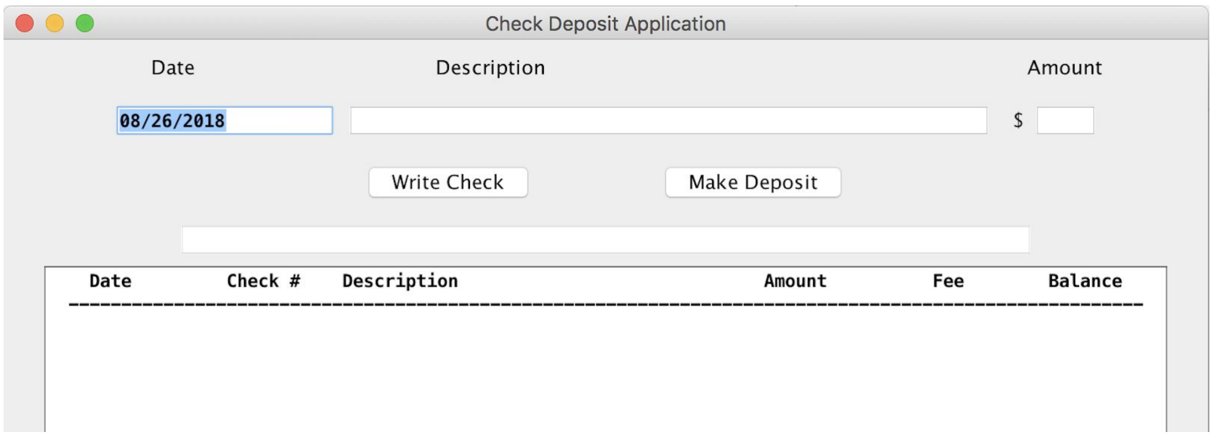
# [Java] Implement Check Deposit Application

Last Modified: Aug. 28th, 2018

## Summary

In this project, you will implement a Java application with a UI (Sometimes the UI part is called GUI, graphical user interface). The application acts like a banking system, or finance management system, where we can deposit checks into the system then your account balance will increase, or you can write checks to other people (or we call it withdraw money) so then your account balance will decrease. In order to do so, the GUI will need to have corresponding input boxes and buttons, and the GUI also contains a large text box to display all historical transactions ordered by transaction date, as well as a text box to display warning message if any input is invalid.

In this project, you do not need to worry about implement the GUI, since the main purpose is for backend and data integration. GUI part is already pre-defined, including size, layout etc. However, you will still need to go through the code along with the explanations from next section, to understand how this GUI is implemented using the java.swing libraries. GUI for this application will look like below:



```
Date        Check #    Description                        Amount        Fee        Balance
------------------------------------------------------------------------------------------
```

## Introduction

In this section, we will introduce the application codebase in details. Clone the project from https://github.com/fangruanruan/check_deposit to an ideal location, and you shall see two java files under the src folder: GUI.java and Data.java. (If unfamiliar with Git, check basics)

```
git clone https://github.com/fangruanruan/check_deposit.git
```

Note that if you want the initial codebase without solutions, do

```
git clone https://github.com/fangruanruan/check_deposit.git
cd check_deposit
git reset --hard cf6b737682fc898f5a2f0354ac42535365740542
```

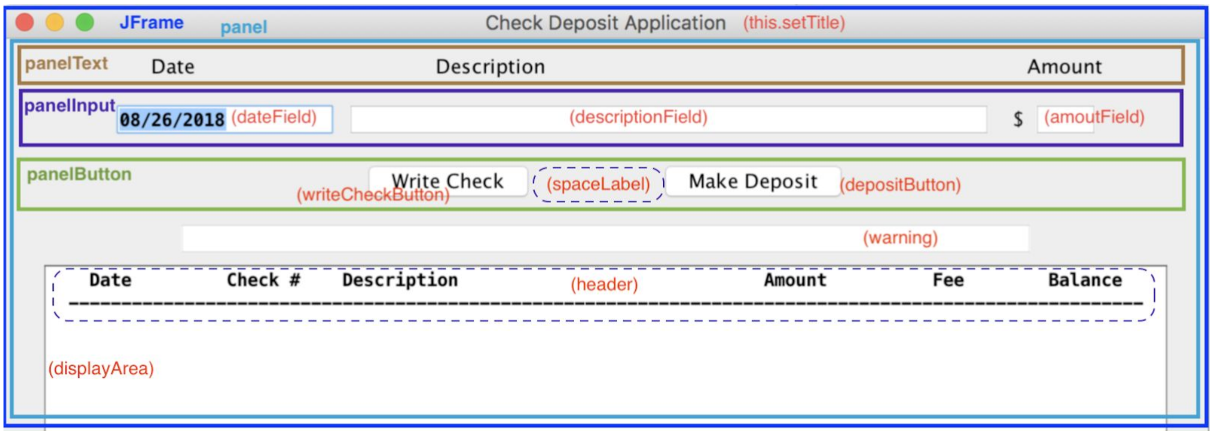
GUI.java is the code for the application GUI, and Data.java is the code for transaction data model class. For Java applications, entry point is always the main method:

```
public static void main(String[] args)
```

