

Documentation for Financial Transactions HTML Page  
Jason N.  
April 26, 2020

# Contents

<b>1</b>	<b>Foreword</b>	<b>1</b>
<b>2</b>	<b>HTML</b>	<b>1</b>
2.1	Preamble and head . . . . .	1
2.2	Inputs . . . . .	1
2.2.1	Common attributes . . . . .	2
2.2.2	Labels . . . . .	2
2.2.3	Date . . . . .	2
2.2.4	Text . . . . .	2
2.2.5	List . . . . .	3
2.2.6	Buttons . . . . .	3
2.3	Table . . . . .	4
2.3.1	thead . . . . .	4
2.3.2	tbody . . . . .	5
<b>3</b>	<b>Javascript</b>	<b>6</b>
3.1	getData() . . . . .	6
3.2	validate() . . . . .	7
3.2.1	Check empty . . . . .	8
3.2.2	Check NaN . . . . .	8
3.2.3	Check date . . . . .	9
3.3	generateId() . . . . .	10
3.4	calculateCostBasis() . . . . .	11
3.5	addTransaction() . . . . .	11
3.6	deleteRow() . . . . .	12
3.7	editRow() . . . . .	13
3.8	saveChanges() . . . . .	14
3.9	discardChanges() . . . . .	15
3.10	sortTable() . . . . .	16
<b>4</b>	<b>CSS</b>	<b>18</b>
4.1	Vertical Scrolling Table . . . . .	18
4.2	Horizontal Scrolling on Overflow . . . . .	18
4.3	Miscellaneous . . . . .	19
4.3.1	Sort buttons . . . . .	19
4.3.2	Editing highlight . . . . .	19
4.3.3	Table borders . . . . .	19
<b>A</b>	<b>HTML Source Code</b>	<b>20</b>
<b>B</b>	<b>Javascript Source Code</b>	<b>24</b>
<b>C</b>	<b>CSS Source Code</b>	<b>29</b>

# 1 Foreword

Some of the code samples in this document were copied by hand. If there are any discrepancies between code in this document and in the source files, refer to the source files.

This does not apply to the appendix. Code in the appendix was generated directly from the source files.

## 2 HTML

### 2.1 Preamble and head

This line declares that the document is an HTML5 document.

```
1 <!DOCTYPE html>
```

`<head>` tags are used to contain meta information about the document.

```
2 <head>
3   <meta charset = "UTF-8"/>
4   <link rel="stylesheet" type="text/css" href="./style.css"/>
5   <script src="./script.js"></script>
6 </head>
```

Within the `head` element:

- The first line defines the character set of the document.
- The second line defines the source of an external CSS document.
- The third line defines the source of an external Javascript document.

### 2.2 Inputs

The input section of this page is contained within `<article>` tags for the purpose of organisation. This can be used to facilitate styling this part of the page with CSS if desired.

```
10 <article id="inputFields">
```

The `article` element has been assigned a unique id for the purpose of styling. Specifically, this id is used to define padding and overflow. This is described in further detail in section 4.2 of this document.

All input fields and buttons are contained within `<form>` tags. Although this is not strictly necessary for the purpose of this project, it is useful for organising data and specifying the fields from which data should be submitted.

```
11 <form onsubmit="return false" autocomplete="off">
```

The attribute `onsubmit` is used to define a Javascript function to be executed when pressed. The form expects that `true` is returned when data is successfully submitted. If so, the default behaviour is to clear

the fields and enter the data in the browser URL bar as arguments. To prevent this behaviour, `onsubmit` is set to `return false`.

The attribute `autocomplete` can be used to specify whether user input from a previous session should be used to populate input fields. This attribute also determines whether or not suggestions are displayed when the user enters data. In this case, `autocomplete` has been set to `off` to prevent these actions from occurring. This does not affect the functionality of the program.

The buttons and input fields within the `form` element are contained within `<section>` tags for organisation. This is primarily done to allow elements to be positioned properly by the CSS file.

### 2.2.1 Common attributes

All `input` elements in this `form` have been assigned a `name` attribute. The `name` attribute is not strictly relevant in this case, but is often used to identify the data when submitting to a database.

All `input` elements have the `required` attribute. Normally this prevents a `form` from being submitted unless all `required` fields contain data. This does not apply to our case as we have disabled the built-in submit function. However, it does still outline missing fields in red.

### 2.2.2 Labels

Each of the inputs are given a label to specify to a user the type of information which should be entered in the given field. This is done with the `input` element.

```
12 <label for="date">Date:</label><br/>
```

The `for` attribute is used to specify an element which corresponds to this label. This is done by setting the attribute to the id of the other element. Labels allow a user to select an input field by clicking the label rather than the field itself. Labels are also used to facilitate the use of assistive technologies.

### 2.2.3 Date

The date of a transaction is specified through the use of an `input` element with a `type` attribute of `date`. This can be used to effectively restrict the input to a valid date format and provides an intuitive method for inputting data.

```
11 <section>
12   <label for="date">Date:</label><br/>
13   <input id="date" name="date" type="date" required/>
14 </section>
```

This type of input field is also useful for interpreting dates in Javascript, as it provides methods which return the date in various formats to facilitate displaying and comparing dates.

### 2.2.4 Text

`input` elements with a `type` attribute of `text` can be used to retrieve a string from a user. This is also the field used for numbers, as these can be easily verified and converted in Javascript.

```

16 <section>
17     <label for="account">Account Number:</label><br/>
18     <input id="account" name="account" type="text" placeholder="Account
    ↪ Number" required/>
19 </section>

```

The advantage of taking numbers from an input field is that it allows for characters such as \$ to be included. In the case of this project, users are able to submit Dollar Amounts as purely numeric values, or in a currency format. Currently, the program only accepts dollars as a currency, however, it is possible to allow and store any number of currencies. These characters, of course, have to be filtered out before the number is interpreted and re-inserted before displaying the value.

