TP 3 - Unit Converter

Adrien Humilière   28/03/2018

Part 1

- Create a new project, of the kind **Single View Application**.

- Using Interface Builder and the Object Library (⌥ + ⌘ + L), add a text label for the converted temperature.

- Adjust the auto layout constraints of the label. It should be at a fixed distance of the top, and horizontally centered.

- Using Interface Builder and the Object Library, add a Picker View to the bottom of the interface and adjust its constraints.

- Run the app (⌘ + R) and attempt to use the picker.

Part 2

- The picker view allow the user to pick a text or a date within defined choices. It uses the data source and delegate patterns.

- Using Interface Builder, set the main View Controller as the picker view datasource by Control-clicking on the picker view, and dragging a connection from the dataSource connection well to the View Controller in the Document Outline (▯).

- Run the app, observe the crash, and inspect the console output.

- The picker view's data source is the view controller, but the ViewController class does not yet implement the methods that conform to the `UIPickerViewDataSource` protocol.

- Using the Xcode Documentation and API Reference (⇧ + ⌘ + 0), explore the `UIPickerViewDataSource` Protocol Reference and the methods `numberOfComponentsInPickerView:` and `pickerView:numberOfRowsInComponent:`.

- Add the `UIPickerViewDataSource` protocol declaration to the controller class.

```
1  class ViewController: UIViewController, ↩
       UIPickerViewDataSource {
```

- Open the Issue Navigator (⌘ + 4), and notice the warnings indicating the methods necessary for conforming to the `UIPickerViewDataSource` protocol.

- Implement `numberOfComponentsInPickerView:` and `pickerView:numberOfRowsInComponent:`.

```
1  func numberOfComponentsInPickerView(pickerView: UIPickerView)↩
       -> Int {
2      return 1
3  }
4
5  func pickerView(pickerView: UIPickerView, ↩
     numberOfRowsInComponent
6      component: Int) -> Int {
7      return 10
8  }
```

- Run the app, and observe that the picker has one scrollable element that contains ten rows.

Part 3

- Observe how the picker view displays the ? character. Without a delegate to determine what to display, the picker view renders a ? by default.

- Using Interface Builder, set the main View Controller as the picker view delegate by Control-clicking the picker view, and dragging a connection from the delegate connection well to the View Controller in the Document Outline.

- Add the `UIPickerViewDelegate` protocol declaration to the controller class.

```
1  class ViewController: UIViewController, ↩
       UIPickerViewDataSource, UIPickerViewDelegate {
```

- Using the Xcode Documentation and API Reference, explore the `UIPickerViewDelegate` Protocol Reference and the methods `pickerView:titleForRow:forComponent:` and `pickerView:didSelectRow:inComponent:`.

- In the `ViewController` class, implement `pickerView:titleForRow:forComponent:`.

```
1  func pickerView(pickerView: UIPickerView, titleForRow row: ↩
       Int, forComponent component: Int) -> String? {
2      return "N Celsius degrees"
3      }
```
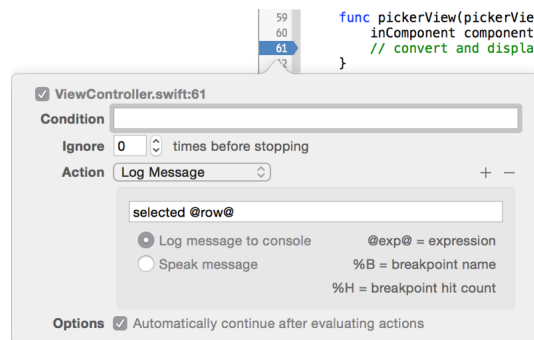
- In the `ViewController` class, implement `pickerView:didSelectRow:inComponent:`.

```
1  func pickerView(pickerView: UIPickerView, didSelectRow row: ↩
       Int, inComponent component: Int) {
2      // convert and display temperature
3      }
```

- Add a custom breakpoint to `pickerView:didSelectRow:inComponent:` that generates a Log message containing `selected:  @row@`.

- Run the app, observe the values displayed in the picker view, flick the picker to select a row, and observe the console message when the row is selected.

Part 4

- We want a "list" of negative and positive Celsius temperatures for the picker view to display, considering the total number of values (how many possible temperatures?), and the range (what minimum and maximum temperatures?).

