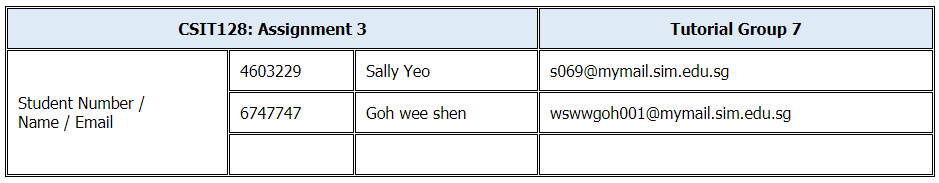
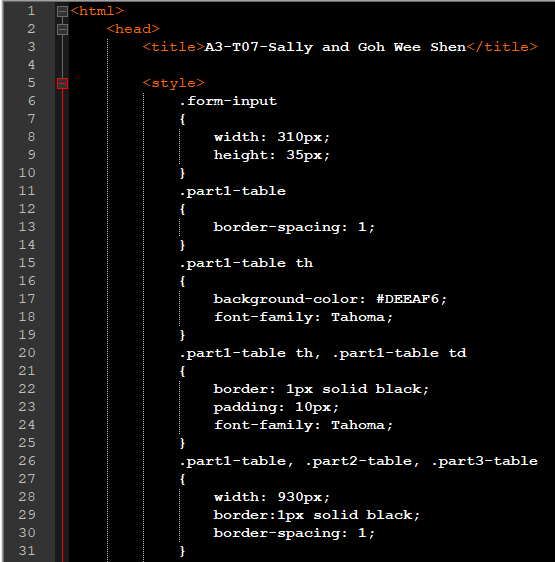
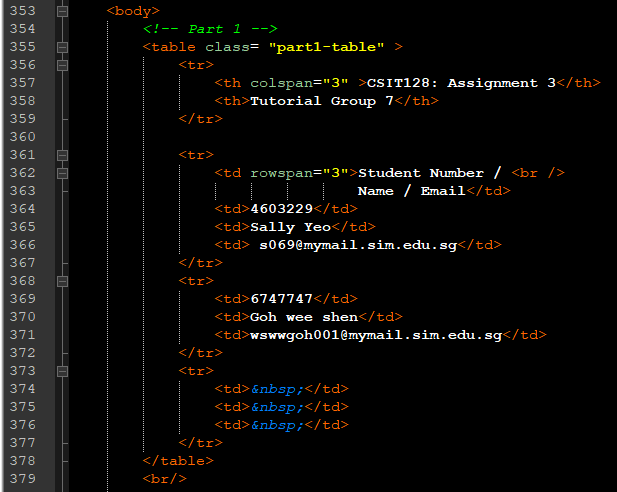
**Part 1 Explanation**

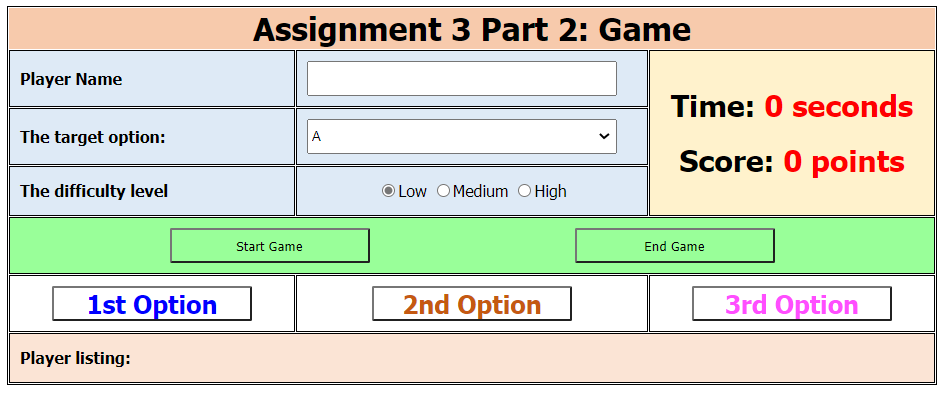


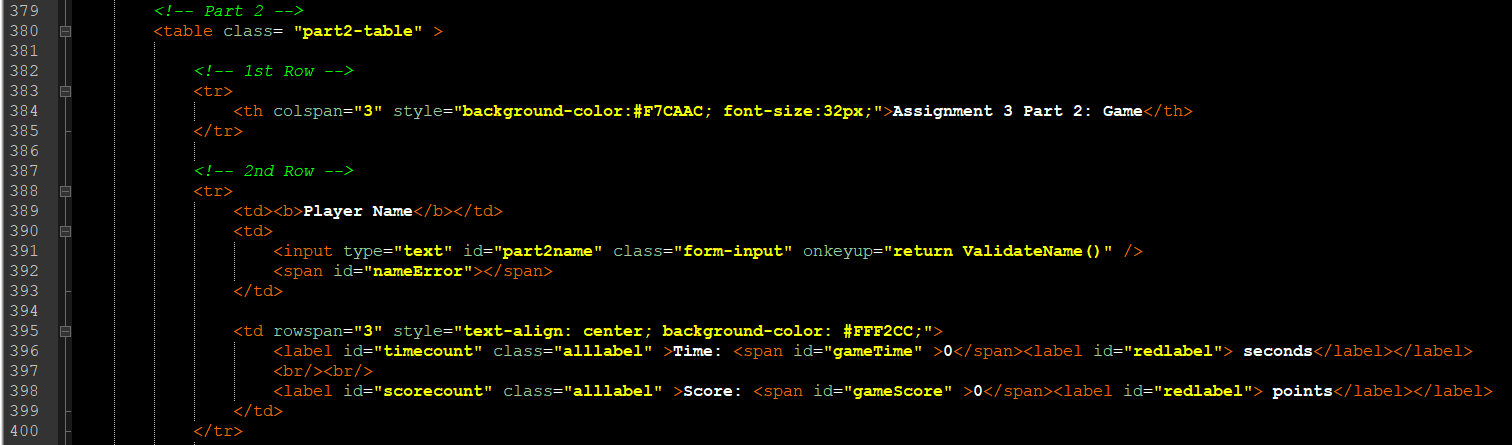




For Part 1, we use class selector for the CSS. The row and columns are merged using colspan and rowspan as shown in line 357 and line 362.