### 2.2.5 List

Dropdown lists are created using `<select>` tags containing `option` elements. Each `option` element represents a possible value, the first element is selected by default.

```

21 <section>
22     <label for="type">Transaction Type:</label><br/>
23     <select id="type" name="type">
24         <option value=""></option>
25         <option value="BUY">BUY</option>
26         <option value="SELL">SELL</option>
27         <option value="DIVIDEND">DIVIDEND</option>
28         <option value="INTEREST">INTEREST</option>
29         <option value="WITHDRAW">WITHDRAW</option>
30         <option value="DEPOSIT">DEPOSIT</option>
31     </select>
32 </section>

```

The `innerHTML` of an `option` element is the text that will be displayed to the user. The `value` attribute of the element is the value that will be read by Javascript. For this project, the `value` and `innerHTML` were made to be identical so that the text in the table would be the same as the text the user had seen in the list.

### 2.2.6 Buttons

`button` elements are clickable elements which can execute Javascript code specified by an `onclick` attribute. Text within the `innerHTML` of the `button` will be displayed as text within the button, which is useful for communicating the purpose of the button.

```

49 <section>
50     <button id="add" type="submit" onclick="addTransactionButton();">Add
    ↪ Transaction</button>
51     <button id="save" type="submit" hidden="true" onclick="saveChanges()
    ↪ ;">Save</button>
52     <button id="discard" type="button" hidden="true" onclick="
    ↪ discardChanges();">Discard</button>
53 </section>

```

In this case, three buttons are present, each set to execute a different Javascript function when clicked.

Two of the three buttons have a **type** attribute of **submit**. This causes each function to trigger the **submit** event along with the Javascript function. However, for this project, this event has been disabled by the **form onsubmit="return false"** attribute. Thus, the only difference is that this causes missing fields to be outlined in red when the button is pressed.

The last button is of **type button**. This element functions exactly the same, except it does not trigger the **submit** event. For this project, this means that missing fields will not be highlighted red, as this is not necessary for the 'Discard Changes' button.

Two of the three buttons also have the **hidden="true"** attribute. This causes the page to render as if these elements did not exist, as these elements are only relevant when editing a row. All three buttons are given unique ids so that **hidden** attributes can be added or removed as needed.

## 2.3 Table

### 2.3.1 **thead**

The header of the table is enclosed in **<thead>** tags. This element includes the first row of the table, denoted by **<tr>** tags, which contains headers for each column.

Every cell in the header is denoted by **<td>** tags. These cells differ from normal cells, such as those in the body of the table, in how they format their contents. Using this element for header cells makes them stand out slightly as well as making it easier to differentiate when styling with CSS.

```
61 <th>
62     <section>
63         Transaction ID
64     </section>
65     <section class="sort">
66         <button type="button" onclick="sortTable(0, true)">^</button>
67         <button type="button" onclick="sortTable(0, false)">V</button>
68     </section>
69 </th>
```

The first 8 header cells are split into two separate **section** elements. This was done to allow for the proper positioning of the header text and the sort buttons. For this reason, the latter **section** element is given the class **sort** to differentiate between the two.

Each of the first 8 header cells contain two buttons for sorting. All sorting buttons call the same function **sort(column, ascending)**, however, they pass different arguments to this function. The first argument is the column number, starting from 0, which allows the Javascript function to determine which column to use when comparing rows. The second argument defines whether data should be sorted in ascending or descending order.

The last header cell contains nothing but text. This column is used to contain the delete and edit buttons created for each row.

```
133 <th>Actions</th>
```

### 2.3.2 tbody

The table body is enclosed in `<tbody>` tags. This element is meant to be the main container of data in a table.

```
136 <tbody id="tableBody">
```

The table body is important for this project as it is the parent element of all data which will be manipulated. For this reason, it has been given a unique id to reference in Javascript. This was not strictly necessary, as it is also possible to reference this element by its tag name, being the only `tbody` element. Nevertheless, I consider this to be good practice as it is clear which element is being referred to in Javascript and allows for other tables to be added in the future if necessary without breaking the current functionality.

## 3 Javascript

The following section describes all Javascript functions used in this project. Functions have been grouped according to their purpose, some functions have been omitted for being too similar to other functions.

Each section contains a section in which these are compared to an equivalent function from the Google Sheets project.

### 3.1 getData()

This function is used to retrieve and format data from the input fields.

```
1 function getData() {
2     var date = document.getElementById("date");
3     var account = document.getElementById("account").value;
4     var type = document.getElementById("type").value;
5     var security = document.getElementById("security").value;
6     var amount = document.getElementById("amount").value;
7     var dAmount = document.getElementById("dAmount").value;
8
9     amount = Number(amount);
10
11     if(dAmount[0] == '$') {
12         dAmount = dAmount.substr(1);
13     }
14     dAmount = Number(dAmount);
15
16     if(validate(date, account, type, security, amount, dAmount)) {
17         var costBasis = calculateCostBasis(amount, dAmount);
18         date = date.value;
19         dAmount = '$' + dAmount.toFixed(2);
20
21         return [ date, account, type, security, amount, dAmount, costBasis
22                 ↪ ];
23     }
24     else return false;
25 }
```

The function checks whether the data is valid by calling the `validate()` function. If so, data is formatted and sent to the function which called `getData()`. Currently, the caller is either `addTransactionButton()` or `saveChanges()`.

The function first stores the `date` element and the raw values of the other input fields. `date` is treated differently as the element includes useful methods for comparing the date in different formats.

```
2 var date = document.getElementById("date");
3 var account = document.getElementById("account").value;
4 var type = document.getElementById("type").value;
5 var security = document.getElementById("security").value;
6 var amount = document.getElementById("amount").value;
7 var dAmount = document.getElementById("dAmount").value;
```



Next, some of the data is processed. `amount` is converted from a string, as it originated from a text field, to a number. This is done with the built-in `Number()` function, which takes a string as an argument and returns it as a numeric value when possible. If the argument cannot be converted, the function returns `NaN` or 'Not a Number'. We are not concerned with validating that the value can be converted at this stage, as we can simply check if the value is `NaN` during the validation stage, therefore it is safe to convert to a number here.

```
9 amount = Number(amount);
```

A similar conversion is performed on the `dAmount` value. However, before this occurs, we check whether the first character in the string is a dollar sign. If so, we remove the dollar sign by taking a substring of `dAmount` which includes everything including and after the second character. This effectively removes the dollar sign from the string, allowing it to be converted to a numeric value.

```
9 if(dAmount[0] == '$') {
10     dAmount = dAmount.substr(1);
11 }
12 dAmount = Number(dAmount);
```

The function then calls `validate` and passes all the stored variables as arguments to determine whether all the data is valid. If not, the function will return `false` and exit, preventing subsequent steps from occurring.

```
16 if(validate(date, account, type, security, amount, dAmount)) {
17     var costBasis = calculateCostBasis(amount, dAmount);
18     date = date.value;
19     dAmount = '$' + dAmount.toFixed(2);
20
21     return [ date, account, type, security, amount, dAmount, costBasis ];
22 }
23 else return false;
```

If all data is valid, the function calculates and stores the `costBasis` by calling `calculateCostBasis()` and passing the necessary values. The function also formats the date and dollar amount in the correct formats to be exported to the table. The `toFixed()` method is used to fix the value to 2 decimal places, and a dollar sign is added to the front of the value. Lastly, the function returns a list including all the data to the caller.

