MDIA 2294 Week 8 In Class Exercises

# Overview

The first three examples this week will mostly serve to review weeks 6 and 7 as well as introduce a couple of smaller new ideas.

Exercise 4 will introduce the idea of user defined objects, which will be expanded upon next week.

# Exercise Steps

The following notations are used:

* **DETAILS**: describes what you will do or suggests that you review some code to understand what's going on. Read this first to ensure you understand what you are about to do
* **STEPS**: these will be actions that you perform. The steps will often be followed by a code block
* Greyed out code is not to be entered and is provided to indicate where the new code should go
* You are NOT recommended to copy/paste code. Often characters will not paste properly from the pdf document and may result in code not working. Also typing code manually will assist memorization.
* **NOTE**: any things to note or explanations on what you just did. Sometimes this will include potential variations or other applications.
* **TEST**: anything you should test for.
* Make sure to save first and then refresh in the browser
* Often if you aren't seeing results you are possible looking at a different file, try changing the <title> tag to double check this

# WK8\_EX1

This exercise is another example of timers both setting as well as clearing an interval. It also introduces a couple of methods of class **Math**.

## Part 1: review the file

* **DETAILS**: the starter file **WK8\_EX1.html** should be copied and added to your local working folder.
* **STEPS**: take a minute to review the file

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>MDIA 2294</title>

</head>

<body>

<h1>MDIA 2294: WK8\_EX2</h1>

<div>Ask your question and then

<button id="cmdShake">Answer</button></div>

<div>Answer: <span id="spnAnswer"></span></div>

<script>

const answers = [ "It is certain",

"It is decidedly so",

"Without a doubt",

"Yes, definitely",

"You may rely on it",

"As I see it, yes",

"Most likely",

"Outlook good",

"Yes",

"Signs point to yes",

"Reply hazy try again",

"Ask again later",

"Better not tell you now",

"Cannot predict now",

"Concentrate and ask again",

"Don't count on it",

"My reply is no",

"My sources say no",

"Outlook not so good",

"Very doubtful"];

</script>

</body>

</html>

* **NOTE**: a script block has already been added and contains a single global variable **answers** which you might recognize as Magic 8 Ball answers. Notice that there are 20 answers, which as they are in an array count from 0 to 19
* **NOTE**: the HTML portion has a single button that will call a function **get\_awnsers()**, which hasn't been added yet. There's also a span for writing an answer from the Magic 8 Ball

## Part 2: Get a random answer

* **DETAILS**: first we'll get a single random answer from the array. Random numbers in computers are generally obtained from 0 to .9999999, then if a specific value is needed we multiply the random number by a scaling factor and then add a shifting factor if need be.
* **STEPS**: add the following function just under the array, at first we'll just use it to see how random numbers work.

const get\_answer = () => {

const roll = Math.random();

document.querySelector("#answer").innerHTML = roll;

} // end of get\_answer

</script>

* **TEST**: if you run the page and check the button you should see that it varies between 0 and 0.9999999999999999
* **STEPS**: next we'll scale the value by multiplying it by the upper range, in this case 19 which is the highest index in the array

function get\_answer()

{

const roll = Math.random() \* 19;

document.querySelector("#answer").innerHTML = roll;

} // end of get\_Answer

* **TEST**: now if you test you'll see that values range from 0 to 18.99999999999
* **STEPS**: next we'll wrap the expression in **Math.round()** to round off the number since otherwise it won't match the array indexes

function get\_answer()

{

const roll = Math.round(Math.random() \* 19);

document.query\_selctor("#answer").innerHTML = roll;

} // end of get\_answer

* **TEST**: now if you test you'll see that values range from 0 to 19
* **STEPS**: finally replace the writing of the variable to the span with the corresponding array index

function get\_answer()

{

const roll = Math.round(Math.random() \* 19);

document.querySelector("#answer").innerHTML = answers[roll];

} // end of get\_answer

* **TEST**: if you test now you should see that random answers from the array are written instead

## Part 3: using a timer and ondblclick

* **DETAILS**: as a review of timers we'll also add functionality to shake the 8 ball to see a number of answers before the final one. This example will also demonstrate that an element can have multiple event handlers attached to it in this case both the **click** as well as **dblclick**
* **STEPS**: first you'll need to add the global variable **timer** to store the interval id as well as another which variable progress which is used to track the number of intervals passed.
* **STEPS**: also add the **shake()** function which will set the interval to call a function **update\_progress()** every 50 milliseconds.

