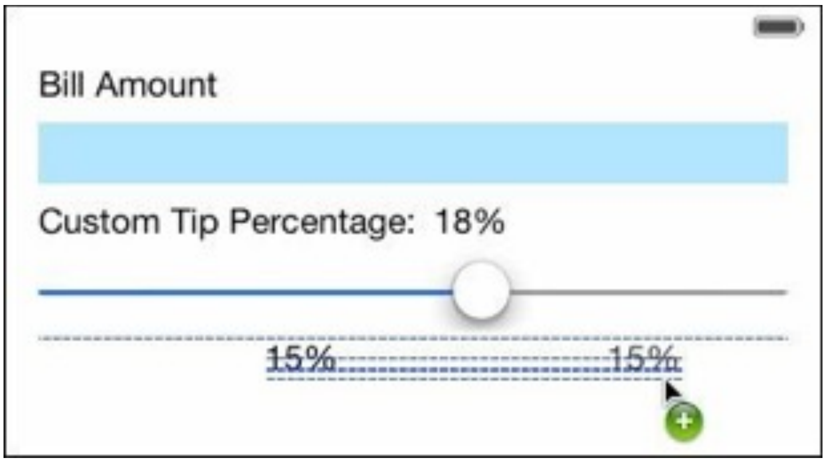
Step 5. Adding the "15%" and "18%" Labels

Next, you will add two more Labels containing the text 15% and 18% to serve as column headings for the calculation results. The app will update the "18%" Label when the user moves the Slider's thumb. Initially, you will position these Labels approximately—later you will position them more precisely.

Drag another Label onto the scene and use the blue guides to position it the recommended distance below the Slider, then set its Text to *15%* and its Alignment to centered.

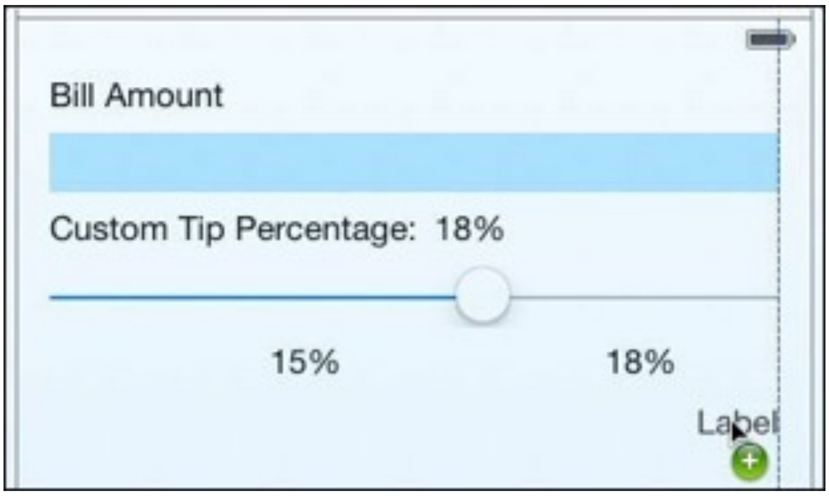


Next you will duplicate the "15%" Label, which copies all of its settings. Hold the option key and drag the "15%" Label to the right. You can also duplicate a UI component by selecting it and typing ⌘ + D, then moving the copy. Change the new Label's text to *18%*.



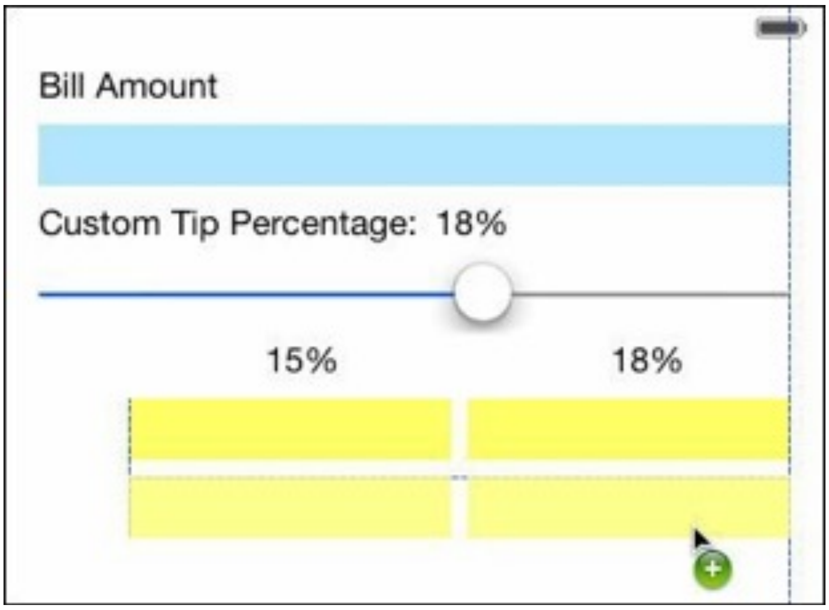
Step 6. Creating the Labels That Display the Tips and Totals

Next, you will add four Labels in which the app will display the calculation results. Drag a Label onto the UI until the blue guides appear as below.

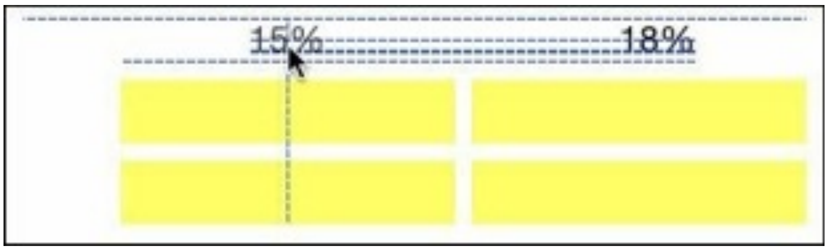


Drag the Label's bottom-center sizing handle until the Label's Height is 30, and drag its left-center sizing handle until the Label's Width is 156. Use the Attributes inspector to clear the Text attribute, set the Alignment so the text is centered and set the Background colour to Banana, which is located in the Colour dialog's Crayons tab in the second row from the bottom.

Next duplicate the yellow Label by holding the option key and dragging the Label to the left to create another Label below the "15%" Label. Select both yellow Labels by holding the Shift key and clicking each Label. Hold the option key and drag any one of the selected Labels down until the blue guides appear as shown below.

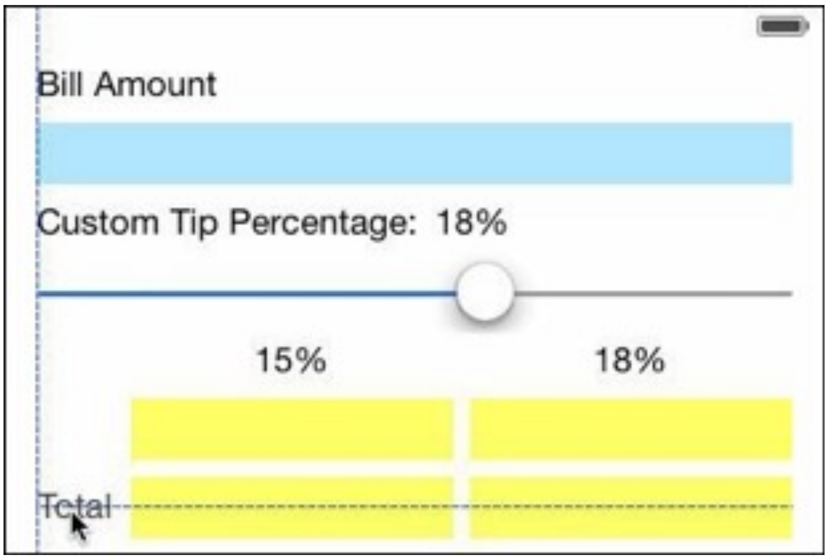


Now you can center the “15%” and “18%” Labels over their columns. Drag the “Tip” Label so that the blue guide lines appear as shown below. Repeat this for the “18%” Label to center it over the right column of yellow Labels.

