


# CISC 131
# Introduction to Programming and Problem Solving
# Spring 2020
# HTML Tables

**Due: Wednesday, April 1, 2020 at the start of class. Must be demonstrated**
**Points: none**

## Solution to Problem from Lecture 4

The Caesar cipher encryption and decryption methods are very similar. You can see the similarity in this sample implementation. If a letter is found in the alphabet, then the corresponding letter is either three characters farther down the alphabet or three characters farther up the alphabet. Adding the number of characters in the alphabet will change any negative value into a non-negative value. The modulus operation prevents the code from falling off the end of the alphabet.

The bolded line in each of the functions is the only difference between the two.

```
function encrypt ( plainText )
{
 var alphabet;
 var character;
 var cipherText;
 var loc;

 alphabet   = getAlphabet();
 cipherText = "";

 plainText = plainText.toUpperCase();
 while ( plainText.length > 0 )
 {
  character = plainText.charAt ( 0 );

  loc = alphabet.indexOf ( character );
  if ( loc >= 0)
  {
   // make sure loc is a valid character number
   // in the alphabet
   loc = (loc + 3) % alphabet.length;
   character = alphabet.charAt (loc);
  }

   cipherText = cipherText + character;

   //  remove the character that was just processed
   plainText  = plainText.substring (1, plainText.length );
 }
 return cipherText;
}
```

```
function decrypt ( cipherText )
{
 var alphabet;
 var character;
 var plainText;
 var loc;

 alphabet   = getAlphabet();
 plainText = "";

 cipherText = cipherText.toUpperCase();
 while ( cipherText.length > 0 )
 {
  character = cipherText.charAt ( 0 );

  loc = alphabet.indexOf ( character );
  if ( loc >= 0)
  {
   // make sure loc is a valid character number
   // in the alphabet
   loc = (loc + alphabet.length - 3 ) % alphabet.length;
   character = alphabet.charAt (loc);
  }

   plainText = plainText + character;

   //  remove the character that was just processed
   cipherText  = cipherText.substring (1, cipherText.length );
 }
 return plainText;
}
```

I hope that subtracting by doing addition causes you to recall how two's complement works – instead of storing a negative integer, the machine stores its additive inverse. A very similar thing is happening here. With that in mind, a single encrypt/decrypt version can be written if it is called with the correct values. Here is the combined function:

```
function encryptDecrypt ( messageIn, delta )
{
 var alphabet;
 var character;
 var loc;
 var messageOut;

 alphabet   = getAlphabet();
 messageOut = "";

 messageIn = messageIn.toUpperCase();
 while ( messageIn.length > 0 )
 {
  character = messageIn.charAt ( 0 );

  loc = alphabet.indexOf ( character );
  if ( loc >= 0)
  {
   // make sure loc is a valid character number
   // messageIn the alphabet
   loc = (loc + delta ) % alphabet.length;
   character = alphabet.charAt (loc);
  }

   messageOut = messageOut + character;

   //  remove the character that was just processed
   messageIn  = messageIn.substring (1, messageIn.length );
 }
 return messageOut;
}
```

To use this function to encrypt using the Caesar cipher, it would be called this way:

```
cipherText = encryptDecrypt ( plainText, 3 );
```

and to decrypt, it would be called this way:

```
plainText  = encryptDecrypt ( cipherText, getAlphabet().length-3 );
```

Study this for a bit. Do you see how you can change the alphabet to contain digits and other things and that you are no longer limited to Caesar cipher's three character movement?

## Start of Today's Lecture

This lecture introduces the HTML *table* tag. Lots of web designers despise the use of this tag and encourage people to use *div* and other things. However, tables are useful and appropriate for displaying certain types of information and will save time when we develop Javascript applications that require a grid-like structure. Remember, this isn't a web development course; it's an introductory programming course that uses HTML as a convenient way to display results and interact with the user.

### HTML Tables

An HTML table consists of rows and each row consists of columns. These rows and columns form a grid. If you have used Excel, its workbook display contains rows and columns just like the HTML table.

There can be any number of rows in a table and any row can have any number of columns. Often, however, the table is designed as a grid: one where there is the same number of columns in each

row. The intersection of a row and a column is where the data is entered. CSS style can be applied to any aspect of the table(table as a whole, row, column, or row/column intersection) and there are convenient ways of assigning rules to tables using some features of CSS that have not yet been introduced in class.