### Comparison to Google Sheets project

There is no equivalent function in the Google Sheets project. The `getData()` function is required to store input values in memory. Google Apps Script had a built-in function to move or copy cells and did not require most values to be stored like this.

## 3.2 validate()

The `validate()` function is used to verify that all fields include valid data.

```
26 function validate(date, account, type, security, amount, dAmount) {
27     if(!validateDate(date)) return false;
28     if(!validateAccount(account)) return false;
29     if(!validateType(type)) return false;
30     if(!validateSecurity(security)) return false;
31     if(!validateAmount(amount)) return false;
32     if(!validateDAmount(dAmount)) return false;
33 }
```

```
34     return true;
35 }
```

The function calls several functions, each of which validates a different input field. If any of the calls returns false, this function returns false. If none of the calls returned false, the function returns true, allowing the caller to proceed.

### Comparison to Google Sheets project

The function uses the same method as the Google Sheets project for validating data, by calling different functions which return **true** or **false**. The difference here is that data is taken as arguments and passed to the validating functions, as this data is no longer read from the sheet. The returns of this function have also been standardised such that **false** always indicated an invalid value, this is done mostly for readability.

#### 3.2.1 Check empty

In cases where the only check necessary is that the field is not empty, the function simply compares the value to an empty string. If the value is equal to an empty string, the function prints an alert and returns false, otherwise it returns true.

```
54 function validateAccount(account) {
55     if(account == '') {
56         alert('Error: Missing Account Number');
57         return false;
58     }
59
60     return true;
61 }
```

This is the template used to check account, transaction type, and security, as these are all strings. Although the transaction type field is not a text box, by setting the default empty option to have a value of an empty string, this template still applies.

### Comparison to Google Sheets project

The function to check for an empty field is now done by the same function that check that a field is valid. This was done to better organise the validation process and allow functions to be modified more easily.

The Google Sheets project used the `isBlank()` method of a cell, as no data was stored and passed to it. This project does pass values to the function, therefore the check can be simplified by comparing it to an empty string.

#### 3.2.2 Check NaN

The function to check whether or not a value is a number is identical to the functions that check for only empty values, except for one key difference. In addition to checking if the value is empty, the function checks if the value is NaN. This is done using the `isNaN()` function, which takes a value as an argument and returns true if the value is NaN.

```
87 if(isNaN(amount)) {
88     alert('Error: Invalid Amount');
89     return false;
```

### Comparison to Google Sheets project

This check was not performed in the Google Sheets project, however, if one were to validate a number in the Google Sheets project, one would check that the value was a numeric type, similar to how the date was validated in Google Sheets. In this case, the number can be validated much simpler, by checking whether or not the conversion was successful.

#### 3.2.3 Check date

Currently, a valid date is a date that is not in the future. In order to check this, the function gets the current date by creating a new `Date` object and storing it as a variable. The function also stores the value of the `date` element in a number format which can be easily compared. This is done by storing the `valueAsNumber` property of the element.

```
37 function validateDate(date) {  
38     realDate = new Date();  
39     inputDate = date.valueAsNumber;  
40  
41     if(!date.checkValidity()) {  
42         alert('Error: Invalid date');  
43         return false;  
44     }  
45  
46     if(realDate.valueOf() < inputDate) {  
47         alert('Error: Date is in the future');  
48         return false;  
49     }  
50  
51     return true;  
52 }
```

The first check performed is whether or not the user had inputted a date that exists. If the date field was left empty or incomplete, or the date is non-existent (e.g. November 31) the `date.checkValidity()` function will return `false`. Therefore, we can reuse the statement that checks whether or not a field contains a valid number.

Next, the function must check that the date is not in the future. This is done by simply comparing the dates in number format, with a greater value indicating a later date.

### Comparison to Google Sheets project

In Google Sheets, a date was validated by checking that it was of the `[object Date]` type. In this project, we can be confident that the object is of the correct type as it was created by a specific type of input, therefore, it is not necessary to validate the type.

To check that the date was not in the future, the Google Sheets project directly compared two date objects. This was not done for the current project, as we have two different types of data, an element and a `Date` object. In order to compare these, both are converted to the same numeric format.

### 3.3 generateId()

A unique id generated with the length specified by `idLength` and with characters specified by `characters`.

```
109 function generateId() {
110     var id = '';
111     var idLength = 6;
112
113     var characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
114     var charactersLength = characters.length;
115
116     var unique = false;
117
118     while(!unique) {
119         for(var i = 0; i < idLength; i++) {
120             id += characters.charAt(Math.floor(Math.random() *
121                                     ↪ charactersLength));
122         }
123
124         unique = true;
125         for(var i = 0; i < document.getElementsByClassName('idCell').
126             ↪ length; i++) {
127             if(document.getElementsByClassName('idCell')[i].innerText ==
128                 ↪ id) {
129                 unique = false;
130                 break;
131             }
132         }
133     }
134     return id;
135 }
```

The `unique` variable is used to store whether the generated id is unique. A `while` loop continuously generates and checks ids until a unique one is found.

```
119 for(var i = 0; i < idLength; i++) {
120     id += characters.charAt(Math.floor(Math.random() * charactersLength));
121 }
```

For every character in the id, a random character is chosen from the character set. A number between 0 and the number of possible characters is generated by `Math.random()* charactersLength` as `Math.random()` generates a number  $0 \leq n < 1$ . This number is converted to an integer by rounding down using the `Math. ↪ floor()` function. The result of this integer is used as the index from which to take a character using the `charAt()` method.

```
123 unique = true;
124 for(var i = 0; i < document.getElementsByClassName('idCell').length; i++)
125     ↪ {
126         if(document.getElementsByClassName('idCell')[i].innerText == id) {
127             unique = false;
128             break;
129         }
130     }
```

Every generated id is initially assumed to be unique. The function loops through every element with a class of `'idCell'`, comparing the `innerText` of this cell to the generated id. If the values match, the id is not unique and another id must be generated. If no matching id has been found, the id is considered unique and the loop exits, passing the id to the caller.