**Part 2 Explanation**

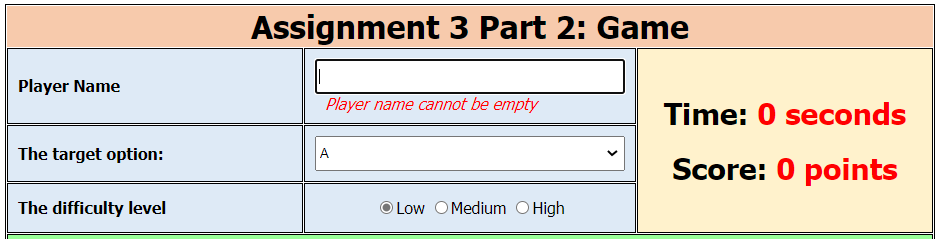




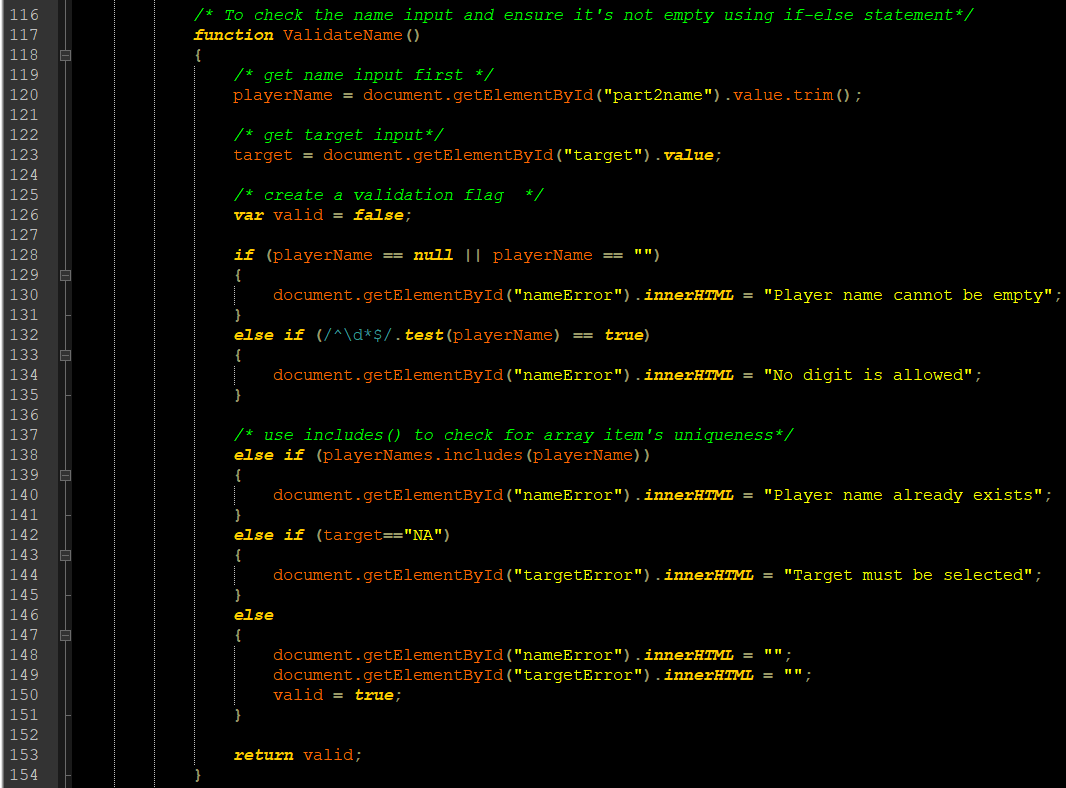
For **player name**, <input> type=”text” with id=”part2name” is used.

Function ValidateName() is called whenever there is a key is released by user.

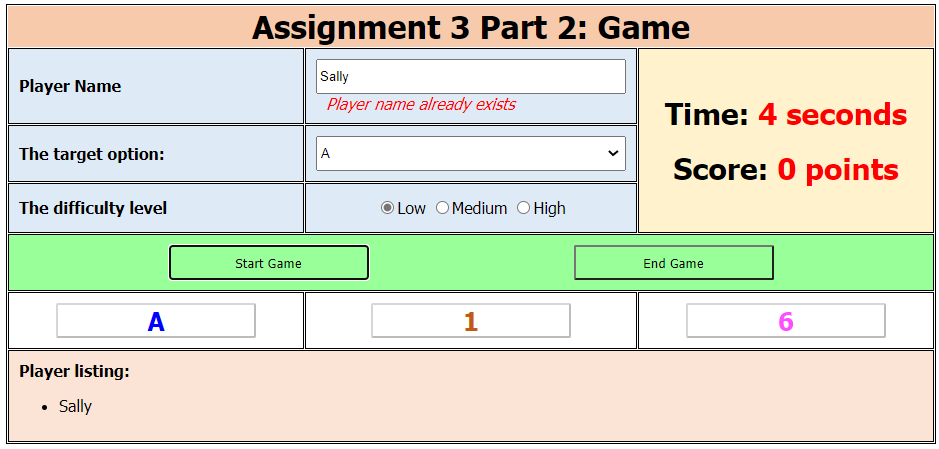
To display the error message, <span> with id=”nameError” is used as shown in the picture below.

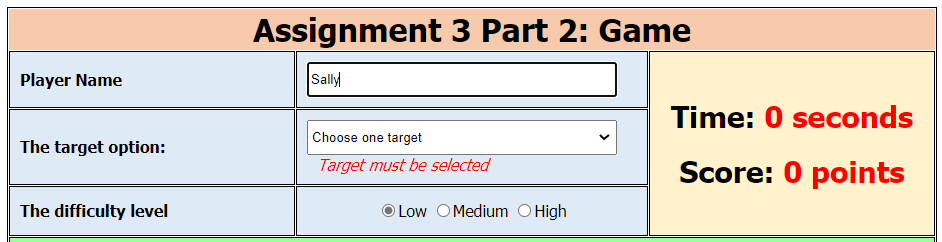


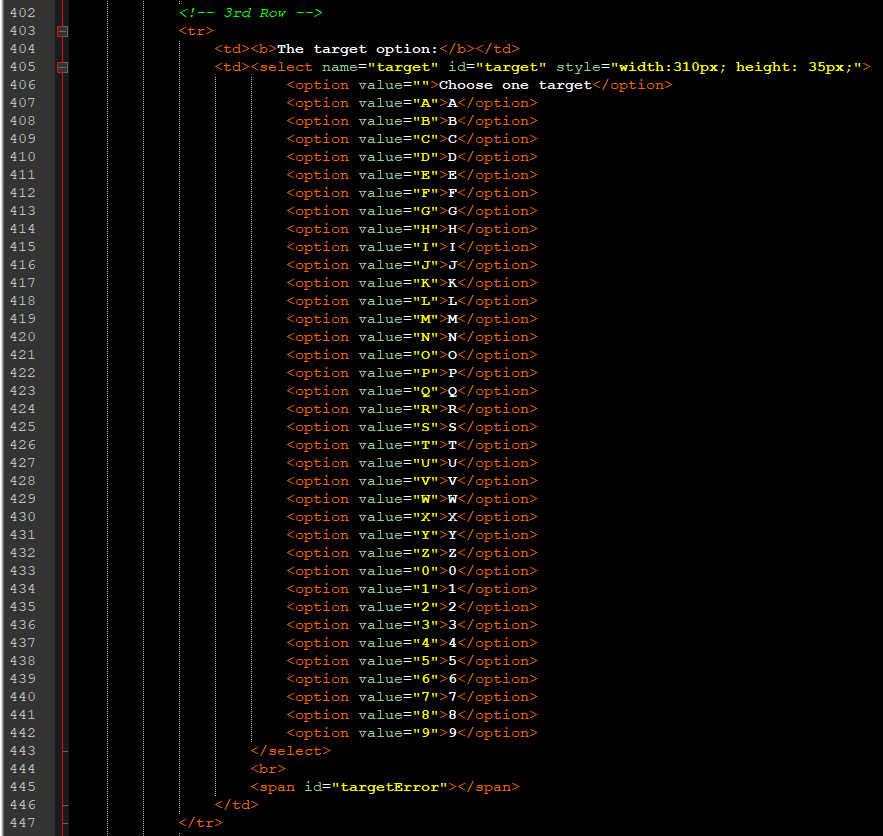
The last column is merged using rowspan=”3”, <span> and <label> is used for this column. To display the time and score.