The HTML tags for tables are:

| | |
|---|---|
| `<table> </table>` | all table rows must be inside these two tags |
| `<tr> </tr>` | all column tags (*td* or *th*) must be inside these two row tags |
| `<td> </td>` | the data for a row – column intersection must be inside these two tags |

An easy way to remember this is: *tr* stands for *table row*, *td* stands for *table data*. There are actually two ways to specify table data. One uses *td* and the other *th*. The *th* stands for *table header* and is given different default styling that *td* elements. You need not use *th* in a table and, of course, its default styling can be overridden by your CSS.

As an example, here is how to create a table with three rows and two columns:

```
<table>
<tr><td>Hi</td><td>There</td></tr>
<tr><td>Bob</td><td>Alice</td></tr>
<tr><td>Judy</td><td>Susan</td></tr>
</table>
```

Place the above HTML inside a *div* element and display the result. As you can see, the default styling for a table is somewhat less than attractive. But notice that the browser makes each column the same width – just wide enough to display the longest content in any *td* or *th* element in that column. To make the table a little more attractive, you can enter these CSS rules

```
table    { border-collapse: collapse;       }
table,td { border         :1px solid black; }
```

All the table elements can have *id* and *class* attributes so that you can access the element from Javascript and style it with your CSS. In addition, if you want a *td* (or *th*) element to use more than a single column, you can add a *colspan* attribute to that *td* (or *th*) element. For example

```
<td colspan="7">Grand Total</td>
```

would place *GrandTotal* in a *td* that was seven columns wide.

There are several convenient ways to specify styling for a table. Actually, these can be used for any HTML element but we are talking about tables here. Assume you wanted all *th* elements in all your tables to use a blue font color. You CSS rule could be specified as:

```
table,th { color: blue; }
```

Or, if your *table* tag contained an *id* of *abc*, you could have all the *td* elements of just that table have a height of one *em* and a background color of green by specifying this CSS rule:

```
#abc td
{
 background-color: green;
 height          : 1em;
}
```

You could also style only the *class* elements within a table. For example, this would make the background color pink for all the elements in the *abc* table that had a CSS *class* of *xyz*:

```
#abc .xyz { background-color: pink; }
```

Enter the necessary HTML and CSS to create a few tables. Be sure to validate both the CSS and the HTML. What do the following look like when they are displayed?

```
<div>Example 1</div
<table>
<tr><td>A</td></tr>
<tr><td>B</td><td>C</td></tr>
<tr><td>D</td><td>E</td><td>C</td></tr>
</table>

<div>example 2</div>

<table>
<tr><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>A</td><td rowspan="2" colspan="2">Z</td><td>A</td></tr>
<tr><td>A</td><td>A</td></tr>
<tr><td>A</td><td>A</td><td>A</td><td>A</td></tr>
</table>
```

Once you are confident you understand the HTML syntax for the *table* element, start the assignment described below.

**Assignment**

In this assignment, you will cause an HTML table to be created and updated using Javascript. Create an HTML file that contains a single *div* whose id is container. The only CSS you will need is:

```
table    { border-collapse: collapse;        }
table,td { border         : 1px solid black; }
```

Create a Javascript program that is associated with the HTML file. As is your standard practice, test to ensure that they are hooked together. Keep the Firefox web console open to catch any errors.

The program you will write requires these previously written functions, so copy them into the Javascript program:

```
createHTMLElement(elementType, id, classInfo, content)
getRandomInteger(upperLimit)
getRandomRGB()
```

**Step 1**

Most of the functions that you will write for this program are very short. Some are only a single line containing a *return* statement. Short functions are easier to write and easier to debug. Some functions are complicated and involve a lot of steps. When implementing complicated functions, it is usually a good idea to implement the complicated function by calling smaller functions each of which does just one of the complicated function's steps. This makes program development and testing much easier and takes much less time.

To demonstrate this idea, here is a function that returns the HTML *id* of the HTML table you are going to create in your program. I used *bob* as my table *id* but you can use any valid HTML *id*.

```
function getTableId (){ return "bob"; }
```

Now, whenever the program needs the HTML *id* of the table element, it simply calls this function. Sometimes you need the reference to the table, not just the *id* of the table, so another useful function is:

```
function getTableElement () { return document.getElementById ( getTableId() ); }
```

Notice how this function uses the *getTableId* function. This is good programming practice. The table *id* occurs in only one place so, if there is a need to change the *id*, there is only one place that change needs to be done. Any function that needs the table *id* will call the *getTableId* function and

any function that needs a reference to the table will call the *getTableElement* function. Both functions are easy to write, easy to understand, and easy to test.

There are several more useful, one line functions that should be written. While the table elements can be accessed in a more direct way than using the *id* of the element, that approach involves some syntax that will make more sense later in the semester when the array data structure is introduced. For now, you will use the same approach you have always used: access an element using its HTML id.

