Workshop 3

For this workshop, we will be starting from a modified version of last week's gradebook project. I have done a few things to simplify the project this week, since you will be adding a lot of new stuff to it, but you might want to compare the final version from this workshop to the final version from the last workshop to see different ways of handling the same data.

The biggest change is that I removed the requirement to pass the **gradebookData** and **assignmentData** arrays to functions. Because we were studying functions last week, I wanted to emphasize the different ways that data is passed (i.e., by reference or by value). However, it is also a valid choice to have a set of "global variables" that can be accessed by any object or function. The textbook has already mentioned one way of protecting this global scope, which is wrapping your entire script in an IIFE so that even "global" variables are scoped to the surrounding function. Another way is to create a "local global" object, which is just an ordinary named object that you create and then attach any global properties and methods to. For this week, however, we will use the global variable namespace built in to JavaScript.

Remember, any variable that is created outside of all functions is considered global, and any variable created within a function that is not declared with **var** (and is not created as a property of an object, e.g., **this.propName**) is also considered global in scope. The danger of having global data structures, even in a single script like this one, is that if several **asynchronous events** call functions that change the global data, cases may arise where one routine handles the data before another one in a way that you haven't counted on! This generally happens when you have a combination of remote network events (which can complete at any moment), timer events (which complete as close to a certain moment as they can), and user-initiated events (such as clicks and keypresses, which happen at the user's whim).

For a program like the one we are working on today, this is not a concern. We are only dealing with the user being able to click on one of two buttons, and button click events are generally queued up so they can't interfere with one another. Each of our functions will very quickly handle the event and return control to the user. However, we aren't dealing with event handling in a serious way until next week.

The focus of today's workshop is manipulating the DOM. Last week's gradebook could only be displayed once after it was built, and any additional rows or columns would not show up (though the data could be viewed via **console.log()**). ***In this week's workshop, you will enhance the gradebook so that the user can add new students and new assignments through a primitive user interface. The user will still have to enter new grades using the console, however, and the new grades won't update in the browser.***

After this workshop, your homework for this week will be to further enhance the gradebook so that the user can update grades via a user interface and can see the changes to the gradebook (including an updated total grade for the class) immediately.

The other major change that I made to the gradebook for this Workshop was the removal of the routine that drew the gradebook. Since you are writing new code that will much better handle the gradebook drawing, updating it as new data is added, it wasn't needed. However, this code also included my method for calculating the student's percentage grade in the class. This is another reason that you may want to refer back to the Workshop 2 solution when you do the homework: there are some hints there.

1. Download **Workshop3.zip** from the Lectures area of Week 3 on Blackboard and unzip it. Open the files in Edge Code or the text editor of your choice.

2. First, look at **index.html**. I have already done all of the work in this file for you, but there are a few things to observe:

- The table has a named **id**. This is so we can add new rows to the table using **appendChild**. A table is the parent of its rows, so table **gradebook** will be the parent to our rows of students.
- The one row in the table also has a named **id**. This is so we can add new cells to each row of the table using **appendChild**. A row is the parent of its cells, so even though we will always be adding an entire column of cells to the gradebook, each row will be in charge of adding its own cell to the column.
- I have added two simple buttons using the HTML5 **button** tag. We will cover these in much more detail next week when we learn about events. For this week, you just need to know that buttons, like any other DOM element, can have a named **id**. This is an attribute of the button and has to be formatted like any other variable. It is not the same as the text displayed to the user, which is a simple **text node** that appears between the opening and closing **button** tags.

3. Next, look at **workshop3.js**. This is where we will spend the rest of the workshop. Start at the bottom and look at the test function calls that I provided for Workshop 2. They are pretty much the same, except I removed the need to pass the two global arrays along with the input data.

***Our goal for the workshop is to write our new button-activated system in such a way that we can still also use these test cases*.** Why would we do that? Well, imagine this was an actual database program that could save out results to a file. We would want to be able to load that file back in and reconstruct our table, right? This would involve reading the data and recreating our students and assignments (tip: just saving the HTML-formatted table with its data would not be an effective storage method!), so we want to keep in the functions that will let us do that.

4. First, let's get the "Add Student" button wired up. To connect a button to a function, you use the **addEventListener** method, which connects a particular action (in this case a click on the button) to a function. The format is:

object.addEventListener('*eventName*', *functionName*, false);

The third value that must be passed to **addEventListener** actually serves a special purpose and is occasionally **true**, but it's usually **false**, so that's all we will say for now. The important notes are:

- The event name is always in quotes, i.e., 'click'
- The function name is NOT followed by a set of parentheses. Remember that a set of parentheses after a function name calls the function right away. We don't want to call the function now; we just want the "add student" button to know that it should call it if the button is clicked!
- Since the function name isn't followed by parentheses, we can't pass any values to the function in the way you've learned. We'll cover how to deal with this next week. For this week, we'll just call functions that don't have any parameters.