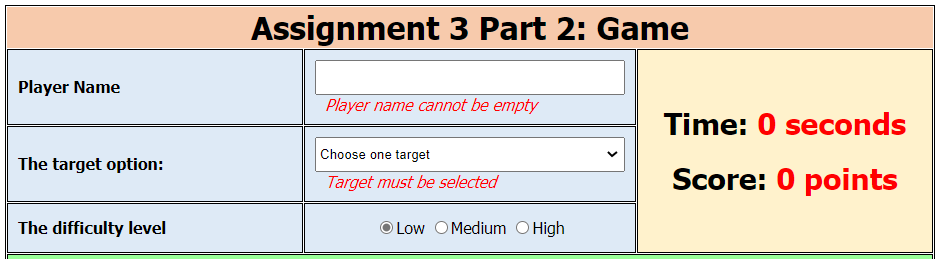
**For name validation**, we use includes() in line 138 in the picture above to check for existing array’s element. If name already exist in playerName, an error message will be shown like the picture below. Target option is also checked using else-if statement to make sure that a target is selected before start game.





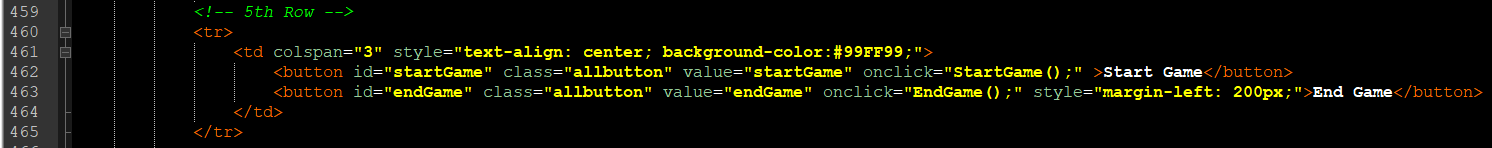


For the **target option**, <select> is used to display the options and <span> to display the error message in the picture below when target is not selected. This is checked when ValidateName() is called.

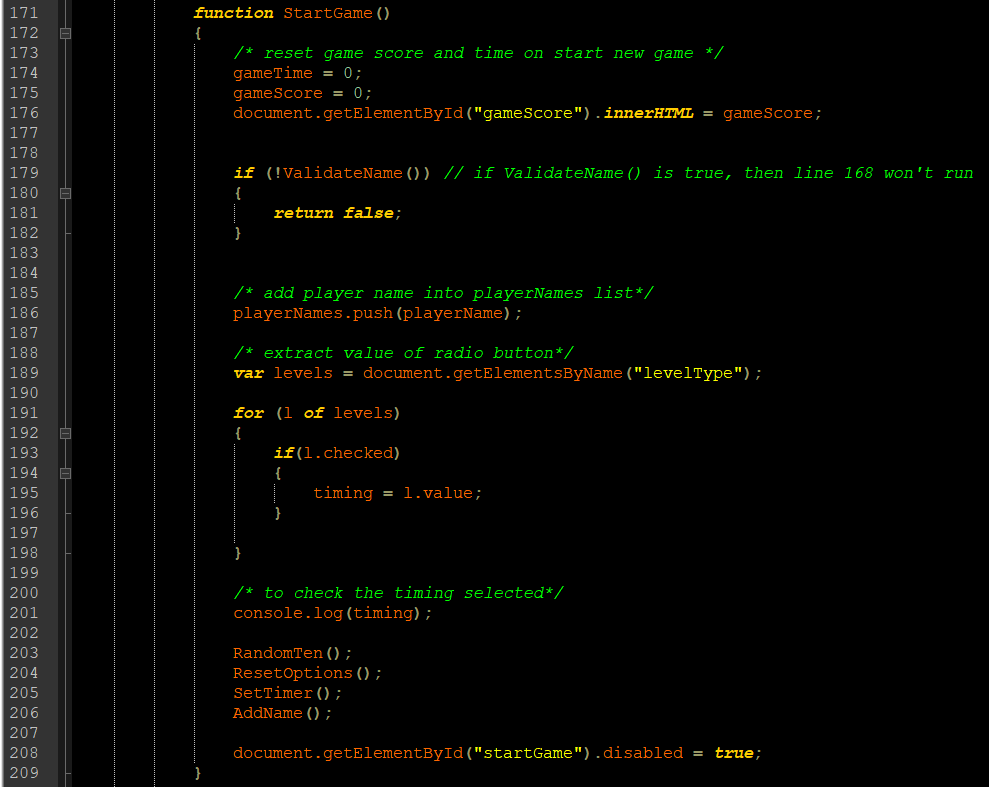




For the **difficulty level**, <input> type=”radio” is used and by default level Low is checked.



For the **start game and end game**, <button> is used. <onclick> event is used to call the function StartGame() and EndGame().



We need to reset gameTime and gameScore because StartGame() is called multiple times.

If there is error in ValidateName(), the function StartGame() will be terminated.

If there is no error in ValidateName(), player name input by user will be push() into playerNames array.