### Comparison to Google Sheets project

The process for generating an id is exactly the same as in the Google Sheets project. The id is compared to the `innerText` of cells exactly as the id was compared to other cells in the first column of the Google Sheets. The two processes are essentially identical, except the process is much faster outside of Google Sheets.

## 3.4 calculateCostBasis()

The function receives two numeric values as arguments, calculates the quotient of the two, and formats the result as a dollar value.

```
134 function calculateCostBasis(amount, dAmount) {  
135     costBasis = '$' + (dAmount / amount).toFixed(2);  
136     return costBasis;  
137 }
```

The `toFixed()` method is used to fix the value to 2 decimal places, and a dollar sign is added to the front of the value.

### Comparison to Google Sheets project

This function is essentially the same as that of the Google Sheets project, except the data is passed to this function, rather than being read from the sheet. Another minor difference is in how data is formatted, as Google Apps Script formatted data by setting a number format, while that format does not exist in pure Javascript and must be set manually.

## 3.5 addTransaction()

The process of adding a transaction is initiated when the `addTransactionButton()` function is called. This function calls the `getData()` function and stores the result. If the result is not `false`, the function generates a unique id for the transaction and adds this to the front of the `data` array using the `unshift()` method. The function then passes the `data` array to the `addTransaction()` function, which creates and populates the row.

```
156 function addTransactionButton() {  
157     var data = getData();  
158     if(data) {  
159         var id = generateId();  
160         data.unshift(id);  
161         addTransaction(data);  
162     }  
163 }  
164 }
```

The `addTransaction` function gets all necessary data as an array argument. It stores the body of the table in a variable by referencing the element using `document.getElementById('tableBody')`. A new row is

created using the `insertRow()` method of the table body and stored as a variable so that contents can be added. The row is assigned a class of `bodyRow` for reference by other functions or styling.

```
139 function addTransaction(data) {
140     var tableBody = document.getElementById('tableBody');
141     var newRow = tableBody.insertRow(0);
142     newRow.classList += "bodyRow";
143
144     var actionsContent = "<button type='button' onclick='editRow(this)''>
        ↪ Edit</button> <button type='button' onclick='deleteRow(this)''>
        ↪ Delete</button>";
145     data.push(actionsContent);
146
147     for(var i = 0; i < data.length; i++) {
148         var newCell = newRow.insertCell(i);
149         newCell.innerHTML = data[i];
150         if(i == 0) {
151             newCell.classList += "idCell";
152         }
153     }
154 }
```

The function creates string containing HTML code for a delete and an edit button and pushes this to the end of the `data` array. The `onclick` functions pass `this` as an argument, which specifies the element that called the function. This is done so that the row to edit or delete can be selected.

```
144 var actionsContent = "<button type='button' onclick='editRow(this)''>Edit</
    ↪ button> <button type='button' onclick='deleteRow(this)''>Delete</
    ↪ button>";
145 data.push(actionsContent);
```

For each item in the array, the function calls the `insertCell()` method of the row to create a new cell. The contents of this cell are defined by the `innerHTML` property, which is set to the corresponding element of the array. For the first element, the cell is also given a special class to identify it as the cell containing the transaction id.

```
148 var newCell = newRow.insertCell(i);
149 newCell.innerHTML = data[i];
150 if(i == 0) {
151     newCell.classList += "idCell";
152 }
```

## Comparison to Google Sheets project

The Google Sheets project copied data from the input range to the output range, whereas this function stores the values in memory then writes the new cells. As the table is within a self-contained element, the more complicated process of moving all contents to a separate area is not required. Therefore, the process of adding a row is significantly simpler in HTML and JS.

### 3.6 deleteRow()

The function takes an element as an argument, this element is supposed to be the button which called the function. The function gets the parent of the parent of the delete button and removes this element. The

first parent of the delete button is the cell, the second parent is the row. By selecting the second parent, we are selecting the row containing the button.

The function uses the `removeChild()` method of the table to remove the specified row element.

```
166 function deleteRow(button) {
167     var row = button.parentElement.parentElement;
168     document.getElementById("tableBody").removeChild(row);
169
170     if(document.getElementsByClassName('editing').length == 0) {
171         document.getElementById('add').removeAttribute('hidden');
172         document.getElementById('save').setAttribute('hidden', true);
173         document.getElementById('discard').setAttribute('hidden', true);
174     }
175 }
```

It is possible that the delete button is used on the row being edited. If this is the case, the save and discard buttons would remain visible, although there would be no row being edited. To resolve this, the function checks if there are any rows with the class `'editing'`. If so, the function hides the save and discard buttons, and unhides the add button. This is performed whenever the function finds no row being edited, however, it has no effect when the row being edited was not deleted.

### Comparison to Google Sheets project

Deleting is far simpler as it is not necessary to transfer the rows to a separate sheet. HTML and JS also allows for a new button to be easily created for every row.

The Google Sheets project did not hide and show buttons depending on whether or not a function was being edited, however, this is done in HTML as the process is much simpler. If one were to do this in Google Sheets, the buttons would have to be indexed before hand so they could be referenced in code.