- In the controller, add a private property for an `Array` of temperature values that the controller will provide to the picker view for display.

```
1  private var temperatureValues = [Int]()
```

- Implement a naive, temporary assignment of the `temperatureValues` property during `viewDidLoad`.

```
1  override func viewDidLoad() {
2      super.viewDidLoad()
3      temperatureValues = [1, 2, 3, 4, 5]
4  }
```

- Update the implementation of `pickerView:titleForRow:forComponent:`.

```
1  func pickerView(pickerView: UIPickerView, titleForRow row: ↩
       Int, forComponent component: Int) -> String? {
2      let celsiusValue = temperatureValues[row]
3      return "\(celsiusValue) celsius degrees"
4  }
```

- Run the app, observe the values displayed in the picker, and flick the picker one row at a time until the app crashes. Observe the console error.

- Update pickerView:titleForRow:forComponent: to use the size of the temperatureValues array to inform the picker of how many rows to display.

- Run the app, observe the temperature values, and interact with the picker.

- Instead of an explicit array initialization([-100, -99, ..., 99, 100]), a programmatic initialization using a loop will be used. Modify viewDidLoad to naively populate the temperatureValues array with a loop.

```
1  override func viewDidLoad() {
2      super.viewDidLoad()
3      let lowerBound = -100
4      let upperBound = 100
5      for index in lowerBound...upperBound {
6          temperatureValues.append(index)
7      }
8  }
```

- map might be used to transform a range into an array of Int values.

- Using the documentation of map, update the temperatureValues property declaration and remove the procedural temperature value generation from viewDidLoad.

- Run the app, and observe the temperature values in the picker.

Part 5

- We now want to convert and display the temperature when selected in the picker view.

- Using Interface Builder and the Assistant Editor (⌥+⌘+↵), create an outlet connection for the label as a controller property.

```
1  @IBOutlet weak var temperatureLabel: UILabel!
```

- Update the ViewController `pickerView:didSelectRow:inComponent:` method. It will perform the conversion $farenheit = 1.8 \times celsius + 32$ and update `temperatureLabel` with the value.

- Run the app, select a temperature with the picker, and observe the displayed temperature.

- We need to convert the converted temperature to an `Int` before updating the `UILabel` text property.

Part 6

- The design pattern MVC (Model – View – Controller) is widely used in iOS development. It helps separating code logic from display. In this project, we will try to create a model taking care of the temperature conversion, to extract this logic from the view controller.

- Add a new Swift class to the project for a UnitConverter model.

```swift
1  import Foundation
2
3  class UnitConverter {
4
5  }
```

- The temperature conversion code in the controller `pickerView:didSelectRow:inComponent:` method belongs in the model. Add a `degreesFahrenheit:` method to the `UnitConverter` class. It should take Celsius degrees in arguments (integers) and return Fahrenheit degrees (integers).

- In the `ViewController` class, declare a new private property for a `UnitConverter` object and affect `UnitConverter()`.

- Update the `pickerView:didSelectRow:inComponent:` method to use the `UnitConverter` `degreesFahrenheit:` method.

Part 7

- The range of temperature values has nothing to do with unit conversion, and therefore the UnitConverter model should not be responsible of generating a range of temperatures. We will create a "view model": a model object whose sole purpose is to serve the view.

- Add a new Swift class to the project for a `TemperatureRange` model.

- We will now establish a `TemperatureRange` object as the picker view's `dataSource`. `TemperatureRange` will have to adopt the `UIPickerViewDataSource` protocol, and the picker view's `dataSource` connection will have to change to a new TemperatureRange object.

- Using Interface Builder and the Object Library, drag an Object to the View Controller Scene in the Document Outline (▯). Rename the Object to `TemperatureRange`.
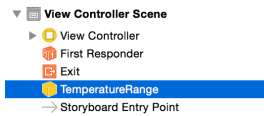
- With the `TemperatureRange` object selected, use the Identity Inspector (⌥ + ⌘ + 3) to set the Class to `TemperatureRange`.
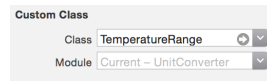
- Using Interface Builder, select the picker view and use the Connections Inspector (⌥ + ⌘ + 6) to delete the `dataSource` connection between the picker view and the controller. Drag a new connection from the picker view's dataSource to the TemperatureRange object in the Document Outline.

- Run the app, observe the crash, and inspect the error displayed in the console. `TemperatureRange` should implement `UIPickerViewDataSource`.