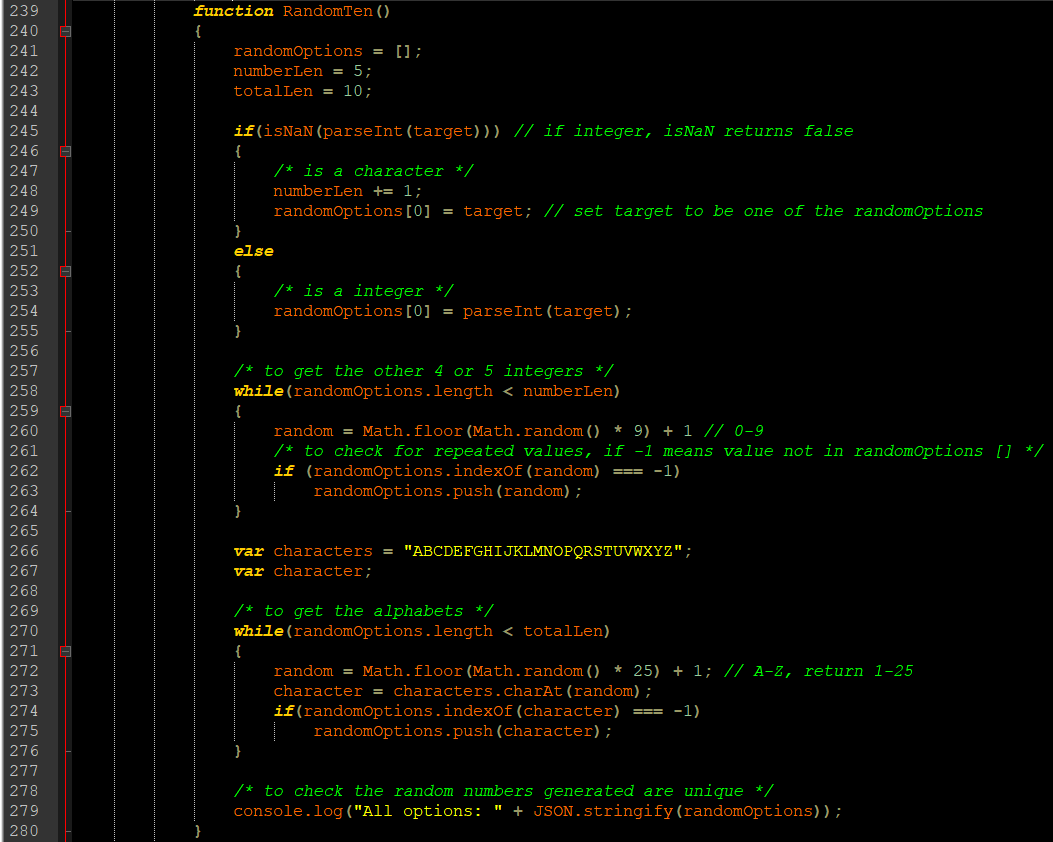
Next, we need to extract the value of radio button by declaring the local variable “levels”.  
since there are 3 levelType, we use **for-of** statement to extract each level and set timing according to the level that is checked.

We use **console.log(timing)** to check at background whether the timing is executed correctly.

Next, **RandomTen()** is called to generate 10 random options with our selected target in the random ten options.

**ResetOption()** is called after each interval to generate a new 3 random options.

**SetTimer()** is called to start the time count and AddName() is called to add the player name to the last row.



**randomOptions** will be set to empty because function **RandomTen()** will generate new list of 10 numbers whenever a new set of game is started.

**numberLen** is set to 5 as we need 5 numbers and 5 alphabets in the random ten numbers.

**totalLen** is set to 10 because we need 10 random numbers.

**If the target selected is an integer**, the if statement will be skipped and else statement will be executed. Target will be assigned to randomOption[0].

**If the target selected is an alphabet**, the if statement will be executed. numberLen needs to be +1 so that we still have at least 5 numbers in the randomOption array.

The **first while loop** is performed to add the rest of the numbers into the randomOptions array.

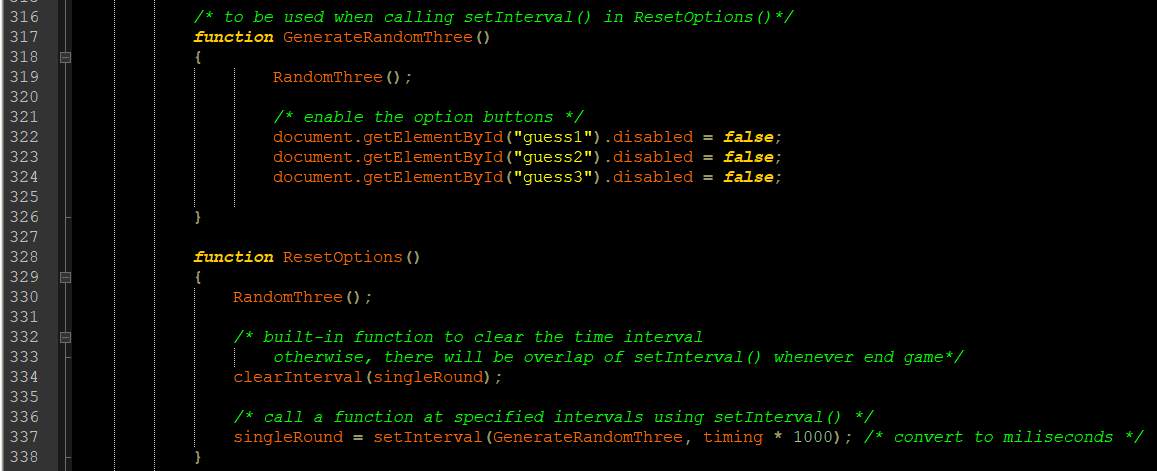
**If statement** randomOptions.indexOf(random) === -1 to make sure that the random number generated is not already existing in the randomOptions array then we can add it into the array.

Next, local variable “characters” is initialized with a string from A to Z. The local variable “character” is declared to be used in the second while loop.

**The second while loop** is performed to add the alphabets into the randomOptions array.

**If statement** randomOptions.indexOf(character) === -1 to make sure that the random number generated is not already existing in the randomOptions array then we can add it into the array.

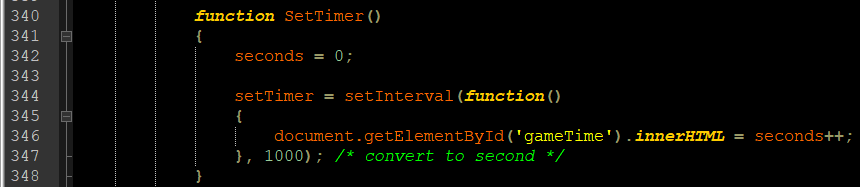
**console.log("All options: " + JSON.stringify(randomOptions));** is performed to check in the background to ensure that our randomOptions are unique and correct.



In the **ResetOptions()**, **RandomThree()** function is called to generate the 1st, 2nd and 3rd option.