## 3.7 editRow()

This element is responsible for highlighting the row being edited and moving the values in the row to the input fields.

```
177 function editRow(button) {
178     if(document.getElementsByClassName('editing').length > 0)
179         document.getElementsByClassName('editing')[0].classList = "bodyRow
180             ↪ ";
181
182     var row = button.parentElement.parentElement;
183     var rowContent = row.getElementsByTagName('td');
184     row.classList = "bodyRow editing";
185
186     document.getElementById('date').value = rowContent[1].innerText;
187     document.getElementById('account').value = rowContent[2].innerText;
188     document.getElementById('type').value = rowContent[3].innerText;
189     document.getElementById('security').value = rowContent[4].innerText;
190     document.getElementById('amount').value = rowContent[5].innerText;
191     document.getElementById('dAmount').value = rowContent[6].innerText;
192
193     document.getElementById('add').setAttribute('hidden', true);
194     document.getElementById('save').removeAttribute('hidden');
```

```

194     document.getElementById('discard').removeAttribute('hidden');
195 }

```

First, the function checks whether or not there is a cell already highlighted for editing. This is done by getting a list of all functions with the `'editing'` class. If so, it resets the class to `'bodyRow'`. This is done to ensure that at most, one row is being edited at a time.

```

178 if(document.getElementsByClassName('editing').length > 0)
179     document.getElementsByClassName('editing')[0].classList = "bodyRow";

```

The function uses the same method as the `deleteRow()` function to get the parent row of the button. The function finds the second parent of the calling button and stores this as a variable for reference. The cells in this row are stored in an array by getting all children of the row with the `td` tag. The row is changed to have the classes `bodyRow` and `editing`.

Each input field is then identified using their id and given a value of the `innerText` of the row.

```

181 var row = button.parentElement.parentElement;
182 var rowContent = row.getElementsByTagName('td');
183 row.classList = "bodyRow editing";
184
185 document.getElementById('date').value = rowContent[1].innerText;
186 document.getElementById('account').value = rowContent[2].innerText;
187 document.getElementById('type').value = rowContent[3].innerText;
188 document.getElementById('security').value = rowContent[4].innerText;
189 document.getElementById('amount').value = rowContent[5].innerText;
190 document.getElementById('dAmount').value = rowContent[6].innerText;

```

Lastly, the add button is hidden by assigning it a `hidden` attribute. The save and discard buttons are revealed by removing the `hidden` attributes.

```

192 document.getElementById('add').setAttribute('hidden', true);
193 document.getElementById('save').removeAttribute('hidden');
194 document.getElementById('discard').removeAttribute('hidden');

```

## Comparison to Google Sheets project

The process is essentially the same, except that the class is used to mark a row as being edited and for formatting, whereas the Google Sheets project marked this by placing a 1 in the I column.

The process of hiding and revealing buttons is unique to this project, as this is made much simpler by the fact that elements can be referred to using a unique id.

## 3.8 saveChanges()

The function calls `getData()` to retrieve data from the input fields and stores this as a variable. If `getData`  $\rightarrow$  () does not return false, the function gets the row being edited by getting the first item with a class of `'editing'`.

The function stores the cells of this row as a an array by getting all child elements with `'td'` tags. Every element in the `data` array is written to a cell in the row, starting from the second cell so that the id is not replaced. The function then sets the class to `bodyRow`, removing the `editing` class.



```

197 function saveChanges() {
198     data = getData();
199     if(data) {
200         rowToEdit = document.getElementsByClassName('editing')[0];
201         cellsToEdit = rowToEdit.getElementsByTagName('td');
202
203         for(var i = 0; i < data.length; i++) {
204             cellsToEdit[i + 1].innerHTML = data[i];
205         }
206         rowToEdit.classList = "bodyRow";
207     }
208
209     document.getElementById('add').removeAttribute('hidden');
210     document.getElementById('save').setAttribute('hidden', true);
211     document.getElementById('discard').setAttribute('hidden', true);
212 }

```

Lastly, the function reveals the 'add transaction' button and hides the 'save' and 'discard' buttons.

### Comparison to Google Sheets project

The function differs in how data is transferred. Instead of using a built-in `copyTo()` method, each cell is individually given data.

This project hides and reveals buttons based on whether or not a row is being edited, as HTML and Javascript makes this process much simpler. This step was not taken in the Google Sheets project.

## 3.9 discardChanges()

This function is responsible for unmarking the editing row and changing the state of buttons.

Firstly, the element with a class of 'editing' is reset to only have the 'bodyRow' class.

Then the 'add transaction' button is revealed, and the 'save' and 'discard' buttons are hidden.

```

214 function discardChanges() {
215     document.getElementsByClassName('editing')[0].classList = "bodyRow";
216
217     document.getElementById('add').removeAttribute('hidden');
218     document.getElementById('save').setAttribute('hidden', true);
219     document.getElementById('discard').setAttribute('hidden', true);
220 }

```

### Comparison to Google Sheets project

The Google Sheets project cleared formatting and removed the edit indicator by clearing the entire I column and resetting formatting for all rows. With Javascript, the process is much simpler as it is easy to select the specific row to reset by referencing its class.

The Google Sheets project also cleared all input fields when changes were discarded. This can be done by setting all input fields to an empty string, but I felt that this was unnecessary and that entering similar data would be easier without this feature.

### 3.10 sortTable()

This function is responsible for sorting each column. The function takes two arguments: the column number, and a true or false value to determine whether the data should be sorted in ascending or descending order.

This function utilises a simple sorting algorithm known as ‘bubble sort’ or ‘sinking sort’. The function steps through every element in the function, comparing it to an adjacent element, and swapping when necessary. This is done until a loop occurs in which no swaps were made. Although the algorithm is considered inefficient compared to other available algorithms, it is far easier to implement and is more than sufficient for this project.

```
222 function sortTable(column, ascending) {
223     var rows = document.getElementsByClassName('bodyRow');
224
225     var sorting = true;
226     while(sorting) {
227         sorting = false;
228         for(var i = 0; i < (rows.length - 1); i++) {
229             rowA = rows[i].getElementsByTagName('td')[column];
230             rowB = rows[i + 1].getElementsByTagName('td')[column];
231
232             var swap = false;
233
234             if(ascending && rowA.innerHTML.toLowerCase() > rowB.innerHTML.
                ↪ toLowerCase()) swap = true;
235             else if(!ascending && rowA.innerHTML.toLowerCase() < rowB.
                ↪ innerHTML.toLowerCase()) swap = true;
236
237             if(swap) {
238                 sorting = true;
239                 document.getElementById('tableBody').insertBefore(rows[i +
                ↪ 1], rows[i]);
240             }
241         }
242     }
243 }
```

First, all the rows are stored in a list of elements for reference.

