Contact List

Introduction

Goal: Build a command line application to help users manage their contacts.

Reminders & Tips:

- If you restart your app, all the contacts will be lost
- Commit and push your progress on a regular basis
- Seed some fake data in so you don't have to create contacts each time you restart the app

Setup

The project will be written in an Object Oriented way. Each contact will be represented by an instance of the `Contact` object. The `Contact` objects will be organized by a `ContactList` object. User input will be handled by an `InputCollector` object. The main application logic, your "controller", will be located main.m.

Task Breakdown

Task 1: Main menu and user input

Yesterday we built a command line app to process strings. Today we're going to build something similar, but using more object-oriented tools.

When the app starts up it should initially display a menu with options. It then prompts them for input. At this stage, it can just ignore the input and reprint the main menu. In the next task, we will start using the input.

The menu:

What would you like to do next?

new - Create a new contact

list - List all contacts

quit - Exit Application

> _

Last time, we just printed all this from the `main.m` file, and things got a bit messy. Now we're going to create an `InputCollector` object to clean this up a bit.

Create the `InputCollector` class, and add a single method to it: `-(NSString *)inputForPrompt:(NSString *)promptString`. This method will take in a single string parameter `promptString`, and return whatever text the user inputs after that prompt.

Back in our `main.m`, `alloc` and `init` an instance of the `InputCollector` class, and use it's one method to display your menu, and capture the result in a local variable.

Note: To help illustrate what the method in `InputCollector` does, an example of how you would call it from `main.m` is: `NSSString *usernameInput = [inputCollector inputForPrompt:@"Enter your username"];`

Task 2: Implement Exit functionality (`quit`command)

Now that you're taking input, it's time to do something with it. The user inputs quit, then the app should break out of the REPL, causing the program to terminate. If you’re feeling generous, wish the user adieu first.

Task 3: Implement contact creation (`new`command)

If the user types in `new` into the prompt at the main menu, the command line app should further prompt the user for information about the contact they wish to create. Eg: take a full name and email (separately).

We can use `InputCollector` to get this data from the user and store them in local variables. Once we have these strings, it's time to build out our model.

We'll need a `Contact` class, with properties for name and email. Once you've made it, return to your `main.m` and create an instance of Contact, and use the user input to set the name and email properties.

At this point we're taking user input, and creating a new model object using our results, but we still need to store them somewhere. To do this, create a new `ContactList` class, with an `NSMutableArray`property where it will store contacts. Make sure to instantiate a mutable array for that property in the ContactList's `init` method.

Now back in `main.m`, create your `ContactList` instance outside your main-menu while loop, so that it isn't reset each loop.

Once all this is set up, add a method to your `ContactList` class called `-(void)addContact:(Contact *)newContact`, and in this method insert the contact into the `ContactList`'s mutable array.

Call your `ContactList`'s `addContact:` method with the `Contact` instance you created a moment ago.

Task 4: Implement Contact index (`list` command)

When on the main menu, the user can type in list to display a list of all contacts within the app, printed one on each line. Each line should be formatted as:

#: <full name> ()

The number (#) should start with 0 and represents an index or unique ID for each contact. Once the contacts are printed out to the screen, the app should go back to the main menu.

Implement the printing code inside your `ContactList` class.

Bonus Features

Bonus 1: Contact details (`show` command)

When on the main menu, the user can type in `show` along with an id (index) of the contact to display their details. If a contact with that index/id is found, display their details, with each field being printed on an individual line. If the contact cannot be found, display a "not found" message.

Bonus 2: Implement Contact search (`find` command)

After typing in a `find` command, along with a search term, the app will search through the names of the contacts and print the contact details of any contacts which have the search term contained within their name or email. (ie. the search term is a substring of the contact's email or name)

Example use: find ted

Bonus 3: Prevent duplicate entries

If a user tries to input a contact with the exact same email address twice, the app should output an error saying that the contact already exists and cannot

be created. If you are asking for name first and then email, for a better user experience, it may make more sense to ask for their email first and then their name.

Bonus 4: Multiple phone numbers

Implement the ability to add contact's phone numbers. Contacts can have a limitless amount of phone numbers. Each phone number has a label and the number itself (eg: "Mobile" and "444-555-3123")

Bonus 5: History

Every time the user enters a command, log it in an array property on your input class. If the user enters `history`, print the last 3 commands the user has entered.