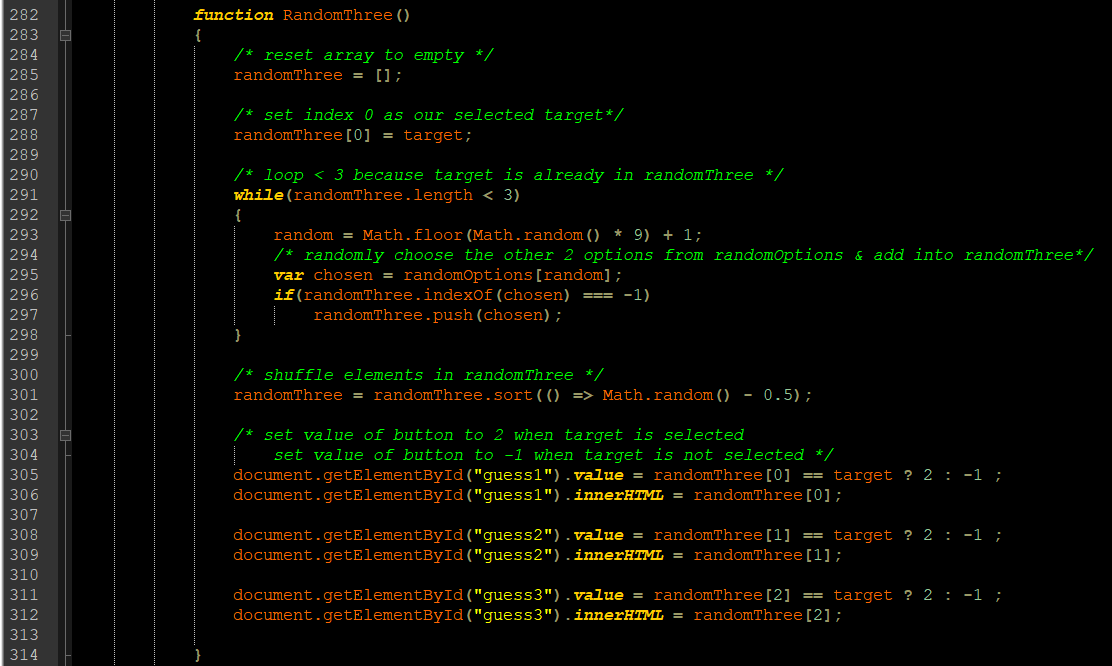
In each call of ResetOptions(), **clearInterval(singleRound)** is executed to ensure there is no overlap of setInterval().

**setInterval(GenerateRandomThree, timing \* 1000)** is executed to call the function GenerateRandomThree() every timing set.



In the **SetTimer()**, seconds is reset to 0 every time a new set of game is started.

setInterval() is used to update the second every 1 second.



In **RandomThree()** function, randomThree array is set to empty array so after every click on the option, there will be new 3 random numbers generated.

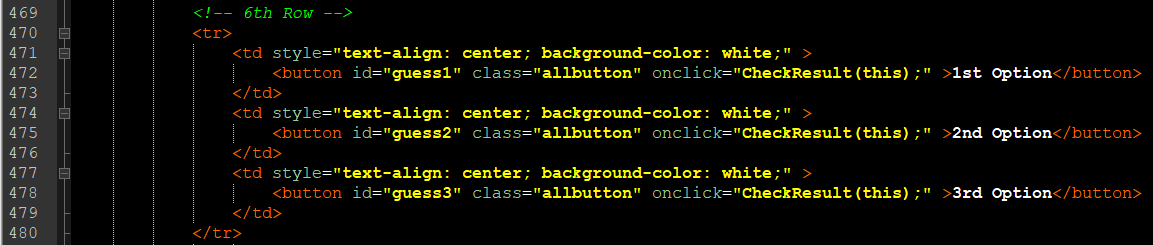
The **randomThree[0]** is set to be the target selected.

The **while loop** is performed to add in another 2 random values into the randomThree array.

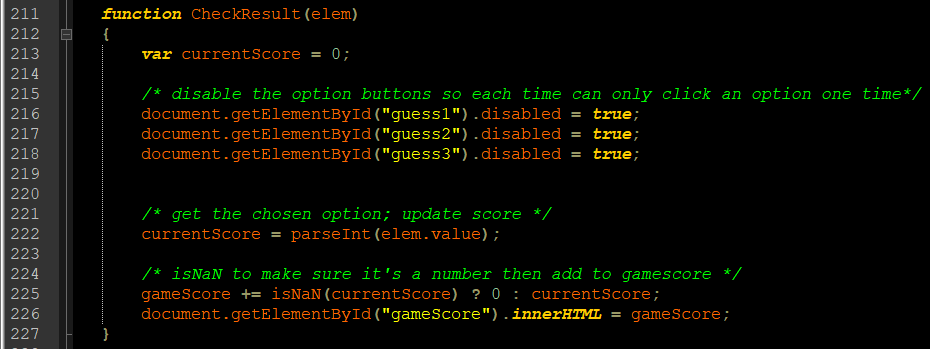
In the **while loop**, there is global variable “randomOptions” and a local variale “chosen”. There is random number generated between 0 to 9 so we can assign variable “chosen” with the value of randomOptions[random]. Next, we use **if statement** to check whether the chosen number exist in the randomThree. If the chosen number does not exist yet, it will be added into randomThree array using randomThree.push(chosen).

Since target is assigned to randomThree[0], we use **randomThree.sort(() => Math.random() - 0.5);** to shuffle the elements in randomThree.

Next, if option clicked is equal to target, we get value 2. If option clicked is not equal to target, we get value -1. This value will be used when we call the function CheckResult(elem)



Whenever the option buttons are clicked, using **onclick=”CheckResult(this)”**, the **CheckResult()** function will be called. We use “this” because there are 3 options. By using “this” as the argument for CheckResult(), whenever the button is clicked we do not have to check which button is clicked as “this” will directly pass in the value of the button to the function

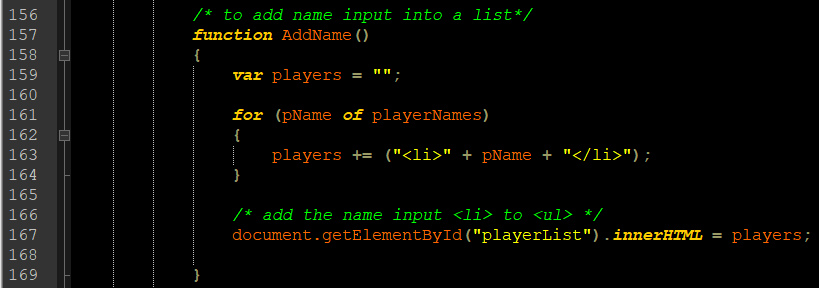


A local variable “currentScore” is declared and set to 0. The currentScore will later on be set to the value of option button that is clicked by user by using **parseInt(elem.value).**

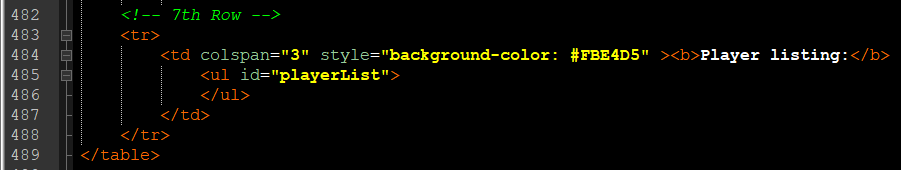
A global variable “gameScore” is used.

