Stopwatch
Lesson 4

Description
Implement a naive updating of the elapsed time, to illustrate the nature of the application main run loop and the consequences of long-running operations on UI responsiveness.

Learning Outcomes
• Analyze application behavior and user expectations to define a requirements statement.
• Experiment with loops in interface event handlers and evaluate how long-running code can block a thread of execution.
• Infer how controller code can unintentionally hinder interface responsiveness.

Vocabulary

<table>
<thead>
<tr>
<th>computed property</th>
<th>accessor</th>
<th>outlet</th>
</tr>
</thead>
<tbody>
<tr>
<td>@IBOutlet</td>
<td>implicitly unwrapped optional</td>
<td>UILabel</td>
</tr>
<tr>
<td>blocking</td>
<td>run loop</td>
<td></td>
</tr>
</tbody>
</table>

Materials
• Stopwatch Lesson 4 Xcode project
• Run Loops and NSTimer presentation
Opening
How might we continuously display the elapsed time in the view?

Agenda

• Discuss the desired behavior of the elapsed time label, and what users expect to happen while the stopwatch is running.

• Consider, "While the stopwatch is running, the controller should update the elapsed time label."

• Modify the Stopwatch model, adding a computed property to indicate that the Stopwatch is running.

    ```swift
    var isRunning: Bool {
        return startTime != nil
    }
    ```

• Explain the shorthand read-only computed property syntax, the `Bool` data type, and how comparing the `startTime` optional property to `nil` can indicate if the Stopwatch is running.

• Using Interface Builder and the Assistant Editor (⌥⌘↩), add an outlet for the elapsed time label to the controller class.

    ```swift
    @IBOutlet weak var elapsedTimeLabel: UILabel!
    ```

• Explain the property declaration, the significance of the `@IBOutlet` attribute, and the implicitly unwrapped optional syntax.

• Using the Xcode Documentation and API Reference (⇧⌘0), explore the `UILabel` class reference.

• Add a naive implementation of `startButtonTapped:`.

    ```swift
    @IBAction func startButtonTapped(sender: UIButton) {
        print("Starting stopwatch")
        stopwatch.start()
        while stopwatch.isRunning {
            print("Updating...")
            elapsedTimeLabel.text = "\(stopwatch.elapsedTime)"
        }
    }
    ```

• Run the app (⌘R), tap the Start button, observe the console (⇧⌘C), and notice how the Start button remains tapped.

• Discuss how `startButtonTapped:` never returns, preventing the view from being visibly updated.
• Present the concept of run loops, and how long-running tasks can block the interface responsiveness.

Closing
Have you experienced using apps that lose their interface responsiveness?

Modifications and Extensions
• Convert the `isRunning` computed property to a stored property, and implement your own custom accessor method. Critique the benefits and drawbacks of both approaches.

Resources