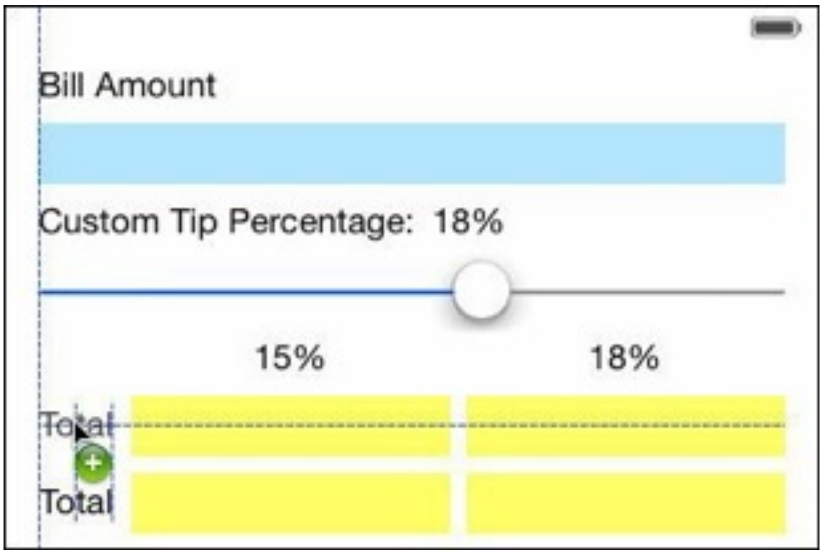


Step 7. Creating the “Tip” and “Total” Labels to the Left of the Yellow Labels

Drag a Label onto the scene, change its Text to *Total*, set its Alignment to right aligned and position it to the left of the second row of yellow Labels as below.



Hold the option key and drag the “Total” Label up until the blue guides appear as shown below. Change the new Label’s text to *Tip*, then drag it to the right so that the right edges of the “Tip” and “Total” Labels align.



Step 8. Creating the Text Field for Receiving User Input

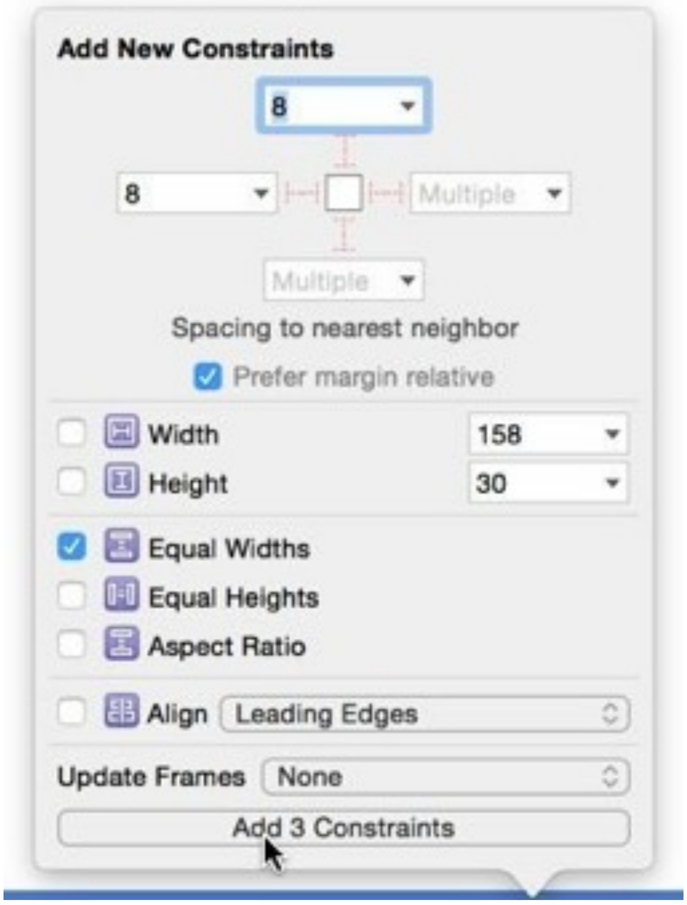
You will now create the Text Field that will receive the user input. Drag a Text Field from the Object library to the bottom edge of the scene, then use the Attributes inspector to set its Keyboard Type attribute to *Number Pad* and its Appearance to *Dark*. This Text Field will be *hidden* behind the numeric keypad when the app first loads. You will receive the user's input through this Text Field, then format and display it in the blue Label at the top of the scene.

Adding the Auto Layout Constraints

You've now completed the TipCalc app's basic UI design, but have not yet added any auto layout constraints. If you run the app in the simulator or on a device, you may notice that—depending on which simulator you use—some of the UI components extend beyond the trailing edge. To get a responsive UI, you must now add auto layout constraints so that the UI components can adjust and display properly on devices of various sizes and resolutions. You will use Interface Builder to add missing constraints automatically, then run the app again to see the results. You will then create some additional constraints so that the app displays correctly in the simulator or on a device.

Step 1. Setting the Yellow Labels to Have Equal Widths

Select all four yellow Labels by holding the shift key and clicking each one. In the auto layout tools at the bottom of the canvas, click the Pin tools icon (⊢◘⊣). Ensure that Equal Widths is checked and click the Add 3 Constraints button, as shown in below. Only three constraints are added, because three of the Labels will be set to have the same width as the fourth.