This is done by taking a list of all elements with the 'bodyRow' class. It would also be possible to do this without the class, as all body rows are contained within <tbody> tags. One could get a list of all rows that are children of the tbody element with `document.getElementsByTagName('tbody')[0].getElementsByTagName('tr');`

However, I believe that the approach above makes the function less readable, and therefore, I have gone with the approach using a separate class.

```
223 var rows = document.getElementsByClassName('bodyRow');
```

Next, the main loop of the function begins. A variable `sorting` is given a value of `true` so that the function is allowed to start. With every loop, `sorting` is set to `false`. The value changes during the loop if any rows have been swapped, otherwise, the rows are considered to be in the correct order and the function terminates as the `while` loop ends.

Every loop, the function steps through every row excluding the last row. It stores the cell of the current row and the row below in the specified column.

```
225 var sorting = true;
226 while(sorting) {
227     sorting = false;
228     for(var i = 0; i < (rows.length - 1); i++) {
229         rowA = rows[i].getElementsByTagName('td')[column];
230         rowB = rows[i + 1].getElementsByTagName('td')[column];
```

A variable `swap` is initially set to false. All text is converted to lowercase for comparison using the `toLowerCase()` method. The values in the cells defined by `rowA` and `rowB` are compared using a `>` or `<` operator depending on whether `ascending` is `true` or `false`. If the values are determined to be in the wrong order, `swap` is set to `true`.

If the rows are required to be swapped, the `sorting` variable is set to true so that the `while` loop does not terminate. The `insertBefore()` method can be used to change the order of the children of an element. This method needs to be called by the parent element, hence why the `tableBody` element is chosen.

The method takes two elements of arguments, and places the first element above the second element. `rows[i]` is always above `rows[i + 1]`, therefore to swap them, we list `rows[i + 1]` as the first argument.

```
232 swap = false;
233
234 if(ascending && rowA.innerHTML.toLowerCase() > rowB.innerHTML.toLowerCase
    ↪ ()) swap = true;
235 else if(!ascending && rowA.innerHTML.toLowerCase() < rowB.innerHTML.
    ↪ toLowerCase()) swap = true;
236
237 if(swap) {
238     sorting = true;
239     document.getElementById('tableBody').insertBefore(rows[i + 1], rows[i
    ↪ ]);
240 }
```

## Comparison to Google Sheets project

In Google Sheets, when buttons are assigned to a function, the arguments are not defined. For this reason, it was necessary to have a different button for ascending and descending orders. In this project, we can define both the column and whether it should sort in ascending or descending by passing different arguments. Therefore, only one function is required to sort any column.

This function is somewhat more complex than the Google Sheets equivalent. For the Google Sheets project, there existed a built-in function to sort a column, whereas this had to be written from scratch for HTML and Javascript.

## 4 CSS

### 4.1 Vertical Scrolling Table

The table id refers to the **article** that contains the table, rather than the table itself. This article was given a max height of 80 visual heights, or approximately 80% of the screen height. **overflow: auto;** specifies that, if necessary, a scroll bar should be present, this is true for both horizontal and vertical scrolling.

```
29 #table {  
30     max-height: 80vh;  
31     overflow: auto;  
32 }
```

The last three properties below are important for keeping the header in place while scrolling.

```
39 th {  
40     min-width: 200px;  
41     width: 10%;  
42     position: sticky;  
43     background: white;  
44     top: 0;  
45 }
```

- **position: sticky;** is used to keep the object in place when scrolling.
- **background: white;** is used to give the element a non-transparent background, so that data cannot be seen through the header.
- **top: 0;** is used to specify that the element should remain at the top of its parent element with no offset.

### 4.2 Horizontal Scrolling on Overflow

The inputFields id is used to identify the **article** element that contains the **form**. The important property here is the **overflow-x: auto;** line, which specifies that, if the child element is wider than this element, a horizontal scroll bar should be present.

The **form** element, which is a child of the **article** is given a minimum width to ensure that the scroll bar is created rather than reducing the width of the element.


```
5 #inputFields {  
6     padding: 10px 0;  
7     overflow-x: auto;  
8 }  
9  
10 form {  
11     min-width: 1900px;  
12 }
```

## 4.3 Miscellaneous

### 4.3.1 Sort buttons

The header cells contain two **sections**, one of which has a special class. Both sections are given a **margin** and **padding** of 0 to minimise wasted space. Both sections are also set to **display: inline-block** to specify that they should be arranged horizontally.

Both sections are given a **width** of 80% of the parent element. However, this is overruled for the element with a class of **sort**, which is assigned a **width** of 10% to ensure that both elements fit horizontally in the parent.

To further reduce wasted space, the **border** and **padding** of the buttons are set to 0. The button's **display**  property is set to **block** as otherwise, it would inherit the **inline-block** property from its parent and attempt to arrange horizontally. Lastly, the buttons are set to take up the entire width of the parent element.

```
47 th > section {
48     width: 80%;
49     display: inline-block;
50     padding: 0;
51     margin: 0;
52 }
53
54 .sort {
55     width: 10%;
56 }
57
58 .sort > button {
59     padding: 0;
60     border: 0;
61     display: block;
62     width: 100%;
63 }
```

### 4.3.2 Editing highlight

```
65 .editing {
66     background-color: yellow;
67 }
```

### 4.3.3 Table borders

```
69 #table,
70 table,
71 td,
72 th {
73     box-shadow: 1px 1px black, inset 1px 1px black;
74 }
```

## A HTML Source Code

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset = "UTF-8"/>
5     <link rel="stylesheet" type="text/css" href="./style.css"/>
6     <script src="./script.js"></script>
7   </head>
8   <body>
9     <article id="inputFields">
10       <form onsubmit="return false" autocomplete="off">
11         <section>
12           <label for="date">Date:</label><br/>
13           <input id="date" name="date" type="date" required/>
14         </section>
15
16         <section>
17           <label for="account">Account Number:</label><br/>
18           <input id="account" name="account" type="text"
19             ↪ placeholder="Account Number" required/>
20         </section>
21
22         <section>
23           <label for="type">Transaction Type:</label><br/>
24           <select id="type" name="type">
25             <option value=""></option>
26             <option value="BUY">BUY</option>
27             <option value="SELL">SELL</option>
28             <option value="DIVIDEND">DIVIDEND</option>
29             <option value="INTEREST">INTEREST</option>
30             <option value="WITHDRAW">WITHDRAW</option>
31             <option value="DEPOSIT">DEPOSIT</option>
32           </select>
33         </section>
34
35         <section>
36           <label for="security">Security:</label><br/>
37           <input id="security" name="security" type="text"
38             ↪ placeholder="Security" required/>
39         </section>
40
41         <section>
42           <label for="amount">Amount:</label><br/>
43           <input id="amount" name="amount" type="text"
44             ↪ placeholder="Unit Amount" required/>
45         </section>
46
47         <section>
48           <label for="dAmount">$ Amount:</label><br/>
49           <input id="dAmount" name="dAmount" type="text"
50             ↪ placeholder="$ Amount" required/>
51         </section>
52       </form>
53     </article>
54   </body>
55 </html>
```

