Penn State University
CMPSC 475

3: Arithmetic Exercises

Due: Noon, 1st February

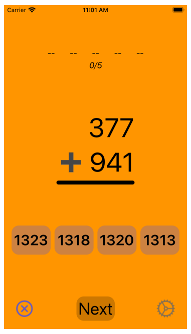
Spring 2022

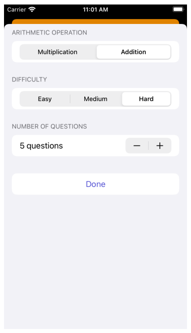
Description

In this assignment you will add preference settings to provide some options to your game. *This is a challenging assignment. Start early!*

Your app should add the following functionality:

1. *Reset function:* Add a reset button that will return the app to its initial state. No math problem should be displayed. This button should always be enabled
2. *Preferences:* Add a preference button that brings up a preference sheet. The button should only be enabled when the game is in its initial state, with no problem displayed. The user should be able to set three preferences:
    a. Difficulty (easy, medium, hard)
    b. Operation (addition, multiplication)
    c. Number of questions per round (an integer between 3 and 7, inclusive)





ARITHMETIC OPERATION

Multiplication [✓ Addition]

DIFFICULTY

Easy Medium [✓ Hard]

NUMBER OF QUESTIONS

5 questions − +

Done

The difficulty preference determines the range of possible operands as follows:

	Addition	Multiplication
Easy	Between 5 and 10	Between 5 and 10
Medium	Between 7 and 99	Between 7 and 15
Hard	Between 50 and 999	Between 12 and 30

3. The app should display math problems consistent with the current preferences.

Walk Through

For this assignment we'll again provide you with plenty of guidance on how to organize your code and what steps to take. As we progress through the course we'll provide less guidance and leave more design decisions up to you. This process is all in preparation for your final project where everything is up to you.

1. **The Model:** Modify your model to add support for the preferences. Think about what data your app needs to represent the state of the game. Define a struct (if you didn't in the previous assignment) that contains all this data. Think about what information the view model can maintain (computed property) based on this data. Keep the model dumb.
    a. Both the operation and difficulty should be represented by enums. I recommend including a preferences struct as a model property. This struct contains all the preferences. A binding to this struct can be passed to your preference view to ensure *Single Source of Truth*.

b. Implement a function for creating a new math problem using the current preferences.

c. You may be tempted to include all the operand ranges as part of the model. Resist this temptation. Let the view model maintain that logic and tell a math problem (upon initialization) what range it should use. Keep the model dumb!

2. The Views

2.1. Create a new SwiftUI View for your preferences and add a `Form`. Use `Pickers` to allow the user to modify your enum preferences. Use a `Stepper` to allow the user to modify the number of questions per round. Include a button to dismiss the view. See Hints section below for more guidance.

2.2. Enable the appearance and removal of your preferences view by adding a button and a sheet modifier to your main view. Enable the preferences button only when the game is in its initial state and not displaying a math problem.

2.3. Add a button to immediately return the game to its initial state.

3. The View Model

1. Initialize your new model with appropriate values (e.g., initial preferences. See Hints.)
2. Implement an intent (function) for resetting the game.
3. Add any published or computed properties for your views. (Perhaps a property with the arithmetic symbol to be displayed). Remember that a new value for a computed property will not automatically trigger views to be redrawn. Convince yourself that any computed properties used by your views only change values when a published property has also changed value.
4. Provide a way to tell the model what operand range to use when creating a new problem. A computed property based on current model preferences might work well here.
5. If your view uses strings to indicate problem status (mine uses unanswered (--), correct (✅) and incorrect (❌)), convince yourself that the view model is the only place for these to be defined. The view model can then tell the model to use these, but all the model knows is that it's been given symbols (strings) it's supposed to provide to the view when it asks for them. Neither the model nor the view should know/care what they are.

4. Clean up your project

4.1. Remember to manage constants in appropriate `Constants` structs.

4.2. Rename any types whose definitions no longer fit. For example, `multiplier` and `multiplicand` are no longer appropriate names if they will also store operands for addition problems.

4.3. Separate views into files as you see fit.

4.4. Organize your project into groups for the different parts of your app architecture.

Testing

Testing of the application is imperative. Be sure that all the required features have been implemented.

1. Your project should build without errors or warnings.
2. Make sure your app doesn't crash!
3. Check each of the requirements in the Description section above.

Hints

1. A range of numbers can be represented as a `Range` or `ClosedRange`. This will be helpful when implementing difficulty in your problems.
2. Use computed properties and functions in your view model to transform your model data into things your views can use.
3. See the discussion section of **`the SwiftUI .tag documentation`** to see how to render the body of your picker with an enum.
4. You may want to use the `.pickerStyle(SegmentedPickerStyle())` modifier on your pickers.
5. Your pickers should employ the *Single Source of Truth* principle. They should use bindings to the preference value in the model. Be impressed with how simple this makes your code and ensures data consistency.
6. The String type has a computed property `capitalized`.
7. View the documentation to find appropriate initializers for views. For the Stepper, you want to provide a range of allowed values.
8. Your Stepper view won't need to take any actions. Its only purpose is to update a preference value. Using the *Single Source of Truth* principle again makes this so easy.
9. You can use **`SFSymbols`** with the `Image(systemName:)` initializer to create nice image icons for use in buttons.
10. In your preferences struct you might find it useful to declare a static property called `default` that is an instance of the struct containing initial/default values.

Troubleshooting

Assignments may also have a section for troubleshooting to call out common pitfalls or areas to watch out for.

1. Be sure enum types conform to any necessary protocols. Strange errors can appear elsewhere if they don't.
2. Remember to have your view model conform to `ObservableObject`.
3. Set breakpoints using the debugger to verify that control is reaching certain parts of your code, to examine values at that point, etc. Once you set a breakpoint by clicking on the line number (if Xcode is not showing line numbers you can enable this feature under the Text Editing tab of Preferences) you can edit it by Control-clicking on the breakpoint. You can add an action such as logging a message (a string) or a debugger command. The two most common commands are `p` and `po` (print and print object). Follow these by a variable name. You can also set an option to have the breakpoint automatically continue after it's hit. Effective use of breakpoints is an essential survival skill in this class. We don't put print statements in our code. *Respect the Craft!*

Submission

Prepare your project for submission:

1. Remove all breakpoints, print statements and any other cruft.
2. Make sure your project compiles cleanly from scratch: choose **Clean** from the Product menu in Xcode, then Build and Run your program. Be sure to remove all warnings.
3. Check that all resources were copied into your project folder.
4. Ensure that you have merged your work back onto the master branch**. Commit and push your project**
5. Create a Bot and run an integration to verify that your code on GitHub is complete.