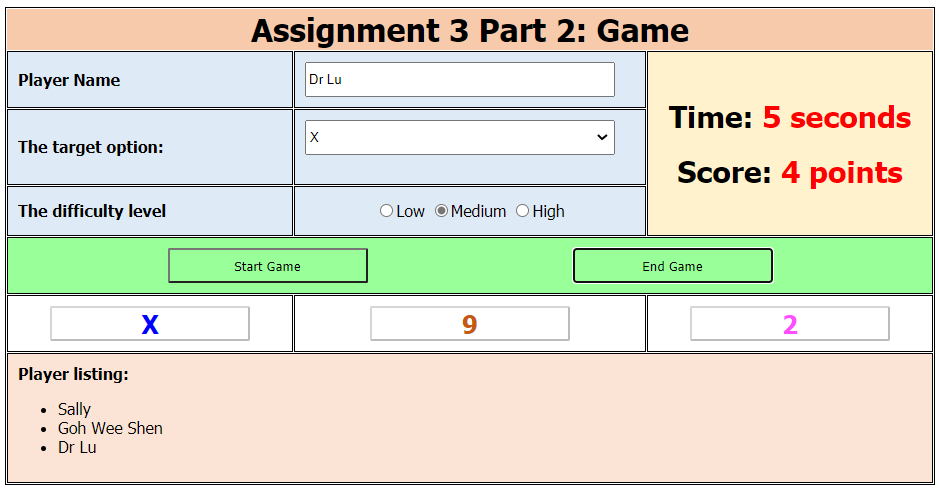
After each click, the option buttons will be disabled to prevent clicking and adding score.

isNaN is used to make sure it is a number before adding to gameScore the global variable.

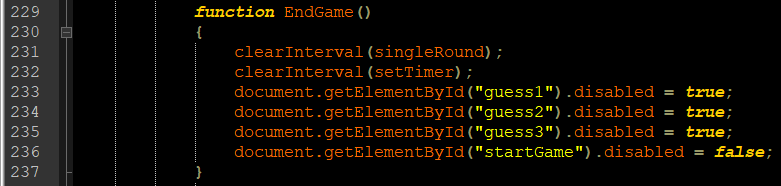


In the **AddName(),** global variable “playerNames” and **for-of** statement is used to add <li> to the <ul> in line 485 picture below.





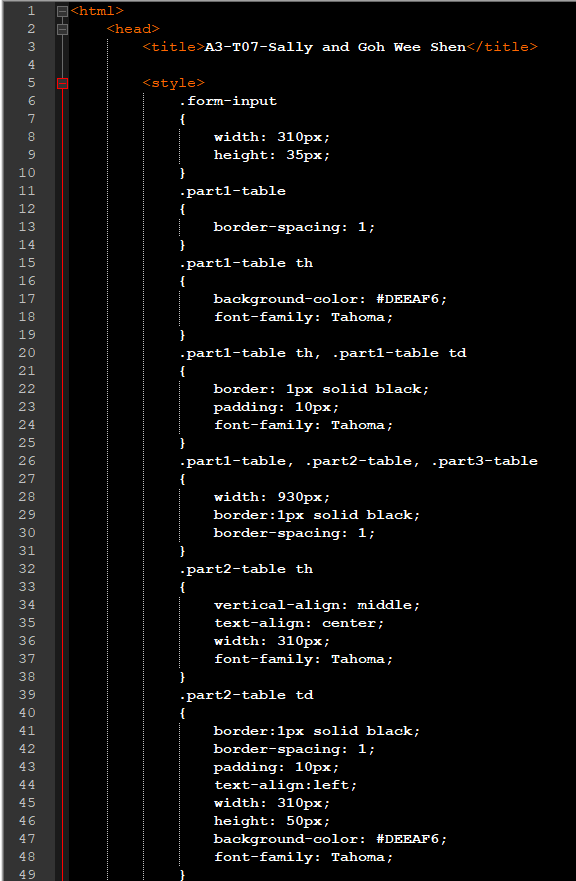
Name input is added to the last row of the table as shown in the picture above.

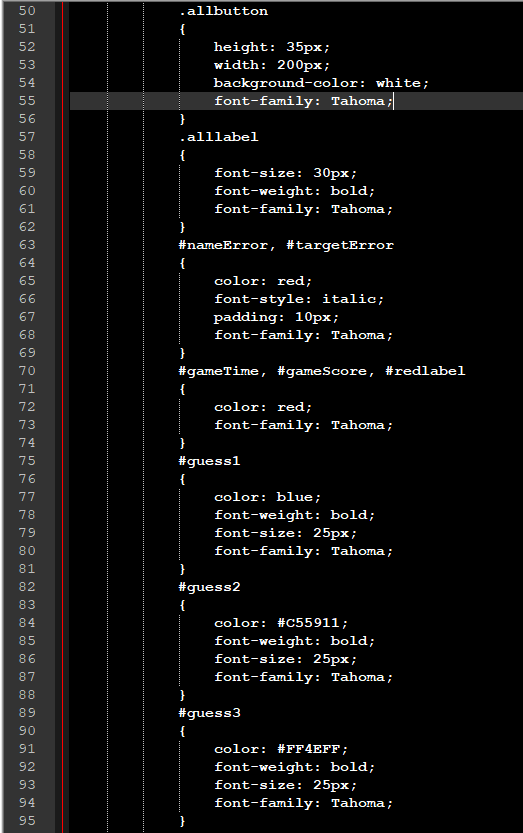


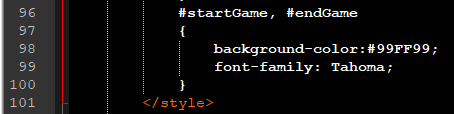
Next, for the **EndGame(),** **clearInterval(singleRound)** is called to prevent overlapping of time interval whenever new set of game is started. Also, **clearInterval (setTimer)** is called to reset the time count.

When End Game button is clicked, the three option buttons will be disabled.

For the CSS, we use inline CSS, id and class selector.

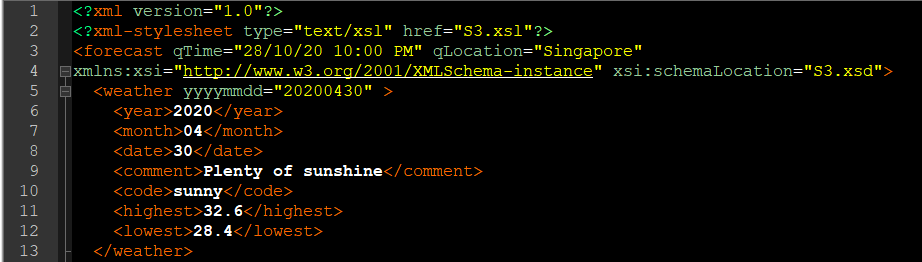






**Part 3 Explanation**

**XML**



The xml file is edited by adding line 2 and line 4 to access the xsl and xsd.