```

49         <section>
50             <button id="add" type="submit" onclick="
                    ↪ addTransactionButton();">Add Transaction</button>
                    ↪ >
51             <button id="save" type="submit" hidden="true" onclick
                    ↪ ="saveChanges();">Save</button>
52             <button id="discard" type="button" hidden="true"
                    ↪ onclick="discardChanges();">Discard</button>
53         </section>
54     </form>
55 </article>
56
57 <article id="table">
58     <table>
59         <thead>
60             <tr>
61                 <th>
62                     <section>
63                         Transaction ID
64                     </section>
65                     <section class="sort">
66                         <button type="button" onclick="sortTable
                                ↪ (0, true)">^</button>
67                         <button type="button" onclick="sortTable
                                ↪ (0, false)">v</button>
68                     </section>
69                 </th>
70                 <th>
71                     <section>
72                         Date
73                     </section>
74                     <section class="sort">
75                         <button type="button" onclick="sortTable
                                ↪ (1, true)">^</button>
76                         <button type="button" onclick="sortTable
                                ↪ (1, false)">v</button>
77                     </section>
78                 </th>
79                 <th>
80                     <section>
81                         Account Number
82                     </section>
83                     <section class="sort">
84                         <button type="button" onclick="sortTable
                                ↪ (2, true)">^</button>
85                         <button type="button" onclick="sortTable
                                ↪ (2, false)">v</button>
86                     </section>
87                 </th>
88                 <th>
89                     <section>
90                         Transaction Type
91                     </section>
92                     <section class="sort">

```

```

93         <button type="button" onclick="sortTable
94             ↪ (3, true)">^</button>
95         <button type="button" onclick="sortTable
96             ↪ (3, false)">v</button>
97     </section>
98 </th>
99 <th>
100     <section>
101         Security
102     </section>
103     <section class="sort">
104         <button type="button" onclick="sortTable
105             ↪ (4, true)">^</button>
106         <button type="button" onclick="sortTable
107             ↪ (4, false)">v</button>
108     </section>
109 </th>
110 <th>
111     <section>
112         Amount
113     </section>
114     <section class="sort">
115         <button type="button" onclick="sortTable
116             ↪ (5, true)">^</button>
117         <button type="button" onclick="sortTable
118             ↪ (5, false)">v</button>
119     </section>
120 </th>
121 <th>
122     <section>
123         $ Amount
124     </section>
125     <section class="sort">
126         <button type="button" onclick="sortTable
127             ↪ (6, true)">^</button>
128         <button type="button" onclick="sortTable
129             ↪ (6, false)">v</button>
130     </section>
131 </th>
132 <th>
133     <section>
134         Cost Basis
135     </section>
136     <section class="sort">
137         <button type="button" onclick="sortTable
138             ↪ (7, true)">^</button>
139         <button type="button" onclick="sortTable
140             ↪ (7, false)">v</button>
141     </section>
142 </th>
143 <th>Actions</th>
144 </tr>
145 </thead>
146 <tbody id="tableBody">

```



```
137         </tbody>
138     </table>
139 </article>
140 </body>
141 </html>
```

## B Javascript Source Code

```
1 function getData() {
2     var date = document.getElementById("date");
3     var account = document.getElementById("account").value;
4     var type = document.getElementById("type").value;
5     var security = document.getElementById("security").value;
6     var amount = document.getElementById("amount").value;
7     var dAmount = document.getElementById("dAmount").value;
8
9     amount = Number(amount);
10
11     if(dAmount[0] == '$') {
12         dAmount = dAmount.substr(1);
13     }
14     dAmount = Number(dAmount);
15
16     if(validate(date, account, type, security, amount, dAmount)) {
17         var costBasis = calculateCostBasis(amount, dAmount);
18         date = date.value;
19         dAmount = '$' + dAmount.toFixed(2);
20
21         return [ date, account, type, security, amount, dAmount, costBasis
22                 ↪ ];
23     }
24     else return false;
25 }
26
27 function validate(date, account, type, security, amount, dAmount) {
28     if(!validateDate(date)) return false;
29     if(!validateAccount(account)) return false;
30     if(!validateType(type)) return false;
31     if(!validateSecurity(security)) return false;
32     if(!validateAmount(amount)) return false;
33     if(!validateDAmount(dAmount)) return false;
34
35     return true;
36 }
37
38 function validateDate(date) {
39     realDate = new Date();
40     inputDate = date.valueAsNumber;
41
42     if(!date.checkValidity()) {
43         alert('Error: Invalid date');
44         return false;
45     }
46
47     if(realDate.valueOf() < inputDate) {
48         alert('Error: Date is in the future');
49         return false;
50     }
51
52     return true;
```

```

52 }
53
54 function validateAccount(account) {
55     if(account == '') {
56         alert('Error: Missing Account Number');
57         return false;
58     }
59
60     return true;
61 }
62
63 function validateType(type) {
64     if(type == '') {
65         alert('Error: Missing Transaction Type');
66         return false;
67     }
68
69     return true;
70 }
71
72 function validateSecurity(security) {
73     if(security == '') {
74         alert('Error: Missing Security');
75         return false;
76     }
77
78     return true;
79 }
80
81 function validateAmount(amount) {
82     if(amount == '') {
83         alert('Error: Missing Amount');
84         return false;
85     }
86
87     if(isNaN(amount)) {
88         alert('Error: Invalid Amount');
89         return false;
90     }
91
92     return true;
93 }
94
95 function validateDAmount(dAmount) {
96     if(dAmount == '') {
97         alert('Error: Missing $ Amount');
98         return false;
99     }
100
101     if(isNaN(dAmount)) {
102         alert('Error: Invalid $ Amount');
103         return false;
104     }
105

```