When accessing *divs*, we developed a naming convention for the *div id* that consisted of a text *id prefix* and an integer suffix. Taken together, these were the *id* of an element. HTML tables form a two dimensional grid of rows and columns. The data part (cell) of the table is the intersection of a row and a column. It makes sense, then, to use the row and column number in the *id* of a cell. Each part will have a text *id prefix* and an integer suffix. Here are two short functions that will return the text id prefix. I used *R* and *C* but you can use any prefix that you want.

```
function getCellRowPrefix()    { return "R"; }
function getCellColumnPrefix() { return "C"; }
```

Now it is easy to use these to write a function that will return the *id* of a cell:

```
function getCellId ( row, column )
{
 return getCellRowPrefix() + row + getCellColumnPrefix() + column;
}
```

And it is also easy to write a function that will return a reference to a cell:

```
function getTableCel ( row, column )
{
 return document.getElementById ( getCellId ( row, column ) );
}
```

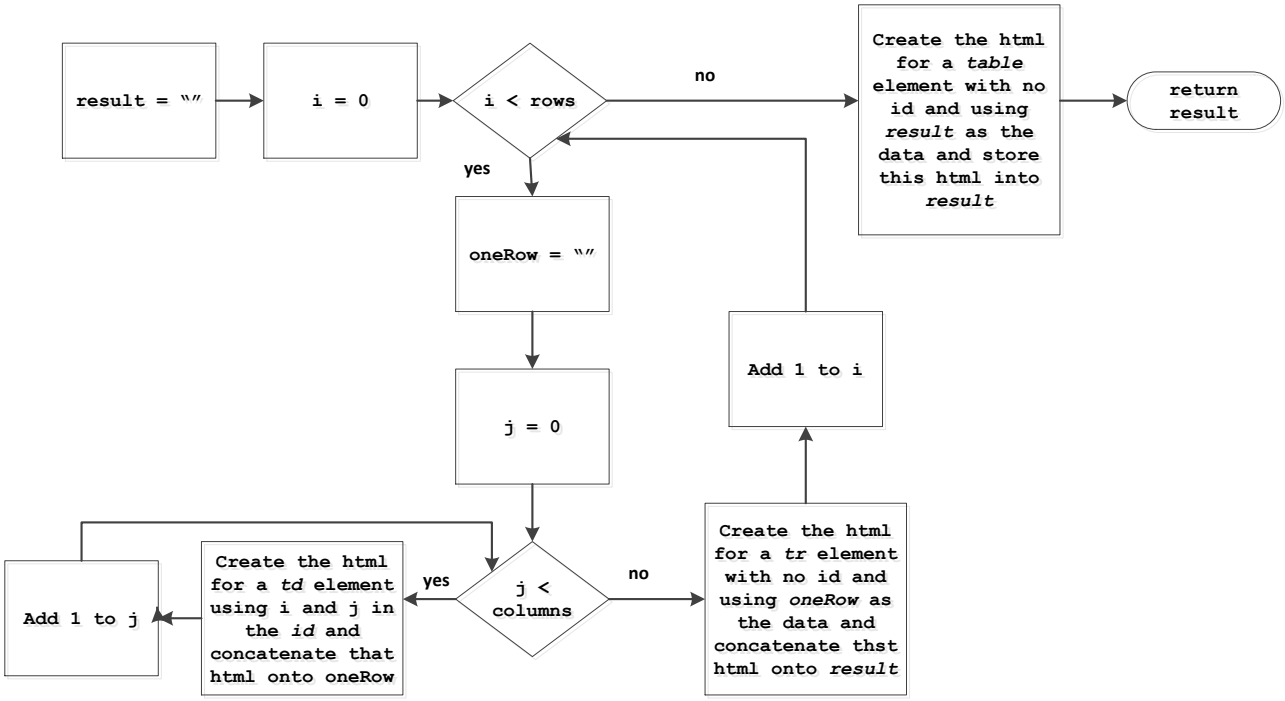
These little utility functions are easy to write and test and they can make it much easier to write and test more complicated functions that use them.

**Step 2**

There is only one complicated function in this program. It is the one that creates the HTML for the table:

```
function createTable ( tableId, rows, columns )
```

This function returns a *String* containing the HTML for the entire table. The function will be called from the *onload* function and the *onload* function will supply the values for the *createTable* formal parameters. The function makes calls to the *createHTMLElement* function and concatenates the return values of those calls. You may find the following flowchart helpful although you are not required to use the logic in the flowchart. You may implement this function in whatever manner you think is best.