Part 8

- Change the `TemperatureRange` class import statement to provide access to the `UIPickerViewDataSource` type.

```
1  import UIKit
```

- Update the `TemperatureRange` class to inherit from `NSObject` and to adopt the `UIPickerViewDataSource` protocol. Remove the `UIPickerViewDataSource` protocol adoption from the `ViewController` class definition.

```
1  class TemperatureRange: NSObject, UIPickerViewDataSource
2  class ViewController: UIViewController, UIPickerViewDelegate
```

- Move the `temperatureValues` property out of the controller and into the `TemperatureRange` class. Remove the private access control modifier, and shorten its name to `values`.

```
1  let values = (-100...100).map { $0 }
```

- Move the controller methods `numberOfComponentsInPickerView:` and `pickerView:numberOfRowsInComponent:` into the `TemperatureRange` class, and replace the reference to `temperatureValues` with `values`.

```
1  func numberOfComponentsInPickerView(pickerView: UIPickerView)↩
       -> Int {
2      return 1
3  }
4
5  func pickerView(pickerView: UIPickerView,
6      numberOfRowsInComponent component: Int) -> Int {
7      return values.count
8  }
```

- Open the `ViewController` class, and observe the red error indicators.

- Using Interface Builder and the Assistant Editor, Control-drag an outlet connection from the `TemperatureRange` object to the controller class, to create a new property.

```
1  @IBOutlet var temperatureRange: TemperatureRange!
```

- Update the controller methods `pickerView:titleForRow:forComponent:` and `pickerView:didSelectRow:inComponent:` to use the new `temperatureRange` property, replacing references to `temperatureValues` with `temperatureRange.values`.

- Run the app, select a temperature, and observe the converted value.

- The remaining controller code only manages communication between the view and the models, and updates the view.

Part 9

- We want to improve the user experience when first starting the app. Notice the default starting temperature in the picker view, and consider how it affects the user experience. We might implement the behavior of specifying a default starting temperature.

- Using Interface Builder and the Assistant Editor, add the picker view as an outlet property within the `ViewController` class.

```
1  @IBOutlet weak var celsiusPicker: UIPickerView!
```

- Set the default selected temperature in `viewDidLoad` with the method `selectRow:inComponent:animated:`.

- Run the app and notice that, while the selected Celsius temperature has changed, the converted temperature label has not updated. Make sure the label is updated in `viewDidLoad`.

- Run the app, observe the default selected temperature in the picker, and observe the converted temperature label.

Part 10

- Run the app, select a temperature, background the app, then foreground the app. Notice how the last selected temperature is still displayed.

- Using the multitasking bar (⇧+⌘+H twice quickly), quit the app, then start the app. The app "forgets" the last selected temperature, and displays the default temperature in the picker view. We want the app to "remember" the last selected temperature, and to use that temperature when it starts, if a last-selected temperature is known.

- `NSUserDefaults` allow developers to save data in the filesystem as property list (.plist) files.

- Enhance `pickerView:didSelectRow:inComponent:` to save the picker view's selected row index.

```
1  let defaults = NSUserDefaults.standardUserDefaults()
2  defaults.setInteger(row, forKey: "defaultCelsiusPickerRow")
3  defaults.synchronize()
```

- The controller method `pickerView:didSelectRow:inComponent:` now has two responsibilities: updating the temperature label and saving the last-selected row. Extract the code for each respective task into two separate, well-named controller methods.

```
1  func displayConvertedTemperatureForRow(row: Int)
2  func saveSelectedRow(row: Int)
```

- Update `pickerView:didSelectRow:inComponent:` to call the two new methods.

- Extract the buried string into a constant placed near the top of the `ViewController` class. Use this constant in `saveSelectedRow:`.

```
1  let userDefaultsLastRowKey = "defaultCelsiusPickerRow"
```

- Run the app, select a temperature. Using the multitasking bar, quit the app, start the app again, and observe that, despite saving the last selected picker row, the default row is selected.

Part 11

- Refactor `viewDidLoad` to use an `initialPickerRow` method, instead of a local variable, to determine the initial selected row index of the picker view.

```
1  override func viewDidLoad() {
2      super.viewDidLoad()
3      let row = initialPickerRow()
4      celsiusPicker.selectRow(row, inComponent: 0, animated: ↩
   false)
5      pickerView(celsiusPicker, didSelectRow: row, inComponent:↩
   0)
6  }
7
8  func initialPickerRow() -> Int {
9      // load from user defaults
10     // if we obtained a last-known row index, return it
11     // otherwise, return the default.
12     return celsiusPicker.numberOfRowsInComponent(0) / 2
13 }
```

- Using the Xcode documentation and API Reference, explore the `NSUserDefaults integerForKey:` method, and observe how it returns `0` when a value for the provided key is not found.

- Implement a functional version of the `initialPickerRow` method, that fetch the corresponding data from `NSUserDefaults` and fallback to the default value if needed.

- Run the app, select a temperature, force quit the app via the multitasking bar, restart the app, and witness the last selected temperature is correctly displayed.