```

106     return true;
107 }
108
109 function generateId() {
110     var id = '';
111     var idLength = 6;
112
113     var characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
114     var charactersLength = characters.length;
115
116     var unique = false;
117
118     while(!unique) {
119         for(var i = 0; i < idLength; i++) {
120             id += characters.charAt(Math.floor(Math.random() *
121                                     ↪ charactersLength));
122         }
123
124         unique = true;
125         for(var i = 0; i < document.getElementsByClassName('idCell').
126             ↪ length; i++) {
127             if(document.getElementsByClassName('idCell')[i].innerText ==
128                 ↪ id) {
129                 unique = false;
130                 break;
131             }
132         }
133     }
134     return id;
135 }
136
137 function calculateCostBasis(amount, dAmount) {
138     costBasis = '$' + (dAmount / amount).toFixed(2);
139     return costBasis;
140 }
141
142 function addTransaction(data) {
143     var tableBody = document.getElementById('tableBody');
144     var newRow = tableBody.insertRow(0);
145     newRow.classList += "bodyRow";
146
147     var actionsContent = "<button type='button' onclick='editRow(this)'"
148         ↪ "Edit</button> <button type='button' onclick='deleteRow(this)'"
149         ↪ "Delete</button>";
150     data.push(actionsContent);
151
152     for(var i = 0; i < data.length; i++) {
153         var newCell = newRow.insertCell(i);
154         newCell.innerHTML = data[i];
155         if(i == 0) {
156             newCell.classList += "idCell";
157         }
158     }
159 }

```

```

155
156 function addTransactionButton() {
157     var data = getData();
158     if(data) {
159         var id = generateId();
160         data.unshift(id);
161
162         addTransaction(data);
163     }
164 }
165
166 function deleteRow(button) {
167     var row = button.parentElement.parentElement;
168     document.getElementById("tableBody").removeChild(row);
169
170     if(document.getElementsByClassName('editing').length == 0) {
171         document.getElementById('add').removeAttribute('hidden');
172         document.getElementById('save').setAttribute('hidden', true);
173         document.getElementById('discard').setAttribute('hidden', true);
174     }
175 }
176
177 function editRow(button) {
178     if(document.getElementsByClassName('editing').length > 0)
179         document.getElementsByClassName('editing')[0].classList = "bodyRow
180         ↪ ";
181
182     var row = button.parentElement.parentElement;
183     var rowContent = row.getElementsByTagName('td');
184     row.classList = "bodyRow editing";
185
186     document.getElementById('date').value = rowContent[1].innerText;
187     document.getElementById('account').value = rowContent[2].innerText;
188     document.getElementById('type').value = rowContent[3].innerText;
189     document.getElementById('security').value = rowContent[4].innerText;
190     document.getElementById('amount').value = rowContent[5].innerText;
191     document.getElementById('dAmount').value = rowContent[6].innerText;
192
193     document.getElementById('add').setAttribute('hidden', true);
194     document.getElementById('save').removeAttribute('hidden');
195     document.getElementById('discard').removeAttribute('hidden');
196 }
197
198 function saveChanges() {
199     data = getData();
200     if(data) {
201         rowToEdit = document.getElementsByClassName('editing')[0];
202         cellsToEdit = rowToEdit.getElementsByTagName('td');
203
204         for(var i = 0; i < data.length; i++) {
205             cellsToEdit[i + 1].innerHTML = data[i];
206         }
207         rowToEdit.classList = "bodyRow";
208     }
209 }

```

```

208
209     document.getElementById('add').removeAttribute('hidden');
210     document.getElementById('save').setAttribute('hidden', true);
211     document.getElementById('discard').setAttribute('hidden', true);
212 }
213
214 function discardChanges() {
215     document.getElementsByClassName('editing')[0].classList = "bodyRow";
216
217     document.getElementById('add').removeAttribute('hidden');
218     document.getElementById('save').setAttribute('hidden', true);
219     document.getElementById('discard').setAttribute('hidden', true);
220 }
221
222 function sortTable(column, ascending) {
223     var rows = document.getElementsByClassName('bodyRow');
224
225     var sorting = true;
226     while(sorting) {
227         sorting = false;
228         for(var i = 0; i < (rows.length - 1); i++) {
229             rowA = rows[i].getElementsByTagName('td')[column];
230             rowB = rows[i + 1].getElementsByTagName('td')[column];
231
232             var swap = false;
233
234             if(ascending && rowA.innerHTML.toLowerCase() > rowB.innerHTML.
                ↪ toLowerCase()) swap = true;
235             else if(!ascending && rowA.innerHTML.toLowerCase() < rowB.
                ↪ innerHTML.toLowerCase()) swap = true;
236
237             if(swap) {
238                 sorting = true;
239                 document.getElementById('tableBody').insertBefore(rows[i +
                ↪ 1], rows[i]);
240             }
241         }
242     }
243 }

```

## C CSS Source Code

```
1 body {
2     font-size: 14px;
3 }
4
5 #inputFields {
6     padding: 10px 0;
7     overflow-x: auto;
8 }
9
10 form {
11     min-width: 1900px;
12 }
13
14     form > section {
15         width: 14%;
16         display: inline-block;
17     }
18
19     input,
20     select {
21         min-width: 100px;
22         width: 80%;
23     }
24
25     button {
26         width: 40%;
27     }
28
29 #table {
30     max-height: 80vh;
31     overflow: auto;
32 }
33
34 table {
35     width: 100%;
36     margin: auto;
37     border-collapse: collapse;
38 }
39     th {
40         min-width: 200px;
41         width: 10%;
42         position: sticky;
43         background: white;
44         top: 0;
45     }
46
47     th > section {
48         width: 80%;
49         display: inline-block;
50         padding: 0;
51         margin: 0;
52     }
```

```
53
54     .sort {
55         width: 10%;
56     }
57
58     .sort > button {
59         padding: 0;
60         border: 0;
61         display: block;
62         width: 100%;
63     }
64
65     .editing {
66         background-color: yellow;
67     }
68
69     #table,
70     table,
71     td,
72     th {
73         box-shadow: 1px 1px black, inset 1px 1px black;
74     }
```