I suggest that you test the function be calling it from the *onload* and using a *window.alert* statement to display its return value. This will help you verify that the function is working correctly. When you are certain it works correctly, call it from the *onload* function passing it the table id, and a small number for the rows and columns. Use the return value as the *innerHTML* of the *div* with the *id* of *container* like this:

```
document.getElementById("container").innerHTML = createTable(getTableId(),2,3);
```

Since the table cells have no data, the table will display as a tiny grid. The next part of the assignment will place data in the cells and the table will start to expand.

**Step 3**

Before you develop the function to add data to the table, it will be useful to develop some more short functions.

```
function getAlphabet() { …
function getRandomAlphabetCharacter() { …
```

The first function returns a *String* that contains any characters you want. My implementation returned the uppercase and lowercase letters of the alphabet as well as the digits zero through nine but you may do what you want here. The second function calls the first one and randomly selects a character from the alphabet and returns it as a *String*. Again, these short functions are easy to write, easy to test, and easy to understand.

The main function you will shortly write will pick a table cell at random and store a random alphabet character into it. The cells in the table use their row and column number as part of their HTML *id*. In order to pick one at random, you will need to know how many rows and how many columns are in the table. The table element can tell you this although accessing that information is a little tricky and involves some syntax that won't be introduced for a while.

Each table row can have any number of columns. The statement *rows[0].cells.length* is accessing the number of columns in row zero of the table. Since table you are creating in this assignment has

the same number of columns in each row, this statement will work for all rows in the table. Here are the implementations of those two functions.

```
function getTableRowCount    () { return getTableElement().rows.length;          }
function getTableColumnCount () { return getTableElement().rows[0].cells.length; }
```

**Step 4**

Now write this function:

```
function updateTable()
```

The function will pick a table cell at random and assign a random character of the alphabet to the *innerHTML* of that cell. Be sure to use the small functions you wrote earlier to help you do this.

In order to see the table grow, you need to add data to the cells. Each time the *updateTable* function is called, one cell will have data placed in it. This process can be automated. Make the last statement in the *onload* function a statement that will cause the *updateTable* function to be called every 250 milliseconds. Now, when you launch the HTML, the table will gradually fill with data and you can watch the rows and columns expand to hold that data.

**Step 5**

Once the table update is working correctly, it is time to have some fun. Add the statement to the *updateTable* function that will set the font of the cell[1] to a random color. Add the statement that will center the text in the cell[2]. Now add the statement that will change the cell's background color to a random color. Launch the HTML page. Now things are more interesting. Here are some additional functions that you should write. After writing and testing each function, call it from the *updateTable* function.

**getRandomFontSize()**

Recall that the font is specified[3] by an integer followed by *pt*. For example: *12pt.* This function returns a random font size between zero and thirty-six.

**getRandomFontStyle()**

The font style[4] is one of the following: *normal*; *italic*; or *oblique*. The function randomly selects one of these and returns that randomly selected String. This is easily implemented using single legged *if* statements.

**getRandomFontWeight()**

The font weight[5] is how heavy or dark the font appears. Font weight is represented by an integer. The greater the integer, the darker the font. HTML supports these nine font weights: *100, 200, 300, 400, 500, 600, 700, 800,* and *900*. The function randomly selects one of these values and returns it.

**getRandomFontFamily()**

The font family[6] is set using a *String* that lists the names of a font and one or two alternate font names that will be used if the system does not have the first font installed. For example, this String:

```
"Arial, Helvetica, sans-serif"
```

---

[1] *elementReference*.style.color = …
[2] *elementReference*.style.textAlign = …
[3] *elementReference*.style.fontSize = …
[4] *elementReference*.style.fontStyle = …
[5] *elementReference*.style.fontWeight = …
[6] *elementReference*.style.fontFamily = …

instructs the browser to use the *Arial* font. If the browser does not have the *Arial* font, it will use the *Helvetica* font. If it also doesn't have the *Helvetica* font, it will use any *sans serif* font.

You may want to do a little research into fonts - how they are specified, and which ones are "web safe". There are a lot of sites that will give you a list that can be copy/pasted. If a font name contains multiple words, the font name must be enclosed in quotation marks. This is easy in Javascript because either the apostrophe or the quotation mark can be used to delimit a String as shown here:

```
'"Courier New", Courier, monospace';
```

You can easily implement the function by getting a random integer and then using single legged *if* statements that test that integer. It's not the best way to do it but it is easy. Soon we will learn a much better method of handling these repeated values

Once you have the *updateTable* function augmented with calls to all these new functions, increase the number of rows and columns in the call to the function in the onload. It is oddly satisfying to sit back and watch the table fill up with letters and colors.

**What You Must Hand In**

Nothing