Add New Constraints

8

8 | Multiple

Multiple

Spacing to nearest neighbor

[✓] Prefer margin relative

[ ] Width 158

[ ] Height 30

[✓] Equal Widths

[ ] Equal Heights

[ ] Aspect Ratio

[ ] Align: Leading Edges

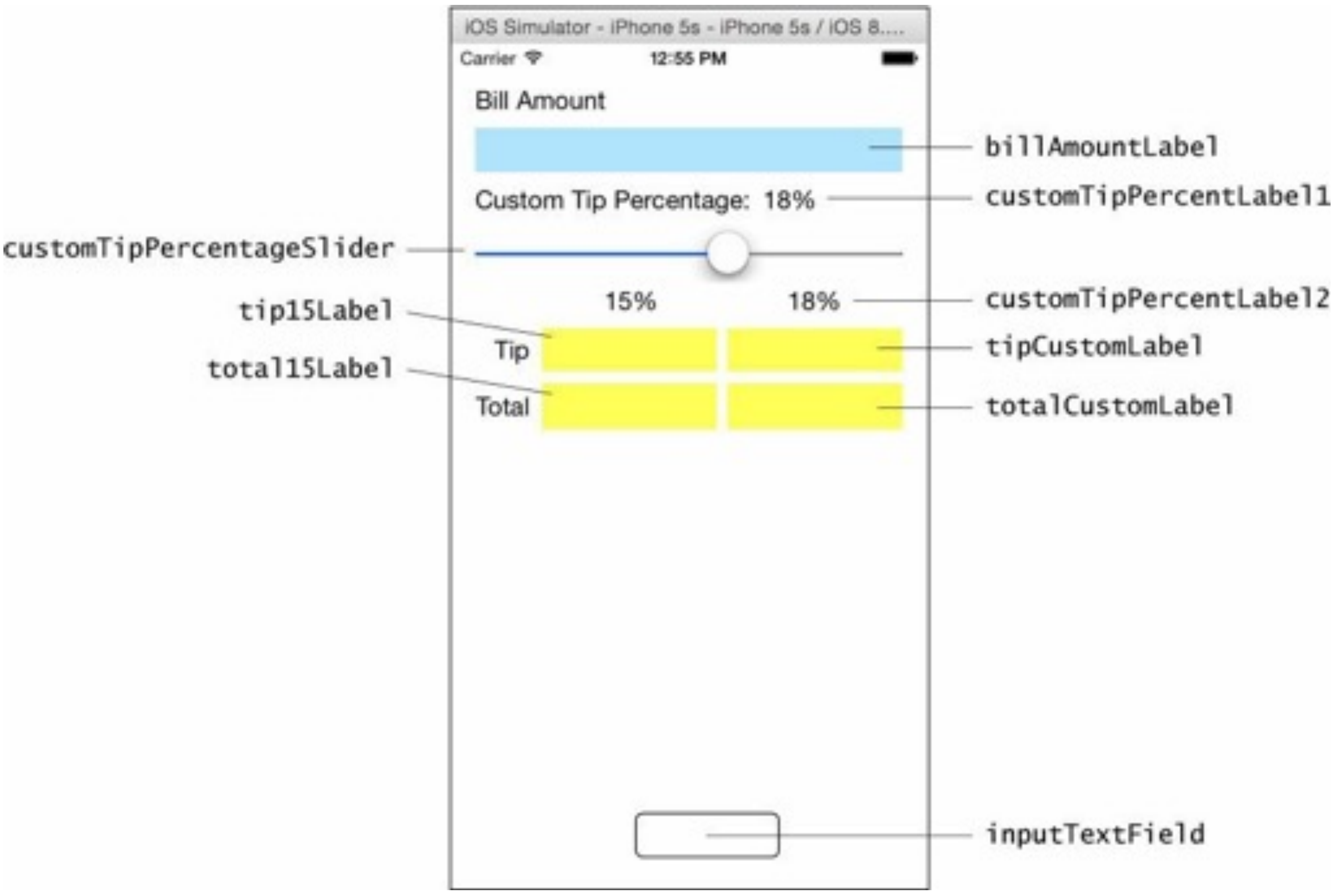
Update Frames: None

Step 2. Adding the Missing Auto Layout Constraints

To add the missing auto layout constraints, click the white background in the design area or select View in the document outline window. At the bottom of the canvas, click the Resolve Auto Layout Issues (|△|) button and under All Views in View Controller select Add Missing Constraints. Interface Builder analyses the UI components in the design and based on their sizes, locations and alignment, then creates a set of auto layout constraints for you. In some cases, these constraints will be enough for your design, but you will often need to tweak the results.

Part 2a – Creating Outlets with Interface Builder

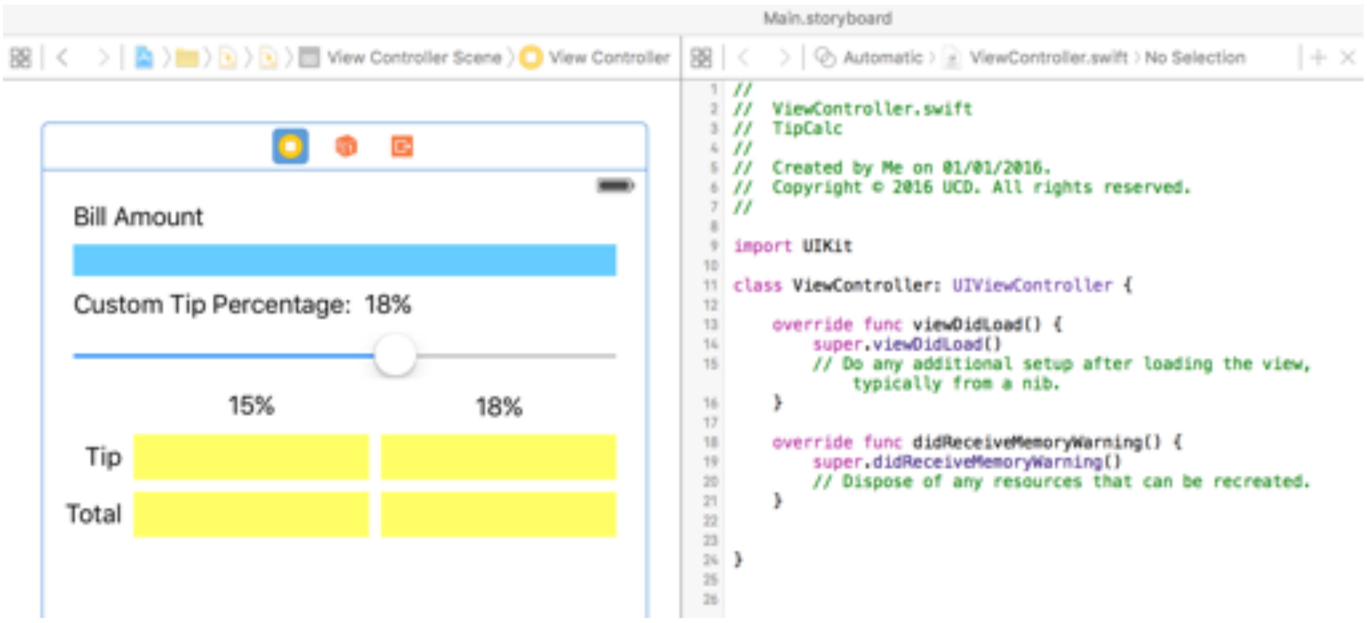
You will now use Interface Builder to create the outlets for the UI components that the app interacts with programmatically. The figure below shows the outlet names that have been specified when creating this app. A common naming convention is to use the UI component's class name without the UI class prefix at the end of an outlet property's name—for example, billAmountLabel rather than billAmountUILabel. Interface Builder makes it easy for you to create outlets for UI components by control dragging from the component into your source code. To do this, you will take advantage of the Xcode Assistant editor.



Opening the Assistant Editor

To create outlets, ensure that your scene's storyboard is displayed by selecting it in the Project navigator. Next, select the Assistant editor button (◎) on the Xcode toolbar (or select View > Assistant Editor > Show Assistant Editor). Xcode's Editor area splits and the file ViewController.swift is displayed to the right of the storyboard (see below).

