Assignment 5

Due Monday, November 8, 11:30 PM
50 points

Purpose

The purpose of this assignment is to

- use string resources
- experiment with different code patterns for listeners

Background

In this assignment, you will create a simple word guessing game. When the user presses the start button, the code will pick a random four-letter word and the user will try to guess it. The user will have the option of giving up, in which case the app will mock them when it tells them what word they were trying to guess.

Instructions

Create a new project named **Assignment 5**. Your code and project should be set up and work according to the following requirements.

1. App label

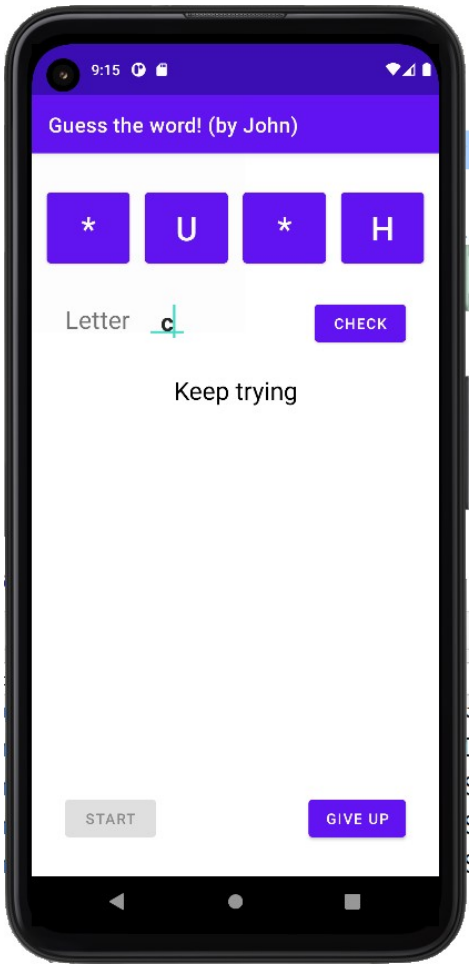
Edit **`app/res/values/string.xml`** and change the label to say **Guess the word! (by <name>)**, where the name is your first name. For example, I changed mine to **Guess the word! (by John)**.

2. Launcher icon

Create a custom launcher icon to replace the default icon created by Android Studio when the project is first created.

3. Design and layout

Place all the widgets in a **`ConstraintLayout`**. It should look similar to the image.



A game in progress. The four buttons in a row are The user just did a **Check**. Two letters are correct, the **U** and the **H**. The **`TextView`** has the message that says the user needs to keep guessing letters.

The user has just typed **c** in the **`EditText`** and is trying to decide which button with an asterisk to push. The **Start** button is disabled, since a game is in progress.

4. Code details

Follow these requirements in your code:

- Open the file **word-code-for-assigment.txt**, which is included with this assignment. It is a predefined list of 500 common four-letter words used on Wikipedia (source Ilya Semenov's wikipedia-word-frequency project). Copy the complete contents of the file and place it in your code. I suggest you place it at the end of your file. All the words in the list have four letters and all the words are in lowercase.
- Except for the exceptions below, all strings used in the Kotlin code should be accessed as string resources. The exceptions to this are:
  - The strings in the list of four-letter words from **word-code-for-assigment.txt**. Leave these as strings.
  - Strings used as the tags for **`Log`** output.
  - Any variable that is a string and is declared as a property. That is, any variable defined where the **`binding`** variable is declared. For example, this is ok:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    private var word = ""  // word is a property.
                           // Ok to use a string literal here

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
```

  Note: this applies to string literals only. If you use character literals, **`char`** in Java, don't define those as string resources.
- Messages in the **`TextView`** used to display information should be color coded.
  - If the message is a warning message for when the user typed something incorrect in the **`EditText`**, the message should be red.
  - When the user wins, the message should be blue.
  - Any other message, the message should be black.
- All buttons will need a listener. You can define the listeners any way you want except you cannot use the pattern we have been using up until where you place **`View.OnClickListener`** on the same line as **`class MainActivity`**. In other words, you cannot have this in your code:

```
// INCORRECT - don't do this
class MainActivity : AppCompatActivity(), View.OnClickListener {
```

  That line must be declared like this:

```
// CORRECT - do this
class MainActivity : AppCompatActivity() {
```

  Use one or more of the patterns covered in *Module 17 Listener Code Patterns*
- When the app first starts,

    - the **`EditText`** and all the buttons except the **Start** button should be disabled. The **Start** button should be the only one that is enabled. (See Help and Hints)
    - The **`TextView`** for messages should have a welcome message.

- When the user presses the **Start** button to start a game,
    - The **`Start`** button should be disabled. All the other buttons and the **`EditText`** should be enabled.
    - The **`TextView`** for messages should have a message indicating a new game has started.
    - The four buttons at the top should have an asterisk * as their text.
    - Initialize any variables in your code, to indicate the start of a new game. For example, the code should pick a random word from the list of words.

- When user presses the **Give up** button,
    - The **`EditText`** and all the buttons except the **Start** button should be disabled. The **Start** button should be the only one that is enabled.
    - The **`TextView`** for messages should have a message that mocks the user for giving up (in a nice way!). It should tell the user what the word was and how many letters the user tried before giving up.

- When trying to guess a word, the user will type a letter in the **`EditText`**. They will then poke one of the four buttons at the top. When the user pokes one of those buttons,
    - If the **`EditText`** is empty or only contains spaces, show an appropriate warning in the **`TextView`** for message.
    - If the **`EditText`** has more than two characters, show an appropriate warning in the **`TextView`** for message.
    - If the **`EditText`** has one character but it is not a letter, show an appropriate warning in the **`TextView`** for message.
    - If the **`EditText`** has exactly one character that is a letter,
        - Copy the letter and make it the letter in the button.
        - Clear the **`EditText`**
        - Increment the count that keeps track of how many times the user guessed a letter.
        - Clear any messages in the **`TextView`** for message

- When the user presses the **Check** button,
    - The code should compare the word created by the text in the buttons to the word that they are trying to guess letter by letter.
        - If a letter is correct, keep the letter in the button's text.
        - If a letter is incorrect, change the button's text back to an asterisk *.
    - If one or more of the letters are incorrect, show a "keep trying"-type message in the **`TextView`**.
    - If all four letters are correct and match the secret word,
        - Display a "You win"-type message in the **`TextView`** for messages. The message should include how many letters the user tried before getter the word right.

- The **`EditText`** and all the buttons except the **Start** button should be disabled. The **Start** button should be the only one that is enabled.

Sample game play

In order to help you understand how the game should work, a video is included with the assignment that walks you through how the app should work.

Help and Hints

These are some hints to help. Depending on your approach to the app, you may or may not need to use all this information.

Local variables vs properties

You will not be able to successfully complete this program by only using local variables. You will need one or more properties (Java calls them instance variables). As an example, the **`binding`** variable, which we have used since we started the class, is a property. In general, other than **`binding`**, your other properties should not be declared with **`lateinit`** unless you know what that means. Even then, don't do it. Make all your properties **`private`**.

Strings and characters

Kotlin and Java strings and characters are similar. String literals are defined by double quotes, e.g. **`"hello"`** . Character literals are defined by single quotes, e.g., **`'h'`**.

In Kotlin, if you want to check a character at a position in a string, treat the string like an array. Use square brackets and an index to get the character you want.

Once a string is created, you cannot change individual letters. Java has this same restriction.

```
var myString = "hello"
val ch = myString[0]  // ch would be assigned the character value 'h'

myString[0] = 'H' // ERROR: would not compile.
                  // Individual letters cannot be changed in strings.
```

Random integers

Choosing a random integer can be done one of two ways:

- This assigns a random integer to **`num`** in the range from 0 up to and including 10.

  ```
  val num = (0..10).random()
  ```

- This assigns a random integer to **`num`** in the range from 0 up to and including 9.

