Challenge requirements:

You have been given a task to create a terminal-based interface/program/application for teachers in NEXT Primary School to access students' details!

The headmaster, Josh Teng, has talked to you about it and the application should meet the following criteria:

Assume the users (teachers) are comfortable with using the command line/terminal.

Schema:

- Follow the basic schema from the previous challenge:
- id, first_name, last_name, gender, birthday, email, phone

Authentication:

- Teachers can type in their email and log into the application to access
- Password is not required for now

Features:

The main page should be a list of options that the user can choose from. The options are:

- List all students in the school
- which includes all the student's details and id
- Create a new student and save him/ her into the school's database
- If the creation failed, repeat the process again
- Find a student with the id and it will show the student's details
- Within that page, teachers can choose to
    - update the student's details
    - delete the student

You may read the following or you may start with the challenge right away if you want to give yourself a challenge.

Some background for this challenge (it's quite a bit of reading)

Before you start the long read, you can have a small refresher on what can you do with a class that inherits from `ActiveRecord`: http://jamesponeal.github.io/projects/active-record-sheet.html

Now, there are currently 2 ways for you to approach how to create an application or how to create a new feature for an application.

1. You immediately jump unto the backend and code it out (database, logic etc.), or
2. You create some sort of interface/visual to get a better grasp of the user flow and then code your backend from there

Most of the time, you can go with any of the approaches:

- Some may prefer the second method as it is easier when you can visually interact with your program,
- while others may prefer to get the backend part sorted, and finally be able to focus on frontend and design.

Though, in this challenge, we'll go with the first approach (in fact, in the previous challenge, we're already doing it!). While we're in it, let's talk a bit about one of the most important part of a web application: **the database**.

Many of our programs/applications require us to read and store data from a data source. Not just that, we want the source to have the ability to present those data the way we want to (eg. get only the students whose age less than 5, order the list of teachers by the first letter of their first name etc.)

For this small application requested by Josh, we can start from handling the layer that communicates to the database, which among the Rails community, we're calling those layers `Model`.

We've created a few model files in the previous AR Student Schema challenge, but what does it actually allow us to do?

The `ActiveRecord` library allows developers to use the concept of classes and objects to interact with the database, while not having to deal with the implementation on how to actually connect to the database/tables in the database. What does that mean?

You no longer need to type SQL commands in the SQL console to get your data. You can leverage on the `ActiveRecord` library which translate **Ruby** code into SQL. This means, you can immediately interact with the database (manipulating the data and do queries on

those data) **inside a Ruby program** using those classes which are inheriting from ActiveRecord::Base!

Though, without an user-friendly interface, these code will be of no use, as users will not want to use your application in its raw form (you surely dont want users, in our case, the teachers, to extract the data by using those Ruby code, don't you?)! Thus it's important to know how do we integrate the Models we've written with whatever kind of interface we are going to build!

For a web application, we would need to know how to use the three web languages: HTML, CSS and Javascript to build an interface for the users. But now, we'll just write code that puts out a simple interface in the terminal, and gets user input from the terminal as well! (Get it now?)

Integrating an interface written in those web languages will be very similar to an interface written with just puts

The example below demonstrates a simple usage of ActiveRecord combined with an interactive menu (You can make changes where you see fit):

```
# app/models/user.rb
class User < ActiveRecord::Base
  validates :email, presence: true
  validates :age, presence: true
end


# app/interface.rb
require_relative '../db/config'
require 'erb'


def user_index
  print "No. Age Email\n"
```

```
  # BONUS: find a built in method that generates index numbers
  index = 1
  User.all.each do |s|
    # BONUS: how to print in a better looking way?
    print "#{index}. #{s.age} #{s.name}\n"
    index += 1
  end
end

def new_user_prompt
  # `params` is a hash that contains all user inputs
  # this way you don't need to have multiple variables
  params = {}
  puts "User's email?"
  print ">> "
  params[:email] = gets.chomp
  puts "User's age?"
  print ">> "
  params[:age] = gets.chomp
  return params
end

def user_create
  loop do
```

```
    params = new_user_prompt

    # BONUS: can you think of a way to do this better?
    user = User.new(first_name: params[:first_name], last_name: params[:last_name])
    if user.save
      puts "User saved!"
      break
    else
      puts "There was an error in your input. Please check your inputs and try again"
      # http://guides.rubyonrails.org/active_record_validations.html#working-with-validation-errors
      p user.errors
    end
  end
end

def print_menu
  # BONUS: refactor to sync menu options with case statements
  puts <<~STR
    Welcome! Please pick an option
    1. List all users
    2. Create a new user
    3. Exit
  STR
end
```

```ruby
# BONUS: clear terminal upon pick option
loop do
  print_menu
  # BONUS: refactor repetitive prompt
  print ">> "
  input = gets.chomp
  case input
  when "1"
    puts "=== All users ===\n"
    user_index
  when "2"
    puts "=== Creating a new user ===\n"
    user_create
  when "3"
    puts "Goodbye!"
    break
  else
    puts "Invalid option!"
  end
end
```

Note: To best understand what does the code above do, you **need** to get your hands dirty. That means you will have to **test out** the code, **find an entry/starting point** to understand, and start dissecting and studying different parts of the code! Reading code can sometimes help you understand different way of solving a

problem, and you may also discover some tricks to achieve something you may have achieved before.
If you run the above code in the command line, you should easily notice that, the part of code starting from line with `loop` is where everything starts. From there, you will further discover that inside the `loop`, it may call different methods, depending of the user's `input` (variable naming can sometimes be important to code reading too!).

Few questions you may want to answer while reading the code:

1. Why don't we write something like this to list out all the users:

```
User.all.each do |u|
  puts u
end
```

1. Why isn't the `id` used for numbering the list of users?
2. Why do we need an `if-else` statement with the `.save` method?
3. Why we shouldn't put these code into the model? (pretty important to know :p)

Once you're ready, let's start to build our application now! Josh is waiting :p

Bonus features for your application:

Once you're done with your application, you can try, and not limited to, the following features:

- Able to list by
- ascending order of the students name
- descending order of date of creation
- A search page that is able to
- Find a student based on gender (Male/Female)
- Find a student based on certain age
    - able to find within a range is also a plus
- Find a student based on the initial letter of name
- Have the total count of students in the main menu every time it's being showed