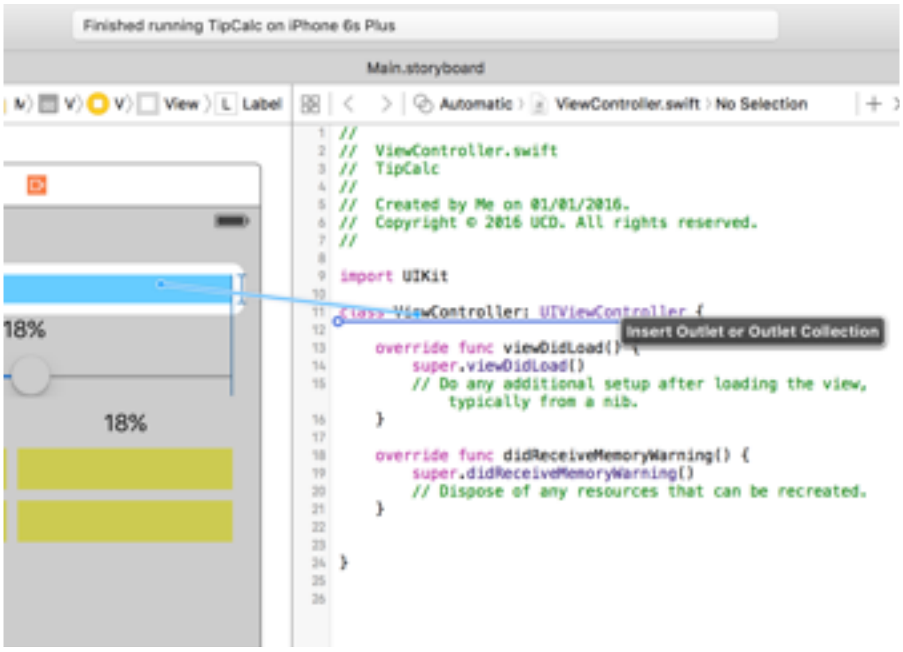
By default, when viewing a storyboard, the Assistant editor shows the corresponding view controller's source code. However, by clicking Automatic in the jump bar at the top of the Assistant editor, you can select from options for previewing the UI for different device sizes and orientations, previewing localized versions of the UI or viewing other files that you'd like to view side-by-side with the content currently displayed in the editor. The comments in lines 1–7 are autogenerated by Xcode. Delete the method didReceiveMemoryWarning in lines 18–21 as we will not use it in this app. We will discuss the details of ViewController.swift and add code to it later.



```
 1  //
 2  //  ViewController.swift
 3  //  TipCalc
 4  //
 5  //  Created by Me on 01/01/2016.
 6  //  Copyright © 2016 UCD. All rights reserved.
 7  //
 8
 9  import UIKit
10
11  class ViewController: UIViewController {
12
13      override func viewDidLoad() {
14          super.viewDidLoad()
15          // Do any additional setup after loading the view,
                typically from a nib.
16      }
17
18      override func didReceiveMemoryWarning() {
19          super.didReceiveMemoryWarning()
20          // Dispose of any resources that can be recreated.
21      }
22
23
24  }
25
26
```

Creating an Outlet

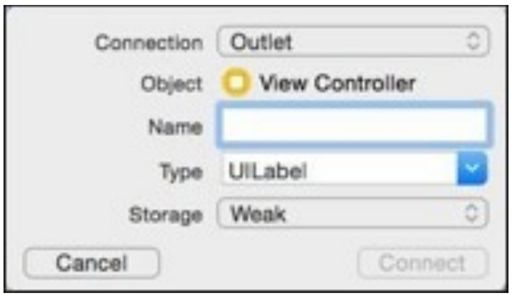
You will now create an outlet for the blue Label that displays the user's input. You need this outlet to programmatically change the Label's text to display the input in currency format. Outlets are declared as properties of a view controller class. To create the outlet, Control drag from the blue Label to below line 11 in **ViewController.swift** (see below) and release.


Finished running TipCalc on iPhone 6s Plus
Main.storyboard
View
Label
Automatic
ViewController.swift
No Selection
Insert Outlet or Outlet Collection
18%
18%

```
 1  //
 2  //  ViewController.swift
 3  //  TipCalc
 4  //
 5  //  Created by Me on 01/01/2016.
 6  //  Copyright © 2016 UCD. All rights reserved.
 7  //
 8
 9  import UIKit
10
11  class ViewController: UIViewController {
12
13      override func viewDidLoad() {
14          super.viewDidLoad()
15          // Do any additional setup after loading the view,
                typically from a nib.
16      }
17
18      override func didReceiveMemoryWarning() {
19          super.didReceiveMemoryWarning()
20          // Dispose of any resources that can be recreated.
21      }
22
23
24  }
25
26
```

This displays a popover for configuring the outlet. In the popover, ensure that Outlet is selected for the Connection type, specify the name *billAmountLabel* for the outlet's Name and click Connect. Xcode inserts the following property declaration in class ViewController and you can now use this property to programmatically modify the Label's text.

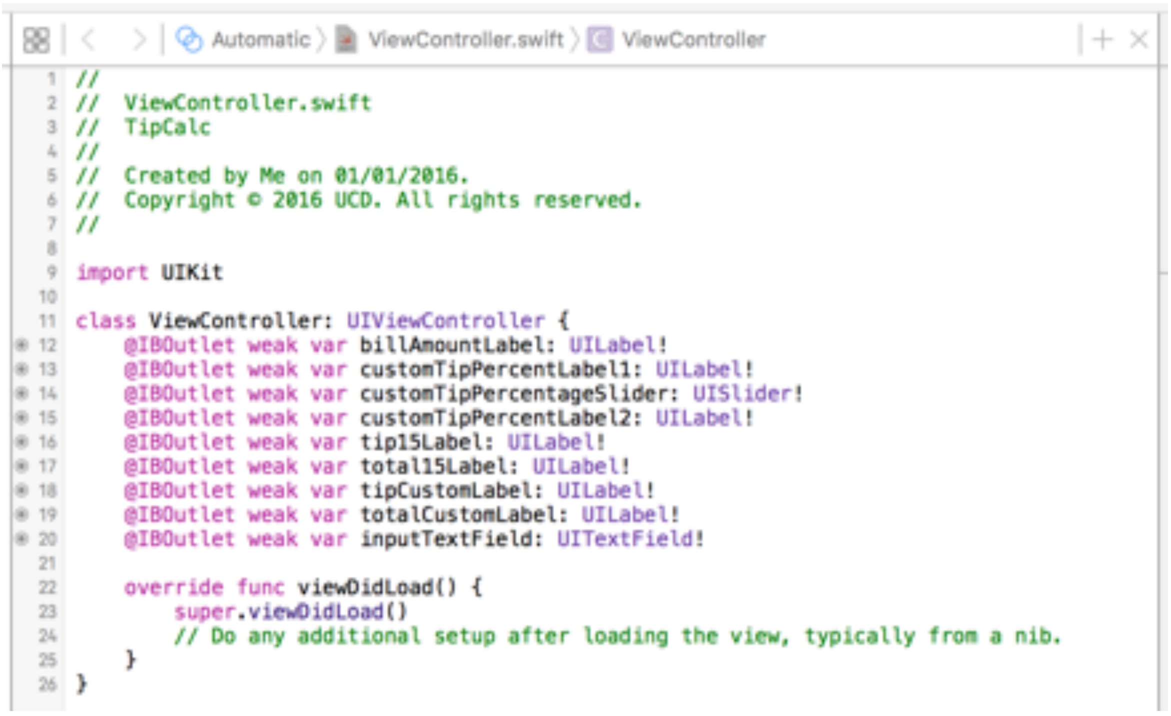
```
@IBOutlet weak var billAmountLabel: UILabel!
```



```
Connection  Outlet
Object      View Controller
Name
Type        UILabel
Storage     Weak
```

Creating the Other Outlets

Repeat the steps above to create outlets for the other labeled UI components. Your code should now appear as shown below. In the gray margin to the left of each outlet property is a small bullseye symbol indicating that the outlet is connected to a UI component. Hovering the mouse over that symbol highlights the connected UI component in the scene. You can use this to confirm that each outlet is connected properly.

