Week 2
(Module 2)

CS 5254

Architecture overview

- Event-driven Graphical User Interface (GUI) code can become overly complicated and inefficient
  - We're going to begin organizing our code into meaningful distinct components
    - This won't be formal yet, but we'll still consider separation of interests as a best practice
- The Model-View-Controller (**MVC**) architecture – or any of its many variants – is useful here
  - This applies to desktop, web, mobile, and most other similar GUI applications
- **Model**
  - Maintains the current state of the system, without any regard for the GUI-related components
- **View**
  - Represents the layout and various components that comprise the graphical interface
- **Controller**
  - Responds to user interactions with the View
    - Might just change the View, perhaps to display other aspects of the Model
    - Might need to change the state of the model, in which case it must:
      - Mutate the Model according to the user interactions, then...
      - ...update the View based entirely upon the updated Model
    - Note that the separation and sequence above are particularly important in Android
      - The Controller shouldn't ever update the View directly
      - The Model shouldn't ever hold references to the View
        - However, the Model should have everything needed to reconstitute the View at any time

Some useful Kotlin features of note

- In Java `if` structures are always statements; in Kotlin `if-else` is actually an expression
  - Returns the last (or only) expression from the true or false block as applicable
  - You can still use `if-else` as a standalone statement
- Iterable objects generally directly support functional-style streaming
  - This is very similar to the `stream()` method of Java 8+ collections
  - Lambda expressions are surrounded by braces, not parentheses, with default parameter `it`
- Useful stream-based functions for the current project:
  - `filter` retains only the elements that match the specified predicate
    - `filterNot` retains only the elements that don't match the predicate
  - `take` retains the first specified number of elements, dropping the remainder
    - `takeLast` retains the last specified number of elements
  - `forEach` is a convenient alternative to for-loops, acting on each element of a stream
  - `zip` combines this stream with another list, to become a stream of `Pair` objects
    - One element is taken from each list, until either runs out of elements
    - The `Pair` can be used as-is, with `first` and `second` as the two individual components...
      - `{ /* some lambda expression involving it.first and it.second */ }`
    - ...or we can destructure the components by specifying parameter names in parentheses
      - `{ (alpha, beta) -> /* some lambda expression involving alpha and beta */ }`

Hints and tips for Project 1A

- Take care when copying classes from the textbook and/or assignment page
  - Please confirm that you copy the whole class content
    - It's easy to accidentally end up with two classes in one file
    - Kotlin also allows (without any warning) a class name that doesn't match the file name
- Keep in mind the important MVC separation of concerns noted earlier
  - This will become crucial for the next parts of this project (and it's a good practice anyway)
    - Each listener should update only the Answer first, then...
    - ...call a function to update the views, based purely on the updated Answer objects