**XSD**



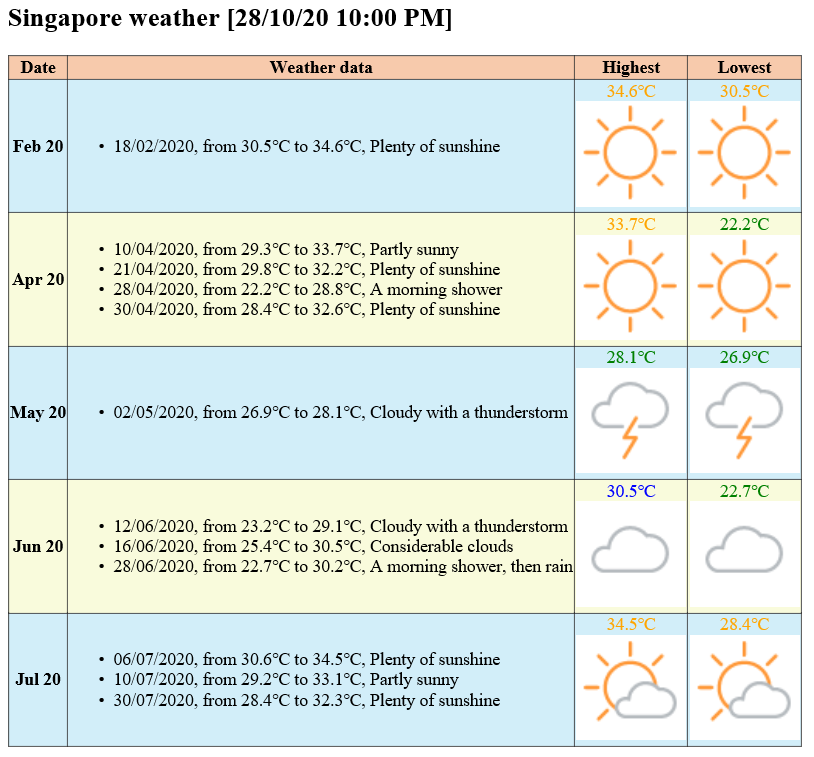
The element forecast is a complex type as it contains element weather and attribute qTime and qLocation.

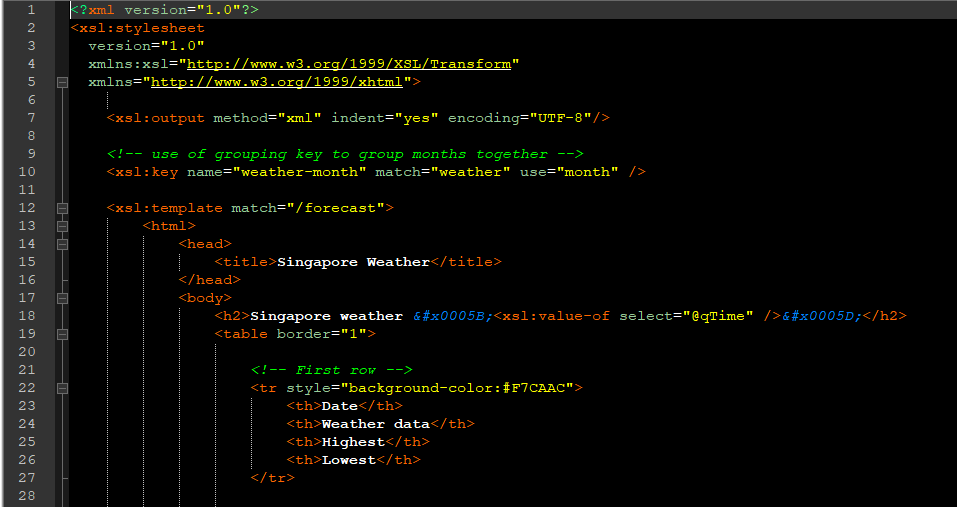
The element weather is a complex type as it contains attributes yyyymmdd and element: year, month, date, comment, code, highest, lowest.

The element: year, month, date, comment, code, highest, lowest are simpleType.

**XSL**

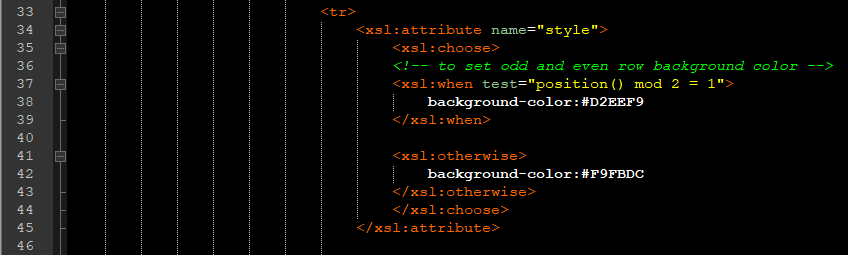
Below is the output of XML running with the xsl stylesheet.



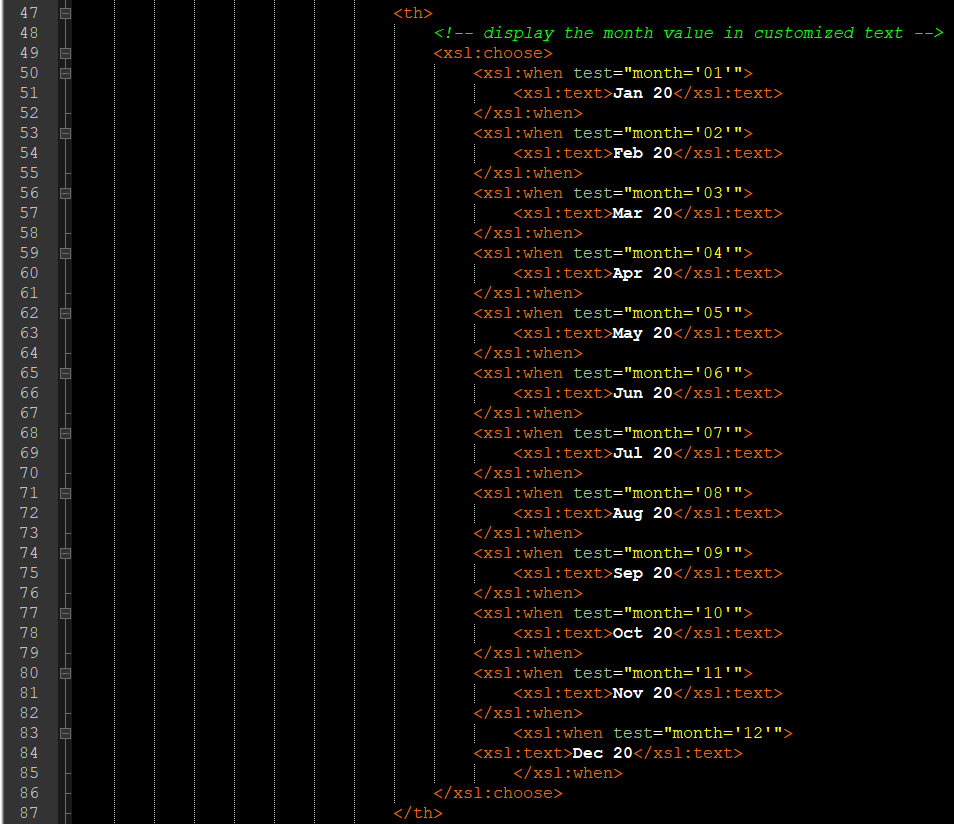


With the xsl:template match="/forecast", it will access the A3.xml file and find the matching <forecast> inorder to access the data.