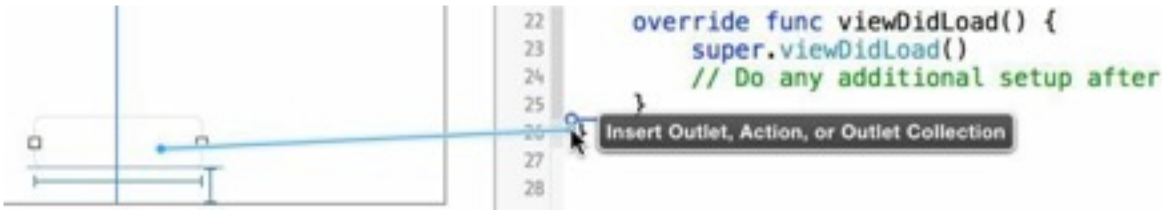


```swift
1  //
2  //  ViewController.swift
3  //  TipCalc
4  //
5  //  Created by Me on 01/01/2016.
6  //  Copyright © 2016 UCD. All rights reserved.
7  //
8
9  import UIKit
10
11 class ViewController: UIViewController {
12     @IBOutlet weak var billAmountLabel: UILabel!
13     @IBOutlet weak var customTipPercentLabel1: UILabel!
14     @IBOutlet weak var customTipPercentageSlider: UISlider!
15     @IBOutlet weak var customTipPercentLabel2: UILabel!
16     @IBOutlet weak var tip15Label: UILabel!
17     @IBOutlet weak var total15Label: UILabel!
18     @IBOutlet weak var tipCustomLabel: UILabel!
19     @IBOutlet weak var totalCustomLabel: UILabel!
20     @IBOutlet weak var inputTextField: UITextField!
21
22     override func viewDidLoad() {
23         super.viewDidLoad()
24         // Do any additional setup after loading the view, typically from a nib.
25     }
26 }
```

Part 2b – Creating Actions with Interface Builder

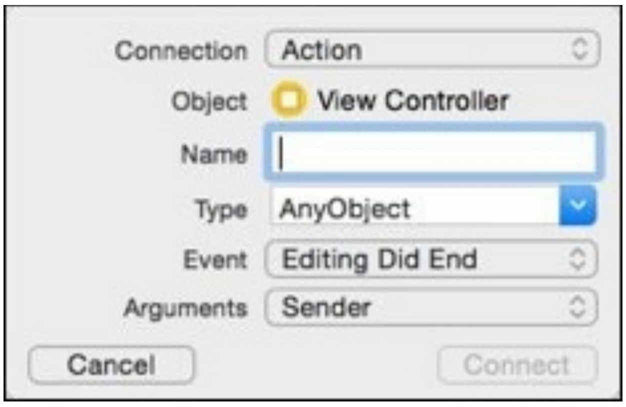
Now that you've created the outlets, you need to create actions (i.e. event handlers) that can respond to the user-interface events. A Text Field's *Editing Changed* event occurs every time the user changes the Text Field's contents. If you connect an action to the Text Field for this event, the Text Field will send a message to the view-controller object to execute the action each time the event occurs. Similarly, the *Value Changed* event repeatedly occurs for a Slider as the user moves the thumb. If you connect an action method to the Slider for this event, the Slider will send a message to the view controller to execute the action each time the event occurs.

In this app, you will create one action method that's called for each of these events. You will connect the Text Field and the Slider to this action using the Assistant editor. Control drag from the Text Field in the scene to ViewController.swift between the right braces at lines 25 and 26 (see below), then release.



```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after
}
```

This displays a popover for configuring an outlet. From the Connection list in the popover, select *Action* to display the options for configuring an action (see below).



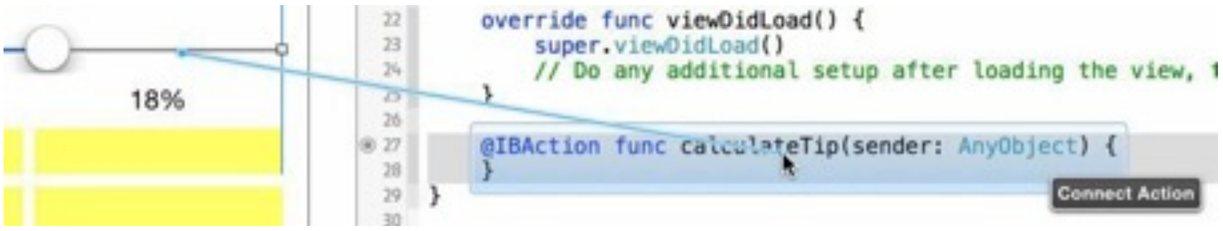
Connection	Action
Object	View Controller
Name	
Type	AnyObject
Event	Editing Did End
Arguments	Sender

In the popover, specify *calculateTip* for the action's Name, select *Editing Changed* for the Event and click Connect. Xcode inserts the following empty method definition in the code and displays a small bullseye symbol in the grey margin to the left of the method indicating that the action is now connected to a UI component. When the user edits the Text Field, a message will be sent to the ViewController object to message the *calculateTip* method. You will define the logic for this method shortly.

```
@IBAction func calculateTip(sender: AnyObject) {
}
```

Connecting the Slider to Method calculateTip

Recall that *calculateTip* should also be called as the user changes the custom tip percentage. You can simply connect the Slider to this existing action to handle the Slider's Value Changed event. To do so, select the Slider in the scene, then hold the control key and drag from the Slider to the *calculateTip* method and release (see below). This connects the Slider's *Value Changed* event to the action.

