Assignment 7

Refer to Canvas for assignment due dates for your section.

Objectives:

- Collaborate with others using a shared repository.
- Write generic classes.
- Apply generic classes to variations of a problem.
- Continue to meet the objectives of previous assignments, where applicable.

General Requirements

Create a new Gradle project for this assignment in your ***group*** GitHub repo.

For this assignment, you only need a single package, which you can name as you see fit. The requirements for repository contents are the same as in previous assignments.

GitHub and Branches

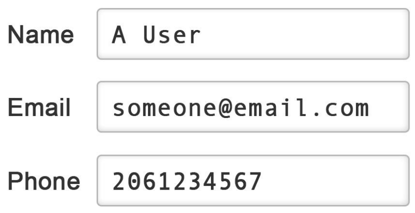
Each individual group member should create their own branch while working on this assignment. Only merge to the **main** branch when you're confident that your code is working well. You may submit pull requests and merge to master as often as you like. Guidelines on working with branches are available from the assignment page in Canvas.

As this is the first group assignment, it is recommended that you practice merging with your group members early on, before you get too deep into the assignment. Getting used to collaborating with GitHub can be painful so don't underestimate this part of the assignment! Two important tips: pull from main before creating a branch and avoid editing files/code that other group members are also working on.

To submit your work, push it to GitHub and create a release on your main branch. Only one person needs to create the release.

The problem

Online forms are widely used to gather information from users for a wide range of purposes such as creating an account, submitting a support request, or making a purchase. An online form typically has the following parts:



Name	A User
Email	someone@email.com
Phone	2061234567

A **GUI** made up of standard form fields (e.g., free text fields, password fields, radio buttons etc.). This is where the user enters their information.

Input validation. As the user enters their information, the system may validate the input and prepare it to be stored or processed elsewhere e.g., a **database** or an email server.

For this problem, you will work on the middle part of a generic form system: validation and processing of input received from standard GUI form fields. You will not work with the GUI directly, but you can assume that a GUI would be a client of your code.

A form is made up of `Fields` that accept different types of input from users. Your task is to implement a generic field class that client code can use to create a variety of forms. Every `Field` will need to track the following information:

- `label` - The String label associated with the form field, e.g., “username”, “password”.
- `value` - The input captured by the GUI. Its data type will be either String for text fields or Boolean for fields such as checkboxes or radio buttons. When a `Field` is first instantiated, `value` should be null.
- `required` - A boolean indicating whether a particular field must be completed before the form can be submitted.
- `validator` - a `Validator` (see below) that will perform input validation.

The methods that a `Field` needs to support are as follows:

- `void updateValue(input)`. The data type of `input` will either be String or Boolean depending on the type of data the field accepts. Update the `Field`’s `value` if the `input` is valid according to the `validator`. If `input` is not valid, throw an exception and *do not update* `value`. Your solution should not provide any other way to set `value`.
- `boolean isFilled()`. This method should return true if the `Field` has been filled out. Client code can use this method to determine if a form is ready to submit. A `Field` will be considered filled under the following conditions: (1) the `Field` is required and its value meets the requirements specified by its validator, or (2) the `Field` is optional (it does not matter what the `Field`’s value is in this case).

A `Validator` stores requirements for user input and provides a method, `boolean isValid(input)`, that determines if the provided input meets the requirements. The data type of `input` will either be String or Boolean depending on the type of data the validator accepts. The `Validators` that your system will need, and their requirements, are as follows:

- `Password` - A password is a String has the following properties:
    - Minimum and maximum acceptable length (inclusive).
    - The minimum number of *lowercase* letters that the password must contain (default = 0, which means that lowercase letters are not required).
    - The minimum number of *uppercase* letters that the password must contain (default = 0).
    - The minimum number of *digits* that the password must contain (default = 0).

To be valid, a password must meet the length requirements and contain at least the minimum number of each character type. Additionally, a valid password cannot contain a space (“ “). To keep things simple, all other characters are allowed.

- `Phone` - A valid phone number is a String that contains only digits and has a specified length supplied by client code. The length must match exactly.
- `Number` - This validator will be used for numeric fields other than phone numbers. The `input` type should be String, rather than a numeric type—the purpose of the validator is to check that text entered by a user can be converted to the appropriate numeric format. Example use cases include dollar amounts (e.g. adding a tip to a food order) and dates (e.g. year of birth). Client code should indicate minimum and maximum values (inclusive) and the maximum number of decimal places allowed. For example, if the number of decimal places is 2, then “3”, “3.1”, and “3.45” would all be considered valid but “3.000” would not. If the number of decimal places is 0, you can assume that the number must be an integer. You may choose how to handle an inappropriate number of decimal places (i.e. a negative number of decimal places).
- `FreeText` - This validator will be used for free text fields such as messages or comments. Client code will provide the number of lines in the text field and the number of characters allowed per line. To be valid, input must be no longer than the number of lines multiplied by the number of characters allowed per line.
- `RadioButton` - Radio button input is a Boolean indicating whether or not the button is selected. To be valid, input cannot be null.
- `CheckBox` - Similar to a radio button, checkbox input is a Boolean indicating whether or not the button is checked. However, unlike a radio button, null is a valid input for a checkbox.

Implement and test `Field` and the required `Validators`. Although it is possible to come up with a good OOD solution to this problem without using generics, for the purposes of this assignment, your **`Field` and `Validator` classes must be generic**. You are encouraged to use inheritance for this problem where appropriate but make sure that your super classes, if applicable, are generic.

Tip: Java’s Character class contains several methods that you may find useful for the `Password` validator.