It's located in GUI.java, and it just initiate the GUI class.

GUI class extends java.swing.JFrame class. The java.swing is a toolkit for Java to provide GUI. JFrame is basically a window that has decorations such as a border, a title, and supports button components. Inside a JFrame, you can add all kinds of swing components to display the UI like you want.

Looking at the member variables, it has a bunch of swing component. JTextField is a text box usually with only one line with a certain length. JTextArea is a text block with multiple lines. JPanel is a component container used to easily group components together (panelText groups all text labels, panelInput groups all input text boxes, panelButton groups all buttons, and panel groups all the panels above to show inside the JFrame). JButton is a button that user can click on. Font is used to define a certain type of font for text to display. JLabel is a text label on UI. All variables is marked on the following graph of the application.



**Now let's look at GUI constructor.**

First, we set the whole JFrame (application window) size to be 800*600, and by default it will display the window from screen top-left corner. In order to display that in the middle of the screen, we obtained the screen size and use setLocation to move it to middle of the screen. Then set the title and default close operation to be EXIT_ON_CLOSE meaning once we close click the close button on window, the application will be terminated. (Otherwise, the application will still be running even if we click the close button)

Then SimpleDateFormat object is used to format a java.util.Date object into a certain type of string format. "MM/dd/yyyy" means it's month/day/year, with 2 digits on month and day, 4 digits on year, i.e. 08/27/2018. You can also use "yyyy-MM-dd", meaning 2018-08-28. In the format string, 'M' means month of the year, 'd' is day of the month, 'y' is year, 'H' is hour in the day 0-23, 'm' is minute in the hour, 's' is second in the minute.

DecimalFormat object is similar to above, but used to format a number to be certain string format. “#,##0.00” means two digit after the decimal dot, usually that’s how money number shows. And before the decimal dot, must be an ‘0’ if number is smaller than 1. Also before the decimal dot, show a comma ‘,’ for every three digits, just like how dollars show. Examples will be “0.29”, “5.23”, “10,233.99”, “1,000,000.00”, etc.

Then we create a Font object with font style Monospaced, Bold, and size 12.

Then setTextInput(), addButton(), addWarning(), and initDisplayArea(), these four methods will correspondingly setup the UI components. These will be explained later.

After that, all components are set up in panel object. We will need to set the panel to be the content pane of the JFrame so that JFrame window will contain all these components. Then set the JFrame to be visible.

**Let’s move to setTextInput method.**

This method sets up everything in the panelText and panelInput in the above graph.

In panelText, it actually only contains one String. The string has some spaces, and word “Date”, and more spaces, then word “Description”, and then spaces, and at last word “Amount”. See the above graph, this is how the UI shows three words in each location, because they are separated by a lot of spaces. But String cannot be added to panelText because only UI component can be added to a JPanel. JLabel is a class to use here. We create a JLabel object containing this string, and add it to panelText.

In panelInput, it contains 4 components, a Date input box (JTextField), a Description input box (JTextField), a money sign (“$”, JLabel), and an Amount input box (JTextField). For dateField, we also set the default value as today’s date. Other two fields will just be empty.

**Let’s now look at addButton method.**

This method sets up everything in the panelButton in the above graph.

In panelButton, there’re basically two JButton objects. One is showing “Write Check”, and one “Make Deposit”. JLabel with a lot of spaces is just an object to stay between two buttons, in order to display them with some spaces in the middle as graph shows. button.addActionListener is the method to give the button a listener. A listener is basically something listens to a specific type of action, whenever this action happens, the listener will execute the corresponding method. In button, the action listener is listening to “click” action, when you click the button. Method “actionPerformed” method is by default the method to execute whenever the action happens. In this case, when you click write check button, it calls the processTransaction method with false input, and if click make deposit button, it calls the processTransaction method with true input.