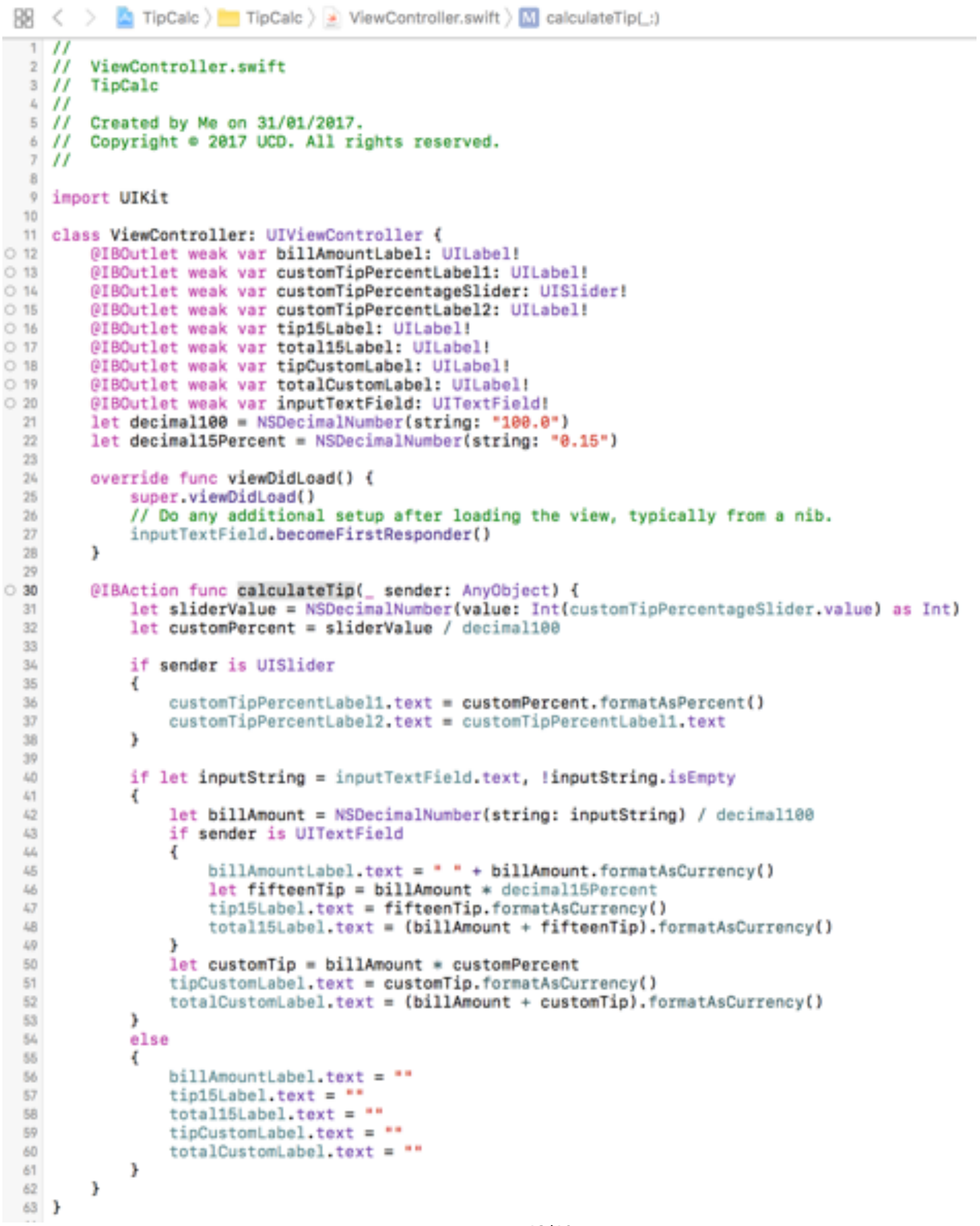


```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, t
}

@IBAction func calculateTip(sender: AnyObject) {
}
}
```

ViewController.swift contains the class implementation for the ViewController, including a utility function used throughout the class to format NSDecimalNumbers as currency and to perform calculations using NSDecimalNumber objects.

To use the iOS frameworks, you must import them into your Swift code. Throughout this app, we use the UIKit framework's UI component classes. Line 9 contains an import declaration indicating that the program uses features from the UIKit framework. All import declarations must appear before any other Swift code (except comments) in your source code files.



```
 1  //
 2  //  ViewController.swift
 3  //  TipCalc
 4  //
 5  //  Created by Me on 31/01/2017.
 6  //  Copyright © 2017 UCD. All rights reserved.
 7  //
 8
 9  import UIKit
10
11  class ViewController: UIViewController {
12      @IBOutlet weak var billAmountLabel: UILabel!
13      @IBOutlet weak var customTipPercentLabel1: UILabel!
14      @IBOutlet weak var customTipPercentageSlider: UISlider!
15      @IBOutlet weak var customTipPercentLabel2: UILabel!
16      @IBOutlet weak var tip15Label: UILabel!
17      @IBOutlet weak var total15Label: UILabel!
18      @IBOutlet weak var tipCustomLabel: UILabel!
19      @IBOutlet weak var totalCustomLabel: UILabel!
20      @IBOutlet weak var inputTextField: UITextField!
21      let decimal100 = NSDecimalNumber(string: "100.0")
22      let decimal15Percent = NSDecimalNumber(string: "0.15")
23
24      override func viewDidLoad() {
25          super.viewDidLoad()
26          // Do any additional setup after loading the view, typically from a nib.
27          inputTextField.becomeFirstResponder()
28      }
29
30      @IBAction func calculateTip(_ sender: AnyObject) {
31          let sliderValue = NSDecimalNumber(value: Int(customTipPercentageSlider.value) as Int)
32          let customPercent = sliderValue / decimal100
33
34          if sender is UISlider
35          {
36              customTipPercentLabel1.text = customPercent.formatAsPercent()
37              customTipPercentLabel2.text = customTipPercentLabel1.text
38          }
39
40          if let inputString = inputTextField.text, !inputString.isEmpty
41          {
42              let billAmount = NSDecimalNumber(string: inputString) / decimal100
43              if sender is UITextField
44              {
45                  billAmountLabel.text = " " + billAmount.formatAsCurrency()
46                  let fifteenTip = billAmount * decimal15Percent
47                  tip15Label.text = fifteenTip.formatAsCurrency()
48                  total15Label.text = (billAmount + fifteenTip).formatAsCurrency()
49              }
50              let customTip = billAmount * customPercent
51              tipCustomLabel.text = customTip.formatAsCurrency()
52              totalCustomLabel.text = (billAmount + customTip).formatAsCurrency()
53          }
54          else
55          {
56              billAmountLabel.text = ""
57              tip15Label.text = ""
58              total15Label.text = ""
59              tipCustomLabel.text = ""
60              totalCustomLabel.text = ""
61          }
62      }
63  }
```

ViewController Class Definition.

Line 11 was generated by the IDE when you created the project and begins with a class definition for class ViewController. The class keyword introduces a class definition and is immediately followed by the class name (ViewController). Class name identifiers use camelCase naming in which each word in the identifier begins with a capital letter. Class names (and other type names) begin with an initial uppercase letter and other identifiers begin with lowercase letters. Each new class you create becomes a new type that can be used to declare variables and create objects.

A left brace (at the end of line 11), {, begins the body of every class definition. A corresponding right brace (at line 63), }, ends each class definition. By convention, the contents of a class's body are indented.

The notation *: UIViewController* in line 11 indicates that class *ViewController* inherits from class *UIViewController*—the UIKit framework superclass of all view controllers. Inheritance is a form of software reuse in which a new class is created by absorbing an existing class's members and enhancing them with new or modified capabilities. This relationship indicates that a *ViewController* is a *UIViewController*. It also ensures that *ViewController* has the basic capabilities that iOS expects in all view controllers, including methods like *viewDidLoad* that help iOS manage a view controller's lifecycle. The class on the left of the : in line 11 is the subclass (derived class) and one on the right is the superclass (base class). Every scene has its own *UIViewController* subclass that defines the scene's event handlers and other logic. Unlike some object-oriented programming languages, Swift classes are not required to directly or indirectly inherit from a common superclass.

@IBOutlet Property Declarations

Lines 12 to 20 show the nine *@IBOutlet* property declarations of the *ViewController* class that were created by Interface Builder when you created the outlets in Part 2a. Typically, you will define a class's properties first followed by the class's methods, but this is not required.

The notation *@IBOutlet* indicates to Xcode that the property references a UI component in the app's storyboard. When a scene loads, the UI component objects are created, an object of the corresponding view-controller class is created and the connections between the view controller's outlet properties and the UI components are established. The connection information is stored in the storyboard. *@IBOutlet* properties are declared as variables using the var keyword, so that the storyboard can assign each UI component object's reference to the appropriate outlet once the UI components and view controller object are created.

Automatic Reference Counting (ARC) and Property Attributes

Swift manages the memory for your app's reference-type objects using Automatic Reference Counting (ARC), which keeps track of how many references there are to a given object. The runtime can remove an object from memory only when its reference count becomes 0.