"Outlook not so good",

"Very doubtful"];

let progress = 0;

let timer;

const shake = () => {

timer = setInterval(updateProgress, 50);

} // end of shake

* **STEPS**: now you'll add the **update\_progress()** function which is the one called by the interval. This function has a conditional which checks if progress is at 20 yet.
* If it’s less than or equal to 20, then a new random answer is shown (using the same **get\_answer()** function) and the progress is incremented by one
* If it's at 20, the interval is cleared (killing the timer) and progress is reset to zero for the next go.

const update\_progress = () => {

if(progress <= 20){

get\_answer();

progress++;

}

else{

// clear the timer and reset the progress for next time

clearInterval(timer);

progress = 0;

}

} // end of update\_progress

* **STEPS:** finally we have to invoke the **shake()** function. To do this we'll use the same button as before but for a shake the user will need to double click (i.e. **ondblclick**). Also change the button text to indicate this option

<div>

shake\_btn.addEventListener("click", get\_answer);

shake\_btn.addEventListener("dblclick", shake);

</div>

* **TEST**: you should be able to test your file both with a single as well as double click

# WK8\_EX2

This exercise is a simple review of validating required entry. However it's also used to show another new event handler, **onchange**, as well as to introduce keyword **this**.

## Part 1: review the file

* **DETAILS**: the starter file **WK8\_EX2.html** should be copied and added to your local working folder.
* **STEPS**: take a minute to review the file

<h1>MDIA 2294: WK8\_EX3</h1>

<table>

<tr>

<td><b>First:</b></td>

<td><input id="first-name" type="text" class="required" data-error-id="first-name-error" />

<span id="first-name-error"></span></td>

</tr>

<tr>

<td>Middle: </td>

<td><input id="middle-name" type="text" /></td>

</tr>

<tr>

<td><b>Last:</b> </td>

<td><input id="last-name" type="text" class="required" data-error-id="last-name-error" />

<span id="last-namer-error"></span></td>

</tr>

<tr>

<td><b>Email:</b> </td>

<td><input id="email" type="text" class="required" data-error-id="email-error" />

<span id="email-error"></span></td>

</tr>

</table>

<div><button id="submit">Submit</button></div>

* **NOTE**: as in previous examples we have both a textbox for data as well as a span next to it for error messages. Also like exercise 1 we use a class to select the desired elements (in this case text boxes that require entry) as well as an attribute for the span to add the notification to

## Part 2: add the submit\_form function

* **DETAILS**: the first function is mostly review.
* **STEPS**: add the following script block and function

<script>

const validate\_form = () => {

const required\_fields = document.querySelectorAll("required");

required\_fields.forEach((field, index) => {

if(field.value.trim() == ""){

document.querySelector("#" + field.dataset.errorId).innerHTML =

"&bull;";

}

else{

document.querySelector("#" + field.dataset.errorId).innerHTML = "";

}

});

} // end of validate\_form

</script>

* **NOTE**: this should be familiar at this point as all elements here have been seen a few times but to review:.
* Create the collection of required elements from the class name and loop through them
* If an element has no value (trimmed to remove spaces) then write a bullet into the span
* Otherwise clear the span in the case that there was a bullet there from before.
* **TEST**: if you test the form you should see notifications for required (bold) fields that are missing text

## Part 3: use onchange event handlers to clear warning as they are corrected

* **DETAILS**: instead of having to click the submit button every time to see if some fields are invalid it's more user friendly to let them know as data is being entered. To that end we'll make a function that checks a single field and call it as the data in fields changes
* **STEPS**: add another function **update\_field()** to the script block

<script>

const update\_field => (field) {

if(field.value.trim() == ""){

document.querySelector("#" + field.dataset.errorId).innerHTML = "&bull;";

}

else{

document.querySelector("#" + field.dataset.errorId).innerHTML = "";

}

### } // end of update\_field

* **NOTE**: the function itself is very similar to the previous with the exception that it doesn’t loop through a collection. Instead notice the parameter variable **caller** which will be a reference to the textbox to check.
* **DETAILS**: the trick here is that the **change** event listener for the given text box will pass a reference to itself to the update\_field function. This is done using keyword **e.target**, which here means the target of the event (the field that was changed).
* **STEPS**: add the following event handler to each of the three required text boxes

document.querySelectorAll(".required").forEach( field => {

field.querySelector("change", e => {

update\_field(e.target);

});

}

### } // end of update\_field

* **TEST**: take a look at your page, notice if you submit your form, get the warnings and then fix the issues by adding data once you leave the text box the change event is triggered. If your change meets the criteria (i.e. some text and not just empty spaces) the bullet will vanish.

# WK8\_EX3

This exercise will introduce user defined objects. Like DOM objects, you can create objects that encapsulate multiple data properties as well as internal methods. Objects can be used to help manage data from an outside source such as a data file, database. Additionally they are often used in conjunction with third party JavaScript libraries such as jQuery.

**NOTE:** the most important part of this exercise for most students will to understand is the literal object notation from part 2. The concepts in the remaining parts are a little more advanced and presented here to give an indication of how you’d use classes in the future.

## Part 1: review the file

* **DETAILS**: the starter file **WK8\_EX3.html** should be copied and added to your local working folder.
* **STEPS**: take a minute to review the file

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>MDIA 2294</title>

</head>

<body>

<h1>MDIA 2294: WK8\_EX4</h1>

<div>Example 1: <span id="example-1"></span></div>

<div>Example 2: <span id="example-2"></span></div>

<div>Example 3: <span id="example-3"></span></div>

<div><button id="examples-btn">Show Examples</button></div>

</body>

</html>

* **NOTE**: the file is quite simple with three areas to write into. We'll use these to write the values of three object instances.