The square bracket is written using entity/hexadecimal/decimal **&#x005B**

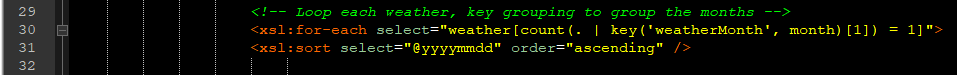


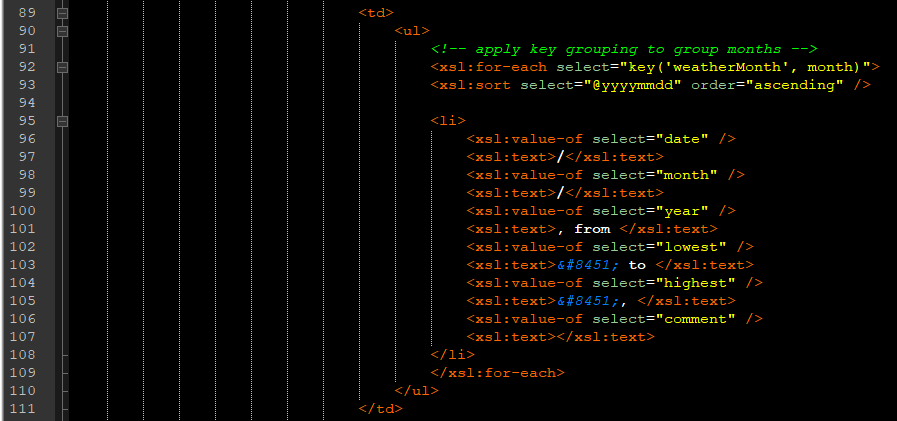
We have applied **<xsl:attribute>**, **<xsl:choose>** and **<xsl:when test=”position() mode 2 = 1">** to alternate the background color for the rows. It tests for odd and even rows.



For displaying the date in the format we need, simple <xsl:choose> and <xsl:when> were used to do the condition check, if condition matched, it will display the information within <xsl:text>information </xsl:text >. For this column we use <th> to bold the text.





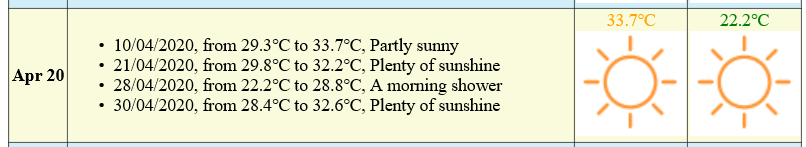


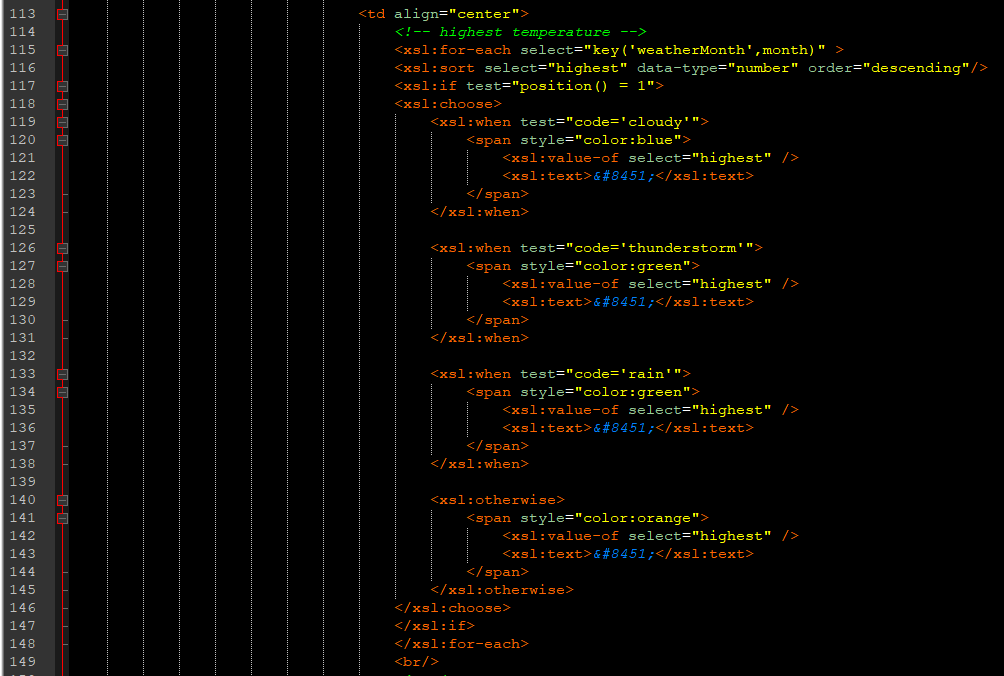
<xsl:key> is used to group the months. This method is searched from following website [XSLT grouping](http://www.jenitennison.com/xslt/grouping/muenchian.html).

<xsl:for-each> is used to loop through the weather and select the grouping key so the months can be grouped together. This will perform loop by different month instead of looping individual data.

<xsl:value-of select="element" />, will access the xml file and get the value of date to display.

Below is the output where the details of month April is shown in one row instead of multiple rows.





To get the different font color for different weather code, we use simple <xsl:choose> <xsl:when> and <span>.

The degree Celsius is written using &#8451.

To get the highest value of the temperature in the same row, we use <xsl:if test=”position() = 1”> with reference from the following website [To get maximum and minumum value under same node](https://stackoverflow.com/questions/17963067/how-to-get-maximum-value-under-same-node-in-xslt).

**Note that the sorting order for the temperature is descending, so the highest value will be in first position and the test=”position() = 1” can be applied.**

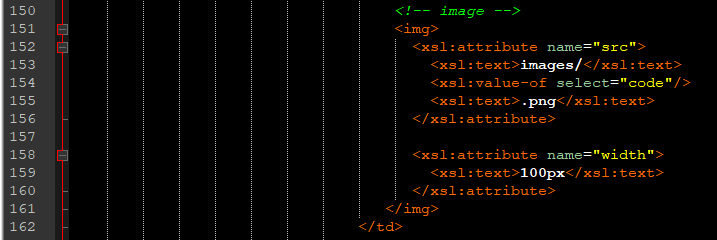
Before that we need to use the key grouping to group the data in one single row.



To get the highest value of the temperature in the same row, we use <xsl:if test=”position() = 1”> with reference from the following website [To get maximum and minumum value under same node](https://stackoverflow.com/questions/17963067/how-to-get-maximum-value-under-same-node-in-xslt).

**Note that the sorting order for the temperature is ascending, so the lowest value will be in first position and the test=”position() = 1” can be applied.**

Before that we need to use the key grouping to group the data in one single row.



To display the image, we created a folder and kept all our pictures in the folder named images.

<xsl:attribute> is used and the <xsl:text>images/</xsl:text> help to direct the path. The <xsl:value-of select="code"/> gets the name of the image to be display out, followed by the hardcoded .png file type.