Property attributes can specify whether a class maintains an ownership or non-ownership relationship with the referenced object. By default, properties in Swift create strong references to objects, indicating an ownership relationship. Every strong reference increments an object's reference count by 1. When a strong reference no longer refers to an object, its reference count decrements by 1. The code that manages incrementing and decrementing the reference counts is inserted by the Swift compiler.

The *@IBOutlet* properties are declared as weak references, because the view controller does not own the UI components—the view defined by the storyboard that created them does. A weak reference does not affect the object's reference count. A view controller does, however, have a strong reference to its view.

Type Annotations and Implicitly Unwrapped Optional Types

A type annotation specifies a variable's or constant's type. Type annotations are specified by following the variable's or constant's identifier with a colon (:) and a type name. For example, line 12 indicates that *billAmountLabel* is a *UILabel!*. The exclamation point indicates an implicitly unwrapped optional type and that variables of such types are initialized to nil by default. This allows the class to compile, because these *@IBOutlet* properties are initialized—they will be assigned actual UI component objects once the UI is created at runtime.

Other ViewController Properties

Lines 21 and 22 shows the class ViewController's other properties, which you should add below the *@IBOutlet* properties. Line 21 defines the constant *decimal100* that's initialized with an *NSDecimalNumber* object. Identifiers for Swift constants follow the same camelCase naming conventions as variables. Class *NSDecimalNumber* provides many initializers—this one receives a *String* parameter containing the initial value ("100.0"), then returns an *NSDecimalNumber* representing the corresponding numeric value. We will use decimal100 to calculate the custom tip percentage by dividing the slider's value by 100.0. We will also use it to divide the user's input by 100.0 for placing a decimal point in the bill amount that's displayed at the top of the app. Initializers are commonly called constructors in many other object-oriented programming languages. Line 22 defines the constant *decimal15Percent* that's initialized with an *NSDecimalNumber* object representing the value 0.15. We will use this to calculate the 15% tip.

When initialising an object in Swift, you must specify each parameter's name, followed by a colon (:) and the argument value. As you type your code, Xcode displays the parameter names for initializers and methods to help you write code quickly and correctly. Required parameter names in Swift are known as external parameter names. In your implementation, neither constant of lines 21 and 22 was declared with a type annotation. Like many popular languages, Swift has powerful type inference capabilities and can determine a constant's or variable's type from its initializer value. Swift infers from the initializers that both constants are *NSDecimalNumbers*.

Overridden UIViewController method viewDidLoad

Method *viewDidLoad*—which Xcode generated when it created class ViewController—is inherited from superclass *UIViewController*. You typically override it to define tasks that can be performed only after the view has been initialized. You should add lines (lines 24 to 28) to the method.

Note that in Swift, a method definition begins with the keyword *func* (line 25) followed by the function's name and parameter list enclosed in required parentheses, then the function's body enclosed in braces ({ and }). The parameter list optionally contains a comma-separated list of parameters with type annotations. This function does not receive any parameters, so its parameter list is empty. This method does not return a value, so it does not specify a return type. You can see methods on lines 65–74 (see further below) extending the existing *NSNumber* class with parameters and specifying a return type.

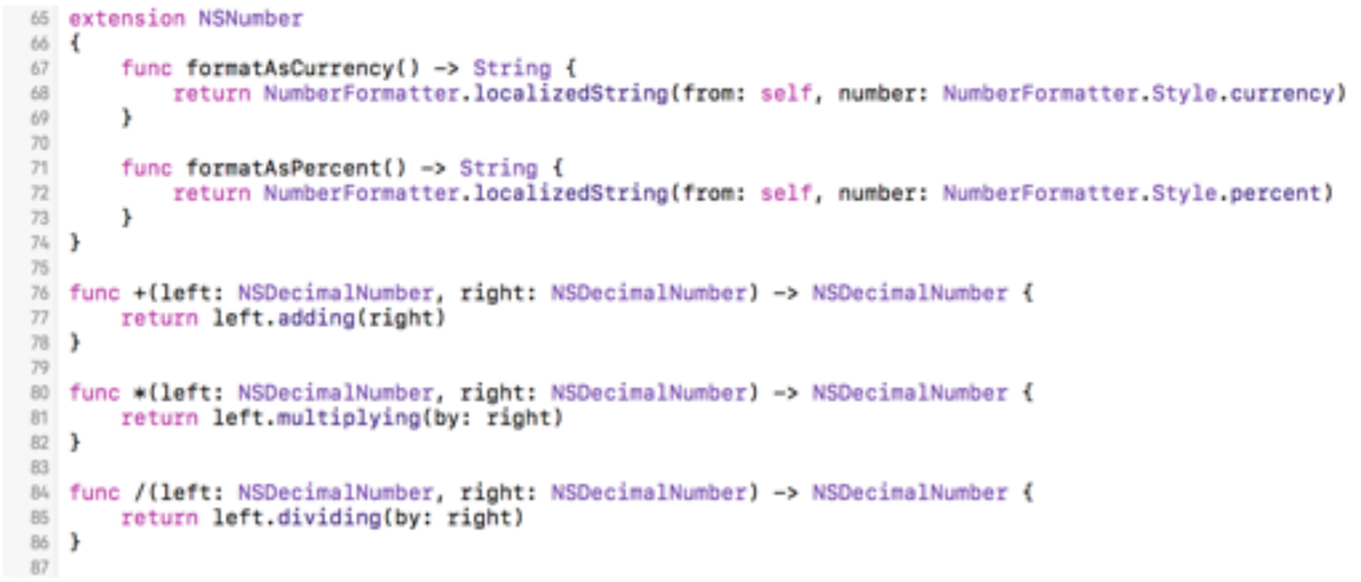
When overriding a superclass method, you declare it with keyword override preceding the keyword *func*, and the first statement in the method's body typically uses the super keyword to invoke the superclass's version of the method (line 24). The keyword super references the object of the class in which the method appears, but is used to access members inherited from the superclass.

Displaying the Numeric Keypad When the App Begins Executing

In this app, we want *inputTextField* to be the selected object when the app begins executing so that the numeric keypad is displayed immediately. To do this, we use property inputTextField to invoke the *UITextField* method *becomeFirstResponder*, which programmatically makes *inputTextField* the active component on the screen—as if the user touched it. You configured *inputTextField* such that when it's selected, the numeric keypad is displayed, so line 27 displays this keypad when the view loads.