The two steps you'll need to take here are (1) get the "Add Student" button from the DOM using **document.getElementById** (watch the weird capitalization on that one!) then (2) assign the **click** event to the function **promptForStudentInfo**.

5. I have already written a small version of the **promptForStudentInfo** function that will pop up an alert box saying **"Prompting for Student Info!"** when you click the button, so open up your **index.html** in the web browser and see if your button does something!

6. Once you have the "Add Student" button working, try to do the same thing for the "Add Assignment" button. Have a click on this button call a function called **promptForAssignmentInfo**. Write a little test version of this function just like I did for **promptForStudentInfo** and make sure both buttons respond.

7. Now let's make the **promptForStudentInfo** function more useful. To do this, you will use the **prompt** command, which pops up a dialogue box and gives the user a text box in which to respond. Using **prompt** and **alert** are generally <u>not</u> good user interface design for something like this. These commands basically stop the user's interaction with the web page until dismissed. In the old days, they took over the entire web browser and you couldn't even switch windows until you got rid of them! We are only using them this week because we haven't yet covered forms and events.

The most common format for prompt is:

answerVariable = prompt("Question goes here");

There is an optional second parameter for **prompt** if you want to provide a default answer, but instead we are going to check the user's answers to make sure they aren't empty.

After deleting my **alert** line from the function, you should write the function to:

- Initialize local variables **firstName** and **lastName** to empty strings
- Using a **do...while** loop, repeatedly ask the user to "Please enter the student's first name" until an answer consisting of 1 or more characters is typed.
- Using a **do...while** loop, repeatedly ask the user to "Please enter the student's last name" until an answer consisting of 1 or more characters is typed.
- Call the **createNewStudent** function, passing it **firstName** and **LastName**

8. Test your function by loading the web page and pressing the "Add Student" button. You should be able to enter a student's first name and then last name. You won't see any changes to the page because we haven't written our update functions yet, but if you go to the console and type **gradebookData**, you should see a set of objects appear: the two students that were created by the test code and your new student. You could also type **gradebookData[2]** to see only the contents of the student you added, since we know the third student would be stored in array position 2.

9. Once you've gotten "Add Student" working, edit your **promptForAssignmentInfo** function to work similarly, calling function **createNewAssignment** at the end. However, be careful of the following changes:

- You should initialize **assignmentName** to an empty string, but you should initialize **assignmentValue** to the number 0.
- When you get back the result from the **prompt** for **assignmentValue**, you should convert the result from a String to a Number. You can do this as follows:

  assignmentValue = Number(prompt("Please enter a point value for the assignment:"));

  Putting a value inside of **Number()** tells JavaScript that you need the result treated as a number if at all possible. If the result can't be turned into a number, it will become **NaN**, the special value for "not a number". JavaScript is fairly loose about data typing, and, given the right context, it can often guess what you intend to do with a Number or a String and automatically make the conversion. However, this is the source of many hidden errors. In fact, I spent at least 15 minutes trying to debug this exercise because I forgot to explicitly ask for a Number here. The result was a very strange math error when the String version of my numeric input made it all the way to my grade calculating function! I'm not going to make you suffer through solving that problem yourself if you promise to learn from my mistake!

10. Thanks to some careful coding, we don't have to touch our **createNewStudent** and **createNewAssignment** functions, so let's move on to figuring out how to make new Assignments appear in the column header row. We're doing this first because it's the easiest bit, but the same techniques apply for creating the student rows and the assignment data cells within each of those rows. Start out by writing:

```
function addGradeColumn(assignmentName, totalPointValue) {
}
```

As a general hint, it's a good idea to close off your functions and parentheses immediately so that you don't forget later on. Many code editors will do this for you, leaving you to just fill in the stuff in between them. Edge Code does this for HTML tags, but for JavaScript, it instead highlights the opening curly brace, bracket, or parenthesis when you type or select a closing one so that you can make sure the sets match the way that you expect them to.

From this function declaration, you can see that we are working with the same values that we obtained from our earlier prompts. We'll worry about how to get those values from there to here later on. For now, you need to get the function to do the following:

- Find the **gradeHeaders** element in the **document** object and assign it to a local variable called **gradeHeaderRow**
- Assign the results of **document.createElement('td')** to a local variable called **gradeColumn**. Note that the method **createElement** takes as a parameter the quoted name of any tagged element ('li', 'p', 'table', etc.) and it returns a pointer to the newly created object.
- Calling **appendChild** from our **gradeColumn**, attach a new text node (created using **document.createTextNode**) with the following expression:

  ```
  assignmentName + " (" + totalPointValue + ")"
  ```

  As you might guess, the idea here is that something like "Assignment 1 (10)" will end up in the column header, just as we had in last week's workshop.