  ```
  val num = (0 until 10).random()
  ```

Lists with `listOf()` and `mutableListOf()`

You create lists with both **`listOf()`** and **`mutableListOf()`**. You can only read values in lists created with **`listOf()`**. You can read and write values in lists created with **`mutableListOf()`**. Once created, you can figure out how many elements are in a list using **`size`**. You can access elements in the lists using square brackets and an index, similar to arrays in Java.

For example, the example below creates a immutable (can't change it) list. The last statement is an error.

```
val data = listOf("Apple", "Banana", "Pear")

Log.i("FRUIT COUNT", data.size) // would output 3
Log.i("FRUIT", data[0])         // would output "Apple"

data[0] = "Chocolate" // ERROR: would not compile
                      // Cannot change a list created with listOf()
```

In this example of a mutable (can change it) list, we can change individual elements.

```
val guess = mutableListOf('w', 'o', 'r', 'd')

Log.i("LETTER COUNT", guess.size) // would output 4
Log.i("LETTER", guess[0])         // would output 'w'
Log.i("WORD  ", guess.toString()) // would output "[w, o, r, d]"

data[3] = 'k' // Changes the last letter from 'd' to 'k'.
              // OK to change a list created with mutableListOf()
Log.i("WORD  ", guess.toString()) // would output "[w, o, r, k]"
```

Enabling and disabling widgets in Kotlin

Widgets like **`Button`** and **`EditText`** can be enabled and disabled. When widgets are enabled, the user can poke them and their listeners will be executed. When widgets are disabled, the user can poke them but nothing will happen because the listeners will not be called. Enabling and disabling widgets may change their color in order to give the user a visual clue about the state of the widget.

Both enabling and disabling a widget is done with **`isEnabled`**. Set it to **`true`** to enable the widget, and set it to **`false`** to disable the widget. For example, if you had a **`Button`** with the id **`button_cancel`**, you could disable it in Kotlin like this:

```
binding.buttonCancel.isEnabled = false
```

If you decide to re-enable it in another part of code, you could write:

```
binding.buttonCancel.isEnabled = true
```

Changing text color

To change text color in Kotlin, use the function **`setTextColor()`**. For now, give it a constant defined in the **`Color`** class. For example, if you had a **`TextView`** with the id **`textview_message`**, you could change the color of the text to red like this:

```
binding.textviewMessage.setTextColor(Color.RED)
```

String resources

Some strings are simple. Just a string. For example

```
<string name="greeting">Hello there, stranger</string>
```

To get this string in Kotlin, you can do this:

```
val message = getResources().getString(R.string.greeting)
binding.someTextView.setText(message)
```

Since this is a simple string, you can use the resource id directly in the call to **`setText()`**

```
binding.someTextView.setText(R.string.greeting)
```

Other strings need information. For example, if we want to turn this into code that uses a string resource:

```
val name = "Mr. Smith"
val message = "Welcome ${name}. How are you?"
binding.someTextView.setText(message)
```

We have to use a place holder, **`%s`**, in the resource file

```
<string name="welcome">Welcome, %s. How are you?</string>
```

Then in Kotlin, we have to use the **`getResources()`** approach. The trick is to add the variable we want to replace the placeholder with as an argument to **`getString()`**. For example:

```
val name = "Mr. Smith"
val message = getResources().getString(R.string.welcome, name)
binding.someTextView.setText(message)
```

You can have more than one placeholder in a string resource. For example, if we want to turn this into a string resource:

```
val name = "Bob"
val amount = 20
val message = "Dear ${name}, you owe ${amount} dollars."
binding.someTextView.setText(message)
```

You could create the string resource like this:

```
<string name="bill">Dear %s, you owe %s dollars.</string>
```

Then in **`getString()`**, you add a variable for each **`%s`**. They should be in the order you need them. For example:

```
val name = "Bob"
val amount = 20
val message = getResources().getString(R.string.bill, name, amount)
binding.someTextView.setText(message)
```

Submission

There one item to turn in. Follow the instructions in **Module 4 Tools overview** on how to submit assignments. Upload the ZIP for your Kotlin Practice project. It should be uploaded to the Assignment 5 DropBox in D2L, which is in **Tasks > Assignments**.

If you have any problems uploading the ZIP to D2L, email me immediately and give me any error messages the D2L gives you. Do not email me your project unless I specifically ask you to do so.