ViewController Action Method calculateTip

Method *calculateTip* is the action (as specified by *@IBAction* on line 30) that responds to the Text Field's *Editing Changed* event **and** the Slider's *Value Changed* event. Add the code in lines 31–61 to the body of *calculateTip*. (If you are entering the Swift code as you read this, you will get errors on several statements that perform *NSDecimalNumber* calculations using overloaded operators that you will define on lines 76–86). The method takes one parameter. Each parameter's name must be declared with a type annotation specifying the parameter's type. When a view-controller object receives a message from a UI component, it also receives as an argument a reference to that component—the event's sender. Parameter sender's type —the Swift type *AnyObject*—represents any type of object and does not provide any information about the object. For this reason, the object's type must be determined at runtime. This dynamic typing is used for actions (i.e. event handlers), because many different types of objects can generate events. In action methods that respond to events from multiple UI components, the sender is often used to determine which UI component the user interacted with (as we do in lines 34 and 43).



```
65 extension NSNumber
66 {
67     func formatAsCurrency() -> String {
68         return NumberFormatter.localizedString(from: self, number: NumberFormatter.Style.currency)
69     }
70
71     func formatAsPercent() -> String {
72         return NumberFormatter.localizedString(from: self, number: NumberFormatter.Style.percent)
73     }
74 }
75
76 func +(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
77     return left.adding(right)
78 }
79
80 func *(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
81     return left.multiplying(by: right)
82 }
83
84 func /(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
85     return left.dividing(by: right)
86 }
87
```

Getting the Current Values of inputTextField and customTipPercentageSlider

Lines 31–32 get the *customTipPercentageSlider's* value property, which contains a *Float* value representing the Slider's thumb position (a value from 0 to 30, as specified in Interface Builder). The value is a *Float*, so we could get tip percentages like, 3.1, 15.245, etc. This app uses only whole-number tip percentages, so we convert the value to an Int before using it to initialize the *NSDecimalNumber* object that's assigned to local variable sliderValue. In this case, we use the *NSDecimalNumber* initializer that takes an *Int* value named integer.

Line 32 uses the overloaded division operator function that we define in lines 84–85 to divide *sliderValue* by 100 (decimal100). This creates an *NSDecimalNumber* representing the custom tip percentage that we will use in later calculations and that will be displayed as a locale- specific percentage *String* showing the current custom tip percentage.

Line 40 stores the value of inputTextField's text property—which contains the user's input— in the local *String* variable *inputString*—Swift infers type *String* because UITextField's text property is a *String*.

Updating the Custom Tip Percentage Labels When the Slider Value Changes

Lines 36–37 update *customTipPercentLabel1* and *customTipPercentLabel2* when the Slider value changes. Line 34 determines whether the sender is a *UISlider* object, meaning that the user interacted with the *customTipPercentageSlider*. The is operator returns *true* if an object's class is the same as, or has an is a (inheritance) relationship with, the class in the right operand.

We perform a similar test at line 43 to determine whether the user interacted with the *inputTextField*. Testing the sender argument like this enables you to perform different tasks, based on the component that caused the event.

Line 36 sets the *customTipPercentLabel1*'s text property to a locale-specific percentage *String* based on the device's current locale using a method defined lines 71–73. *NSNumberFormatter* class method *localizedStringFromNumber* returns a *String* representation of a formatted number. The method receives two arguments: the first is the *NSNumber* to format (Class *NSDecimalNumber* is a subclass of *NSNumber*, so you can use an *NSDecimalNumber* anywhere that an *NSNumber* is expected) ; the second argument (which has the external parameter name *numberStyle*) is a constant from the enumeration *NSNumberFormatterStyle* that represents the formatting to apply to the number—the *.PercentStyle* constant indicates that the number should be formatted as a percentage. Because the second argument must be of type *NSNumberFormatterStyle*, Swift can infer information about the method's argument. As such, it's possible to write the expression *NSNumberFormatterStyle.PercentStyle* with the shorthand notation: *.PercentStyle.*

Line 37 assigns the same string to *customTipPercentLabel2*'s text property.

Updating the Tip and Total Labels

Lines 42–52 update the tip and total Labels that display the calculation results. Line 43 uses the Swift *String* type's *isEmpty* property to ensure that *inputString* is not empty—that is, the user entered a bill amount. If so, lines 42–52 perform the tip and total calculations and update the corresponding Labels; otherwise, the *inputTextField* is empty and lines 56–60 clear all the tip and total labels and the *billAmountLabel* by assigning the empty *String* literal ("") to their text properties.

Lines 42 use *inputString* to initialize an *NSDecimalNumber*, then divide it by 100 to place the decimal point in the bill amount—for example, if the user enters 8808, the amount used for calculating tips and totals is 88.08.

Lines 42–52 execute only if the event's sender was a *UITextField*—that is, the user tapped keypad buttons to enter or remove a digit in this app's *inputTextField*. Line 45 displays the currency-formatted bill amount in *billAmountLabel* by calling the *formatAsCurrency* extension method of class *NSNumber* (defined lines 67–69). Line 46 calculates the 15% tip amount by using an overloaded multiplication operator function for *NSDecimalNumbers* (defined in lines 80–82). Then line 47 displays the currency-formatted value in the *tip15Label*. Next, line 48 calculates and displays the total amount for a 15% tip by using an overloaded addition operator function for *NSDecimalNumbers* (defined in lines 76–78) to perform the calculation, then passing the result to the *formatAsCurrency* function. Lines 50–52 calculate and display the custom tip and total amounts based on the custom tip percentage.

Extending the class NSNumber with new formatting behaviours

Lines 65–74 adds new methods *formatAsCurrency* and *formatAsPercent* to existing class *NSNumber*. A method definition begins with the keyword *func* followed by the method's name and parameter list enclosed in required parentheses, then the method's body enclosed in braces ({ and }). A method may also specify a return type by following the parameter list with -> and the type the method returns—this method returns a *String*. A method that does not specify a return type does not return a value—if you prefer to be explicit, you can specify the return type *Void*. A function with a return type uses a return statement (e.g. line 68) to pass a result back to its caller.

Lines 65–74 invoke *NSNumberFormatter* class method *localizedStringFromNumber*, which returns a locale-specific *String* representation of a number. This method receives as arguments the *NSNumber* to format—*formatAsCurrency's* number parameter—and a constant from the *NSNumberFormatterStyle* enum that specifies the formatting style—the constant *CurrencyStyle* specifies that a locale-specific currency format should be used. Once again, we could have specified the second argument as *.CurrencyStyle*, because Swift knows that the numberStyle parameter must be a constant from the *NSNumberFormatterStyle* enumeration and thus can infer the constant's type.

Defining Overloaded Operator Functions for Adding, Subtracting and Multiplying NSDecimalNumbers

Lines 76–86 create global functions that overload the addition (+), multiplication (*) and division (/) operators, respectively. Global functions (also called free functions or just functions) are defined outside a type definition (such as a class). These functions enable us to add two *NSDecimalNumbers* with the + operator, multiply two *NSDecimalNumbers* with the * operator, divide two *NSDecimalNumbers* with the / operator. Overloaded operator functions are defined like other global functions, but the function name is the symbol of the operator being overloaded. Each of these functions receives two *NSDecimalNumbers* representing the operator's left and right operands.

The addition (+) operator function returns the result of invoking *NSDecimalNumber* instance method *decimalNumberByAdding* on the left operand with the right operand as the method's argument—this adds the operands. The multiplication (*) and division (/) operator function invoke similar arithmetic methods on class *NSDecimalNumber*.

Since each of these *NSDecimalNumber* instance methods receives only one parameter, the parameter's name is not required in the method call. Unlike initializers and methods, a global function's parameter names are not external parameter names and are not required in function calls unless they are are explicitly defined as external parameter names in the function's definition.

For this assignment, testing will be done by running the project in the simulator. We will be looking at the following:

1. **Your project should build without errors or warnings.**
2. **Your project should run without crashing.**
3. Each of the sections in the assignment will be considered to verify that you've completed the assignment correctly.
4. Your project should have a clean user interface. User interface elements should be arranged logically, be aligned nicely , etc.
5. If your project doesn't build and run correctly, verify that you have the 'iPhone 6s' selected for the active scheme in the project window toolbar.

End of Assignment 2