- Once you have **gradeColumn** all set up with its text node child, use **appendChild** again to attach the **gradeColumn** to **gradeHeaderRow**.
- End the function with

  ```
  return gradeColumn;
  ```

  We are going to store the column that corresponds to each assignment in our Assignment objects, so we need to have this data available to us later!

11. For this next step, we are going to be a bit naughty. It is good programming practice to separate out your data from your display code, but for just this exercise, we are going to have our new display columns get created when we create a new Assignment object. This means we will be adding our call to **addGradeColumn** to the Assignment object constructor function. This might seem quite logical— the column appears when we create it!— but how do we deal with getting rid of the

column's visual component if we delete the Assignment object? Some languages have a destructor function that is called when an object is removed from memory, but JavaScript doesn't (officially) have one. Fortunately, we aren't going to allow removal of Assignments this week, so we can be sloppy just this once.

All that being said, all you have to do is add this line to the bottom of your **Assignment** constructor function:

this.gradeColumn = addGradeColumn(assignmentName, totalPointValue);

You don't have to change anything, just write the line as-is. This line adds a gradeColumn property to each Assignment object. This property stores the DOM location of the corresponding grade column header so that we could, later on, update it if we changed the name or point value of an assignment.

12. Once you've done this step, you should be able to load up your page and use the "Add Assignment" button to add a new assignment column header! If that's working, then it's time to work on adding the students to our gradebook!

13. Adding a new student row is a bit more complicated than adding a gradebook column header. Remember that the DOM doesn't really treat tables as "rows" and "columns" as we do, but rather as "rows" and "data cells within rows". This means that we not only have to create a row for the student, but we also have to create all of the cells for the row: the student's ID and name, which we know, but also a cell for the student's cumulative class score and cells for each assignment that has already been created! To begin, look at the function header:

```
function addStudentRow(studentID, firstName, lastName) {
}
```

Notice that I changed "id" to **studentID** here. The beauty of function parameters is that they don't have to be named the same as the variables being passed to them (otherwise they wouldn't be so flexible), and since in this function we will want **id** to refer to the table cell containing the student's ID (not to be confused with the **id** attribute!), we use **studentID** to receive the number that represents the student.

I have written the first part of this function for you, but you should go through the lines of code to make sure you understand what is being created. After my eight lines of code, the first three cells of the row have been created and populated with the information passed to the function (We fill the text node within **studentPercent** with "0" to start because the we assume new students will have done no assignments yet).

14. To complete this function, you need to write a loop that will execute once for each assignment in the array **assignmentData** (HINT: You won't be using any data from **assignmentData**; you just need to count the **length** of it!). Each time through the loop, a new <td> element should be created and appended as a child of the studentRow, and a new TextNode containing the value "0" (zero, but quoted as a string) should be appended as a child of each new <td>.

15. After the loop completes, you should return **studentRow** to the calling function, which will be our **Student** constructor function. After you add the **return** statement to the end of **addStudentRow**, go up to the **Student** constructor function and add this line to the end:

this.studentRow = addStudentRow(this.id, firstName, lastName);

Note that **this.id** had to be passed instead of just **id** because **id** is actually a property of Student. **firstName** and **lastName** could have also been passed using **this** since they are also properties, but here I chose to pass the same variables that came in to the Student constructor, and thus didn't use the **this**. Forgetting to use **this** when you need it is a very common mistake, so look out for it as you work!

16. Now test your code again. Add an Assignment, then add a Student. If all goes well, your new Student will have all of the current Assignments! Next, add another Assignment and another Student. What happens? Your second new student gets all of the Assignments, but your previous students don't get the new Assignment!

17. We ran into a similar problem last week, but between what we learned then and what you just learned here, you should be able to modify function **addGradeColumn** to iterate through all of the student rows (good thing we stored **studentRow** on each **Student** object, right?) and add a new <td> filled with a "0" to the end of the row. HINT: You will iterate through the **gradebookData** array, appending a newly created **td** element as a child of each **studentRow** and then append a newly created **TextNode** element as a child of each **td** that you've just created.

18. Test your page one more time. At this point, when you load the page, you should see Adam Anders, Beth Booker, and Homework #1 and Homework #2. There should be 0 scores for everything including their total scores. If you add a new student or a new assignment, you should see a new row or column appear with 0s in all of the right places. At this point, congratulations! You have built the foundation for your homework assignment, which involves dealing with the inability to SEE updates to the student grades. Although you have done very well in getting this far, I still recommend that you start the homework from the provided starter files that will unlock at the end of this workshop, since I have provided additional hints and comments there!