**Then it’s addWarning method.**

This method only set up the warning JTextField in the UI. We set the background color to be white, but foreground color to be red, meaning the text message is red. That’s usually how warning message shows.

**At last it’s initDisplayArea method.**

This method sets up the displayArea JTextArea object in the above graph.

You can consider JTextArea as just a text editor that can show multiple lines. We always need to display the header part. So we first use the StringBuilder to build out the header string. String header has two lines, first line is the column names, and second line is just a lot of ‘-’ as a separation. Then we create the object of JTextArea, input 20 and 106 will be the height and width. displayArea.append(header) will basically put the header two lines in the display area by default.

The text display area may have more lines than the 20x106 space can display. We need a vertical scroll bar for the display area. We created a JScrollPane, which is basically scroll bar. By default it will apply horizontally and veritically. We need to display horizontal scroll bar as we don’t need it.

**About processTransaction method.**

This method is not about UI set up. We used this method whenevery buttons are clicked. When this method is called, it will get the amountField value (default it’s a string), parse it to double. Then get the description field value. Then get the dateField value, and try parse it to a java.util.Date type. Whenever a parse fails, we will display error on warning field to tell user.

After all inputs are obtained and validated, we will reset all input box to be empty. Then create the Data type with date, description and amount. At this point, amount will be positive if it’s deposit, negative if not. After data added to the dataArray, we will sort the dataArray in order to have it order by date. Then resizeDataArray will check whether dataArray is full, if yes then resize the array, if not just skip it. Finally, we call printDetails to display updated information in the displayArea (we clear what was showing in displayArea, and re-add the header and all transactions). We use necessary spaces in order to display each data to be aligned in the area, just like a table.

In general, this processTransaction method covers everything after a button is clicked.

## Task 1 - Dynamically Resize the Data Array

Comparing to List or other data structures, Array has fixed size. No matter how big we initiate the array, it’s always possible for the array to be full, then new data has no place to store. Here we will use the concept of dynamic array, where we create a new array with double size and copy the data over, every time when the original array is full.

1. Add data to array
2. Resize array
   a. If array is not full, skip
   b. If array is full, create a new array with double size of current array, copy current array data to the first half of the new array

This will solve the potential problem with fixed size. But we need to do some extra works to copy the data over. Will this change the time complexity of any method? Array usually has two types, add new data and read from array. Because it’s still an array, read operation using index will definitely be still O(1). About add operation, we are doing extra works O(n) if we need to copy the old array with size n, but we don’t do this for every add operation. Using Amortized Analysis, we will now get that such kind of add operation on average is still O(1) (Examples section for the wikipedia page has a brief prove for this).

Please go to the GUI.java file, and find the resizeDataArray method with TODO comment. Implement this method so that it will resize the dataArray object if it’s full.

To test your implementation: Run the application, try “write check” or “deposit check” multiple times and make sure it will not throw exception. (Before application will throw exception if you do them more than twice)

## Task 2 - Implement Data Class

From the Introduction section, you will see that Data object is the actual backend data for what’s added into the system or what’s displayed on the UI. Every time user input a transaction, a new Data object will be created to represent that transaction, including date, amount, description. We can tell from the sign of the amount whether it’s a deposit or withdraw (positive means deposit, negative means withdraw).

We also included a transaction fee concept here. But for simplicity, we will set the transaction fee to be statically 3% of the transaction amount if it’s a withdrawal, and 1% if it’s a deposit.

Data class in this case is a typical data model class, where it represent one kind of data used in the application. And Data object has few public methods to get the date, amount, fee, etc. There’s no corresponding set method for these properties, because once a Data object is created, there’s no need to modify the detail information. This kind of concept in larger scale projects is very important, and usually it’s called the Access Control or simply Visibility, where you strictly control what kind of information can people read or modify. In this project, try to strictly control the visilibity in Data class.

Please go to the Data.java file, implement all member methods and constructor so that it’s correctly storing the transaction information, and UI will also show correct information as expected. DO NOT implement the static sortData method in this section.

To test your implementation: Run the application, try do some operations with valid inputs, and make sure transaction is shown correctly in the display area, with correct balance and historical information.

## Task 3 - Sort Historical Transactions by Date

Now the display area will show historical transactions in the order of user input, because the static sortData method is doing nothing. We would like to display transactions in time order, from early to recent.

Please go to Data.java file, implement the static sortData method with input of dataArray and size. Because dataArray may not be full, size will indicate the actual range of the index, you should sort the data from index 0 to index size (exclusively).

To test your implementation: Run the application, try do some operations with dates in random order, and you should see the transactions in display area shown from early to recent.