## Part 2: create a literal object

* **DETAILS**: objects can be created as literals in a similar fashion to arrays. The key differences are:
* Object definitions are encased in the curly brackets instead of square brackets.
* Object members (**properties**) have the property name, a colon and then the value.
* **STEPS**: add the following script block with the two functions, **showExamples** and **displayMovie**

<script>

const show\_examples = () => {

// first example literal object no issues

const movie\_1 = {

title: "Everything Everywhere All at Once",

director: " Daniel Kwan and Daniel Scheinert",

year: 2022

};

document.querySelector("#example-1").innerHTML = display\_movie(movie\_1);

} // end of show\_examples

const display\_movie = movie => {

return movie.title + " by " + movie.director + " (" + movie.year + ")";

} // end of dipsplay\_movie

</script>

* **NOTE**: the object we created is a collection of data about a movie. In the declaration we've set values for three properties: **title**, **director** and **year**. Notice that the string values are in quote while the year is not as we want to store it as a number
* **NOTE**: also specifically pay attention that unlike most blocks the object block has a **semi-colon** after it to close the definition
* **NOTE**: we've also added a function called **display\_movie()** that has a parameter of the passed movie object. Notice that from this instance we can access the three parameters via dot notation. In this example we simply concatenate some of the data together.
* **TEST**: run the page, click the button and the movie details should be added to the first span

## Part 3: demonstrating a risk with literal definitions

* **DETAILS**: one of the problems with defining objects in this way is that if you are creating more than one you can easily misspell, forgot or otherwise create properties inconsistently
* **STEPS**: add another **literal object** for a second movie.

function showExamples(){

  // first example literal object no issues

const movie\_1 = {

title: "Everything Everywhere All at Once",

director: " Daniel Kwan and Daniel Scheinert",

year: 2022

};

document.querySelector("#example-1").innerHTML = display\_movie(movie\_1);

// first example literal object no issues

const movie\_2 = {

title: "Entergalatic

director: "Flecther Morris",

year: 2022

};

document.querySelector("#example-2").innerHTML = display\_movie(movie\_2);

} // end of showExamples

* **NOTE**: since white space between commas doesn't matter we've line-returned to give each property its own line. Conventionally this is a more common approach than the first part.
* **TEST**: However, notice that we've mistyped **year** as **year\_made** so when you run the page you should see **undefined** for that property. You can fix it if you'd like but it's unnecessary as the point is proven

## Part 4: constructing an object

* **DETAILS**: When we are dealing with more than one instance of an object or using objects in a re-usable fashion it makes sense to create the structure for the object and then create objects from this definition which is known as a **constructor function**.
* **STEPS**: first add the constructor function block at the top of the script

<script>

// this is a Class

// convention is that the name is capitalized

class Movie {

constructor(t, d, y) {

this.title = t;

this.director = d;

this.year = y;

}

} // end of class Movie

function show\_examples(){

* **NOTE**: there are a few things to pay attention to in the syntax of the constructor
* First the convention for constructors is that they begin with capital letters as in this case with **Movie**
* The passed parameters are assigned to the properties which are defined by instantiating them using keyword **this**
* Finally a call to this constructor will return the constructed object but you don't need to use keyword return as it is implied
* **STEPS**: now try constructing an instance of the object in you **show\_examples()** function

document.getElementById("spnBeta").innerHTML = displayMovie(myMovieBeta);

// third example we call the constructor

const movie\_3 = new Movie("Brick", "Rian Johnson", 2005);

document.querySelector("#example-3").innerHTML = display\_movie(movie\_3);

} // end of show\_examples

* **NOTE**: you can see that constructor is called by way of keyword **new**, which we've seen before when constructing DOM objects such as **Dates**. Otherwise the object behaves as the other two do, the key differences here are:
* We can reuse the same call for all instances of the same object which besides being cleaner should also cut down on potential inconsistencies
* We could build data correction into the constructor for example defaulting to values that haven't been provided

## Part 5: adding a method

* DETAILS: finally we'll add a method to the object, in this case it will provide the same functionality as **display\_movie()**.
* **STEPS**: add the following method to the **Movie** class constructor

// this is a Class

// convention is that the name is capitalized

class Movie {

constructor(t, d, y) {

this.title = t;

this.director = d;

this.year = y;

}

full\_title() {

return this.title + " by " + this.director + " (" + this.year + ")";

}

} // end of class Movie

* **NOTE**: like properties a method is also defined using keyword **this**. This method is quite simple but all the rules we've learned so far about function **parameterization** as well as **return** values still apply.
* **STEPS**: change the statement in the **showExamples()** function to call the object method **toDisplayString()** instead of the generic **displayMovie()** function

~~document.querySelector("#example-3").innerHTML = display\_movie(movie\_3);~~

document.querySelector("#example-3").innerHTML = movie\_3.full\_title();

} // end of showExamples

* **NOTE**: notice that the object calls the method by way of the dot operator, rather than by being passed as a parameter.